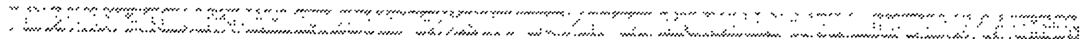
THINKING MACHINES CORPORATION

CM5

TECHNICAL SUMMARY

The Connection Machine CM-5 Technical Summary



October 1991

Thinking Machines Corporation
Cambridge, Massachusetts

October 1991

****

THINKING MACHINES CONFIDENTIAL

The information in this document is confidential and proprietary to Thinking Machines Corporation. It is the property of Thinking Machines Corporation and shall not be disclosed to persons outside the company or generally distributed within the company.

The information in this document is subject to change without notice and should not be construed as a commitment by Thinking Machines Corporation. Thinking Machines Corporation assumes no responsibility for any errors that may be contained in this document.

****

Connection Machine® is a registered trademark of Thinking Machines Corporation.
CM, CM-1, CM-2, CM-2a, CM-200, CM-5, and DataVault are trademarks of Thinking Machines Corporation.
CMOST and Prism are trademarks of Thinking Machines Corporation.
C*® is a registered trademark of Thinking Machines Corporation.
Paris, *Lisp, and CM Fortran are trademarks of Thinking Machines Corporation.
C/Paris, Lisp/Paris, and Fortran/Paris are trademarks of Thinking Machines Corporation.
CMMD, CMSSL, and CMX11 are trademarks of Thinking Machines Corporation.
Thinking Machines is a trademark of Thinking Machines Corporation.
VAX, ULTRIX, and VAXBI are trademarks of Digital Equipment Corporation.
Symbolics, Symbolics 3600, and Genera are trademarks of Symbolics, Inc.
Sun, Sun-4, SunOS, Sun Workstation, and SPARCstation are trademarks of Sun Microsystems, Inc.
UNIX is a registered trademark of AT&T Bell Laboratories.
The X Window System is a trademark of the Massachusetts Institute of Technology.
Motif is a trademark of The Open Software Foundation, Inc.
StorageTek is a registered trademark of Storage Technology Corporation.
Ethernet is a trademark of Xerox Corporation.
VMEbus is a trademark of Motorola Corporation.
AVS is a trademark of Stardent Computer Inc.
Explorer is a trademark of Silicon Graphics, Inc.

Copyright © 1991 by Thinking Machines Corporation. All rights reserved.

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

Contents

Part I Introduction

Chapter 1 Supercomputing and Parallelism **3**

1.1 Parallelism 4
1.2 Parallel Programming 5
1.3 Advantages of a Universal Architecture 9
1.4 Looking Ahead 11

Chapter 2 The Basic Components of the CM-5 .... **13**

2.1 Processors 13
2.2 Networks 15
2.3 I/O 17
2.4 A Universal Architecture 18

Chapter 3 Data Parallel Programming **19**

3.1 Data Sets and Distributed Memory 19
3.2 Interconnected Data Structures 21
3.3 Interprocessor Communications 22
3.4 Conditionals 27
3.5 In Summary 27
3.6 More Information To Come 29

Part II CM-5 Software

Chapter 4 Connection Machine Software 33

4.1 Base System Software 33
4.2 Languages and Libraries 33
4.3 CM Software Summarized 35

Chapter 5 The Operating System: CMOST 37

5.1 CMOST and the CM-5 Architecture 38
5.2 CMOST and the Users 40
5.3 CMOST and the Administrator 41
5.4 I/O and File Systems 41

Chapter 6 The Programming Environment: Prism 45

6.1 Using Prism 46
6.2 Analyzing Program Performance 47
6.3 Visualizing Data 48
6.4 On-Line Help and Documentation 49

Chapter 7 The Program Execution Environment 51

7.1 Checkpointing 51
7.2 Timers 52
7.3 Timesharing 52
7.4 NQS 53

Chapter 8 The CM Fortran Programming Language 55

8.1 Structuring Parallel Data 55
8.2 Computing in Parallel 57
8.3 Communicating in Parallel 58
8.4 Transforming Parallel Data 60

Chapter 9 The C* Programming Language **63**

9.1 Structuring Parallel Data 63
9.2 Computing in Parallel 65
9.3 Communicating in Parallel 67
9.4 Transforming Parallel Data 68

Chapter 10 The *Lisp Programming Language **71**

10.1 Structuring Parallel Data 72
10.2 Computing in Parallel 74
10.3 Communicating in Parallel 76
10.4 Transforming Parallel Data 77

Chapter 11 CM Scientific Software Library **79**

11.1 Linear Algebra Routines 79
11.2 Fast Fourier Transforms 86
11.3 Random Number Generators 87
11.4 Statistical Analysis 88

Chapter 12 Data Visualization **89**

12.1 A Distributed Graphics Strategy 89
12.2 An Integrated Environment 90
12.3 The X11 Protocol 91
12.4 The CMX11 Library 91
12.5 Visualization Environments 92

Chapter 13 CM Message Passing Library **93**

13.1 Initialization 94
13.2 Message Passing 94
13.3 Informational Routines 95
13.4 Global Synchronization 95
13.5 Global Operations 97

Part III CM-5 Architecture

Chapter 14 Architecture Overview **101**

14.1 Processors 101

14.2 Networks and I/O 103

14.3 Further Information 105

Chapter 15 The User-Level Virtual Machine **107**

15.1 Communications Facilities 108

15.2 Data Parallel Computations 110

15.3 Low-Level User Programming 114

Chapter 16 Local Architecture **115**

16.1 Control Processor Architecture 115

16.2 Processing Node Architecture 116

16.3 Vector Unit Architecture 120

Chapter 17 Global Architecture **131**

17.1 The Network Interface 131

17.2 The Control Network 133

17.3 The Data Network 137

Chapter 18 System Architecture and Administration **139**

18.1 The System Console 139

18.2 Allocation of Resources 140

18.3 Partitions and Networks 141

18.4 Resource Allocation and Management 143

18.5 Accounting, Monitoring, and Error Reporting 143

18.6 Physical Monitoring Systems 144

18.7 Fault Detection and Recovery 144

Chapter 19 Input/Output Subsystem **147**

19.1 I/O Architecture 148

19.2 File System Environment 149

19.3 I/O Interfaces and Device Implementation 152

Part I
Introduction

Chapter 1

Supercomputing and Parallelism

The Connection Machine system CM-5 provides high performance plus ease of use for large, complex, data-intensive applications. Its architecture is designed to scale to teraflops or teraops performance for terabyte-sized problems. It features

- independent scalability of processing, communication, and I/O
- extremely high floating-point and integer execution rates
- high processor-memory bandwidth
- efficient execution of high-level languages
- multiple job execution, both timeshared and partitioned
- multi-user network access
- security between users
- flexible high-bandwidth I/O
- balanced scalar and parallel execution
- balanced I/O, processing, and memory
- high reliability and high availability

The CM-5 continues and extends support for the parallel programming model that has proved so successful in the CM-2. To achieve its goals, the CM-5 takes advantage of the latest developments in high-speed VLSI, new compiling technologies, RISC microprocessors, operating systems, and networking. It combines the best features of existing parallel architectures — including fine- and coarse-grained concurrence, MIMD and SIMD control, and fault tolerance — in a single, integrated, "universal" architecture.

1.1 Parallelism

One of the most notable advances in computing technology over the past decade has been in the use of parallelism, or concurrent processing, in high-performance computing. Of the many types of parallelism, two are most frequently cited as important to modern programming:

- *control parallelism*, which allows two or more operations to be performed simultaneously. (Two well-known types of control parallelism are *pipelining*, in which different processors, or groups of processors, operate simultaneously on consecutive stages of a program, and *functional parallelism*, in which different functions are handled simultaneously by different parts of the computer. One part of the system, for example, may execute an I/O instruction while another does computation; or separate addition and multiplication units may operate concurrently. Functional parallelism frequently is handled in the hardware; programmers need take no special actions to invoke it.)

- *data parallelism*, in which more or less the same operation is performed on many data elements by many processors simultaneously.

While both control and data parallelism can be used to advantage, in practice the greatest rewards have come from data parallelism. There are two reasons for this.

First, data parallelism offers the highest potential for concurrency. Each type of parallelism is limited by the number of items that allow concurrent handling: the number of steps that can be pipelined before dependencies come into play, the number of different functions to be performed, the number of data items to be handled. Since in practice the last of these three limits is almost inevitably the highest (being frequently in the thousands, millions, or more), and since data parallelism exploits parallelism in proportion to the quantity of data involved, the largest performance gains can be achieved by this technique.

Second, data parallel code is easier to write, understand, and debug than control parallel code.

The reasons for this are straightforward. Data parallel languages (such as the Connection Machine system's CM Fortran, C*, and *Lisp) are nearly identical to standard serial programming languages. Each provides some method for defining parallel data structures: CM Fortran uses the Fortran 90 array features, while the other two languages add a new data type. Once the data sets (arrays, matrices, structures, etc.) are defined, a single sequence of instructions, as in serial code, causes operations to be performed concurrently either on the full data

sets or on selected sections thereof. 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.

The flow of control in a data parallel language is also nearly identical to that of its serial counterpart. Since this control flow, rather than processor speed, determines the order of execution, race conditions and deadlock cannot develop. The programmer does not have to add extra code to ensure synchronization within a program; the compilers and other system software maintain synchronization automatically. Moreover, the order of events, being essentially identical to that in a serial program, is always known to the programmer, which eases debugging and program analysis considerably.

1.2 Parallel Programming

Prior to the CM-5, the most successful implementation of the data parallel programming model was the so-called SIMD (Single Instruction, Multiple Data) architecture. As implemented on the Connection Machine model CM-2, the SIMD architecture has shown itself to be extremely efficient and powerful. Arrays that are hundreds or thousands of elements in size are laid out across hundreds or thousands of processors, one element per processor, in a format whose logical structure matches that of the data set itself and the operations to be performed on it. (See Figure 1.) When there are more array elements than processors, the processors subdivide themselves into "virtual processors" and give each element its own virtual processor. Instructions are then executed upon each element simultaneously. For example, given three 400 x 400 arrays, **A**, **B**, and **C**, the statement **C = A + B** is a single statement — and is executed as such — in data parallel programming.

But "data parallel" and "SIMD" are not necessarily synonymous terms. Consider, for example, finite difference codes. Boundary elements in these codes usually require special treatment, which means conditional branching. In data parallel languages, such branching is frequently coded along the lines of

```
where (boundary_elements)
    do_a
elsewhere
    do_b
end where
```

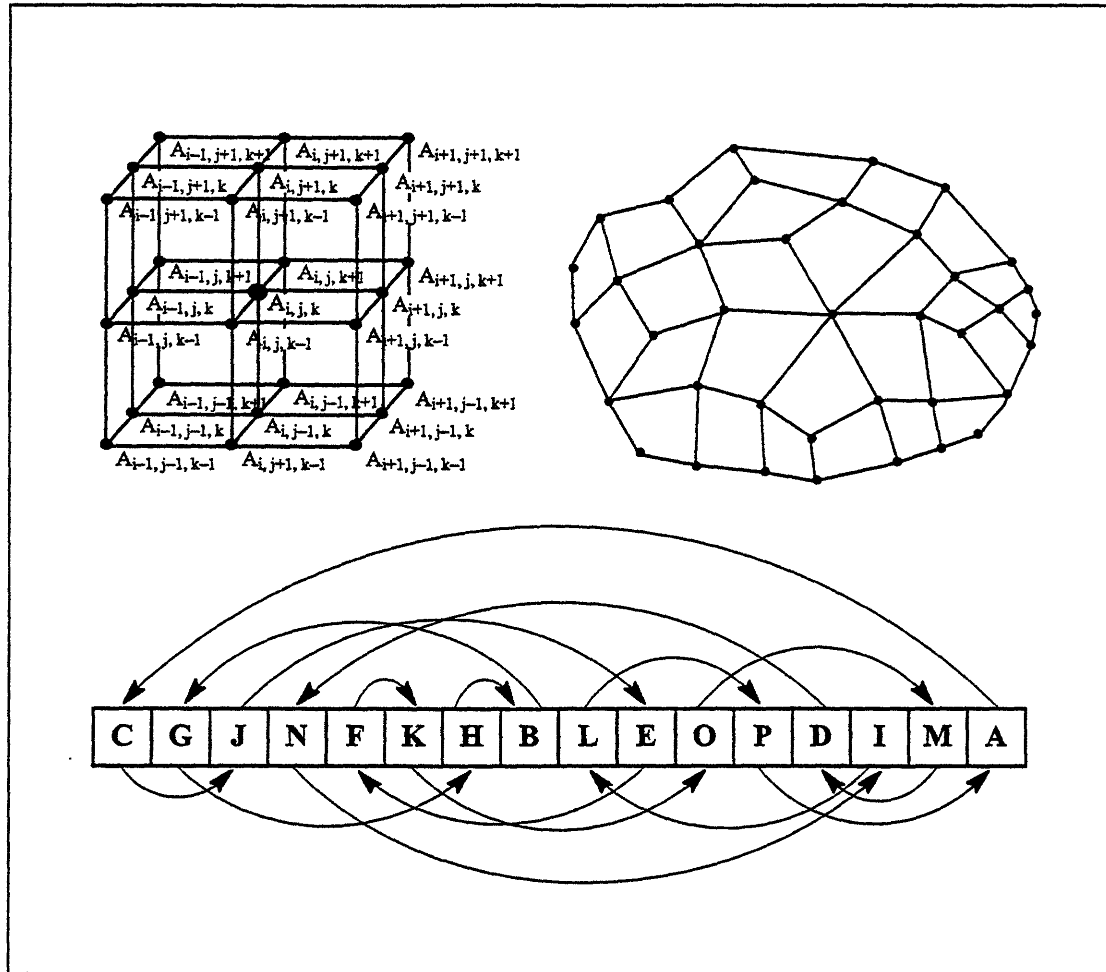


Figure 1. Examples of data sets.

Some problems involve data sets organized as multidimensional grids. The calculation for each data point relies on the values of neighboring data pieces. The pattern of interaction is both local and regular. Finite difference methods are typical of this category.

Other problems, exemplified by finite element methods, operate on data that is less rigidly structured. The calculation for each data point again relies on the values of nearby data points, but the pattern of interaction is irregular. In some cases the pattern of interaction may change over time, as dictated by the content of the data (for example, to make the mesh finer in regions of interest).

For tasks such as sorting, the manner in which data points interact depends greatly on the data values; the pattern of communication will be both nonlocal and irregular.

The communications networks of the CM-5 are designed to support both regular and irregular patterns of communication. Patterns that are predominantly local are rewarded with higher throughput.

A pure SIMD implementation of such code will execute the **where** branch for all boundary elements, and then execute the **elsewhere** branch for all interior elements. A MIMD (Multiple Instructions, Multiple Data) implementation will execute both branches simultaneously, with each processor making its own decision whether to fetch and execute instructions for the **where** branch or for the **elsewhere** branch for each element. When all processors have finished execution, the program will proceed to the next statement. Note that both implementations use the same code; both are undoubtedly data parallel programming. The only externally visible difference will be in performance; the second implementation, by using functional parallelism in support of data parallelism, can run faster than the first.

Note that the order of events in either case is identical to the order that would obtain for serial code. Even if the **where** branch takes several times as long as the **elsewhere** branch to execute, no processors will proceed to the subsequent statement until all have finish executing the **where** block. System software implements this control; the programmer does not have to worry about it. Only where events have no dependencies on each other, so that their order does not matter, will the order be unknown. (Figure 2 illustrates the combined independence and synchronization of program execution in this MIMD implementation of the data parallel programming model.)

Extensions to the Data Parallel Model

Although data parallel programming provides the biggest gains among known techniques of parallelism, it may sometimes be usefully extended by mixing in other parallel techniques. For example, some applications may perform best when divided into sections, each section making use of data parallel programming and all sections together acting as a pipeline. Thus, one process might gather data and do some preliminary selecting or compacting; it would then pass its results to a second process, which would do more intense computing on the smaller data set; and that process would then pass its results to a third process, which would perform some visualization or reporting function. On the CM-5, all three processes can run in parallel, either timesharing on a single partition or perhaps each having exclusive use of a separate partition. In the latter case, each process has its own physical computing resources; I/O for the first process and computation for the second occur simultaneously, with no impact on each other or on the third process.

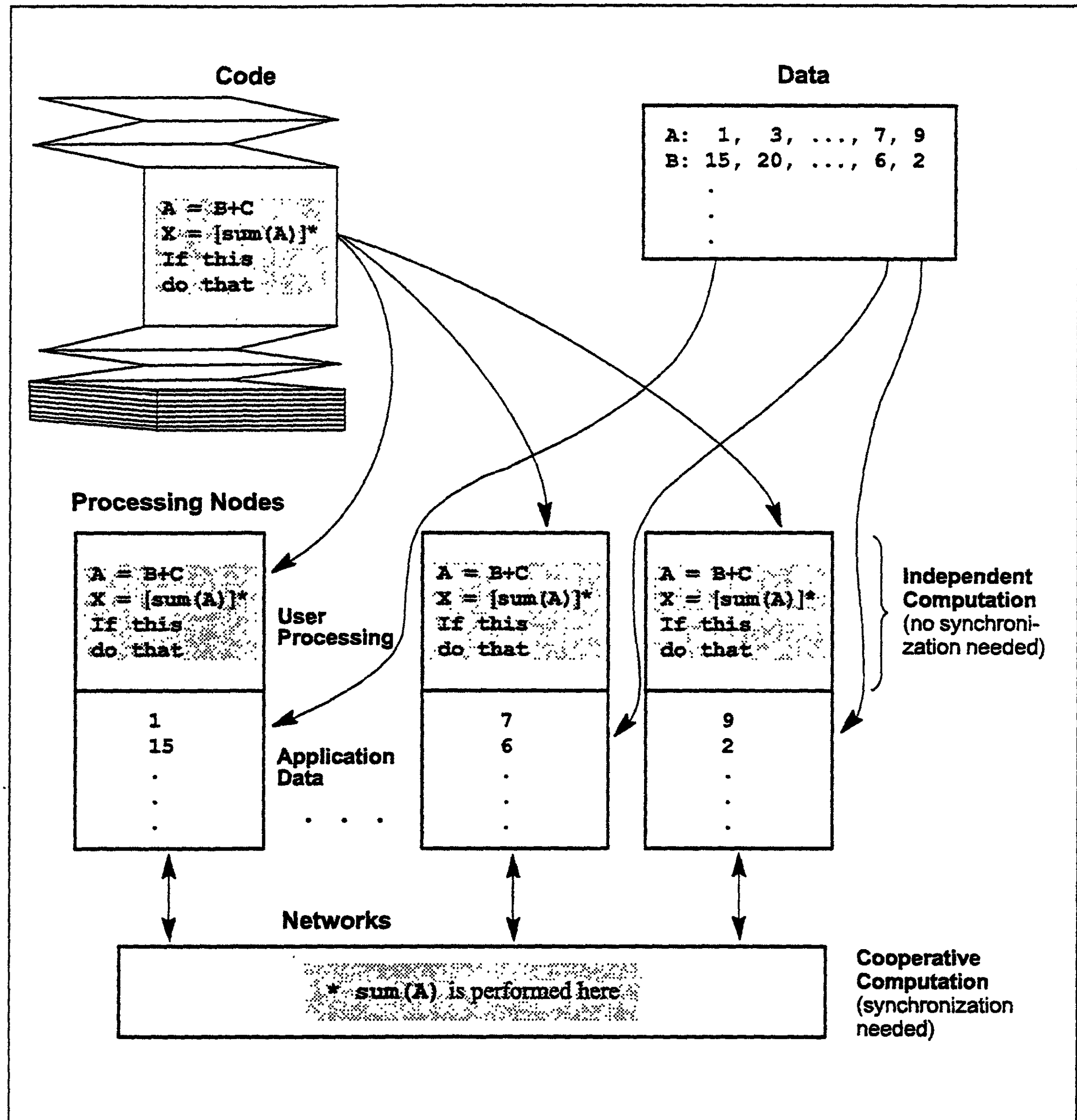


Figure 2. Code running on a CM-5.

A partition manager loads identical code onto every processing node in a partition. Data is distributed across the nodes: Given an array of *m* values and a partition of *n* nodes, each node handles *m*/*n* values.

Each node executes its program independently, branching according to its own data values. As long as computation remains local, no synchronization or communication is needed.

When data needs to be transferred among processors — for example, when processors must each contribute values to a global sum — the communications networks carry the data and enforce the necessary synchronization. (For global combining operations such as sum, the Control Network performs the reduction.)

The CM-5 thus extends the data parallel programming model developed for the CM-2 to incorporate an even broader and more widely useful mix of parallel techniques. Optimized for data parallelism, the CM-5 nonetheless supports other forms of parallelism that can either enhance data parallelism or allow the porting of programs from other architectures. This extended model, which we may call *coordinated parallelism*, represents the best that is known about parallel programming today.

1.3 Advantages of a Universal Architecture

In the past, programmers of supercomputers were forced to choose between MIMD machines, which were good at independent branching but bad at synchronization and communication, and SIMD machines, which were good at synchronization and communication but poor at branching. The CM-5 supports the full data parallel model by providing high performance for branching and synchronization alike — and, indeed, for all aspects of both SIMD-style and MIMD-style architectures.

This extended data parallel model allows new flexibility in writing programs specifically for use on the Connection Machine system. It also allows the use, on the CM-5, of programs and libraries written with other architectures in mind.

Figure 3 shows some of the ways in which applications originally written for other systems can migrate to the CM-5. (The illustration is based on the Fortran language, but the CM-5 supports C and Lisp as well.)

- Existing CM-2 Fortran programs can be moved directly onto the CM-5; recompiling is all that is needed.
- In some cases, partial recoding of CM-2 programs can bring better performance by taking advantage of new compiler features.
- Applications written using a message-passing programming model for distributed memory computers can run on the CM-5 by substituting calls to a CM-5 message-passing library for the original calls.
- With some additional recoding, message-passing programs can be tuned to take advantage of the superior hardware facilities for cooperative computation offered by the CM-5.

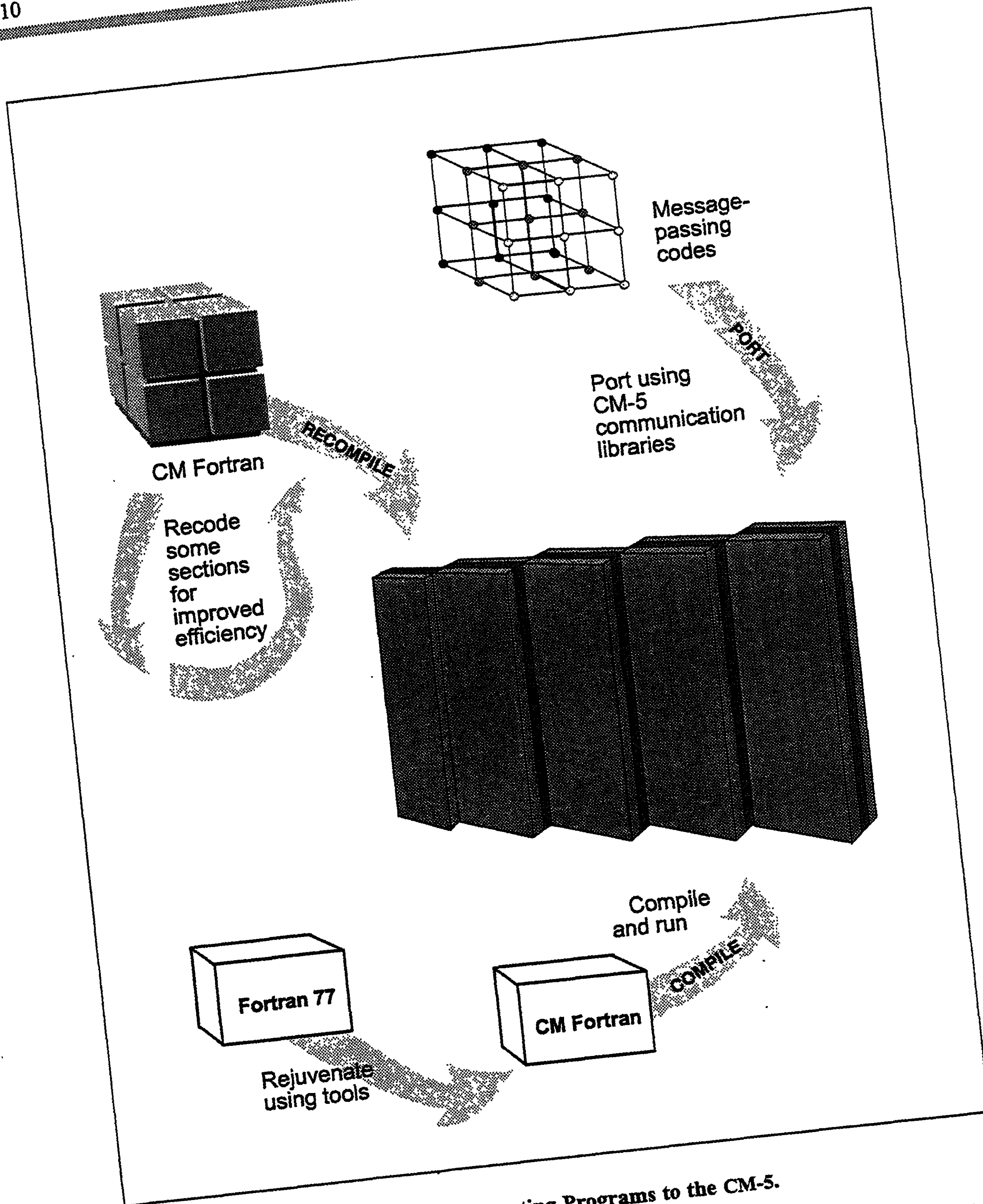


Figure 3. Transporting Programs to the CM-5.

CM Fortran programs written for the CM-2, Fortran 77 programs written for execution on serial computers, and message-passing programs designed to run on MIMD-only architectures are all easily ported to the universal architecture of the CM-5.

- Existing Fortran 77 codes can be migrated to Fortran 90, using commercially available tools supplied by third-party vendors, and then compiled by the CM Fortran compiler. This allows many widely used codes to function effectively on the CM-5.

1.4 Looking Ahead

The next two chapters explain further what coordinated parallelism on the CM-5 offers. Chapter 2 shows how the CM-5 hardware is optimized to support coordinated parallelism, while Chapter 3 provides further explanation of the features to be found in data parallel languages.

Chapter 2

The Basic Components of the CM-5

At its best, parallel processing brings many processors, working in close coordination, to bear on large quantities of data. An effective parallel-processing system must provide a large amount of memory to hold this data and must provide effective access to the data for hundreds or thousands of processors. The CM-5 system meets this goal. Moreover, it allows its memory and processor resources to be applied equally effectively to a single large problem or to job requests from dozens of simultaneous users.

Traditional computer architectures, such as the generic system diagrammed in Figure 4, link one or a few processors to a shared memory via a system bus. This worked well when processing speeds were slower and the number of processors was small. Nowadays it is much more cost-effective to use many processors than to try to make the processors faster. With many processors, a simple bus is a bottleneck, and the complex switches that can provide fast access to a shared memory for every memory reference are both expensive and complicated. Two more changes to the early model are therefore needed to balance communication speed with processing speed: memory must be distributed, rather than shared; and a high-bandwidth network, rather than a bus, must be used. Figure 5 diagrams this second architecture as it appears in the CM-5.

2.1 Processors

A CM-5 system may contain hundreds or thousands of parallel processing nodes. Each node has its own memory. Nodes can fetch from the same address in their respective memories to execute the same (SIMD-style) instruction, or from individually chosen addresses to execute independent (MIMD-style) instructions.

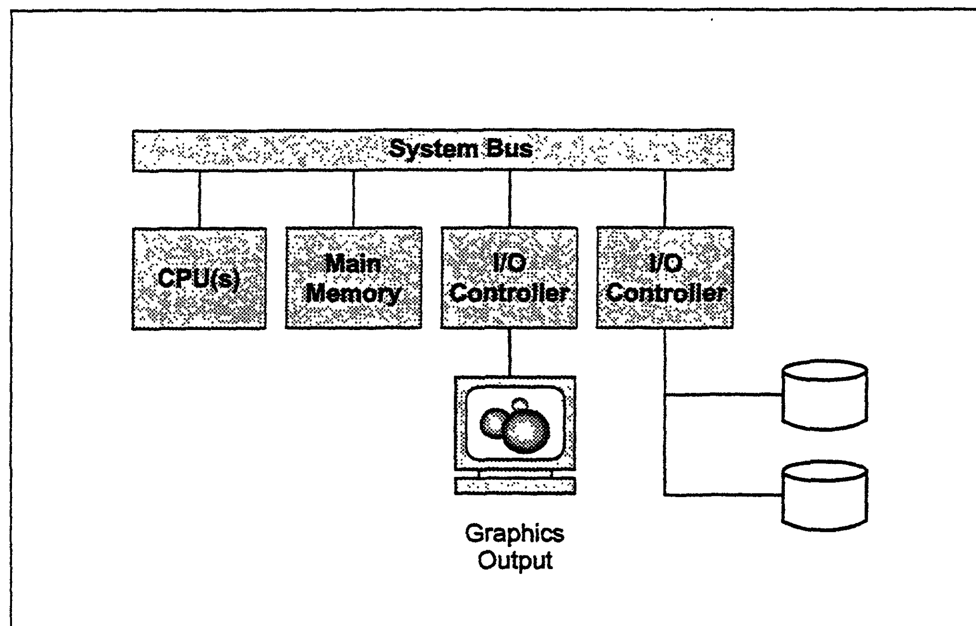


Figure 4. Organization of a traditional computer.

The processing nodes are supervised by a control processor, which runs an enhanced version of the UNIX operating system. Program loading begins on the control processor; it broadcasts blocks of instructions to the parallel processing nodes and then initiates execution. When all nodes are operating on a single control thread, the processing nodes are kept closely synchronized and blocks are broadcast as needed. (There is no need to store an entire copy of the program at each node). When the nodes take different branches, they fetch instructions independently and synchronize only as required by the algorithm under program control.

To maximize system usefulness, a system administrator may divide the parallel processing nodes into groups, known as *partitions*. There is a separate control processor, known as a *partition manager*, for each partition. Each user process executes on a single partition, but may exchange data with processes on other partitions. Since all partitions utilize UNIX timesharing and security features, each allows multiple users to access the partition while ensuring that no user's program interferes with another's.

Other control processors in the CM-5 system manage the system's I/O devices and interfaces. This organization allows a process on any partition to access any I/O device, and ensures that access to one device does not impede access to other

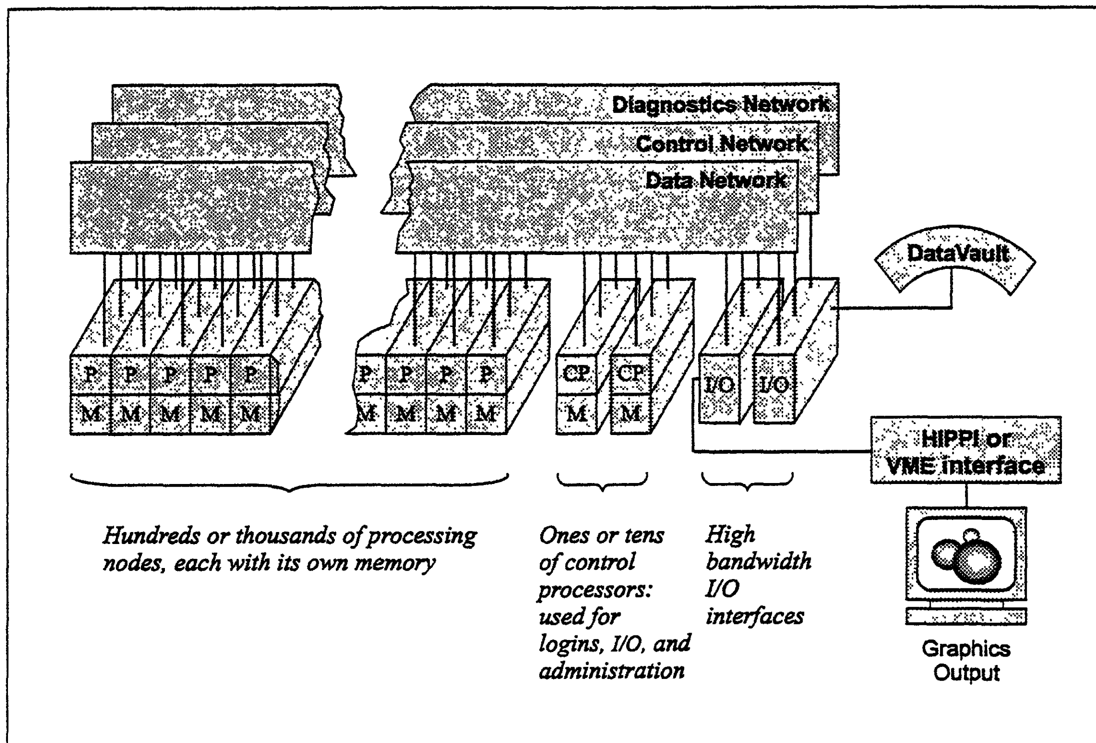


Figure 5. Organization of the Connection Machine system.

devices. (Figure 6 shows how this distributed control works with the CM-5's interprocessor communication networks to enhance system efficiency.)

2.2 Networks

Every control processor and parallel processing node in the CM-5 is connected to two scalable interprocessor communication networks, designed to give low latency combined with high bandwidth in any possible configuration a user may wish to apply to a problem. Any node may present information, tagged with its logical destination, for delivery via an optimal route. The network design provides low latency for transmissions to near neighboring addresses, while preserving a high, predictable bandwidth for more distant communications.

The two interprocessor communications networks are the Data Network and the Control Network. In general, the Control Network is used for operations that involve all the nodes at once, such as synchronization operations and broadcasting;

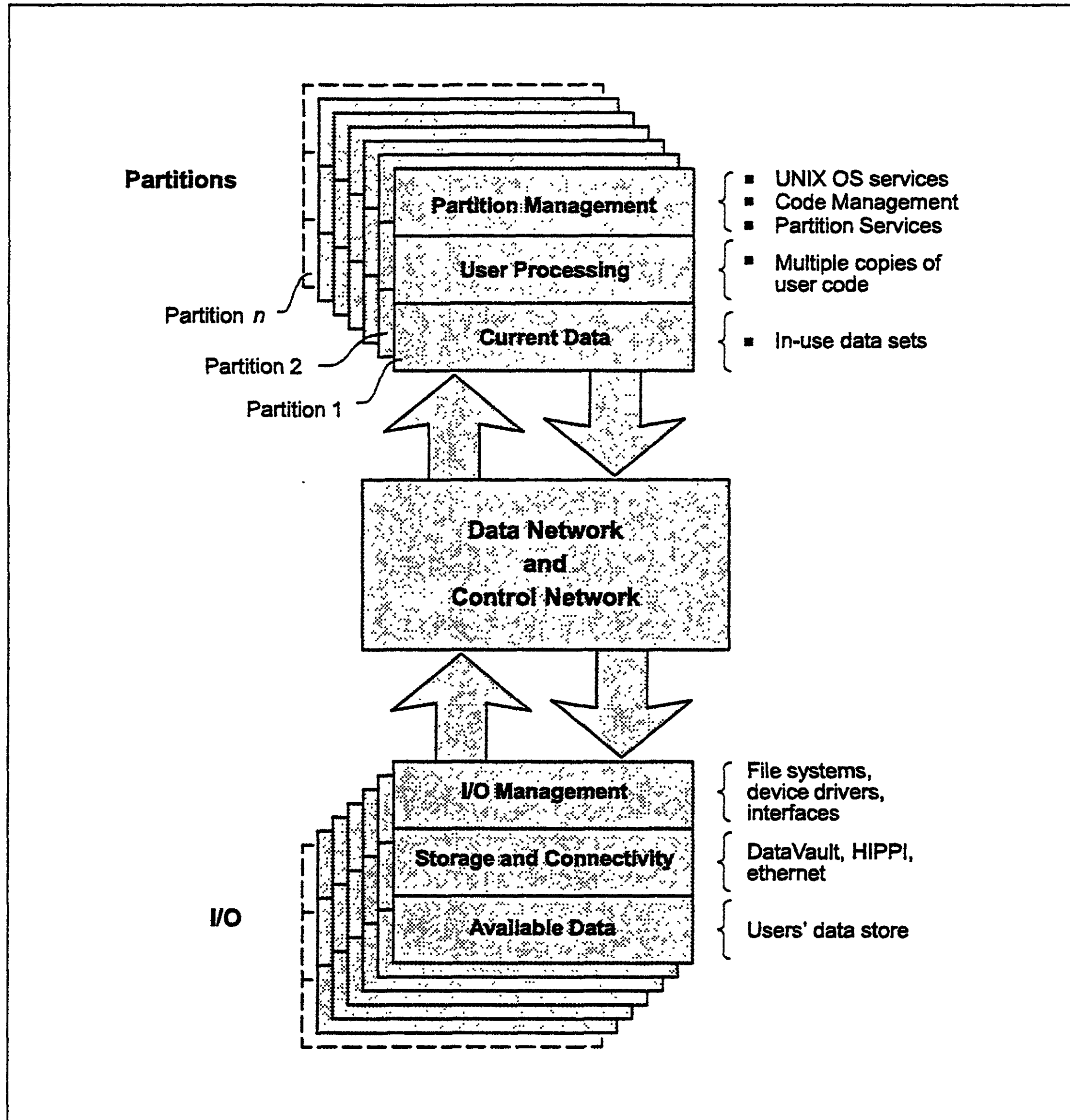


Figure 6. Distributed control on the CM-5.

Functionally, the CM-5 is divided into three major areas. The first contains some number of partitions, which manage and execute user applications; the second contains some number of I/O devices and interfaces; and the third contains the two interprocessor communication networks that connect all parts of the first two areas. (A fourth functional area, covering system management and diagnostics, is handled by a third interprocessor network and is not shown in this drawing.)

Because all areas of the system are connected by the Data Network and the Control Network, all can exchange information efficiently. The two networks provide high bandwidth transfer of messages of all sorts: downloading code from a control processor to its nodes, passing I/O requests and acknowledgments between control processors, and transferring data, either among nodes (whether in a single partition or in different partitions) or between nodes and I/O devices.

the Data Network is used for bulk data transfers where each item has a single source and destination.

A third network, the Diagnostics Network, is visible only to the system administrator; it keeps tabs on the physical well-being of the system.

External networks, such as Ethernet and FDDI, may also be connected to a CM-5 system via the control processors.

2.3 I/O

The CM-5 runs a UNIX-based operating system; it provides its own high-speed parallel file system, and also allows full access to ordinary NFS file systems. It supports both HIPPI (high-performance parallel interface) and VME interfaces, thus allowing connections to a wide range of computers and I/O devices, while using standard UNIX commands and programming techniques throughout. A CMIO interface supports mass storage devices such as the DataVault and enables sharing of data with CM-2 systems.

I/O capacity may be scaled independently of the number of computational processors. A CM-5 system of any size can have the I/O capacity it needs, whether that be measured in local storage, in bandwidth, or in access to a variety of remote data sources. Communications capacity scales both with processors and with I/O. Customers may choose both the processing power and the I/O capabilities that meet their needs, and the CM's communications capacity is automatically scaled to match.

Just as every partition is managed by a control processor, every I/O device is managed by an input/output control processor (IOCP), which provides the software that supports the file system, device driver, and communications protocols. Like partitions, I/O devices and interfaces use the Data Network and the Control Network to communicate with processes running in other parts of the machine. If greater bandwidth is desired, files can be spread across multiple I/O devices: a striped set of eight DataVaults, for example, can provide eight times the I/O bandwidth of a single DataVault.

The same hardware and software mechanisms that transfer data between a partition and an I/O device can also transfer data from one partition to another (through a named UNIX pipe) or from one I/O device to another.

2.4 A Universal Architecture

The architecture of the CM-5 is optimized for data parallel processing of large, complex problems. The Data Network and Control Network support fully general patterns of point-to-point and multi-way communication, yet reward patterns that exhibit good locality (such as nearest-neighbor communications) with reduced latency and increased throughput. Specific hardware and software support improve the speed of many common special cases. Chapter 3 outlines the nature of this support, which is discussed in even greater detail in later chapters.

Two more key facts should be noted about the CM-5 architecture. First, it depends on no specific types of processors. As new technological advances arrive, they can be moved with ease into the architecture. Second, it builds a seamlessly integrated system from a small number of basic types of modules. This creates a system that is thoroughly scalable and allows for great flexibility in configuration.

Chapter 3

Data Parallel Programming

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 database, 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 continually and repeatedly 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. 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 gathering data, one regards it as global. When computing on the gathered data, one thinks of it as local data, and of the computations themselves as being carried out in multiple local memories.

3.1 Data Sets and Distributed Memory

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.

In fact, the basic unit of data in a Connection Machine is the array, or some other form of parallel variable. Arrays are spread across the distributed memory of the Connection Machine system so that each element is in the memory of a separate processor. If the number of elements in the array matches the number of physical processors, then each local memory receives one element. If the number of elements in the array exceeds the number of physical processors, then several elements are placed in the memory of each processor. The elements remain distinct. Each is considered to have its own "virtual processor" and is handled accordingly.

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 the expression B + C requires an integer or floating-point add instruction. In the same way, a compiler for a data parallel language determines whether B + C requires a single addition operation or thousands.

Array Layout

A user program runs within a partition of a CM-5. Defined by the administrator, a partition may represent part or all of the CM-5 system. In order to allow a program compiled with a CM compiler to run on a partition of any size, the precise mapping of data elements to processors occurs at run time; the run-time system lays out the array for best efficiency. Compiler directives in each language allow programmers to request that the mapping be optimized for particular purposes.

Local Computations

Unless the programmer has specified otherwise, arrays of equal size and shape will have identical layouts. Thus, identical elements of each such array will share the memory of a particular processor. When a computational statement such as

`C = A + B` is executed, each processor locates and stores the needed data in its own memory; no interprocessor data movement is required, and the operation proceeds very quickly.

3.2 Interconnected Data Structures

The inherent structure of most data sets links each data element to some, but not all, other elements. Often the linkages are to neighboring elements, in which case the structure is said to be localized.

A matrix, for example, is generally thought of as having row and column structure. Elements that share one subscript 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 brought together for computation. If a data structure is converted from the spatial domain to the frequency domain, then a butterfly pattern may be required 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 may 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.

Establishing Linkages among Data Elements

Data parallel languages use pointers or array subscripts to establish connections between processors and hence between their data elements. If the required patterns are regular and local, such as processors sharing data with their nearest neighbors, then each processor can easily calculate the address of its neighbors as needed. For irregular arrays, an array of pointers, itself a parallel data structure, establishes an arbitrary pattern of intercommunication.

3.3 Interprocessor Communications

There are four important categories of interprocessor communications:

- replication
- reduction
- permutation
- parallel prefix

Each of these four types of data transfer can be applied to regular or irregular data sets: to vectors, matrices, multidimensional arrays, variable-length vectors, linked lists, and completely irregular patterns. All these combinations are supported by data parallel software within the CM-5. In addition, the most common or otherwise important cases are supported directly by special hardware built into the Control Network. In all cases, the CM-5's high performance is a result of having all the processors act cooperatively to achieve the needed data transfers.

Replication

Replication consists of taking some data values and making a larger number of data values by copying them. (See Figure 7.) A single value, for example, may be *broadcast* to all processors for use in a computation. A vector may be copied into each column of a matrix, or into each row. (The general case of making many copies of an array to fill a higher-dimensional array is called *spreading*.) A less regular pattern is the division of a collection into arbitrary subsets of varying size, and one may wish to broadcast a different value within each subset. If the subsets are ordered and not interleaved, one may regard them as a collection of vectors of various sizes; this common case can be implemented more efficiently than the general case.

Most data parallel programming languages support broadcasting implicitly; if `A` and `B` are arrays and `X` is a scalar quantity, the statement `A = B + X` implicitly broadcasts `X` to all processors so that the value of `X` can be added to every element of `B`. The general case of replication is typically supported through parallel array indexing, that is, indexing the same array with many index values. If some of the index values are the same, then the same array element will be copied to many places. Intrinsic functions (such as `SPREAD` in Fortran) cover important special cases.

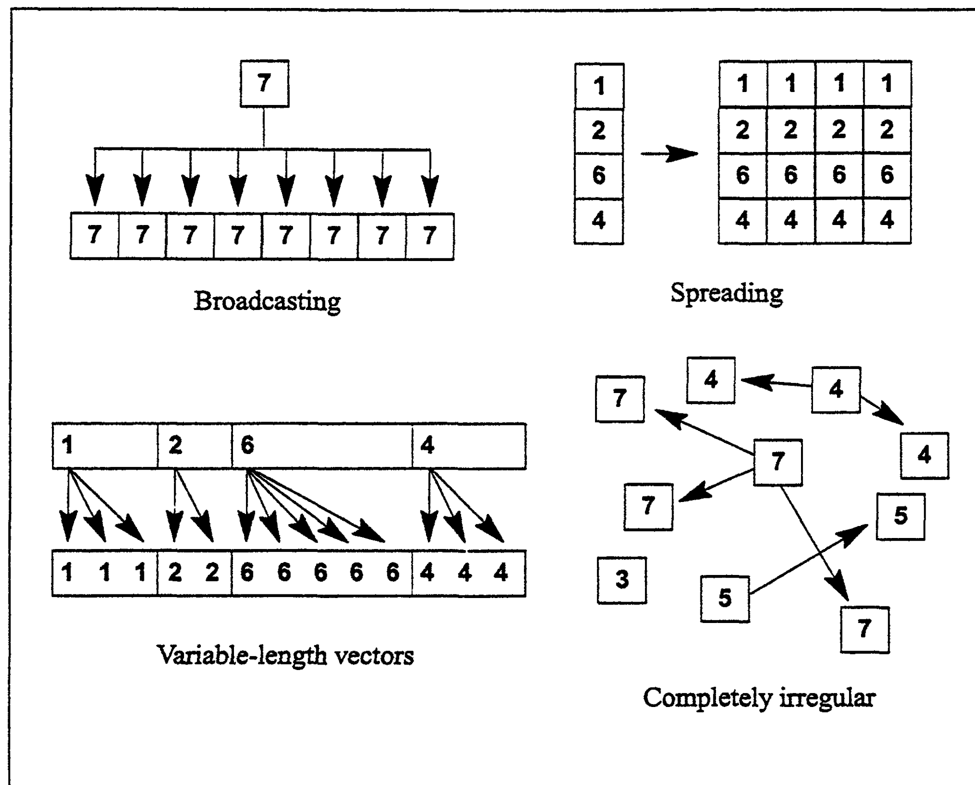


Figure 7. Replication.

Reduction

Reduction is the opposite of replication: Reduction consists of taking some data values and making a smaller number of data values by combining them. (See Figure 8. Note that it is similar to Figure 7 except that the arrows all point the other way.) A single value, for example, may be produced by computing the *sum* of a set of values; here the combining operation is addition. Other important reduction operations include taking largest or smallest value (maximum or minimum), logical AND (are all results true?), and logical OR (is any result true?). All these start with a large collection of values and reduce them to a single result.

More complex patterns of reduction mirror related patterns of replication. The rows of a matrix may be summed to produce the elements of a column-vector result; this is the opposite of a SPREAD operation. A collection of variable-length vectors may be reduced, producing a separate sum for each vector. Completely general patterns may be specified by index values or pointers.

Most data parallel languages provide a collection of operators or intrinsic functions for expressing various patterns of reduction. For example, the Fortran statement `X = SUM(A)` sums all the elements of the array `A` and places the scalar result in `X`. The same computation can be expressed in C* as `x = (+= a);` and if the old value of `x` is to be included in the sum one may simply write `x += a;` (which says that every element of `a` is to be added into `x`).

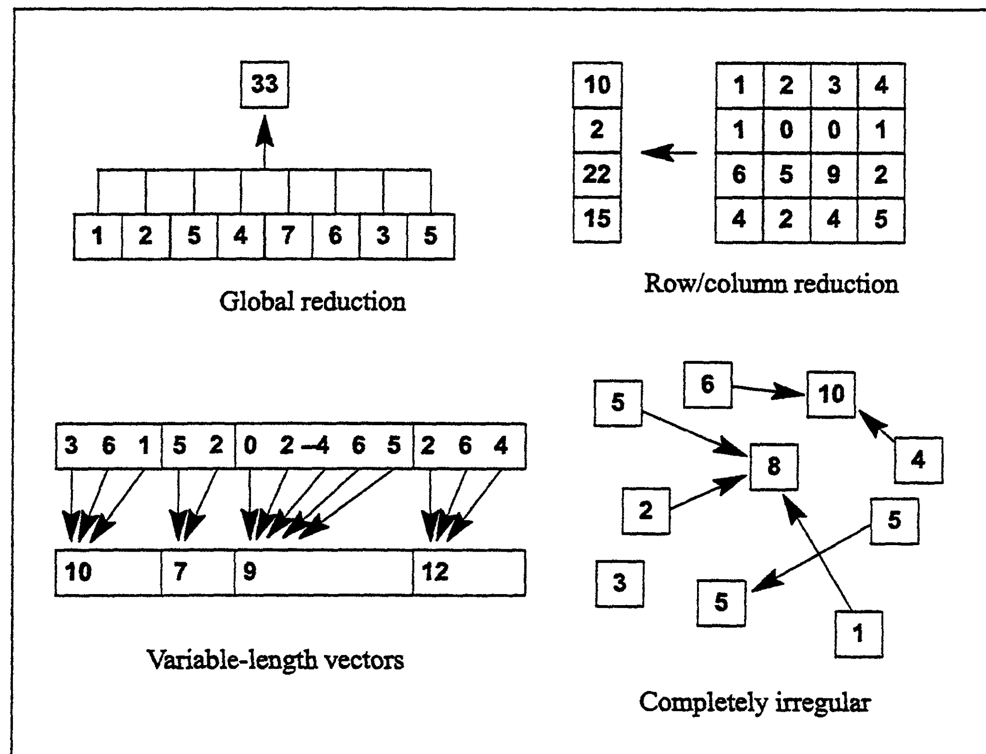


Figure 8. Reduction.

Permutation

Permutation rearranges its inputs to produce the same number of results; every data value comes from one place and goes to one place. (See Figure 9.) Transposing a matrix, reversing a vector, shifting a multidimensional grid, and FFT butterfly patterns are all examples of permutation.

Data parallel languages usually express permutation through parallel array indexing and special-purpose intrinsic functions. A typical example of use might

be a finite-difference grid used in the discretization of Laplace's Equation, in which the average of four nearest neighbors is iteratively computed:

```
    C = 0.25 * ( CSHIFT(A,1,+1)
   &           + CSHIFT(A,1,-1)
   &           + CSHIFT(A,2,+1)
   &           + CSHIFT(A,2,-1) )
```

Here **CSHIFT** is a Fortran intrinsic that shifts (or rotates) an array with periodic boundary conditions. Elements shifted off one edge are circularly shifted into the opposite edge; thus no elements are lost in this operation. In constrast, **EOSHIFT** performs an end-off shift that discards shifted-out elements and introduces a pad value, usually zero, into vacated positions; this operation is thus technically a hybrid of permutation (of array elements) and replication (of the pad value).

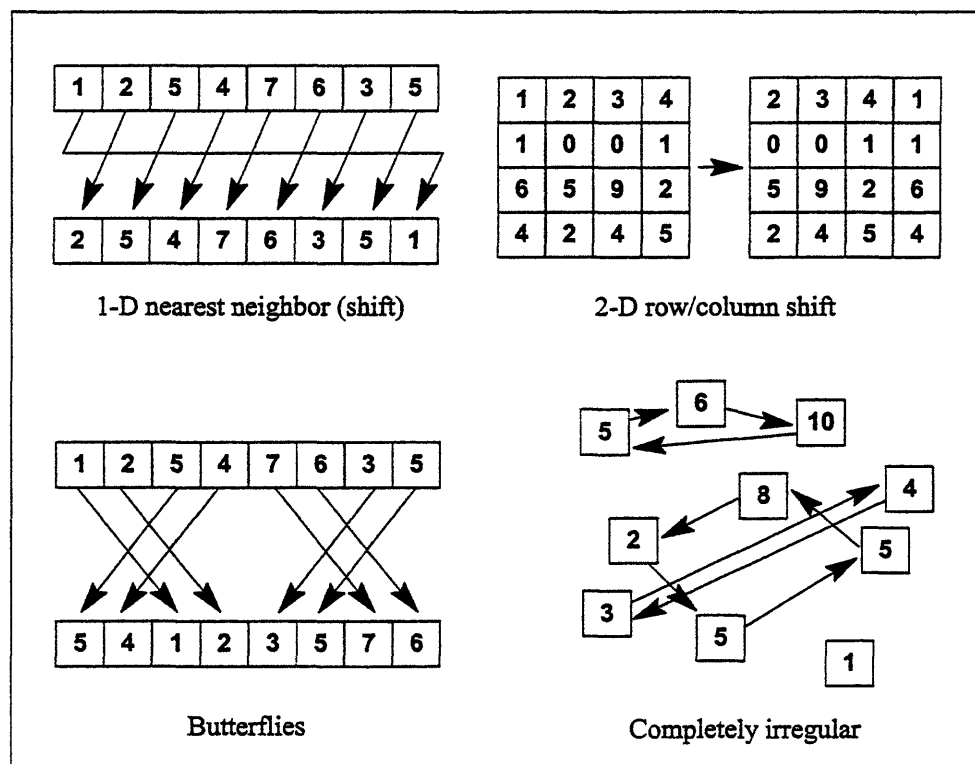


Figure 9. Permutation.

Parallel Prefix

A parallel prefix operation is a very specific compound operation; it produces as many results as inputs, but each result may be a reduction of many inputs, and each input may contribute to many results. There happens to be a rapid and efficient parallel method for performing this complex compound operation; the CM-5 supports it with a combination of hardware and library software. It is of particular use in parallel computations because it permits rapid parallel execution of operations that at first glance appear to be inherently sequential.

The simplest example of a parallel prefix operation is computing the running totals of a list of numbers. The *k*th result is the sum of the first *k* inputs. (See Figure 10.) There is a simple sequential implementation of such a computation:

```
RUNNING_TOTAL = 0.0
DO J = 1,1000
  RUNNING_TOTAL = RUNNING_TOTAL + B(J)
  A(J) = RUNNING_TOTAL
END DO
```

This would appear to be an inherently sequential process, scanning the array B from one end to the other, but by bringing many processors to bear in parallel, one can perform this computation in 10 steps instead of 1000 steps (10 is approximately the base-2 logarithm of 1000).

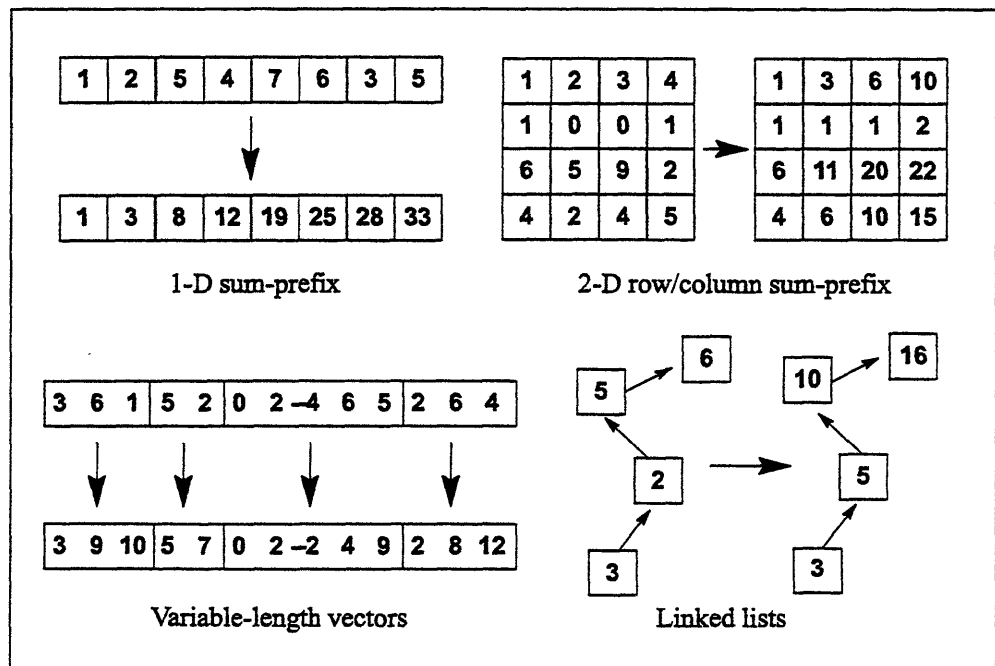


Figure 10. Parallel prefix.

3.4 Conditionals

Conditional operations are an essential part of data parallel programming, as of serial programming. Some of the control constructs (`IF`, `CASE`) are identical; others (`WHERE`, `FORALL`) are specific to parallel usage.

Data parallel programs implement conditionals by limiting the impact of operations to a certain 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. The specified operation is then performed only on elements for which the condition is true, while either an alternate operation, or no operation, is performed on the other elements. As in serial programs, conditionals may be nested in very general ways.

3.5 In Summary

The data parallel model of computation makes it easy to program massively parallel computers. The model is also suitable for use on sequential computers, including vector processors, and on shared-memory parallel computers. High-level data parallel languages support the data parallel style. The CM-5 architecture is specifically designed for efficient execution of data parallel programs on large data sets.

Data parallel programming provides a practical framework for organizing inter-processor communication. An analogy may be drawn with the way "structured programming" has provided a practical framework for organizing control flow in sequential programs. Each model begins with primitive computations and uses a fixed set of standard combining forms to impose structure on the program.

Structured programming begins with simple assignment statements and observes that most patterns of control flow can be expressed in terms of sequencing (`BEGIN-END`), conditional branching (`IF-THEN-ELSE`), and looping (`WHILE-DO`). If these structures are conventionally used wherever appropriate, then use of a low-level construct such as a `GOTO` is a strong indication, and a useful one, that something unusual is going on; maintenance programmers should pay special attention, and language designers should ask whether the situation represents a class of problems that could be addressed more generally. Conventional syntax has evolved for certain frequently used compound patterns, such as `CASE` statements and `DO` loops.

Similarly, data parallel programming begins with local computations and observes that most patterns of interprocessor communication can be expressed in terms of replication, reduction, permutation, and parallel prefix. If these structures are conventionally used wherever appropriate, then use of a low-level construct such as explicit message-passing is a strong indication, and a useful one, that something unusual is going on; maintenance programmers should pay special attention, and language designers should ask whether the situation represents a class of problems that could be addressed more generally. Conventional syntax has evolved for certain frequently used compound patterns, such as shifting of regular grids, sorting, and fast transforms such as FFT.

As Figure 11 suggests, the data parallel model simplifies the programmer's job by providing for parallel programs the conventional structure and discipline that structured programming provides for sequential programs. Indeed, the data parallel model is the only programming methodology yet put forward that provides a coherent global organization for structuring programs that operate on thousands of processors.

	Structured Programming	**Data Parallel Programming**
Primitive Computations	assignment statements	local computations
Basic Patterns	BEGIN . . . END IF . . . THEN . . . ELSE . . . WHILE . . . DO . . .	replication reduction permutation parallel prefix
Common Compound Patterns	DO loops CASE statements REPEAT . . . UNTIL . . .	regular grids, stencils fetch-with-add sorting, fast transforms
Low-Level Mechanisms	GOTO	message passing

Figure 11. Structuring programs.

3.6 More Information To Come

This introduction barely begins to present the features and capabilities of the CM-5. The remainder of this book presents them in somewhat more detail (although still at a summary level). Part II discusses the software that supports application programming on the CM-5. Part III discusses the various aspects of the system's architecture.

For information beyond this, you can turn to technical reports on Connection Machine programming, and to the CM-2 and CM-5 documentation sets. Especially recommended for new users are the manuals *Getting Started in C** and *Getting Started in CM Fortran.*

Part II
CM-5 Software

Chapter 4

Connection Machine Software

The Connection Machine system provides a well-designed, thoroughly integrated software environment to facilitate applications programming. The environment seamlessly blends industry standards with data parallel enhancements to provide both high performance and ease of use.

4.1 Base System Software

The use of industry standards begins with the UNIX operating system and its network file system (NFS). Full X11 support provides windowing capability; the NQS batch system allows submission of batch jobs locally or across a network. Networking support includes Ethernet and FDDI for local area networking, and VME, HIPPI, and UltraNet for high-performance networking.

Ease of use, meanwhile, is enhanced by Prism, the windowed, integrated development environment for program editing, debugging, and performance analysis. Another CM enhancement, a parallel, high-performance file system, provides excellent I/O performance and allows use of extremely large files.

4.2 Languages and Libraries

For programming, users choose among the popular languages C, Fortran, and Lisp. The CM offers data parallel versions of each language, extending the languages' own constructs in intuitive ways to support the data parallel model.

In addition, specialized libraries offer support for graphics, communications, and mathematical and scientific programming. All are available from the high-level languages; low-level programming is not required to achieve high performance on the Connection Machine supercomputer.

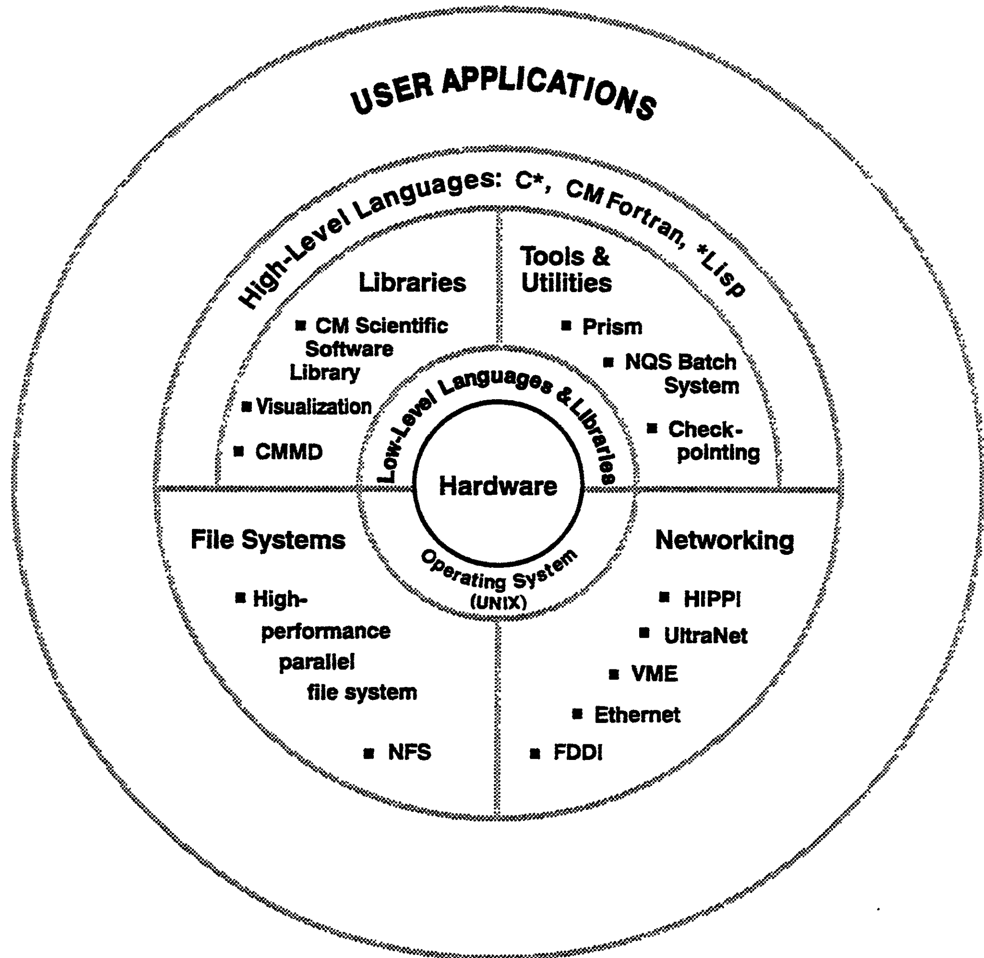


Figure 12. Layered software of the Connection Machine system.

4.3 CM Software Summarized

Figure 12 summarizes the layered software of the Connection Machine system. This software is discussed in the following chapters:

Operating system, file systems, I/O programming .... Chapter 5

Prism (the development environment) .... Chapter 6

NQS batch system, checkpointing,
the execution environment .... Chapter 7

CM Fortran programming language .... Chapter 8

C* programming language .... Chapter 9

*Lisp programming language .... Chapter 10

CM Scientific Software Library
(linear algebra, Fast Fourier Transforms,
random number generation, histograms) .... Chapter 11

Visualization .... Chapter 12

CMMD (message-passing communications library) ...... Chapter 13

Chapter 5

The Operating System: CMOST

The CM-5 operating system, CMOST, is an enhanced version of the UNIX operating system. The enhancements optimize computation, communication, and I/O performance within the CM-5 system itself, while the adherence to UNIX standards allows the CM-5 to interact efficiently with other computers in a heterogeneous, networked environment.

Because the CMOST operating system is built upon standard UNIX, it can provide all the services that any standard network server provides:

- timesharing and batch processing
- standard UNIX protection, security, and user interfaces
- support for all standard UNIX-based communications protocols
- exchange of data with other systems in an open, seamless fashion
- the ability to access files on other systems via NFS protocols and to supply data to other systems by acting as an NFS server
- the Network Queuing System (NQS) and other standard network-oriented programs
- for scalar programs, binary compatibility with SunOS

Enhancements provide higher-performance services and expanded functionality for users within the CM-5 system:

- high-speed file access
- fast parallel interprocessor communications capabilities
- other parallel operations for optimal utilization of CM-5 hardware

- central administration and resource management for all CM-5 computational and I/O facilities
- support for extended models of data parallel programming, such as data parallel pipes
- support for other parallel programming models
- checkpointing

5.1 CMOST and the CM-5 Architecture

The computational nodes on a CM-5 are grouped into partitions. A partition can be as small as 32 processors, or as large as the entire machine. The partitioning is flexible and is controlled by the system administrator, who can create and alter partitions as needed to meet site requirements. Each partition operates independently under the control of a control processor acting as a partition manager (PM). Users log into (or `rsh` onto) the PM and, once logged in, have full access to the PM itself, to all the computational nodes it controls, and—through the operating system—to all the I/O resources, partitions, and network connections of the CM-5 system. Figure 13 shows a user's-eye view of the CM-5.

Each partition manager runs a full version of the CMOST operating system. The PM makes all operating system resource allocation decisions and all swapping decisions for its partition, as well as most system calls for process execution, memory management, and I/O.

Each processing node runs an operating system microkernel, which supports the mechanisms required to implement the policy decisions made in the partition manager. All operating system code operates in supervisor mode, allowing it to access any network address and memory address in the machine.

When a user process begins running, its partition manager downloads code to the processing nodes and broadcasts identical memory maps to each node. The nodes then execute the provided code, each acting on its own data and executing computations and branches accordingly.

All nodes in a partition operate on the same process at the same time. Interprocessor communication between nodes within an application is handled entirely by user code, without any operating system overhead. For external communications the user process calls on the operating system, which requests and supervises the

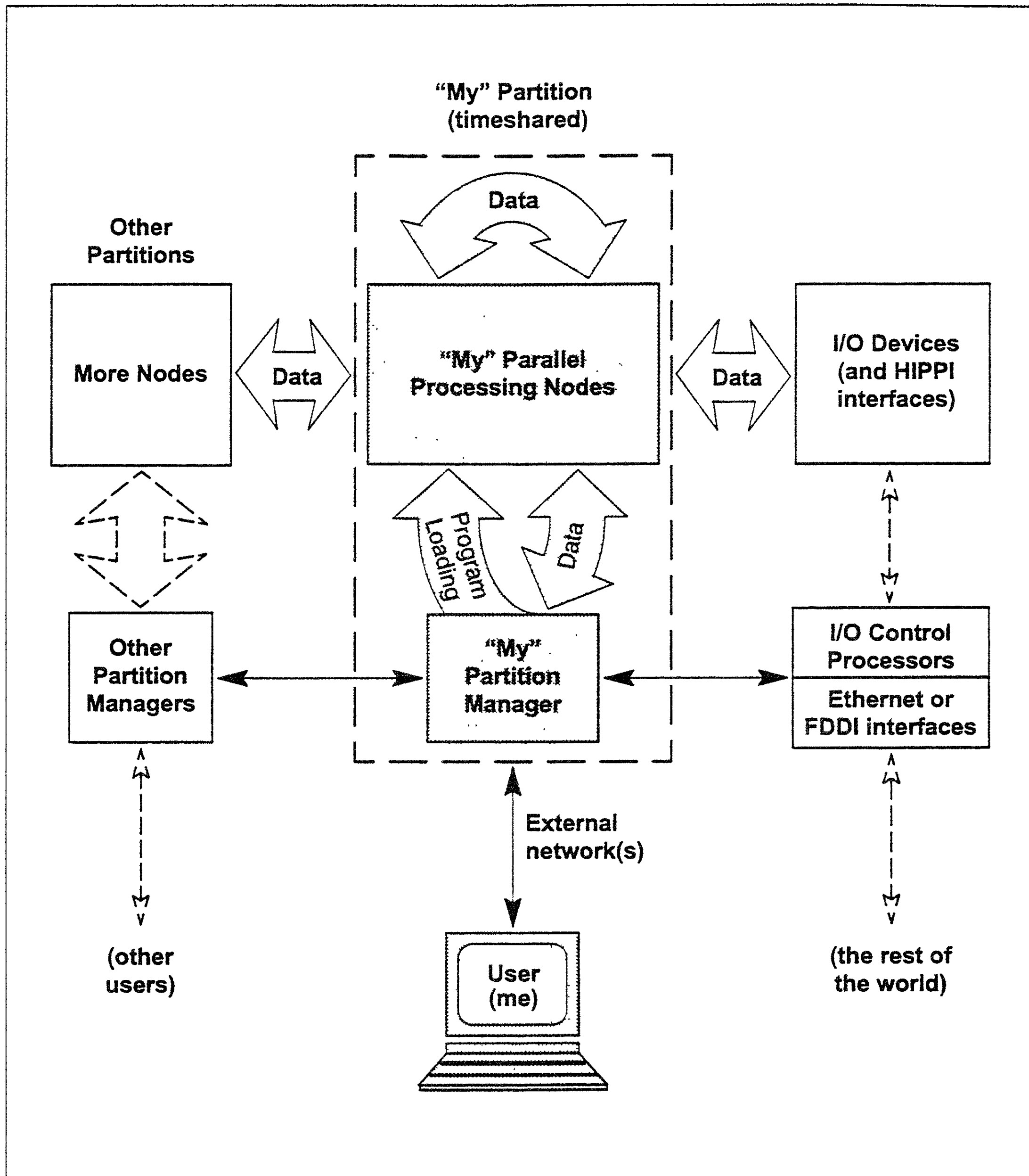


Figure 13. A user's view of a CM-5.

Users access a CM-5 system by running `rlogin` or `rsh` commands on a specific partition manager. A user program begins execution on the PM, downloads code to the nodes, then runs on nodes and PM both, passing data as needed among processors.

If a program needs to exchange data with an I/O device or with another process, the PM arranges the transfer, via system calls to other control processors. Data then flows directly between the nodes and the I/O device, nodes of another partition, or external network interface, thus ensuring that a parallel process gets the full benefit of the CM-5 Data Network bandwidth.

transfer on behalf of the user process. Data may be transferred between two processes running timeshared in the same partition or between two processes running concurrently in different partitions.

Interprocess communication is based on parallel extensions to UNIX sockets and pipes and is managed by the operating system. I/O transfers are handled in the same manner as transfers between partitions.

5.2 CMOST and the Users

Users typically access the CM-5 through an external network, either in batch mode, via the NQS `qsub` command, or interactively, via `rlogin` or `rsh` commands.

Each PM and IOCP within the CM-5 is a separate host on the network. Users can log in to any PM or IOCP for which they have appropriate privileges. Once logged in, a user has access to the full resources controlled by that control processor and to both local and networked file systems; the user can then run processes that use a control processor alone or a full partition of PM plus processing nodes. Since the set of control processors (PMs and IOCPs) within a CM-5 form a loosely coupled network of UNIX computers, a user with appropriate privileges can also run programs on any processor within the CM-5 using the normal UNIX networking commands.

The Program Development Environment

The program development environment available to Connection Machine users offers the full capabilities of UNIX and the X Window System. In addition, it offers enhancements specific to CM parallel programming: parallel languages, specialized libraries, and tools for parallel debugging and performance analysis. Prism, the CM-5's integrated programming environment, facilitates programmers' use of the machine (see Chapter 6).

The Program Execution Environment

The program execution environment on the CM-5 supports both interactive, time-shared program execution and batch execution using the NQS batch system.

Several facilities, such as automatic checkpointing and Prism, the CM programming environment, aid program development and robustness during execution. (See Chapter 7.)

5.3 CMOST and the Administrator

CMOST provides the administrator with tools for efficient and flexible resource management. It allows the administrator to partition the CM-5 for spacesharing among users, to set up the NQS batch system and the accounting system, and to monitor system usage, error logging, and power and environmental concerns. In addition, it provides all the standard UNIX capabilities, such as setting process priorities for use with process scheduling, setting disk quotas to control disk space usage, backing up and restoring user data, and setting up user permissions.

CM-5 administration is centralized at a system console, using commands that are modeled on SunOS 4.1 commands. The commands execute through a set of daemon processes that run (depending on their tasks) on the system console processor, the diagnostic console processor, or the partition managers.

5.4 I/O and File Systems

I/O programming on the CM-5 uses standard UNIX mechanisms, including sockets, pipes, character devices, block devices, and serial files. All I/O operations are modeled as reads and writes to files, regardless of the type of device used for storage.

CMOST extends the UNIX I/O environment to support parallel reads and writes and to support very large files, including files above the size supported in most current UNIX implementations. The virtual file system interface supports device-independent file behavior and supports many different file system types, including the standard UNIX file system, the Network File System (NFS), and the CM's own high-performance file system. Operating over the CM-5 Data Net-

external networks, NFS supports distributed file system management, allowing external devices to access CM files and CM-5 processes to access external files.

The CM-5 arranges communications to allow maximum simultaneous performance of computation and I/O. Transfers from one partition do not affect the performance of other partitions. Simultaneous transfers from several partitions see minimal interactions unless they require access to the same I/O device. Direct I/O-to-I/O transfers allow direct movement of data between a remote machine and a CM-5 I/O device, or between primary and secondary I/O devices on a CM-5, without affecting activities in partitions.

The CM-5 File System

The CM-5's high-performance file system manages the CM's high-speed disk storage, or DataVaults.

Within CM-5 files, data is stored in canonical (serial UNIX) ordering, thus allowing its use by both serial and parallel systems and processes. When a serial process does I/O, data remains in canonical order throughout; for parallel I/O, data moves between the canonical order and the ordering required by the computational nodes.

This reordering serves two important purposes. First, it allows a program to run on partitions of any size without affecting its I/O: a file written by a process running on a partition of one size may be read with equal ease by a process running on a partition of a different size. Second, it allows the same file to be read by parallel or serial processes. A serial process may read a file written by a parallel process, and vice versa.

For further information on the CM-5 file system and I/O, see Chapter 19.

Network Communications

Data can travel through sockets directly between CM-5 processes and other machines on the network. A user process can create a socket, send parallel data to it, and have that data received as a serial stream by a serial or vector computer. The same socket can carry serial data from control processors; as with file I/O, network communication uses standard protocols and data ordering for transmission, and uses parallel ordering only within the parallel computational nodes.

The User's View

From the user's point of view, data from any file system, on any device, appears the same and is handled in the same manner. A CM-5 control processor, accessing data over the data network, sees no difference between data stored on any CM-5 I/O device and data stored on any other UNIX file system.

Similarly, user processes are not concerned with the storage media on the CM. Whether data is stored on a single device or striped across multiple devices, the process accesses it as a single file. The only user-visible difference is in performance.

Chapter 6

The Programming Environment: Prism

The Prism programming environment is an integrated Motif-based graphical environment within which users can develop, execute, debug, and analyze the performance of programs written for the Connection Machine system. It provides an easy-to-use, flexible, and comprehensive set of tools for performing all aspects of Connection Machine programming.

Users can either load an executable program into Prism, or start from scratch by calling up an editor and a UNIX shell within Prism and using them to write and compile the program.

Once an executable program is loaded into Prism, users can (among other things):

- **Execute the program.** Users can simply start the program running or single-step through it. Execution can be interrupted at any time.
- **Debug the program.** Users can perform standard **dbx**-like debugging operations such as setting breakpoints and traces, printing the value of a variable or expression, and displaying and moving through the call stack.
- **Analyze the program's performance.** Data on execution time, broken down by procedures or by lines of source code, may be displayed as histograms. See Section 6.2.
- **Visualize data.** The values of interactively specified variables or expressions may be displayed in a variety of textual and graphical formats. See Section 6.3.

Prism operates on terminals or workstations running the X Window System.

6.1 Using Prism

Figure 14 shows the main window of Prism, with a program loaded. It is within this window that users debug and analyze their programs. Users can operate with a mouse, use keyboard equivalents of mouse actions, or issue text commands.

Clicking on items in the *menu bar* along the top of the window displays pulldown menus that provide access to most of Prism's functionality.

Frequently used menu items can be moved to the *tear-off region*, below the menu bar, to make them more accessible.

The *status:* area displays messages about the program's status.

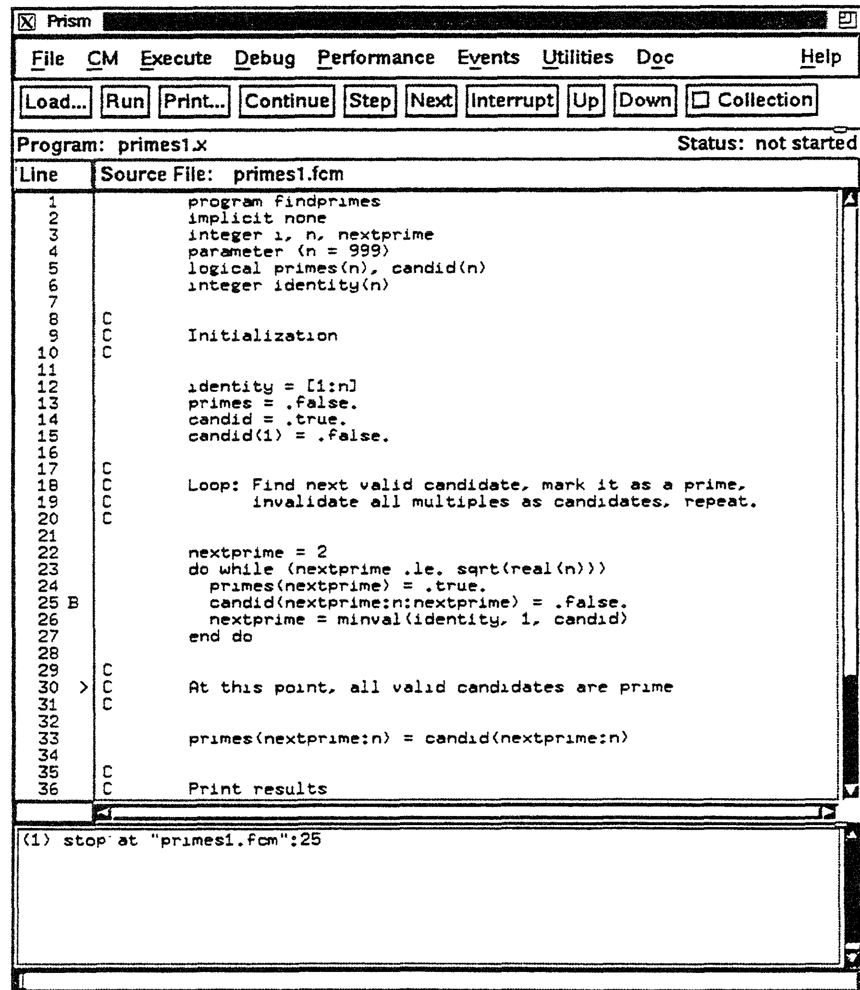


Figure 14. The main window of Prism.

The *source window* displays the source code for the executable program. The user can scroll through this source code or display a different source file. When a program stops execution, the source window is automatically updated to show the code currently being executed. The user can click on variables or expressions in the source code to print their values.

The *line-number region* is associated with the source window. Clicking to the right of the line number sets a breakpoint at that line.

The *command window* at the bottom of the main window displays messages and output from Prism. The user can also type commands in the command window, rather than use the graphical interface.

6.2 Analyzing Program Performance

In cooperation with the compilers and run-time library routines, Prism provides the performance data essential for effectively analyzing and tuning programs. The data includes:

- control processor user and system time
- processing time
- time spent transferring data between control processor and nodes
- time spent in general Data Network communication
- time spent doing specific patterns of Data Network communications, such as nearest-neighbor on a grid
- time spent doing reductions and parallel prefix

The performance data is displayed as histograms and percentages. For each type of time measurement, the user can also see the data broken down for each procedure and each source line in the program. The data on procedures is available in two versions. One gives a flat per-procedure view of the utilization of the resource; the other shows utilization using the dynamic call graph of the program.

6.3 Visualizing Data

In data parallel computing, it is often important to obtain a visual representation of the data elements that make up a parallel variable or expression. In Prism, the user can create *visualizers* for variables or expressions. Available representations include:

- *Text*, where the data is shown as numbers or characters
- *Pixel*, where each data element is mapped to a single color pixel, based on a range specified by the user
- *Boolean*, where each data element is mapped to a single pixel, either black or white, based on a cutoff value specified by the user

A *data navigator* allows manipulation of the display window relative to the data being visualized. If a parallel array is multidimensional, the visualizer displays a slice through the array; the data navigator provides controls for selecting the array axes to be displayed and the position of the slice. The user can update a visualizer or save a snapshot of it.

Figure 15 shows a text visualizer for a two-dimensional array.

a

File Options

1

101	102	103	104	105
201	202	203	204	205
301	302	303	304	305
401	402	403	404	405
501	502	503	504	505
601	602	603	604	605
701	702	703	704	705
801	802	803	804	805
901	902	903	904	905
1001	1002	1003	1004	1005
1101	1102	1103	1104	1105
1201	1202	1203	1204	1205
1301	1302	1303	1304	1305
1401	1402	1403	1404	1405
1501	1502	1503	1504	1505
1601	1602	1603	1604	1605
1701	1702	1703	1704	1705
1801	1802	1803	1804	1805
1901	1902	1903	1904	1905
2001	2002	2003	2004	2005
2101	2102	2103	2104	2105
2201	2202	2203	2204	2205

Figure 15. A visualizer.

6.4 On-Line Help and Documentation

Prism features a comprehensive on-line help system. Help is available for each pulldown menu and dialog box. Moreover, the Help Index, shown in Figure 16, contains a list of entries on which the user can obtain information. Each help topic has a list of related topics, subtopics, terms, and commands associated with it; clicking on any of these opens a new window displaying information about the selected item.

Prism also provides an interface to on-line documentation for the Connection Machine system. The user can call up a **man** page for a CM command or library routine, or view the portions of the Connection Machine documentation set that are most relevant to a specific question.

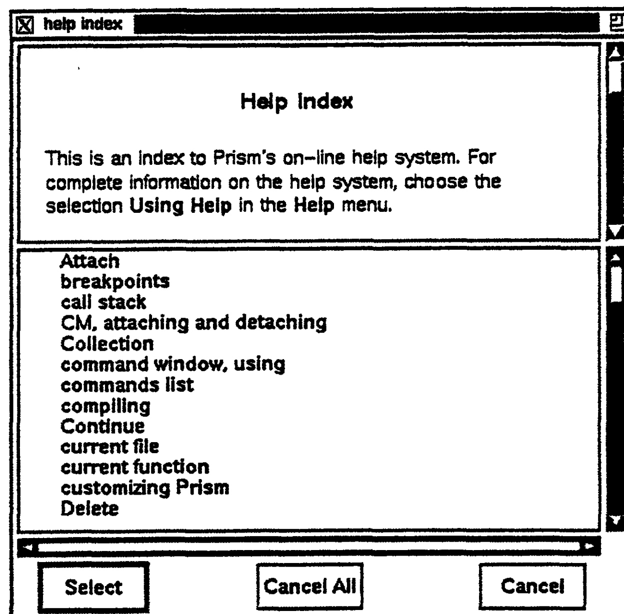


Figure 16. The Help Index.

Chapter 7

The Program Execution Environment

The program execution environment on a CM-5 partition supports both interactive program execution and batch execution, using the Network Queueing System (NQS) batch system. In either case, the program executes on the partition manager and accesses the associated set of processing nodes, plus I/O devices and other devices, such as graphics workstations, as needed.

Access to the interactive environment is achieved through remote login or remote shell commands. Access to the batch environment is achieved through NQS's `qsub` command, delivered either from the CM itself or from a remote machine.

The interactive environment is, by default, a timeshared environment. Access to partitions may be limited by UNIX permissions, however, allowing exclusive use of a given partition by some particular user, project, or batch queue. Thus, the system administrator can choose not only how to partition the system, but which partitions to make available as what sort of environment: open for general access, open for exclusive access, or open to batch jobs only.

To further enhance the program execution environment, the CM-5 offers facilities for performance analysis and for checkpointing programs. The Prism programming environment (discussed in the previous chapter), and CM timers provide the former; CM checkpointing routines provide the latter.

7.1 Checkpointing

Many applications that run on the Connection Machine system require extended execution time. Users may need to be able to interrupt and later restart such a program for any number of reasons: to allow it to run only when the system is

not needed for other use, to allow for scheduled machine downtime, to 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 partition manager (PM) and nodes, a list of the files that the program has 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 batch jobs and interactive jobs, including those running under **cmdbx** and Prism. It can be used on programs that execute on the PM only, as well as those that use both the PM and the nodes.

7.2 Timers

A CM timer calculates, with microsecond precision, both the total elapsed time (wall-clock time) and the amount of time during which the nodes are 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.

7.3 Timesharing

The Connection Machine system uses the UNIX timesharing mechanisms, with all the administrative flexibility they provide. Each partition manager controls timesharing on its partition, switching processes in and out as necessary. (Because a data parallel process running on the PM plus the nodes is a single process, it is switching as a single entity.)

7.4 NQS

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 directed to a specific PM, and which run on the partition that is controlled by that PM at the time the job is submitted; and pipe queues, which feed jobs (via batch queues) to any suitable partition that is available to run them. The pipe queue can be directed to any available partition, or only to partitions that meet specified minimal resources. NQS queries current partitions to find one suitable for running jobs from these queues.

NQS allows the administrator to control the number and characteristics of queues at a site and to define the hours during which each queue will accept and execute jobs. Note that the two sets of hours are not necessarily identical: 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

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 electronic mail of a 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.

Chapter 8

The CM Fortran Programming Language

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.

8.1 Structuring Parallel Data

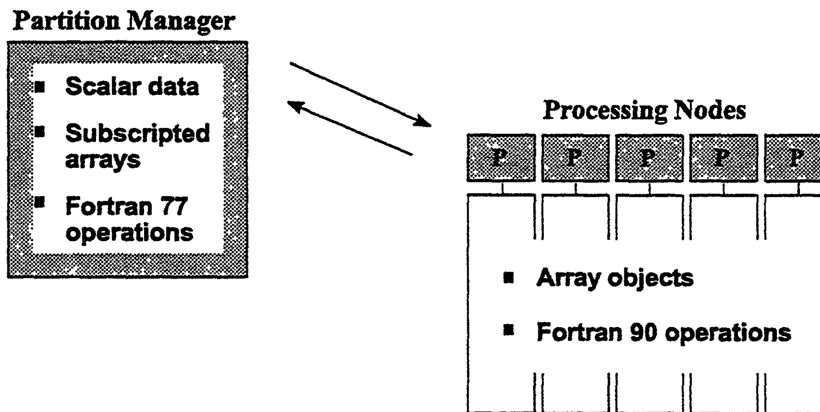
Fortran 90 allows an array to be treated either as a set of scalars or as a first-class object. As a set of scalars, array elements must be referenced explicitly in a DO construct. In contrast, a reference to an array object is an implicit reference to all its elements (in unspecified order). For example,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

A as a set of scalars	A as an object

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

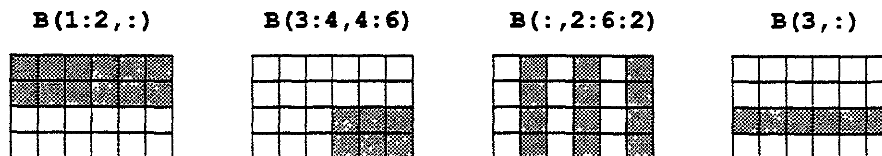
To operate on multidimensional arrays, DO loops must be nested to reference each element explicitly. In the statement `A = A + 1`, however, A could be a scalar, a vector, a matrix, or a higher-dimensional array.

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 partition manager in the normal serial manner. Any array that is referenced as an object is stored in node memory, one element per processor, and processed in parallel. In essence, the partition manager executes all of CM Fortran that is Fortran 77, and the nodes execute all the array extensions drawn from Fortran 90. No new data structure is required to express parallelism.



The simple array reference A may be written more explicitly using a *triplet subscript*, `A(1:100:1)`, which resembles the control specification of a DO loop. Using triplet subscripts, you can replace one or more DO loops with an array reference that indicates all the elements of interest — and thereby cause the array to be processed in parallel.

An implicit triplet — that is, the array name alone — is usually used for whole arrays. You can, however, explicitly specify any of the index 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.

8.2 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. An assignment statement where the entire array is referenced as an object has this effect. For example, consider the following assignment statement for an 8 x 8 x 8 array `C`:

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

The CM system allocates one element of `C` in each of 512 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*2.0 + 1.0 + 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 unspecified). 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
```

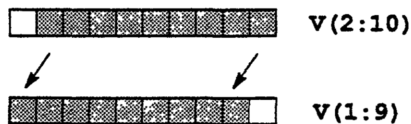
8.3 Communicating in Parallel

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

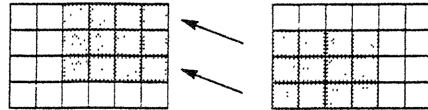
Many applications, such as convolutions and image rotation, need to move data in regular grid patterns. One way to specify such motion in Fortran 90 is by assigning array sections. For example, to shift vector values to the left:

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



To shift data on more than one dimension:

```
A(1:3,3:6) = A(2:4,1:4)
```



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 )
```

One notable use of **CSHIFT** is in so-called "stencils," array expressions that compute a weighted sum of neighboring points of a specific grid point. A simple example would be

```
A = C3*B + C1*CSHIFT (B,DIM=1,SHIFT=-1) + C2*CSHIFT (B,DIM=2,SHIFT=-1)
```

The CM Fortran compiler includes optimizations that provide particularly high performance for stencils.

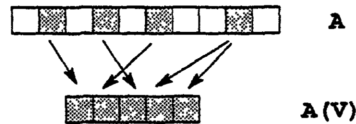
General Communication

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 an array containing 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) )
```

8.4 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**.

Chapter 9

The C* Programming 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.

9.1 Structuring Parallel Data

In C*, data is allocated on the processing nodes 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 specifies the number of dimensions (or *axes*) in the shape and the number of

positions along each dimension. Positions correspond to processors (or virtual processors). For example,

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

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

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 **ints** 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.

9.2 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

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). The corresponding compound assignment operators **<?=** and **>?=** are also included. The operator **%%** provides a true modulus operation (as compared to the remainder operator **%**).

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.

9.3 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

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 17.

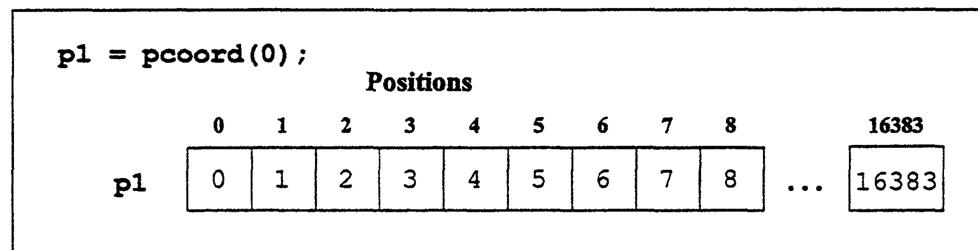


Figure 17. The use of pcoord with a one-dimensional shape.

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 assignment

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

the parallel variable **dest** gets values from **source**; the values in **index** indicate 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.

9.4 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 10

The *Lisp Programming Language

The *Lisp language is a high-level programming language for the Connection Machine system. Based on the Common Lisp programming language, *Lisp allows you to write data parallel programs for the CM using the data types, programming constructs, and programming style of Lisp. Programs written in *Lisp make full use of CM 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 calculations, interprocessor communication, and data shape specification.

A *Lisp program is simply a Common Lisp program that includes calls to *Lisp operators. A call to a *Lisp operator causes all active CM processors to execute that operation in parallel. Thus, *Lisp is fully compatible with Common Lisp; programs written in Common Lisp will run unmodified in *Lisp.

*Lisp functions and macros are defined via **`defun`** and **`defmacro`**, just as in Common Lisp. *Lisp programs are compiled by the *Lisp compiler, which includes (and is invoked in the same ways as) the Common Lisp compiler. This means that programs in *Lisp and Common Lisp can be written, compiled, and tested with the same editors and debuggers.

10.1 Structuring Parallel Data

Scalar and Parallel Data

*Lisp is an extension of Common Lisp and therefore includes all the standard Common Lisp data types. These data types are collectively referred to as *scalar* data. *Lisp also supports an additional parallel data type, called a *pvar*. 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 simultaneously by all active CM processors, with each processor modifying only its own value for the pvar. Many of the scalar data types in Common Lisp have corresponding pvar equivalents. The eight basic pvar data types are boolean, integer, floating-point, complex, character, array, structure, and front-end value.

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

(defpvar my-five-pvar 5)  ;; Allocating a permanent pvar

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

As these examples show, *Lisp supports temporary, permanent, and local pvars.

- Temporary pvars are allocated by the `!!` (bang-bang) function, which takes a single scalar value as its argument and returns a temporary pvar with that value in every processor.
- Local pvars are allocated by the **`*let`** and **`*let*`** functions. They exist for the duration of a body of *Lisp code.
- Permanent pvars are allocated by the **`defpvar`** function. They remain in existence until specifically deallocated.

Defining the Shape of the Data

The shape of the data stored in a pvar is determined by a grid of processors that the CM is currently simulating. The defining property of a processor grid is its *geometry*: 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). For example, the expression

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

defines a VP set named **`my-vp-set`** with 64 x 64 processors and associates two permanent pvars with it: an integer pvar **`x`** and a single-precision floating-point pvar **`y`**.

Because the CM can simulate many grids within a single program, *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. If no VP set has been defined, all pvar operations occur within a *default VP set* that is automatically defined whenever *Lisp starts up.

Processor Addressing

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, accessible via the *Lisp operation **`(self-address!!)`**.

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

*Lisp allows 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. For example:

- **(pref my-pvar 10)** returns the value of **my-pvar** in processor 10.
- **(*setf (pref my-pvar 10) 123)** stores the quantity 123 into processor 10 of **my-pvar**.
- **(*setf (pref my-pvar (cube-from-grid-address 5 7)) 111)** stores 111 into **my-pvar** at grid location (5,7).
- **(*set pvar1 pvar2)** copies the contents of **pvar2** into **pvar1** in all active processors.
- **(*set pvar1 5)** stores the value 5 into **pvar1** in all active processors.

The *Lisp operation **ppp** (short for **pretty-print-pvar**) displays 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
```

10.2 Computing in Parallel

The parallel operations supplied by *Lisp are modeled very closely on the existing scalar operations of Common Lisp and include 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 (*!! pvar1 pvar2)))
```

multiplies the values of **pvar1** and **pvar2** in all active processors, adds the value of **pvar1**, then stores the result 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).

In addition, *Lisp redefines many Common Lisp operations so that they will accept pvar arguments and will call the appropriate *Lisp operations to compute the result. This means that the above ***set** example can be rewritten as:

```
(*set pvar2 (+ pvar1 (* pvar1 pvar2)))
```

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 not changed. *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 code, not whether the code will be evaluated at all. 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 19)
0 -1 2 -3 4 -5 6 -7 8 -9 10 -11 12 -13 14 -15 16 -17 18
```

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.

10.3 Communicating in Parallel

Like all CM languages, *Lisp supports both regular and irregular communication. For example:

- **`news!!`** causes each active processor to get a value from another processor a fixed distance away on the grid.
- **`*news`** causes each active processor to send a value to another processor a fixed distance away on the grid.
- **`pref!!`** allows each active processor to get a value from any other processor in the grid.
- **`*pset`** allows each active processor to send a value to any other processor in the grid.

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).

10.4 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. Some examples:

- **scan!!** and **segment-set-scan!!** 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.

  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.

- **spread!!** replicates the value of a pvar at a given coordinate to all processors along a selected dimension of the currently selected grid. A related operation, **reduce-and-spread!!**, combines the operations of scanning and spreading.

- 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 11

CM Scientific Software Library

The Connection Machine Scientific Software Library (CMSSL) is a constantly growing set of numerical routines that support computational applications while exploiting the massive parallelism of the Connection Machine system.

CMSSL provides data parallel implementations of familiar numerical routines, providing new solutions to problems of both performance and algorithm choice and design.

CMSSL provides immediately useful routines for users whose work demands solutions in such areas as partial differential equations, optimizations, signal processing, and statistical analysis. It also provides a strong base for the development of further tools.

While CMSSL routines have been designed to meet the needs of Fortran users, any CMSSL routine may be called from any CM programming language that supports the data formats required by that routine.

The current version of the library concentrates on four critical areas of scientific programming: linear algebra, Fast Fourier Transforms, random number generation, and statistical analysis. Certain communications primitives important to linear algebra are also provided.

11.1 Linear Algebra Routines

- *Matrix Multiplication*. Multiplies 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.
- *Outer Product.* Computes the outer product of two vectors 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.
- *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.
- *Sparse Matrix Vector Product.* Computes the product of an arbitrary sparse matrix, whose non-zero elements are stored in a packed vector, and a vector. An associated setup routine provides options that may improve performance.
- *Block Sparse Matrix Operations.* Computes the product of a block sparse matrix with a vector or a dense matrix. Gathers elements from the source vector or matrix, and scatters solution elements to the product vector or matrix, using pointers provided by the application. An associated setup routine provides options that may improve performance.
- *Sparse Matrix Gather Utility.* Gathers elements of a vector into an array using pointers supplied by the application. Preprocessing is performed by an associated setup routine.
- *Sparse Matrix Scatter Utility.* Scatters elements of an array to a vector using pointers supplied by the application. Preprocessing is performed by an associated setup routine.
- *All-to-All Broadcast.* Given a real or complex array and a designated axis, performs an in-place, stepwise broadcast of every array value on the axis to every location along the axis.

- *Multidirectional NEWS.* Performs multidirectional and/or multidimensional array shifts in an array geometry.

Further routines in areas such as LU decomposition and Eigenvalues will soon be added.

Multiple Instances

Most linear algebra routines are designed to support multiple instances. They allow multiple, independent matrices to be solved, transformed, or multiplied concurrently. In addition, they allow multiple vectors or multiple right-hand sides, where relevant, to be associated with each matrix to be multiplied or solved.

Matrix vector multiplication, for example, may be performed with a single matrix and a single vector by specifying each as an object whose elements are spread across many processing nodes. 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 node.

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 and layout 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.

To solve a single system, one specifies the upper, main, and lower arguments as one-dimensional (see Figure 18).

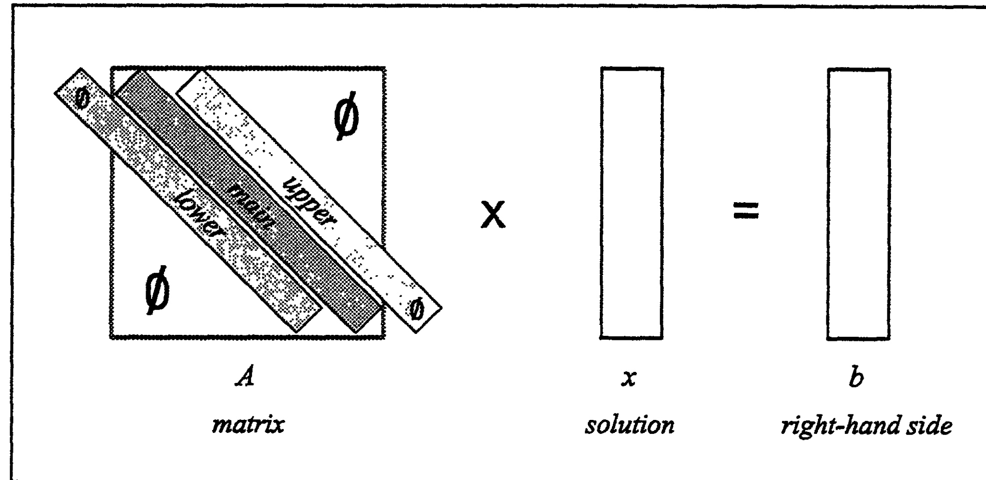


Figure 18. A single tridiagonal system with a single right-hand side.

To solve for multiple right-hand sides, one gives the right-hand-side argument (which will be replaced by the solutions) an in-processor (serial) dimension equal to the number of right-hand sides (*nrhs*) (see Figure 19).

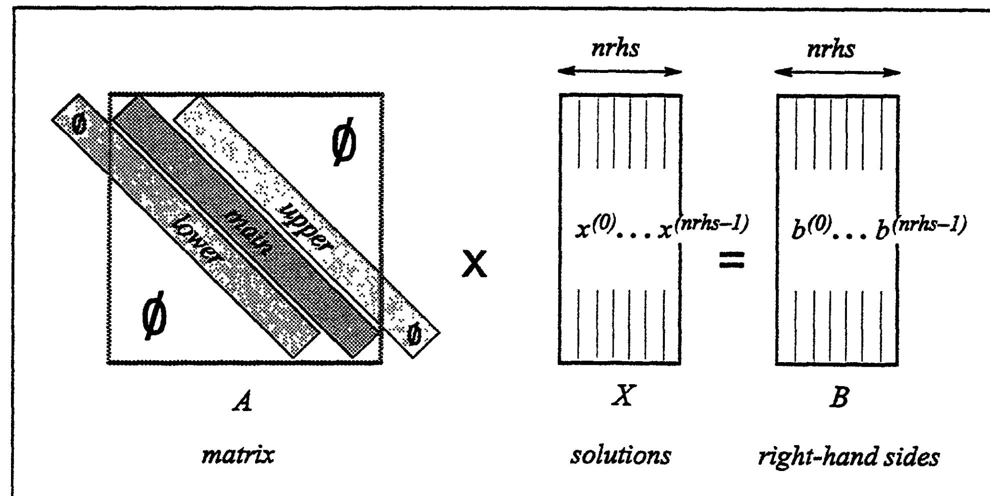


Figure 19. Single tridiagonal system with multiple right-hand sides and solutions.

To solve multiple systems, one specifies 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 is similarly specified in two dimensions (see Figure 20).

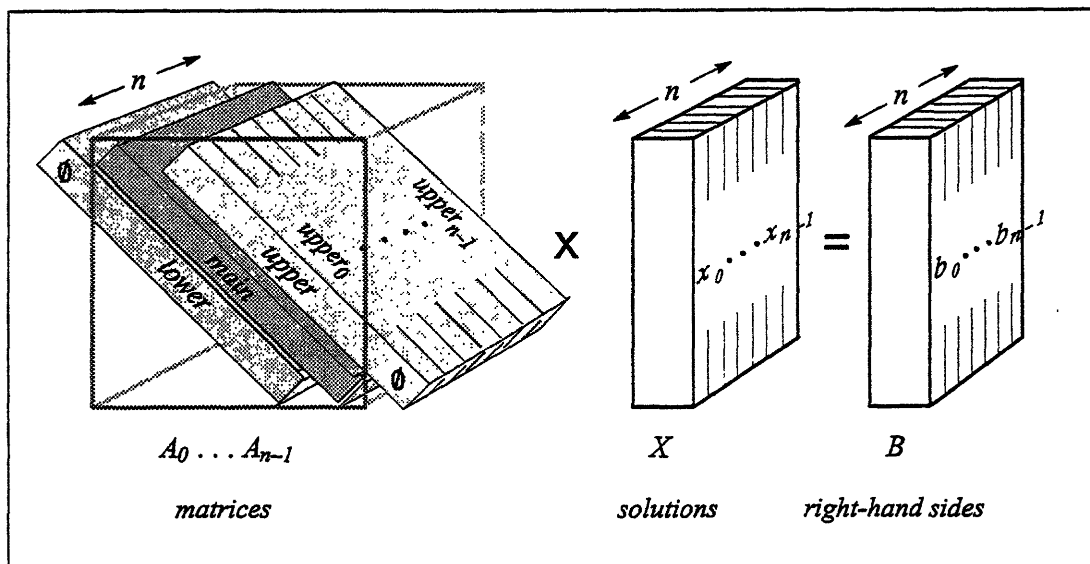


Figure 20. Multiple tridiagonal systems with single right-hand side for each system.

To solve multiple systems each with multiple right-hand sides, one specifies the right-hand-side (solution) argument 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 21).

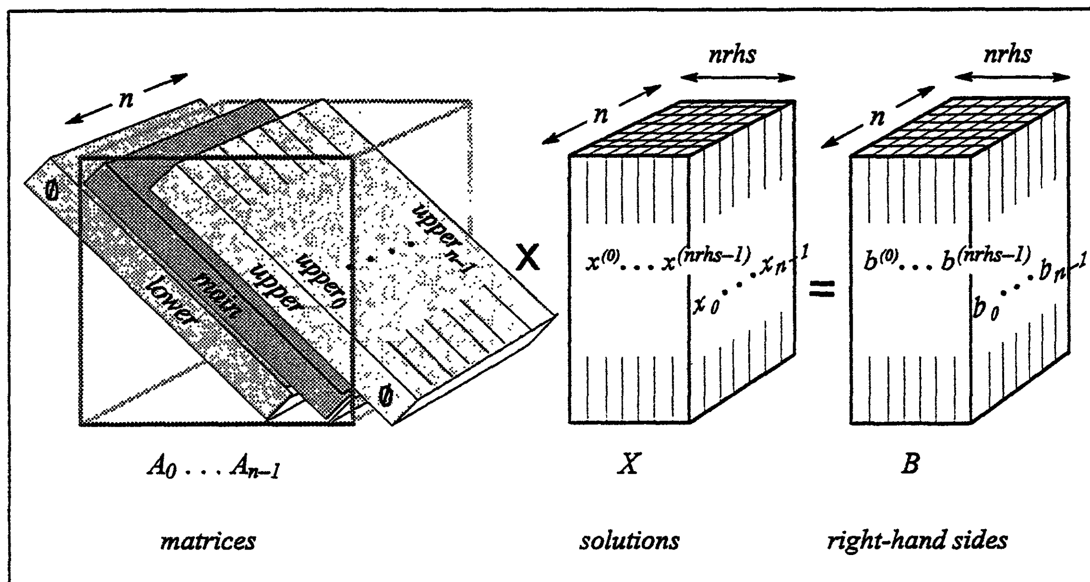


Figure 21. Multiple tridiagonal systems with multiple right-hand sides for each system.

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 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.

Solving Dense Systems

For dense matrices, 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 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 pivoting strategy to improve the numerical stability of this solver is currently planned.

Sparse Matrix Operations

CMSSL includes routines for multiplying an elemental sparse matrix by a vector, and for multiplying a block sparse matrix by a vector or dense matrix. An elemental sparse matrix is stored as a packed vector; a block sparse matrix is stored as a three-dimensional array, with two axes representing the rows and columns of the blocks and the third axis identifying the blocks themselves. The elements to be multiplied with each block are *gathered* from the source vector or matrix, and the results are *scattered* to form the product vector or matrix. The application supplies two pointer arrays that represent the sparsity of the matrix and specify the gathering and scattering patterns. Utility routines allow applications to perform pre-processed gather and scatter operations separately from other computations.

Both the elemental and the block sparse matrix operations include options for improving performance. The setup routines that perform the pre-processing required for the multiplication can save the communication pattern, or *trace*, associated with the system's sparsity. One call to the setup routine can be followed by multiple calls to the routines that compute the products; the overhead associated with the setup can thus be amortized over any number of multiplica-

tions. Randomization of the source and product arrays may provide additional performance improvements by minimizing the routing conflicts that occur during the data motion phase of the multiplication.

Communication Routines

CMSSL provides two communication primitives: all-to-all broadcast and multi-directional NEWS. These operations exploit the full communication bandwidth of the Connection Machine for regular communication patterns. These functions can be significantly faster than the equivalent handwritten functions. In particular, the improvement in the performance of the all-to-all broadcast function varies between a factor of 1 and d/2, where $d = \log_2(N/32)$ and N is the number of physical processors. Typically, the performance of the multi-directional NEWS function is better than the equivalent handwritten Fortran code by a factor of nearly two. Applications involving higher-dimensional lattices may see a much larger performance improvement.

11.2 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. This pattern always operates between data elements at a distance of 2^N.

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

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

- Simple FFT, used to transform a data set in the same direction along all axes
- 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 implementation limits.

11.3 Random Number Generators

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

- Fast RNG
- VP RNG

These random number generators use a lagged-Fibonacci algorithm 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

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*. The Fast RNG allocates one state table per physical Connection Machine node. The VP RNG allocates one state table per virtual processor (that is, per array position). The Fast RNG thus consumes substantially less memory than the VP RNG. The VP RNG can produce identical results on differently sized partitions.

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

11.4 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 12

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 way to interpret the large data sets and complex problems common to the applications run on the Connection Machine system.

12.1 A Distributed Graphics Strategy

In keeping with its role as a network resource, the CM-5 uses a distributed graphics strategy to support a wide range of user applications. The key items in this strategy are

- the parallel processing power of the Connection Machine supercomputer
- the specialized power and interactive visualization environments provided by dedicated graphics display stations
- the use of standard protocols, such as X11, to allow communication among a variety of hardware and software

A full range of interconnections is supported, from high-speed HIPPI interfaces through FDDI and Ethernet for longer-distance communications, to allow fast communication between the CM and graphics display stations.

Basically, the pattern is as follows: Computations carried out by the CM's parallel processing nodes manipulate data to create graphics primitives, which can then be sent to a graphics display station anywhere on the network. This strategy lets users maximize the value of existing hardware and software, while taking advantage of the computational speed and power of the CM, the high bandwidth of CM I/O, and the rendering power and speed of graphics workstations (such as those from Silicon Graphics, Stardent, and Sun), which implement many advanced rendering techniques in hardware and offer extensive visualization environments to make interactive rendering easy for the user.

Following this strategy, for example, a scientific visualization program can use the CM to compute image geometry (including, for example, polygon coordinates and color information) and then send it from the CM directly to local memory on the graphics workstation, where the results of simulations done on the CM can be interactively displayed and analyzed.

At the workstation, users benefit from the ability to create and use graphical user interfaces (GUIs). GUIs are widely used today and growing in popularity, as their use enhances productivity for applications programmers and users alike, allows tighter coupling of simulation and visualiation, and allows such activities as simulation steering. Many tools exist for the creation of such interfaces, and all are now available to the CM programmer.

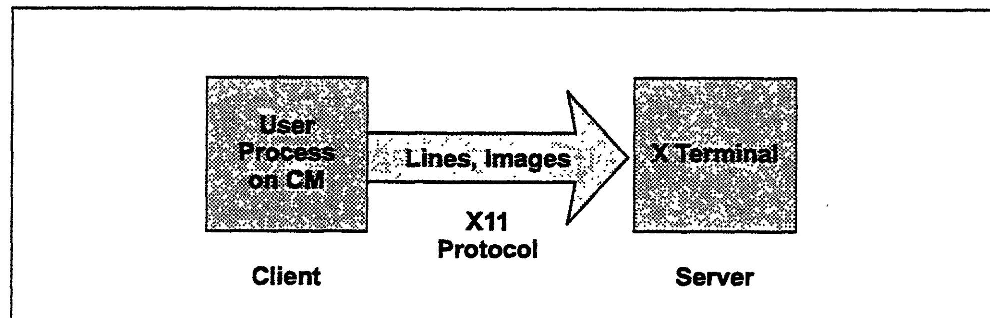


Figure 22. Distributed graphics.

12.2 An Integrated Environment

By using the distributed graphics strategy described above, together with an underlying protocol such as X11, programmers can create and use a wide variety of

integrated enviroments for their computational and visualization tasks. Connection Machine software provides an environment that permits the exchange of very large data sets between the CM and framebuffers, workstations, or X window terminals.

The CM programming environment, Prism, exports data using a common, easy-to-understand format; thus, programs running within Prism can generate data for use within a visualization environment.

12.3 The X11 Protocol

Support for the network-based X graphics protocol is integral to the CM distributed graphics strategy, since use of this protocol facilitates both data transfer and the use of GUIs, and allows considerable portability: data from a CM can be displayed on any X workstation.

But simple portability is not the only issue involved. As useful as graphics workstations are, the extra-large data sets typically used in CM applications frequently provide more data than such workstations can readily handle. The solution to this problem lies partly in using the CM's power to reduce the volume of information contained in the data sets so that the workstations can handle it rapidly, and partly in the successful integration of visualization environments, workstations, and high-speed framebuffers into a coherent system for rendering scientific data.

12.4 The CMX11 Library

The CMX11 library provides routines that allow the transfer of parallel data between the CM and any X11 terminal or workstation. It contains routines that draw and fill points, lines, rectangles, and arcs; draw text strings, polygons, and image-text strings; and draw and get images. The CMX11 library thus extends the X11 libraries by providing parallel network calls that substitute parallel variables for serial arrays. For example, where the X library offers an **XDrawPoint** routine, the CMX library offers **CMXDrawPoint**:

```
CMXDrawPoint(Display *display, Drawable *d,
             GC gc, int x, int y)
```

where x and y are pointers to parallel variables, and all other arguments are identical to the serial call.

Similarly, the CMX version of the X11 **XPutImage** routine uses the arguments and semantics of the original to provide a parallel transfer of an image that exists as a parallel array:

```
CMXPutImage(display, d, gc, data, depth,
            src_x, src_y, dest_x, dest_y,
            width, height)
```

Note that no X protocol extensions are necessary, since the underlying CM socket mechanism makes the data source entirely transparent to the server. In most cases, the user simply makes the parallel version of the normal call, and the parallel data is inserted into the data stream in the same format and position as it would have been in the equivalent serial call. This greatly facilitates the user's task.

12.5 Visualization Environments

Ongoing research in all areas of distributed visualization — for example, data transfer protocols, distributed application interfaces, and visual programming languages — will expand Connection Machine visualization support to include full support for distributed visualization environments. These environments will allow several processes to communicate large data sets among themselves and to cooperate in producing the visualization. Interfaces are currently being developed for existing visualization environments such as AVS (Stardent) and Explorer (SGI). Since the CM-5 system communicates with the outside world via UNIX sockets, users can easily integrate CM applications into these visualization systems. In addition, visualization modules such as filters, volume visualization tasks, or polygon renderers may be developed to execute on the CM-5 itself and thus directly handle the large data sets commonly associated with data parallel applications. The ability to write data to tape at any intermediate stage for later processing will also be supported.

Chapter 13

CM Message Passing Library

Users who have written C and Fortran programs for machines with MIMD-only architectures can port these programs to the CM-5 by replacing the original message-passing library calls with calls to the CM message-passing library, CMMD. CMMD routines permit cooperative message passing among processing nodes, thus providing simple processor communication that falls outside the range of the data parallel languages.

CMMD supports a programming model frequently referred to as host/node programming. One program runs on the host (a CM-5 partition manager), and independent copies of the node program run on each of the processing nodes. The host may have little involvement aside from initially invoking the node program and perhaps providing user interface services.

CMMD permits concurrent processing in which synchronization occurs only between matched sending and receiving nodes and only during the act of communication. At all other times, computing on each node proceeds asynchronously.

The initial release of CMMD supports primarily *blocking* message sending and receiving. Blocking routines are synchronized routines in which senders wait for their recipients to respond before continuing execution, and vice versa. Failure on the part of the program to ensure that each call to a sending routine is matched with a call to a receiving routine in the destination processor is likely to deadlock the user's process. (The CM-5 timesharing system ensures that any such deadlock affects only the erring program; deadlock has no effect on other programs sharing the partition.)

This initial release does provide limited support for *non-blocking* sending and receiving of short messages. Future releases are expected to provide further support for asynchronous message passing.

In addition, global functions — in which all nodes must participate — provide for broadcasting data from the host, for scan and reduce operations, and for global synchronization. These global functions make use of the CM-5 interprocessor networks; thus, they take direct advantage of the CM-5's hardware support for global communications.

The library functions can be called from C and Fortran. They are summarized in the paragraphs below.

13.1 Initialization

Four functions toggle the system between message-passing and data parallel communication. The function **CMMD_sys_enable()** changes network participation to that required for message passing, and initializes the data structures required by the message-passing environment. During program execution, **CMMD_suspend** and **CMMD_resume** temporarily toggle the processing mode. When the program finishes, **CMMD_sys_disable()** reverts the network to its previous state.

13.2 Message Passing

Routines are provided for both separate and simultaneous sending and receiving.

- **CMMD_send** sends a message from one processor (node or host) to another.
- **CMMD_receive** receives a message sent from another processor. (A call to **CMMD_msg_pending** can determine whether a message is waiting to be received.)
- **CMMD_send_and_receive** provides for simultaneous sending to one node and reception from one (generally different) node. This call is particularly useful for structured communications, most commonly found in grid topologies. For instance, it is frequently used for shifts and other "nearest neighbor" operations, with many nodes simultaneously sending in one direction and receiving from another.
- **CMMD_swap** allows two processors to trade messages with each other.

Each of these routines has a vector version, which allows the sending or receiving of data elements regularly spaced within the buffers by a specified stride. Because regular and vector calls can be mixed, the CMMD message-passing routines can perform scatter/gather behavior. (See Figure 23.)

Routines also exist for the separate sending and reception of non-blocking messages. One pair of these routines allows programmers to create their own protocols.

13.3 Informational Routines

CMMD routines are available to provide information on such matters as partition size, host and node IDs, and the size, tag, and sender of the last message sent or received, or of a message waiting to be received.

13.4 Global Synchronization

CMMD provides routines that explicitly synchronize all nodes (and possibly the host as well). It also provides informational routines that allow, but do not enforce, synchronization. **CMMD_sync_host_with_nodes** synchronizes the host (partition manager) with all nodes. This function, called on the host, does not return until the corresponding routine, **CMMD_sync_with_host** is called from all the nodes.

CMMD_barrier_sync, called from the host, synchronizes the host with completion of all currently executing node functions.

CMMD_sync_with_nodes synchronizes the node with all other nodes. This function will not return until it has been called in all nodes. (Note that this function does not involve the host.)

CMMD_set_global_or, which is callable on either the host or the nodes, contributes the value 0 or 1 from the processor to a global OR function.

CMMD_get_global_or similarly returns the current value of a global OR function over all nodes including the host.

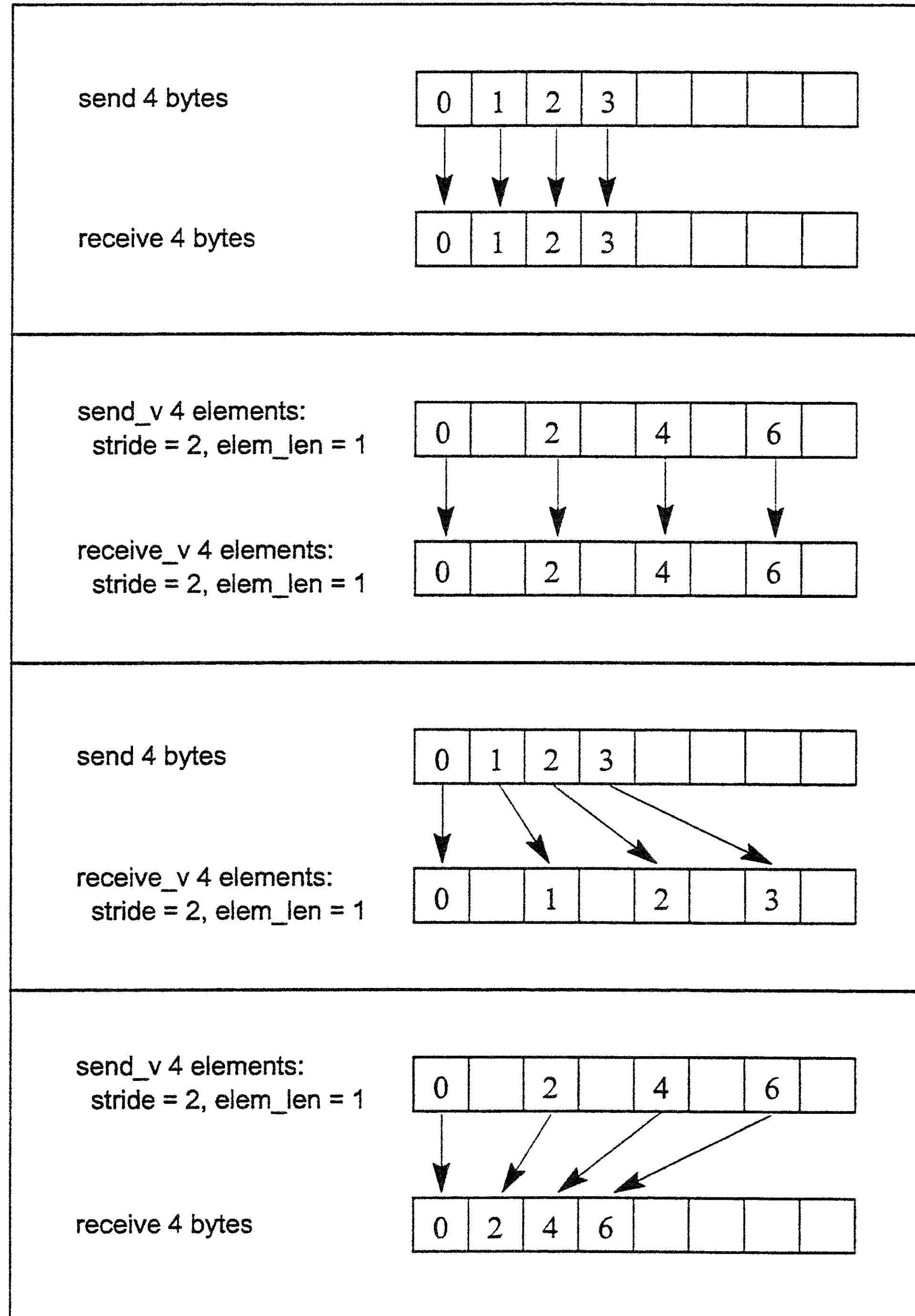


Figure 23. Sending and receiving data.

13.5 Global Operations

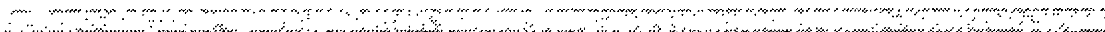
The CMMD library provides a number of global functions. These functions perform their operations over all the nodes; some include and some exclude the host.

Global functions include

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

Reduce and parallel prefix operations can perform summation, find a maximum or minimum value, or perform bitwise AND, OR, or XOR.

Part III
CM-5 Architecture



Chapter 14

Architecture Overview

A Connection Machine Model CM-5 system contains thousands of computational processing nodes, one or more control processors, and I/O units that support mass storage, graphic display devices, and VME and HIPPI peripherals. These are connected by the Control Network and the Data Network. (For a high-level sketch of these components, see Figure 24.)

14.1 Processors

Every processing node is a general-purpose computer that can fetch and interpret its own instruction stream, execute arithmetic and logical instructions, calculate memory addresses, and perform interprocessor communication. The processing nodes in a CM-5 system can perform independent tasks or collaborate on a single problem. Each processing node has 8, 16, or 32 Mbytes of memory; with the high-performance arithmetic accelerator, it has the full 32 Mbytes of memory and delivers up to 128 Mips or 128 Mflops.

The control processors are responsible for administrative actions such as scheduling user tasks, allocating resources, servicing I/O requests, accounting, enforcing security, and diagnosing component failures. In addition, they may also execute some of the code for a user program. Control processors have the same general capabilities as processing nodes but are specialized for performing managerial functions rather than computational functions. For example, control processors have additional I/O connections and lack the high-performance arithmetic accelerator. (See Figure 25.)

In a small system, one control processor may play a number of roles. In larger systems, individual control processors are often dedicated to particular tasks and

referred to by names that reflect those tasks. Thus, a control processor that manages a partition and initiates execution of applications on that partition is referred to as a partition manager (PM), while a processor that controls an I/O device is called an I/O control processor (IOCP).

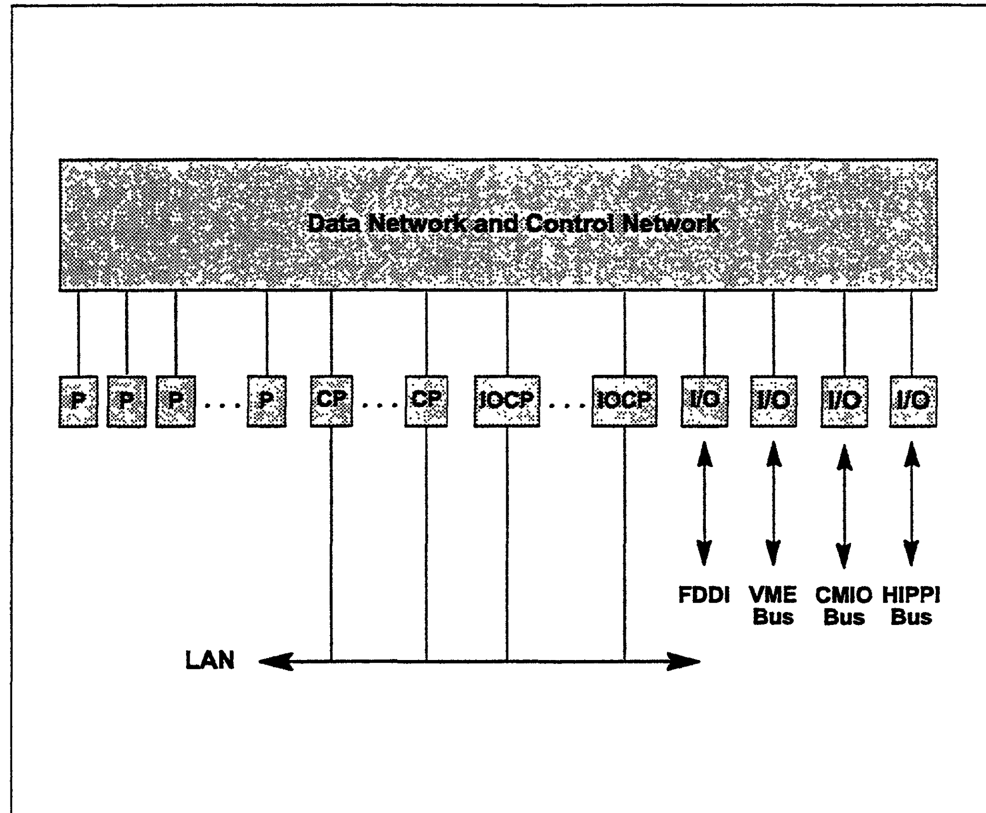


Figure 24. System components.

A CM-5 system contains tens, hundreds, or thousands of processing nodes, each with up to 128 Mflops of 64-bit floating-point performance. It also contains a number of I/O devices and external connections. The number of I/O devices and external connections is independent of the number of processing nodes. Both processing and I/O resources are managed by a relatively small set of control processors. All these components are uniformly integrated into the system by two internal communications networks, the Control Network and the Data Network. The Control Network provides multiway operations that coordinate thousands of participants, while the Data Network supports high-bandwidth bulk data transfers. The capacity of each network scales up with the size of the system; every processing node or I/O device gets the network capacity it needs.

14.2 Networks and I/O

The Control Network provides tightly coupled communications services. It is optimized for fast response (low latency). Its functions include synchronizing the processing nodes, broadcasting a single value to every node, combining a value from every node to produce a single result, and computing certain parallel prefix operations.

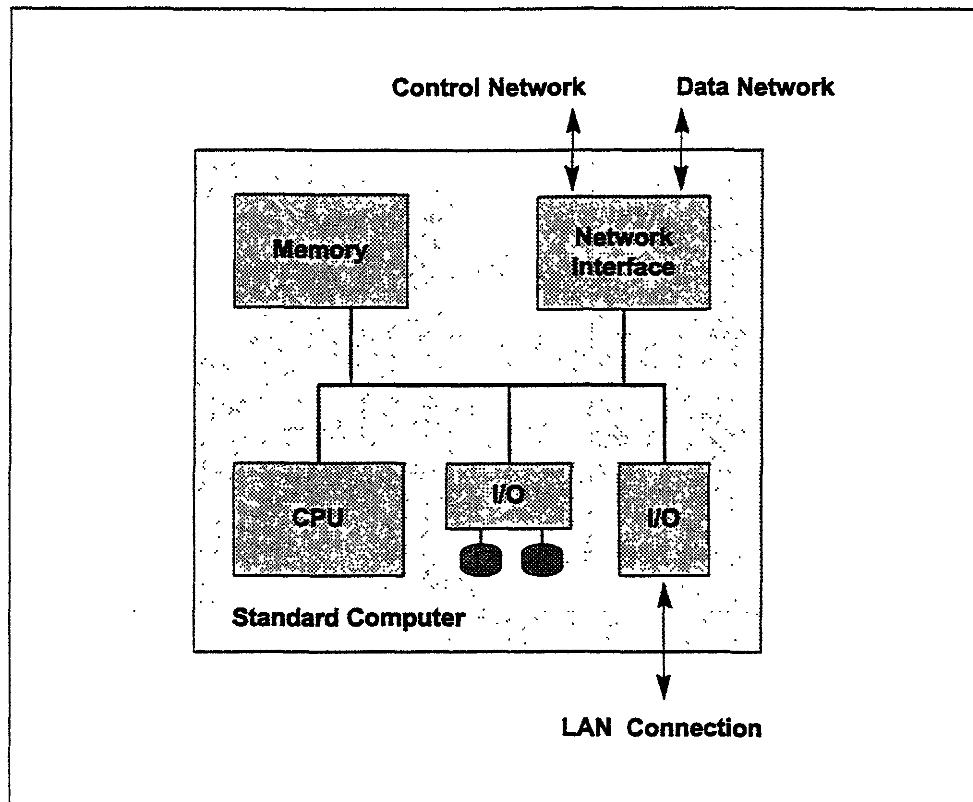


Figure 25. Control processor.

The basic CM-5 control processor consists of a RISC microprocessor, memory subsystem, I/O (including local disks and Ethernet connections), and a CM-5 Network Interface, all connected to a standard 64-bit bus. Except for the Network Interface, this is a standard off-the-shelf workstation-class computer system. The Network Interface connects the control processor to the rest of the system through the Control Network and Data Network. Each control processor runs CMOST, a UNIX-based operating system with extensions for managing the parallel-processing resources of the CM-5. Some control processors are used to manage computational resources and some are used to manage I/O resources.

The Data Network provides loosely coupled communications services. It is optimized for high bandwidth. Its basic function is to provide point-to-point data delivery for tens of thousands of items simultaneously. Special cases of this functionality include nearest-neighbor communication and FFT butterflies. Communications requests and data delivery need not be synchronized. Once the Data Network has accepted a message, it takes on all responsibility for its eventual delivery; the sending processor can then perform other computations while the message is in transit. Recipients may poll for messages or be notified by interrupt on arrival. The Data Network also transmits data between the processing nodes and I/O units.

A standard Network Interface (NI) connects each node or control processor to the Control Network and Data Network. This is a memory-mapped control unit; reading or writing particular memory addresses will access network control registers or trigger communication operations.

The I/O units are connected to the Control Network and Data Network in exactly the same way as the processors, using the same Network Interface. Many I/O devices require more data bandwidth than a single NI can provide; in such cases multiple NI units are ganged. For example, a HIPPI channel interface contains 6 NI units, which provide access to 6 Data Network ports. (At 20 Mbytes/sec apiece, 6 NI units provide enough bandwidth for a 100 Mbyte/sec HIPPI interface with some to spare.)

Individual I/O devices are controlled by dedicated I/O control processors (IOCP). Some I/O devices are interfaces to external buses or networks; these include interfaces to VME buses and HIPPI channels. Noteworthy features of the I/O architecture are that I/O and computation can proceed independently and in parallel, that data may be transferred between I/O devices without involving the processing nodes, and that the number of I/O devices may be increased completely independently of the number of processing nodes.

Lurking in the background is a third network, the Diagnostic Network. It can be used to isolate any hardware component and to test both the component itself and all connections to other components. The Diagnostic Network pervades the hardware system but is completely invisible to the user; indeed, it is invisible to most of the control processors. A small number of the control processors include command interfaces for the Diagnostic Network; at any given time, one of these control processors provides the System Console function.

14.3 Further Information

The following chapters discuss the CM-5 architecture in more detail. Chapter 15 contains a sketch of the user-level *virtual machine*, the programming model that is visible to a single user job. This virtual machine is supported by a combination of hardware, operating system, and run-time libraries.

In Chapter 16, *local architecture* is considered: the structure of individual processors and associated memory. This is the view seen from any single processor in the system; it is the level of architecture where program code is executed.

Chapter 17 discusses *global architecture*. This specifies how various components of the system operate together to solve a single problem. This level of architectural specification provides a framework for understanding the flow of control and the management of data in a massively parallel application.

Chapter 18 describes the *system architecture*, which addresses support of multiple user jobs, communication between jobs, I/O transfers, fault diagnosis and repair, and system administration.

Chapter 19 presents the *I/O architecture*, including the design of individual I/O devices and how they fit into the system structure.

Chapter 15

The User-Level Virtual Machine

The virtual machine provided by the hardware and operating system to a single user task consists of a control processor acting as a partition manager (PM), a set of processing nodes, and facilities for interprocessor communication. Each node is an ordinary general-purpose microprocessor capable of executing code written in C, Fortran, or assembly language. The processing nodes may also have optional vector units for high arithmetic performance.

The operating system is CMOST, a version of SunOS enhanced to manage CM-5 processor, I/O, and network resources. The PM provides full UNIX services through standard UNIX system calls. Each processing node provides a limited set of UNIX services.

A user task consists of a standard UNIX process running on the PM and a process running on each of the processing nodes. Under timesharing, all processors are scheduled *en masse*, so that all are processing the same user task at the same time. Each process of the user task, whether on the PM or on a processing node, may execute completely independently of the rest during their common time slice.

The Control Network and Data Network allow the various processes to synchronize and transfer data among themselves. The unprivileged control registers of the network interface hardware are mapped into the memory space of each user process, so that user programs on the various processors may communicate without incurring any operating system overhead.

15.1 Communications Facilities

Each process of a user task can read and write messages directly to the Control Network and the Data Network. The network used depends on the task to be performed.

The Control Network (CN) is responsible for communications patterns in which many processors may be involved in the processing of each datum. One example is broadcasting, where one processor provides a value and all other processors receive a copy. Another is reduction, where every processor provides a value and all values are combined to produce a single result. Values may be combined by summing them, finding the maximum input value, or taking the logical OR or exclusive OR of all input values; the combined result may be delivered to a single processor or to all processors. (Software provides minimum-value and logical AND operations by inverting the inputs, applying the hardware maximum-value or logical OR operation, then inverting the result.) Note that the control processor does not play a privileged role in these operations; a value may be broadcast from, or received by, the control processor or any processing node with equal facility.

The Control Network contains integer and logical arithmetic hardware for carrying out reduction operations. This hardware is distinct from the arithmetic hardware of the processing nodes; CN operations may be overlapped with arithmetic processing by the processors themselves. The arithmetic hardware of the Control Network can also compute various forms of parallel prefix operations, where every processor provides a value and receives a result; the nth result is produced by combining the first n input values. Segmented parallel prefix operations are also supported in hardware.

The Control Network provides a form of two-phase barrier synchronization (also known as "fuzzy" or "soft" barriers). A processor can indicate to the Control Network that it is ready to enter the barrier. When all processors have checked in, the Control Network relays this fact to all processors. A processor can thus overlap unrelated processing with the possible waiting period between the time it has checked in and the time it has been determined that all processors have checked in. This allows thousands of processors to guarantee the ordering of certain of their operations without ever requiring that they all be exactly synchronized at one given instant.

The Data Network is responsible for reliable, deadlock-free point-to-point transmission of tens of thousands of messages at once. Neither the senders nor the receivers of messages need be globally synchronized. At any time, any processor may send a message to any processor in the user task. This is done by

writing first the destination processor number, and then the data to be sent, to control registers in the Network Interface (NI). Once the Data Network has accepted the message, it assumes all responsibility for eventual delivery of the message to its destination. In order for a message to be delivered, the processor to which it was sent must accept the message from the Data Network. However, processor resources are not required for forwarding messages. The operation of the Data Network is independent of the processing nodes, which may carry out unrelated computations while messages are in transit.

There is no separate interface for special patterns of point-to-point communication, such as nearest neighbors within a grid. The Data Network presents a uniform interface to the software. The hardware implementation, however, has been tuned to exploit the locality found in commonly used communication patterns.

Data Network performance follows a simple model. The Data Network provides enough bandwidth for every Network Interface to sustain data transfers at 20 Mbytes/sec to any other NI within its group of 4; at 10 Mbytes/sec to any other NI within its group of 16; or at 5 Mbytes/sec to any other NI in the system. (Two Network Interfaces are in the same group of 2^k if their network addresses differ only in the k lowest-order bits.) These figures are for maximum sustained network hardware performance, which is sufficient to handle the transfer rates sustainable by node software. Note that worst-case performance is only a factor of 4 worse than best-case performance. Other network designs have much larger worst/best ratios.

To see the consequences of this performance model, consider communication within a two-dimensional grid. If, say, the processors are organized so that each group of 4 represents a 2 x 2 patch of of the grid, and each group of 16 processors represents a 4 x 4 patch of the grid, then nearest-neighbor communication can be sustained at the maximum rate of 20 Mbytes/sec per processor. For within each group of 4, 2 of the processors have neighbors in a given direction (North, East, West, South) that lie within the same group, and therefore can transmit at the maximum rate. The other 2 processors have neighbors outside the group of 4. But the Data Network provides bandwidth of 40 Mbytes/sec out of that group, enough for each of the 4 processors to achieve 10 Mbytes/sec within a group of 16. That is enough to provide 20 Mbytes/sec apiece to the remaining 2 processors. The same argument applies to the 4 processors in a group of 16 that have neighbors outside the group: not all processors have neighbors outside the group, so their outside-the-group bandwidth can be borrowed to provide maximum bandwidth to processors that do have neighbors outside the group.

There are two mechanisms for notifying a receiver that a message is available. The arrival of a message sets a status flag in a Network Interface control register; a user program can poll this flag to determine whether an incoming message is available. The arrival of a message can also optionally signal an interrupt. Interrupt handling is a privileged operation, but the operating system converts an arrived-message interrupt into a signal to the user process. Every message bears a four-bit tag; under operating system control, some tags cause message-arrival interrupts and others do not. (The operating system reserves certain of the tag numbers for its own use; the hardware signals an invalid-operation interrupt to the operating system if a user program attempts to use a reserved message tag.)

The Control Network and Data Network provide flow control autonomously. In addition, two mechanisms exist for notifying a sender that the network is temporarily clogged. Failure of the network to accept a message sets a status flag in a Network Interface control register; a user program can poll this flag to determine whether a retry is required. Failure to accept a message can also optionally signal an interrupt.

Data can also be transferred from one user task to another, or to and from I/O devices. Both kinds of transfer are managed by the operating system using a common mechanism. An intertask data transfer is simply an I/O transfer through a named UNIX pipe.

15.2 Data Parallel Computations

While the user may code arbitrary programs for the various processors and put the general capabilities of the network interface to any desired use, the CM-5 architecture is designed to support especially well the data parallel model of programming. Parallel programs are often structured as alternating phases of local computation and global communication. Local computation consists of operations by each processor on the data in its own memory. Global communication includes any transfer of data between or among processors, possibly with arithmetic or logical computation on the data as it is transferred. By managing data transfers globally and coherently rather than piecemeal, the data parallel model often realizes economies of scale, reducing the overhead of synchronization for interprocessor communication. Frequently used patterns of communication are captured in carefully tuned compiler code generators and run-time library routines; they are presented as primitive operators or intrinsic

functions in high-level languages so that the programmer need not constantly reinvent them.

The following sections discuss various aspects of the data parallel programming model and sketch the ways in which each is supported by the CM-5 architecture and communications structure.

Elemental and Conditional Computations

Elemental computations, which involve operating on corresponding elements of arrays, are purely local computations if the arrays are divided in the same way among the processors. If two such matrices are to be added together, for example, every pair of numbers to be added reside together in the memory of a single processing node, and that node takes responsibility for performing the addition.

Because each processing node executes its own instruction stream as well as processing its own local data, conditional operations are easily accommodated. For example, one processing node might contain an element on the boundary of an array while another might contain an interior element; certain filtering operations, while allowing all elements to be processed at once, require differing computations for boundary elements and interior elements. In the CM-5 data parallel architecture, some processors can take one branch of a conditional and others can take a different branch simultaneously with no loss of efficiency.

Replication

Replication consists of making copies of data. The most important special case is broadcasting, in which copies of a single item are sent to all processors. This is supported directly in hardware by the Control Network.

Another common case is spreading, in which copies of elements of a lower-dimensional array are used to fill out the additional dimensions of a high-dimensional array. For example, a column vector might be spread into a matrix, so that each element of the vector is copied to every element of the corresponding row of the matrix. This case is handled by a combination of hardware mechanisms.

If the processors are partitioned into clusters of differing size, such that the network addresses within each cluster are contiguous, then one or two

parallel-prefix operations by the Control Network can copy a value from one processor in each cluster to all others in that cluster with particular speed.

Reduction

Reduction consists of combining many data elements to produce a smaller number of results. The most important special case is global reduction, in which every processor contributes a value and a single result is produced. The operations of integer summation, finding the integer maximum, logical OR, and logical exclusive OR are supported directly in hardware by the Control Network. Floating-point reduction operations are carried out by the nodes with the help of the Control Network and Data Network.

A common operation sequence is a global reduction immediately followed by a broadcast of the resulting value. The Control Network supports this combination as a single step, carrying it out in no more time than a simple reduction.

The cases of reduction along the axes of a multidimensional array correspond to the cases of spreading into a multidimensional array and have similar solutions. The rows of a matrix might be summed, for example, to form a column matrix. This case is handled by a combination of hardware mechanisms.

If the processors are partitioned into clusters of differing size, such that the network addresses within each cluster are contiguous, then one or two parallel-prefix operations by the Control Network can reduce values from all processors within each cluster and optionally redistribute the result for that cluster to all processors in that cluster.

Permutation

The Data Network is specifically designed to handle all cases of permutation, where each input value contributes to one result and each result is simply a copy of one input value. The Data Network has a single, uniform hardware interface and a structure designed to provide especially good performance when the pattern of exchange exhibits reasonable locality. Both nearest-neighbor and nearest-but-one-neighbor communication within a grid are examples of patterns with good locality. These particular patterns also exhibit regularity, but regularity is not a requirement for good Data Network performance. The irregular polygonal tesselations of a surface or a volume that are typical of finite-element methods

lead to communications patterns that are irregular but local. The Data Network performs as well for such patterns as for regular grids.

Parallel Prefix

Parallel prefix operations embody a very specific, complex yet regular, combination of replication and reduction operations. A parallel prefix operation produces as many results as there are inputs, but each input contributes to many results and each result is produced by combining multiple inputs. Specifically, the inputs and results are linearly ordered; suppose there are n of them. Then result j is the reduction of the first j inputs; it follows that input j contributes to the last $n-j+1$ results. (For a reverse parallel prefix operation—also called a parallel suffix operation—these are reversed: result j is the reduction of the last $n-j+1$ inputs, and input j contributes to the first j results.)

The Control Network handles parallel prefix (and parallel suffix) operations directly, in the same manner and at the same speed as reduction operations, for integer and logical combining operations. The input values and the results are linearly ordered by network address.

The Control Network also directly supports segmented parallel prefix operations. If the processors are partitioned into clusters of differing size, such that the network addresses within each cluster are contiguous, then a single Control Network operation can compute a separate parallel prefix or suffix within each cluster.

More complex cases of parallel prefix operations, such as on the rows or columns of a matrix or on linked lists, are variously handled through the Control Network or Data Network in cooperation with the nodes.

Virtual Processors

Data parallel programming provides the high-level programmer with the illusion of as many processors as necessary; one programs as if there were a processor for every data element to be processed. These are often described as *virtual processors*, by analogy with conventional virtual memory, which provides the illusion of having more main memory than is physically present.

The CM-5 architecture, rather than implementing virtual processors entirely in firmware, relies primarily on software technology to support virtual processors.

CM-5 compilers for high-level data parallel languages generate control-loop code and run-time library calls to be executed by the processing nodes. This provides the same virtual-processor functionality made available by the Paris instruction set on the Connection Machine Model CM-2, but adds further opportunities for compile-time optimization.

15.3 Low-Level User Programming

Low-level programs may be written for the CM-5 in C or Fortran 77. Assembly language is also available, though C should be adequate for most low-level purposes; all hardware facilities are directly accessible to the C programmer. A special assembler allows hand-coding of individual vector instructions for the processing nodes.

One writes low-level programs as two pieces of code: one piece is executed in the control processor, and the other is replicated at program start-up and executed by each processing node. One speaks of writing a program in "C & C" (a C program for the control processor and a C program for the nodes); one may also write in "Fortran & Fortran" ("F & F") or in "C & assembler," etc.

A package of macros and run-time functions supports common communications operations within a message-passing framework (see Chapter 13). Such low-level communications access allows the user to experiment with MIMD program organizations other than data parallel, to port programs easily from other MIMD architectures, and to implement new primitives for use in high-level programs.

Chapter 16

Local Architecture

16.1 Control Processor Architecture

A control processor (CP) is essentially like a standard high-performance workstation computer. It consists of a standard RISC microprocessor, associated memory and memory interface, and perhaps I/O devices such as local disks and Ethernet connections. It also includes a CM-5 Network Interface, providing access to the Control Network and Data Network.

A control processor acting as a partition manager (PM) controls each partition and communicates with the rest of the CM-5 system through the Control Network and Data Network. For example, a PM initiates I/O by sending a request through the Data Network to a second CP, an I/O Control Processor. A PM initiates task-switching by using the Control Network to send a broadcast interrupt to all processing nodes; privileged operating-system support code in each node then carries out the bulk of the work. To access the Control Network and Data Network, each CP uses its Network Interface, a memory-mapped device in the memory address space of its microprocessor.

The microprocessor supports the customary distinction between user and supervisor code. User code can run in the control processor at the same time that user code for the same job is running in the processing nodes. Protection of the supervisor, and of one user from another, is supported by the same mechanisms used in workstations and single-processor time-shared computers, namely memory address mapping and protection and the suppression of privileged operations in user mode. In particular, the operating system prevents a user process from performing privileged Network Interface operations; the privileged control registers simply are not mapped into the user address space.

The initial implementation of the CM-5 control processor uses a SPARC microprocessor. However, it is expected that, over time, the implementation of the CP will track the RISC microprocessor technology curve to provide the best possible functionality and performance at any given point in time; therefore it is recommended that low-level programming be carried out in C as much as possible, rather than in assembly language.

16.2 Processing Node Architecture

The CM-5 Processing Node is designed to deliver very good cost-performance when used in large numbers for data parallel applications. Like the control processor, the node makes use of industry-standard RISC microprocessor technology. This microprocessor may optionally be augmented with a special high-performance hardware arithmetic accelerator that uses wide datapaths, deep pipelines, and large register files to improve peak computational performance.

The node design is centered around a standard 64-bit bus. To this node bus are attached a RISC microprocessor, a CM-5 Network Interface, and memory. Note that all logical connections to the rest of the system pass through the Network Interface.

The node memory consists of standard DRAM chips and a 2 Kbyte boot ROM; the microprocessor also has a 64 Kbyte cache that holds both instructions and data. All DRAM memory is protected by ECC checking, which corrects single-bit failure and detects two-bit errors and DRAM chip failures. The boot ROM contains code to be executed following a system reset, including local processor and memory verification and the communications code needed to download further diagnostics or operating system code.

The memory configuration depends on whether the optional high-performance arithmetic hardware is included. Without the arithmetic hardware, the memory is connected by a 72-bit path (64 data bits plus 8 ECC bits) to a memory controller that in turn is attached to the node bus. (See Figure 26.) In this configuration the memory size can be 8, 16, or 32 Mbytes. (This assumes 4-Mbit DRAM technology. Future improvements in DRAM technology will permit increases in memory size. The CM-5 architecture and chip implementations anticipate these future improvements.)

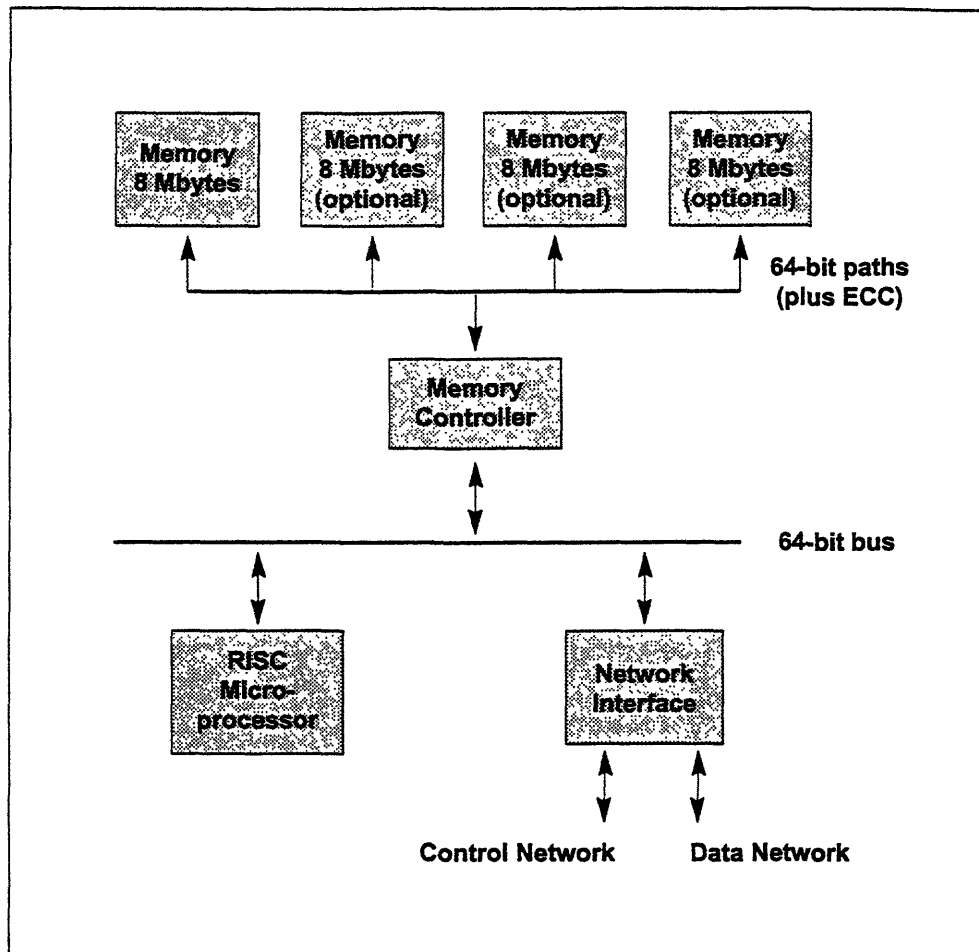


Figure 26. Processing node.

The basic CM-5 processing node consists of a RISC microprocessor, memory subsystem, and a CM-5 Network Interface all connected to a standard 64-bit bus. The RISC microprocessor is responsible for instruction fetch, instruction execution, processing data, and controlling the Network Interface. The memory subsystem consists of a memory controller and either 8 Mbytes, 16 Mbytes, or 32 Mbytes of DRAM memory. The path from each memory back to the memory controller is 72 bits wide, consisting of 64 data bits and 8 bits of ECC code. The ECC circuits in the memory controller can correct single-bit errors and detect double-bit errors as well as failure of any single DRAM chip. The Network Interface connects the node to the rest of the system through the Control Network and Data Network.

If the high-performance arithmetic hardware is included, then the node memory is divided into four independent banks, each with a 72-bit (64 data bits plus 8 ECC bits) access path. (See Figure 27.)

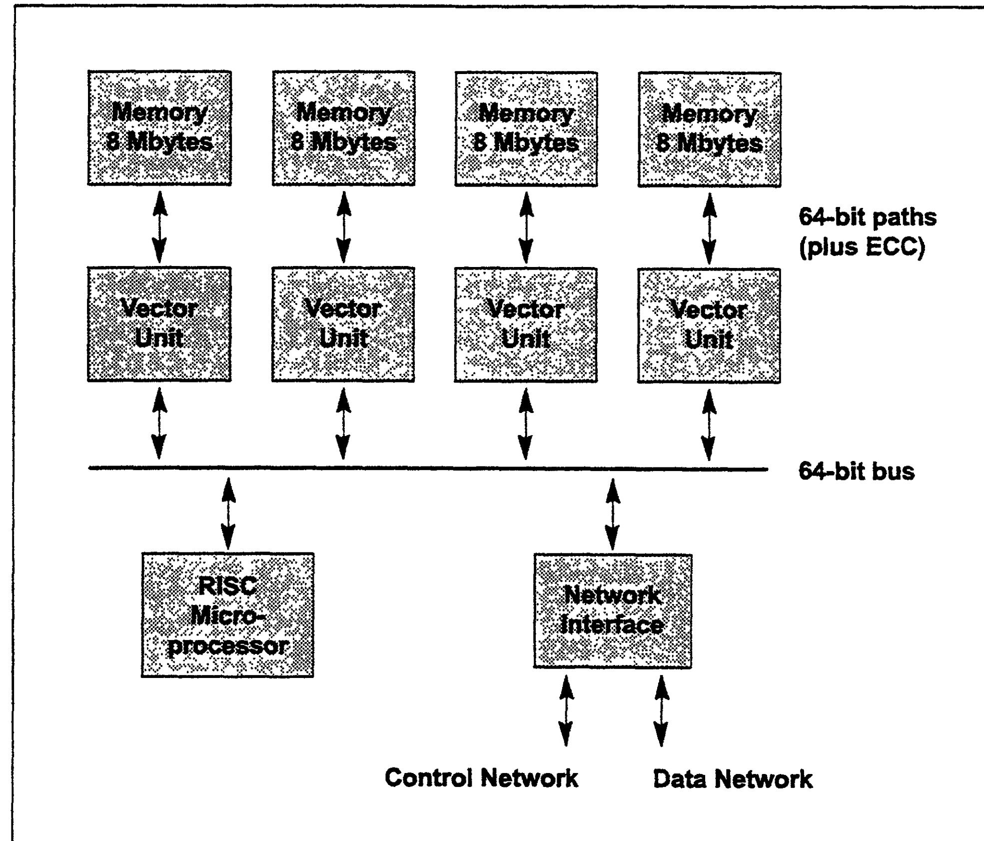


Figure 27. Processing node with vector units.

A CM-5 processing node may optionally contain an arithmetic accelerator. In this configuration the node has a full 32 Mbytes of memory, four banks of 8 Mbytes each. The memory controller is replaced by four vector units. Each vector unit has a dedicated 72-bit path to its associated memory bank, providing peak memory bandwidth of 128 Mbytes/sec per vector unit, and performs all the functions of a memory controller, including generation and checking of ECC bits. Each vector unit has 32 Mflops peak 64-bit floating-point performance and 32 Mops peak 64-bit integer performance. The vector units execute vector instructions issued to them by the RISC microprocessor. Each vector instruction may be issued to a specific vector unit (or pair of units), or broadcast to all four vector units at once. The microprocessor takes care of such "housekeeping" computations as address calculation and loop control, overlapping them with vector instruction execution. Together, the vector units provide 512 Mbytes/sec memory bandwidth and 128 Mflops peak 64-bit floating-point performance. A single CM-5 node with vector units is a supercomputer in itself.

The special arithmetic hardware consists of four vector units (VU), one for each memory bank, connected separately to the node bus. (See Figure 28.) In this configuration the memory size is 8 Mbytes per VU for a total of 32 Mbytes per node. (Again, this figure assumes 4-Mbit DRAM technology and will increase as industry-standard memories are improved.) Each VU also implements all memory

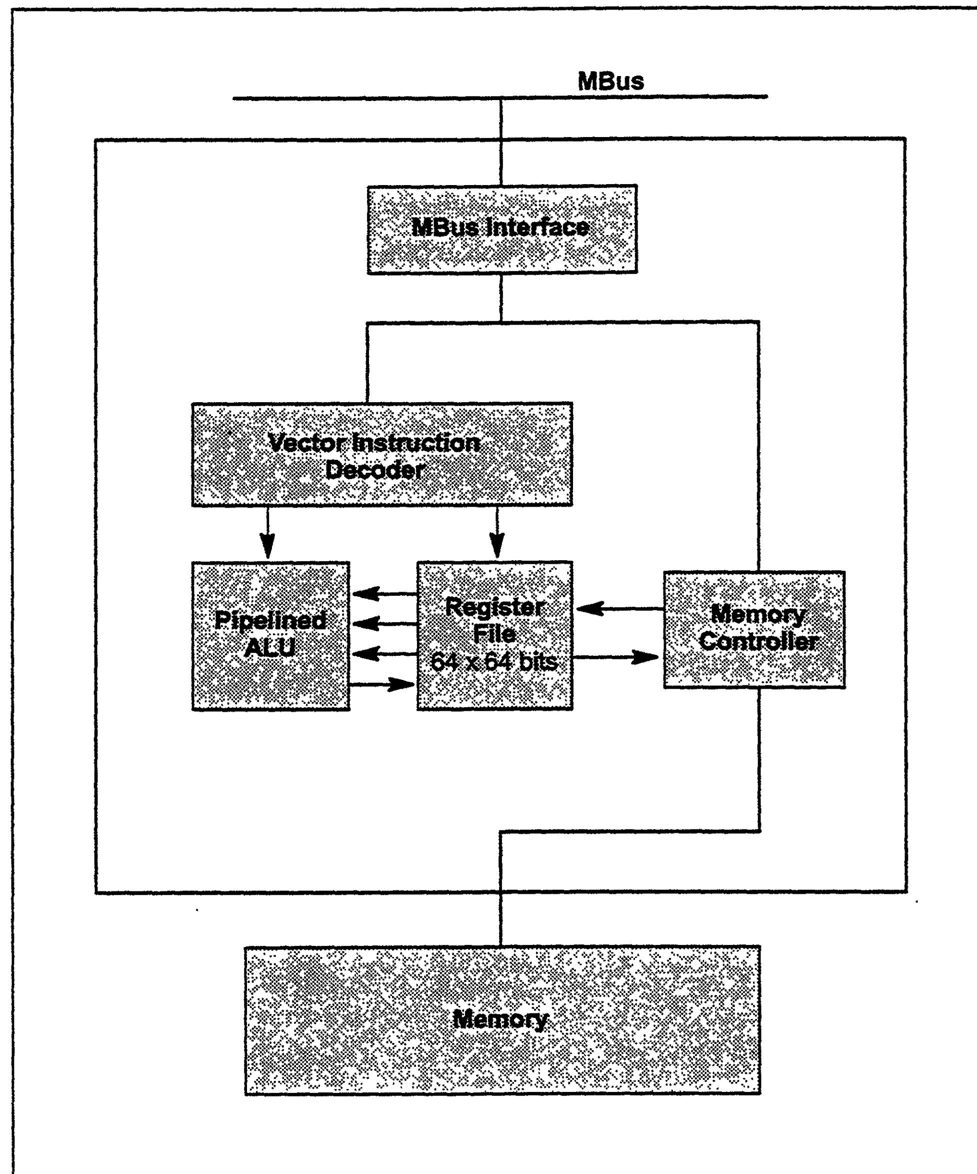


Figure 28. Vector unit functional architecture.

controller functions, including ECC checking, so that the entire memory appears to be in the address space of the microprocessor exactly as if the arithmetic hardware were not present.

The memory controller or vector unit also provides a word-based interface to the system Diagnostics Network (see Section 18.7). This provides an extra communications path to the node; it is designed to be slow but reliable and is used primarily for hardware fault diagnosis.

As with the control processors, the implementation of the CM-5 processing node is expected to track the RISC microprocessor technology curve to provide the best possible functionality and performance at any given point in time; therefore it is recommended that low-level programming be carried out in C as much as possible, rather than in assembly language. The initial implementation of the CM-5 node uses a SPARC microprocessor.

16.3 Vector Unit Architecture

Each vector unit (VU) is a memory controller and computational engine controlled by a memory-mapped control-register interface. When a read or write operation on the node bus addresses a VU, the memory address is further decoded. High-order bits indicate the operation type:

- For an ordinary *memory transaction*, the low-order address bits indicate a location in the memory bank associated with the VU, which acts as a memory controller and performs the requested memory read or write operation.
- For a *control register access*, the low-order address bits indicate a control register to be read or written.
- For a *data register access*, the low-order address bits indicate a data register to be read or written.
- For a *vector-unit instruction*, the node memory bus operation must be `write` (an attempt to `read` from this part of the address space results in a bus error). The data on the memory bus is not written to memory but is interpreted as an instruction to be executed by the vector execution portion of the VU. The low-order address bits indicate a location in the memory bank associated with the VU; the instruction will use this address if it in-

cludes operations on memory. A vector-unit instruction may be addressed to any single VU (in which case the other three VUs will ignore it), to a pair of VUs, or to all four VUs simultaneously.

The first two types of operation are identical to those performed by the memory controller when vector units are absent. The third type permits the microprocessor to read or write the register file of any vector unit. The fourth type of operation initiates high-performance arithmetic computation. This computation has both vector and parallel characteristics: each VU can perform vector operations, and a single instruction may be issued simultaneously to all four. If the vector length is 16, then issuing a single instruction can result in as many as 64 individual arithmetic operations (or 128 if the instruction specifies a compound operation such as `multiply-add`).

Vector units cannot fetch their own instructions; they merely react to instructions issued to them by the microprocessor. The instruction format, instruction set, and maximum vector length have been chosen so that the microprocessor can keep the vector units busy while having time of its own to fetch instructions (both its own and those for the vector units), calculate addresses, execute loop and branch instructions, and carry out other algorithmic bookkeeping.

Each vector unit has 64 64-bit registers, which can also be addressed as 128 32-bit registers. There are some other control registers as well, most notably the 16-bit Vector Mask (VM) and the 4-bit Vector Length (VL) registers. The Vector Mask register controls certain conditional operations and optionally receives single-bit status results for each vector element processed. The Vector Length register specifies the number of elements to be processed by each vector instruction.

The vector unit actually processes both vector and scalar instructions; a scalar-mode instruction is handled as if it were a vector-mode instruction of length 1. Thus scalar-mode instructions always operate on single registers; vector-mode instructions operate on sequences of registers. Each register operand is specified by a 7-bit starting register number and a 7-bit stride. The first element for that vector operand is taken from the starting register; thereafter the register number is incremented by the stride to produce a new register number indicating the next element to be processed. Using a large stride has the same effect as using a negative stride, so it is possible to process a vector in reverse order. Most instruction formats use a default stride of 1 for 32-bit operands or 2 for 64-bit operands, so as to process successive registers, but one instruction format allows arbitrary strides to be specified for all operands, and another allows one vector operand to take its elements from an arbitrary pattern of registers by means of a mechanism for indirect addressing of the register file.

Each vector unit includes an adder, a multiplier, memory load/store, indirect register addressing, indirect memory addressing, and population count. Every vector-unit instruction can specify at least one arithmetic operation and an independent memory operation. Every instruction also has four register-address fields: three for the arithmetic operation and one for the memory operation. All binary arithmetic operations are fully three-address; an addition, for example, can read two source registers and write into a third destination register. The memory operation can address a completely independent register. If, however, a load operation addresses a register that is also a source for the arithmetic operation, then load-chaining occurs, so that the loaded memory data is used as an arithmetic operand in the same instruction. Indirect memory addressing supports scatter/gather operations and vectorized pointer indirection.

Two mechanisms provide for conditional processing of vector elements within each processing node. Each vector unit contains a vector mask register; vector elements are not processed in positions where the corresponding vector mask bit is zero. Alternatively, a vector-mask enumeration mechanism may be used in conjunction with the scatter/gather facility to pack vector elements that require similar processing; after bulk application of unconditional vector operations, the results are then unpacked and scattered to their originally intended destinations.

Vector-unit instructions come in five formats. (See Figure 29.) The 32-bit short format allows many common scalar and vector operations to be expressed succinctly. The four 64-bit long formats extend the basic 32-bit format to allow additional information to be specified: a 32-bit immediate operand, a signed memory stride, a set of register strides, or additional control fields (some of which can update certain control registers with no additional overhead).

The short format includes an arithmetic opcode (8 bits), a load/store opcode (3 bits), a vector/scalar mode specifier (2 bits), and four register fields called rLS, rD, rS1, and rS2 that designate the starting registers for the load/store operation and for the arithmetic destination, first source, and second source, respectively. The vector/scalar specifier indicates whether the instruction is to be executed once (scalar mode) or many times (vector mode). It also dictates the expansion of the 4-bit register specifiers into full 7-bit register addresses. The short format is designed to support a conventional division of the uniform register file into vector registers of length 16, 8, or (for 64-bit operands only) 4, with scalar quantities kept in the first 16 registers. For a scalar-mode instruction, the 4-bit register field provides the low-order bits of the register number (which is then multiplied by 2 for 64-bit operands); for a vector-mode instruction, it provides the high-order bits of the register number. The rS1 field is 7 bits wide; in some cases these specify a full 7-bit register number for arithmetic source 1 and in other cases 4 bits specify a vector register and the other 3 bits convey stride information.

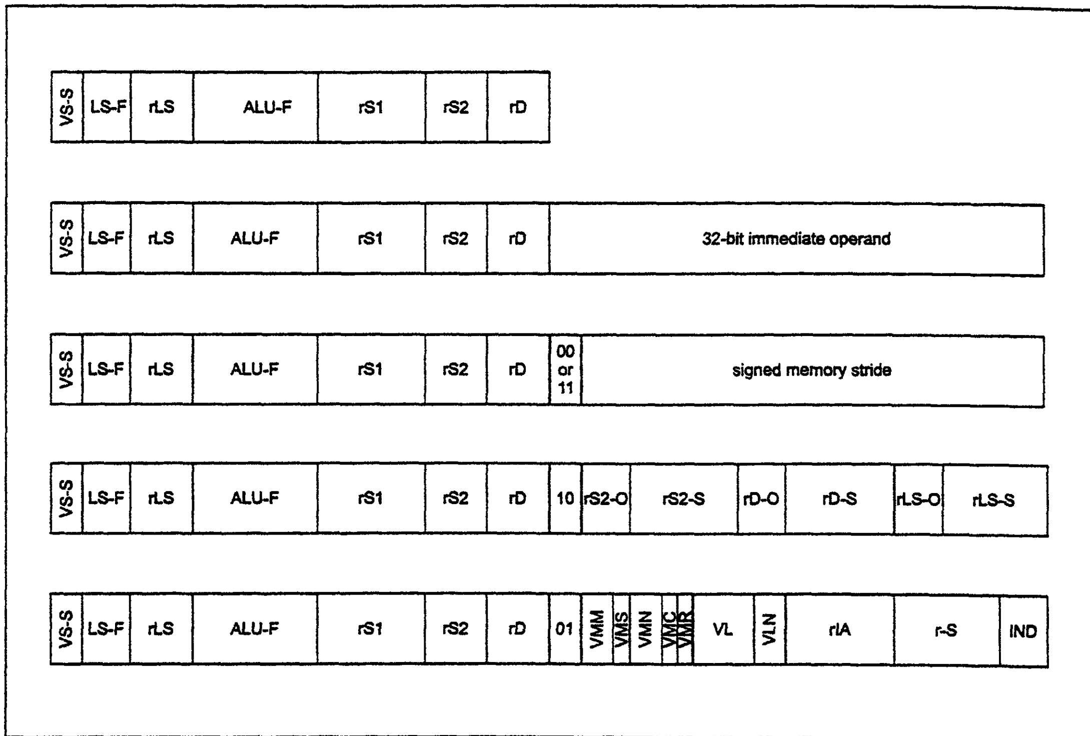


Figure 29. Vector unit instruction formats.

Each instruction issued by the RISC microprocessor to the vector units is 32 bits or 64 bits wide. The 32-bit format is designed to cover the operations and register access patterns most likely to arise in high-performance compiled code. The 32 high-order bits of the 64-bit format are identical to the 32-bit format. The 32 low-order bits provide an immediate operand, a signed memory stride, or specifications for more complex or less frequent operations.

A short scalar-mode instruction can therefore access the first 16 32-bit or 64-bit elements of the register file, simultaneously performing an arithmetic operation and loading or storing a register. (The memory address that accompanies the issued instruction indicates the memory location to be accessed.) One of the arithmetic operands (S1) may be in any of the 128 registers in the register file.

A short vector-mode instruction can conveniently treat the register file as a set of vector registers:

16	4	x	64-bit vector registers
8	8	x	64-bit vector registers
4	16	x	64-bit vector registers
16	8	x	32-bit vector registers
8	16	x	32-bit vector registers

Many options are available for vector-mode instructions. These include a choice between a default memory stride and the last explicitly specified memory stride, as well as a choice of register stride for the S1 operand (last specified, 1, or 0 — stride 0 treats the S1 operand as a scalar to be combined with every element of a vector).

The long instruction formats are all compatible extensions of the short format: the most significant 32 bits of a 64-bit instruction are decoded as a 32-bit instruction, and the least significant 32 bits specify additional operations or operands. If the rS2 field of a long instruction is zero, then the low-order 32 bits of the instruction constitute an immediate scalar value to be used as the S2 operand. If the arithmetic operation requires a 64-bit operand, then the immediate value is zero-extended left if an unsigned integer is required, sign-extended left for a signed integer, or zero-extended right for a floating-point number.

If the rS2 field of a long instruction is not zero, then the two high-order bits of the low 32 are decoded. If the two bits match, then the low-order 32 bits are an explicit signed memory stride. (Note that it is possible to specify such a stride even in a scalar-mode long instruction, in order to latch the stride in preparation for a following vector-mode instruction that might need to use another of the long formats.) Code 01 indicates additional register number and register stride information, allowing specification of complete 7-bit register numbers and register strides for the rLS, rD, and rS2 operands. This enables complex regular patterns of register access. Code 10 indicates a variety of control fields for such mechanisms as changing the Vector Length, controlling use of the Vector Mask, indirect addressing, S1 operand register striding, and population count.

The arithmetic operations that can be specified by the ALU-F instruction field are summarized in Table 1. Note the large set of three-operand multiply-add instructions. These come in three different addressing patterns: accumulative, which adds a product into a destination register (useful for dot products); inverted, which multiplies the destination by one source and then adds in the other (useful for polynomial evaluation and integer subscript computations); and full triadic, which takes one operand from the load/store register so that the destination register may be distinct from all three sources. The triadic multiply-add operations are provided for signed and unsigned integers as well as for floating-point operands, in both 32-bit and 641–bit sizes. Unsigned 64-bit multiply-boolean operations are also provided. (Note that multiplying by a power of two has the effect of a shift.)

Table 1. Summary of vector unit arithmetic instructions (Part I).

imove	dimove	umove	dumove	fmove	dfmove	Move: D = S1 + 0
itest	ditest	utest	dutest	ftest	dftest	Move and generate status
icmp	dicmp	ucmp	ducmp	fcmp	dfcmp	Compare
iadd	diadd	uadd	duadd	fadd	dfadd	Add
isub	disub	usub	dusub	fsub	dfsub	Subtract
isubr	disubr	usubr	dusubr	fsubr	dfsubr	Subtract reversed
imul	dimul	umul	dumul	fmul	dfmul	Multiply (low 64 bits for integers)
	dimulh		dumulh			Integer multiply (high 64 bits)
				fdiv	dfdiv	Divide
				finv	dfinv	Invert: D = 1.0/S1
				fsqrt	dfsqrt	Square root
				fisqt	dfisqt	Inverse square root: D = 1.0/SQRT(S2)
ineg	dineg			fneg	dfneg	Negate
iabs	diabs			fabs	dfabs	Absolute value
iaddc	diaddc	uaddc	duaddc			Integer add with carry
isubc	disubc	usubc	dusubc			Integer subtract with borrow
isbrc	disbrc	usbrc	dusbrc			Integer subtract reversed with borrow
		ushl	dushl			Integer shift left
		ushlr	dushlr			Integer shift left reversed
		ushr	dushr			Integer shift right logical
		ushrr	dushrr			Integer shift right logical reversed
ishr	dishr					Integer shift right arithmetic
ishrr	dishrr					Integer shift right arithmetic reversed
		uand	duand			Bitwise logical AND
		uandc	duandc			Bitwise logical AND with Complement
		unand	dunand			Bitwise logical NAND
		uor	duor			Bitwise logical OR
		unor	dunor			Bitwise logical NOR
		uxor	duxor			Bitwise logical XOR
		unot	dunot			Bitwise logical NOT
		umrg	dumrg			Merge: D = (if mask then S2 else S1)
		uffb	duffb			Find first 1-bit

Table 1. Summary of vector unit arithmetic instructions (Part II).

imada	dimada	umada	dumada	fmada	dfmada	rD = (rS1 * rS2) + rD
imsba	dimsba	umsba	dumsba	fmsba	dfmsba	rD = (rS1 * rS2) – rD
imsra	dimsra	umsra	dumsra	fmsra	dfmsra	rD = – (rS1 * rS2) + rD
inmaa	dinmaa	unmaa	dunmaa	fnmaa	dfnmaa	rD = – (rS1 * rS2) – rD
imadi	dimadi	umadi	dumadi	fmadi	dfmadi	rD = (rS2 * rD) + rS1
imsbi	dimsbi	umsbi	dumsbi	fmsbi	dfmsbi	rD = (rS2 * rD) – rS1
imsri	dimsri	umsri	dumsri	fmsri	dfmsri	rD = – (rS2 * rD) + rS1
inmai	dinmai	unmai	dunmai	fnmai	dfnmai	rD = – (rS2 * rD) – rS1
imadt	dimadt	umadt	dumadt	fmadt	dfmadt	rD = (rS1 * rLS) + rS2
imsbt	dimsbt	umsbt	dumsbt	fmsbt	dfmsbt	rD = (rS1 * rLS) – rS2
imsrt	dimsrt	umsrt	dumsrt	fmsrt	dfmsrt	rD = – (rS1 * rLS) + rS2
inmat	dinmat	unmat	dunmat	fnmat	dfnmat	rD = – (rS1 * rLS) – rS2
			dumsa			rD = lower(rS1 * rS2) AND rD
			dumhsa			rD = upper(rS1 * rS2) AND rD
			dumma			rD = lower(rS1 * rS2) AND NOT rD
			dumhma			rD = upper(rS1 * rS2) AND NOT rD
			dumoa			rD = lower(rS1 * rS2) OR rD
			dumhoa			rD = upper(rS1 * rS2) OR rD
			dumxa			rD = lower(rS1 * rS2) XOR rD
			dumhxa			rD = upper(rS1 * rS2) XOR rD
			dumsi			rD = lower(rS2 * rD) AND rS1
			dumhsi			rD = upper(rS2 * rD) AND rS1
			dummi			rD = lower(rS2 * rD) AND NOT rS1
			dumhmi			rD = upper(rS2 * rD) AND NOT rS1
			dumoi			rD = lower(rS2 * rD) OR rS1
			dumhoi			rD = upper(rS2 * rD) OR rS1
			dumxi			rD = lower(rS2 * rD) XOR rS1
			dumhxi			rD = upper(rS2 * rD) XOR rS1
			dumst			rD = upper(rS1 * rLS) AND rS2
			dumhst			rD = upper(rS1 * rLS) AND rS2
			dummt			rD = lower(rS1 * rLS) AND NOT rS2
			dumhmt			rD = upper(rS1 * rLS) AND NOT rS2
			dumot			rD = lower(rS1 * rLS) OR rS2
			dumhot			rD = upper(rS1 * rLS) OR rS2
			dumxt			rD = lower(rS1 * rLS) XOR rS2
			dumhxt			rD = upper(rS1 * rLS) XOR rS2

Table 1. Summary of vector unit arithmetic instructions (Part III).

			fclas	dfclas	Classify operand
			fexp	dfexp	Extract exponent
			fmant	dfmant	Extract mantissa with hidden bit
	uenc	duenc			Make float from exponent (S1) and mantissa (S2)
			fnop		No arithmetic operation
cvtfi					Convert integer to float*
cvtf					Convert float to float*
cvtir					Convert float to integer (round)*
cvti					Convert float to integer (truncate)*
trap					Generate debug trap
etrap					Generate trap on enabled exception
ldvm					Load vector mask
stvm					Store vector mask

* The rS2 field encodes the source and result sizes and formats for these instructions.

The LS-F instruction field specifies one of 5 load/store operations:

- no operation
- 32-bit load
- 64-bit load
- 32-bit store
- 64-bit store

The load/store size (32 or 64 bits) need not be the same as the arithmetic operand size. They should be the same, however, if load chaining is used. There is no distinction between integer and floating-point loads and stores. A 64-bit load or store may be used to load or store an even-odd 32-bit register pair.

Executing Vector Code

All instruction fetching and control decisions for the vector units are made by the node microprocessor. When vector units are present, all instructions and data reside in the memory banks associated with the vector units. A portion of each memory bank is conventionally reserved for instruction and data areas for the

microprocessor. The memory management hardware of the microprocessor is used to map pages from the four memory banks so as to make them appear contiguous to the microprocessor.

While the microprocessor does not have its own memory, it does have a local cache that is used for both instruction and data references. Thus, the microprocessor and vector units can execute concurrently so long as no cache misses occur.

When a cache block must be fetched from memory, the associated vector unit may be in one of three states. If it is not performing any local operations, then the cache block is fetched immediately. If it is performing a local load or store operation, then the block fetch is delayed until the operation completes. If the vector unit is doing an operation that does not require the memory bus, then the block fetch proceeds immediately, concurrently with the executing vector operation.

The microprocessor issues VU instructions by storing to a specially constructed address: the microprocessor fetches the instruction itself from its data memory, calculates the special vector-unit destination address for issuing the instruction, and executes the store. The time it takes the microprocessor to do this is generally less than the time it takes a vector unit to execute an instruction with a vector length of 4. Moreover, the tail end of one vector instruction may be overlapped in time with the beginning of the next, thus eliminating memory latency and vector start-up overhead. With careful programming, therefore, the microprocessor can sustain delivery of vector instructions so as to keep the vector units continuously busy.

The vector unit is optimally suited for a vector length of 8; with vectors this long, the timing requirements are not so critical, and the microprocessor has time to spare for bookkeeping operations. The short vector-unit instruction format supports addressing of length-8 register blocks for either 32-bit or 64-bit operands. This provides 8 vector registers for 64-bit elements or 16 vector registers for 32-bit elements, with the first two such register blocks also addressable as 16 scalar registers. This is only a conventional arrangement, however; long-format instructions can address the registers in arbitrary patterns.

Flow control of instructions to the vector units is managed using the hardware protocol of the node bus. When a vector instruction is issued by the microprocessor, any addressed vector unit may stall the bus if it is busy. A small write buffer and independent bus controller within the microprocessor allows it to continue local execution of its own instructions while the bus is stalled by a vector unit. If the microprocessor gets far enough ahead, the small write buffer becomes full, causing the microprocessor to stall until the vector unit(s) catch up.

Each vector instruction either completes successfully or terminates in a hard error condition. Exceptions and other non-fatal conditions are signaled in sticky status registers that may be either polled or enabled to signal interrupts. Hard errors and enabled exception conditions are signaled to the microprocessor as interrupts via the Network Interface.

The memory addresses on the node bus are physical addresses resulting from memory-map translation in the microprocessor. The memory map provides the necessary protection to ensure that the addressed location itself is in fact within a user's permitted address space, but cannot prevent accesses to other locations by execution of vector instructions that use indirect addressing or memory strides. Additional protection is provided in each vector unit by bounds-checking hardware that signals an interrupt if specified physical address bounds are exceeded.

Certain privileged vector unit operations are reserved for supervisor use. These include the interrupt management and memory management features. The supervisor can interrupt a user task at any time for task-switching purposes and can save the state of each vector unit for transparent restoration at a later time.

Chapter 17

Global Architecture

A single user process (as shown in Chapter 15) "views" the CM-5 system as a set of processing nodes plus a partition manager, with I/O and other extra-partitional activities being provided by the operating system.

To support such processes, however, requires that the underlying system software make appropriate use of the global architecture provided by the CM-5's communications networks.

All the computational and I/O components of a CM-5 system interact through two networks, the Control Network and the Data Network. Every such component is connected through a standard CM-5 Network Interface. The NI presents a simple, synchronous 64-bit bus interface to a node or I/O processor, decoupling it both logically and electrically from the details of the network implementation.

The Control Network supports communication patterns that may involve all the processors in a single operation; these include broadcasting, reduction, parallel prefix, synchronization, and error signalling. The Data Network supports point-to-point communications among the processors, with many independent messages in transit at once.

17.1 The Network Interface

The CM-5 Network Interface provides a memory-mapped control-register interface to a 64-bit processor memory bus. All network operations are initiated by writing data to specific addresses in the bus address space.

Many of the control registers appear to be at more than one location in the physical address space. When a control register is accessed, additional information is

conveyed by the choice of which of its physical addresses was used for the access; in other words, information is encoded in the address bits. For example, when the control network is to be used for a combining operation, the first—and perhaps only—bus transaction writes the data to be combined, and the choice of address indicates which combining operation is to be used. One of the address bits indicates whether the access has supervisor privileges; an error is signalled on an attempt to perform a privileged access using an unprivileged address. (Normally the operating system maps the unprivileged addresses into the address space of the user process, thereby giving the user program zero-overhead access to the network hardware while prohibiting user access to privileged features.)

The logical interface is divided into a number of functional units. Each functional unit presents two FIFO interfaces, one for outgoing data and one for incoming data. A processor writes messages to the outgoing FIFO and pulls messages from the incoming FIFO, using the same basic protocol for each functional unit. Different functional units, however, respond in different ways to these messages. For example, a Data Network unit treats the first 32 bits of a message as a destination address to which to send the remainder of the message; a Control Network combining unit forwards the message to be summed (or otherwise combined) with similar messages from all the other processors.

Data is kept in each FIFO in 32-bit chunks. The memory-bus interface accepts both 32-bit and 64-bit bus transactions. Writing 64 bits thus pushes two 32-bit chunks onto an output FIFO; reading 64 bits pulls two chunks from an input FIFO.

For outgoing data, there are two control registers called **send** and **send_first**. Writing data to the **send_first** register initiates an outgoing message; address bits encode the intended total length of the message (measured in 32-bit chunks). Any additional data for that message is then written to the **send** register. After all the data for that message has been written, the program can test the **send_ok** bit in a third control register. If the bit is 1, then the network has accepted the message and bears all further responsibility for handling it. If the bit is 0, then the data was not accepted (because the FIFO overflowed) and the entire message must be re-pushed into the FIFO at a later time. The **send_space** control register may be checked before starting a message to see whether there is enough space in the FIFO to hold the entire message; this should be treated only as a hint, however, because supervisor operations (such as task switching) might invalidate it. In many situations throughput is improved by pushing without checking first, in the expectation that the FIFO will empty out as fast as new data is being pushed. It is also permissible to check the **send_ok** bit before all the data words for the message have been pushed; if it is 0, the message may be retried immediately.

For incoming data, a processor can poll the **`receive_ok`** bit until it becomes 1, indicating that a message has arrived; alternatively, it can request that certain types of messages trigger an interrupt on arrival. In either case, the program can then check the **`receive_length_left`** field to find out how long the message is and then read the appropriate number of data words from the **`receive`** control register.

The supervisor can always interrupt a user program and send its own message; this is done by deleting any partial user message, sending the supervisor message, and then forcing the **`send_ok`** bit for that unit to 0 before resuming the user program. To the user program it merely appears that the FIFO was temporarily full; the user program should then retry sending the message. The supervisor can also lock a send-FIFO, in which case it appears always to be full, or disable it, in which case user access will cause an interrupt. The supervisor can save and transparently restore the contents of any receive-FIFO.

Each Network Interface records interrupt signals and error conditions generated within its associated processor; exchanges error and interrupt information with the Control Network; and forwards interrupt and reset signals to its associated processor.

17.2 The Control Network

Each Network Interface contains an assortment of functional units associated with the Control Network. All have the same dual-FIFO organization but differ in detailed function.

Every Control Network operation potentially involves every processor. A processor may push a message into one of its functional units at any time; shortly after all processors have pushed messages, the result becomes available to all processors. Messages of each type may be pipelined; a number of messages may be sent before any results are received and removed. (The exact depth of the pipeline varies from one functional unit to another.) The general idea is that every processor should send the same kinds of messages in the same order. The Control Network, however, makes no restrictions about when each processor sends or receives messages. In other words, processors need not be exactly synchronized to the Control Network; rather, the Control Network is the very means by which processors conduct synchronized communication *en masse*.

There are exceptions to the rule that every processor must participate. The functional units contain mode bits for *abstaining*. A processor may set the appropriate mode bit in its Network Interface in order to abstain from a particular type of operation; each operation of that type will then proceed without input from that processor or without delivering a result to that processor. A *participating* processor is one that is not abstaining from a particular kind of Control Network operation.

Broadcasting

The *broadcast* unit handles broadcasting operations. There are actually three distinct broadcasting units: one for user broadcast, one for supervisor broadcast, and one for interrupt broadcast. Access to the supervisor broadcast unit or interrupt broadcast unit is a privileged operation.

Only one processor may broadcast at a time. If another processor attempts to send a broadcast message before completion of a previous broadcast operation, the Control Network signals an error.

A broadcast message is one to fifteen 32-bit words long. Shortly after a message is pushed into the broadcast send-FIFO, copies of the message are delivered to all participating processors. The user broadcast and supervisor broadcast units are identical in function except that the latter is reserved for supervisor use.

An interrupt broadcast message causes every processor to receive an interrupt or reset signal. A processor can abstain from receiving interrupts, in which case it ignores interrupt messages when it receives them; but a processor cannot abstain from a reset signal (which causes the receiving NI and its associated processor to be reset).

As an example of the use of broadcast interrupts, consider a partition manager coordinating the task-switching of user processes. When it is time to switch tasks, the PM uses the Control Network to send a broadcast interrupt to all nodes in the partition. This transfers control in each node to supervisor code, which can then read additional supervisor broadcast information about the task-switch operation (such as which task is up next).

Combining

The *combine* unit handles reduction and parallel prefix operations. A combine message is 32 to 128 bits long and is treated as a single integer value. There are four possible message types: reduction, parallel prefix, parallel suffix, and router-done. The router-done operation is simply a specialized logical OR reduction that assists the processors in a protocol to determine whether Data Network communications are complete. Reduction, parallel prefix, and parallel suffix may combine the messages using any one of five operators: bitwise logical OR, bitwise logical XOR, signed maximum, signed integer addition, and unsigned integer addition. (The only difference between signed and unsigned addition is in the reporting of overflow.) The message type and desired combining operation are encoded by address bits when writing the destination address to the **`send_first`** register.

As an example, every processor might write a 64-bit integer to the combine interface, specifying signed integer addition reduction. Shortly after the last participating processors write their input values, the signed sum is delivered to every participating processor, along with an indication of whether overflow occurred at any intermediate step.

As another example, every processor might write a 32-bit integer to the combine interface, specifying signed maximum parallel prefix. Shortly after the last participating processors write their input values, every participating processor receives the largest among the values provided by itself and all lower-numbered processors.

The combine interface also supports segmented parallel prefix (and parallel suffix) operations. Each combine unit contains a **`scan_start`** flag; when this flag is 1, that NI is considered to begin a new segment for purposes of parallel prefix operations. Such an NI will always receive the very value that was pushed.

Every participating processor must specify the same message type and combining operation. If, in the course of processing combine requests in order, the Control Network encounters different combine requests at the same time, it signals an error.

Global Operations

Global bit operations produce the logical OR of one bit from every participating processor. There are three independent global operation units, one synchronous

and two asynchronous, which may be used completely independently of each other and of other Control Network functions. This makes them useful for signalling conditions and exceptions.

The synchronous global unit is similar to the combine unit except that the operation is always logical OR reduction and each message consists of a single bit. Processors may provide their values at any time; shortly after the last participating processors have written their input bits, the logical OR is delivered as a single-bit message to every participating processor.

Each asynchronous global unit produces a new value any time the value of any input is changed. Input values are continually transported, combined, and delivered throughout the Control Network without waiting for all processors to participate. Processors may alter their input bits at any time. These units are best used to detect the transition from 0 to 1 in any processor or to detect the transition from 1 to 0 in all processors. (The NI will signal an interrupt, if enabled, whenever a transition from 0 to 1 is observed.)

There are two asynchronous global units, one for the user and one for the supervisor. Access to the supervisor asynchronous global unit is a privileged operation.

Synchronization

Both the synchronous global unit and the combine unit may be used to implement barrier synchronization: if every processor writes a message and then waits for the result, no processor will pass the barrier until every processor has reached the barrier. The hardware implementation of this function provides extremely rapid synchronization of thousands of processors at once. Note that the router-done combine operation is designed specifically to support barrier synchronization during a Data Network operation, so that no processor abandons its effort to receive messages until all processors have indicated that they are no longer sending messages.

Flushing the Control Network

There is a special functional unit for clearing the intermediate state of combine messages, which may be required if an error or task switch occurs in the middle of a combine operation. A flush message behaves very much like a broadcast message: shortly after one processor has sent such a message, all processors are

notified that the flush operation has completed. Access to the flush functional unit is a privileged operation.

Error Handling

The Control Network is responsible for detecting certain kinds of communications errors, such as an attempt to specify different combining operations at the same time. More important, it is responsible for distributing error signals throughout the system. Hard error signals are collected from the Data Network and all Network Interfaces; these error signals are combined by logical OR operations and the result is redistributed to every Network Interface.

17.3 The Data Network

Each Network Interface contains one Data Network functional unit. The first 32-bit chunk of a message is treated as a destination address; it must be followed by one to five additional 32-bit chunks of data. This data is sent through the Data Network and delivered to the receive-FIFO of the Network Interface at the specified destination address. Each message also bears a 4-bit tag, which is encoded by address bits when writing the destination address to the **send_first** register. The tag provides a cheap way to differentiate among a small number of message types. The supervisor can reserve certain tags for its own use; any attempt by the user to send a message with a reserved tag signals an error. The supervisor also controls a 16-bit interrupt mask register; when a message arrives, an interrupt is signalled to the destination processor if the mask bit corresponding to the message's tag value is 1.

A destination address may be physical or relative. A physical address specifies a particular Network Interface that may be anywhere in the system and is not checked for validity. Using a physical address is a privileged operation. A relative address is bounds-checked, passed through a translation table, and added to a base register. A relative destination address is thus very much like a virtual memory address: it provides to a user program the illusion of a contiguous *address space* for the nodes running from 0 to one less than the number of processing elements. Access to the bounds register, translation table, or base register is a privileged operation; thus the supervisor can confine user messages within a partition.

While programs may use an interrupt protocol to process received messages, data parallel programs usually use a receiver-polls protocol in which all processors participate. In the general case, each processor will have some number of messages to send (possibly none). Each processor alternates between pushing outgoing messages onto its Data Network send-FIFO and checking its Data Network receive-FIFO. If any attempt to send a message fails, that processor should then check the receive-FIFO for incoming messages. Once a processor has sent all its outgoing messages, it uses the Control Network combine unit to assert this fact; it then alternates between receiving incoming messages and checking the Control Network. When all processors have asserted that they are done sending messages and all outstanding messages have been received, the Control Network asserts the **`router_done`** signal to indicate to all the processors that the communications step is complete and they may proceed.

For task-switching purposes, the supervisor can put the Data Network into All Fall Down (AFD) mode. Instead of trying to route messages to their intended destinations, the Data Network drops each one into the nearest node. The advantage of this strategy is that no node will receive more than a few hundred bytes of AFD messages, even if they were all originally intended for a single destination. The supervisor can then read them from the Data Network receive-FIFO and save them in memory as part of the user task state, re-sending them when that user task is resumed.

Chapter 18

System Architecture and Administration

The CM-5 system architecture provides for multiple task execution partitions, I/O devices, and fault detection and recovery. It supports a centralized system administration facility that gives the administrator flexibility to optimize the use of system resources. All these tasks are handled through various extended capabilities and privileged features of the Control Network and Data Network, with the assistance of a third network, the Diagnostic Network.

18.1 The System Console

Administration is managed from a system console, a process executing on a control processor that has a Diagnostics Network interface. Large CM-5 systems will typically have a dedicated processor for administration; on small CM-5 systems, the administration process may run on a control processor that also has other tasks.

The system console processor has a Diagnostics Network connection that allows it to address the entire system. It is responsible for configuring the system on power-up, for partitioning the system, and for managing the system as it changes due to repartitioning and hardware failures. A database containing the status of the overall system, kept up to date by the Diagnostics Network, helps it perform these tasks.

18.2 Allocation of Resources

The CM-5 system provides flexible allocation of computational resources. The administrator can subset processing nodes into partitions; the administrator can also allocate control processors to single or multiple I/O devices.

Partitions

The set of computational and network resources in use at any given instant by a single user task is called a *partition*. Each partition constitutes a complete task execution system that may be used for timesharing, batch processing, or both.

The system administrator creates partitions dynamically, to best accommodate the site's workload. Some administrators may use a partitioning strategy that involves changing the partitioning two or three times during the course of a day. Other sites may stick with a single set of partitions for several days at a time.

An administrator might, for example, create three partitions on a system: one dedicated to a production run of a single large application, a second one used for timeshared program development by day and scheduled batch processing by night, and a third small one dedicated to around-the-clock timeshared access.

All partitions are joined by the Control Network and Data Network into a single integrated system. Resources can therefore be reallocated from one partition to another when necessary. For example, all partitions might be joined to form one giant partition in order to tackle a single giant application. As another example, if processors were to fail in the partition dedicated to a production run, they could be replaced (by reconfiguring the networks) with processors borrowed from another partition. The production run could then be rolled back to a prior checkpoint and resumed with minimal disruption, while the failed processors were powered down and, at a convenient time, physically replaced.

I/O

I/O devices and interfaces, like processing nodes, reside in specific areas of the network address space and are managed by control processors. The I/O resources they control are available to processes running on any partition. The Data Network transfers data between I/O devices and partitions, while the Control

Network is used by the operating system to monitor the transfers and signal errors.

18.3 Partitions and Networks

From a system view, the Control Network and Data Network are designed to provide

- the capability for flexible partitioning of computing resources
- the isolation of each partition's network activity
- high throughput for all cases of data transfer

To see how this works, we look at the way in which the address space on these two networks is managed.

Figure 30 shows a simplified view of address space management in the networks. As this figure suggests, each of the superficially homogeneous networks is logically split by hardware-supported, software-configured mechanisms so as to devote a portion to each partition or I/O resource. Additional network capacity is dedicated to carrying traffic between the various partitions and devices that make up the system at any given time. Network resources allotted to one partition do not overlap those associated with another. Moreover, traffic from one partition to another, or between a partition and an I/O device, consumes no network resources belonging to any intervening partition. The network design thus guarantees that network traffic within one partition cannot affect the behavior or the performance of traffic in another partition. (The only exception occurs when processors fail and are logically replaced for the nonce by more distant processors from another partition.) The design also allows the Data Network to guarantee each processing node at least 5 Mbytes/sec of I/O bandwidth, no matter where it is in the network. However the nodes are divided into partitions, there is always enough Data Network to serve each partition and enough left over to guarantee the stated I/O rate.

When a CM-5 system is first powered up, reset, and bootstrapped, the networks form a single partition that spans the entire system. The operating system then creates a temporary partition for initializing the nodes. It also initializes the I/O devices. After the startup procedures have been completed, administration software establishes one or more operating partitions.

Within each partition, the Network Interfaces are assigned virtual network addresses starting at zero. User programs use virtual network addresses; they are translated by hardware into physical network addresses wherever necessary, in exactly the same way that a memory management unit translates virtual memory addresses to physical memory addresses. Therefore, a user program need not concern itself with the physical network addresses of the partition being used to execute it.

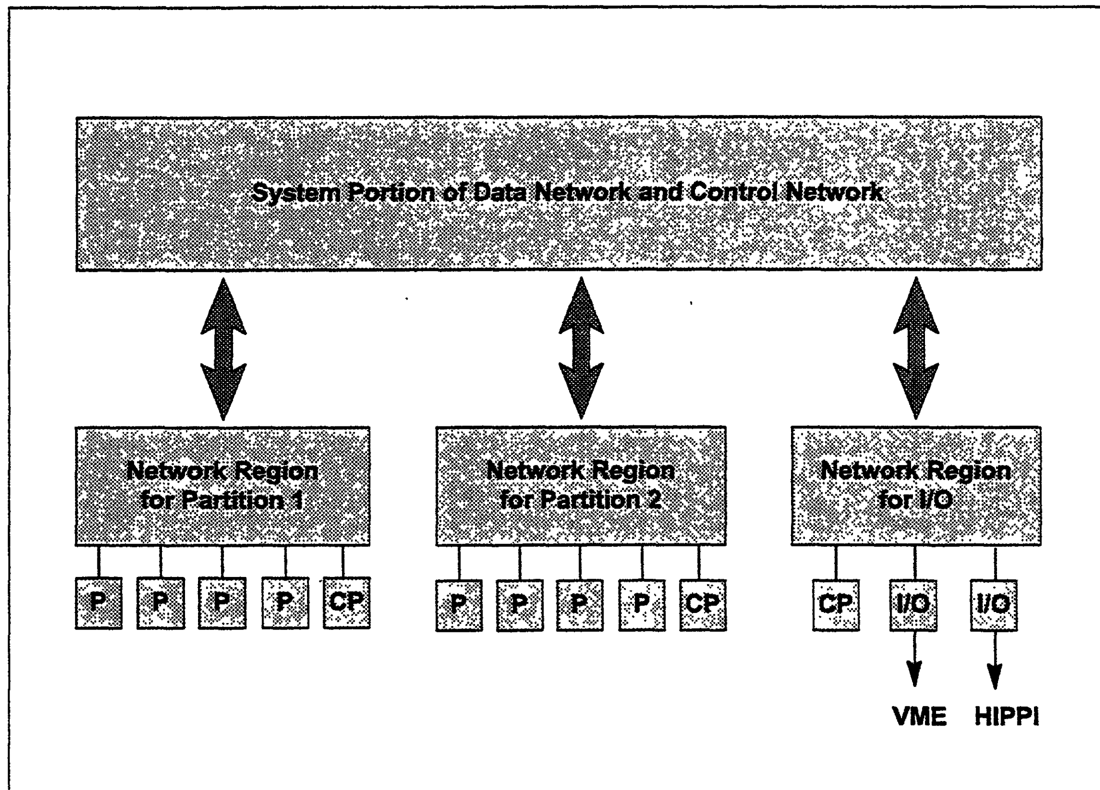


Figure 30. Network support for multiple partitions.

The processing nodes of a CM-5 system can be configured into two or more partitions. Each partition is assigned to a partition manager, a control processor that bears the responsibility for managing the processes executing in that partition.The operating system configures the Control Network and Data Network to match the partition structure. Each partition has a dedicated portion of each network sufficient to provide that partition with the guaranteed minimum network bandwidth of 20 Mbytes/sec for the Control Network and 5 Mbytes/sec per node for the Data Network, regardless of destination. No matter how the partitions are configured, there is always additional network capacity for carrying data between partitions and I/O devices or from one partition to another. Therefore, system-wide data traffic does not interfere with or impede traffic that stays within a partition.

The translation of virtual network addresses includes protection checking that prevents a user process from sending messages to destinations outside its partition. The supervisor can send messages from one partition to another; the mechanism is identical except that it is not subject to the same protection checks because for this purpose the supervisor uses absolute physical network addresses.

I/O is coordinated by the operating system. User processes may transfer data to and from I/O devices, or to and from other user processes (through the facility of UNIX named pipes). In both cases, the operating system breaks up the data into messages and sends the messages through the Data Network. If the two user processes happen to be in the same partition, the message traffic will be confined to that partition, not because of protection (the supervisor is responsible for sending the messages in this case) but simply as a consequence of the structure of the Data Network.

18.4 Resource Allocation and Management

CM-5 administration uses standard UNIX mechanisms to control the usage of various resources (disk usage, CPU usage, memory usage, and so on). These are enhanced for the CM-5 when necessary: for instance, stack and heap management can be set for the nodes in a partition as well as for the control processors.

Similarly, standard UNIX procedures govern the mounting and maintenance of file systems.

18.5 Accounting, Monitoring, and Error Reporting

Standard UNIX kernel and device drivers collect information on system activity. Accounting information is collected by ordinary UNIX tools, including NQS, and is logged to a central facility on the system console.

Errors occurring during normal operation of the CM system are detected by the operating system, collected and distributed by the Control Network.

Hard error signals are collected from the Data Network and from every Network Interface. These signals are combined and distributed according to the current partitioning. Errors detected within a partition are signaled to every Network

Interface in that partition, and are reported if appropriate to the user process running at the time of the error. Errors detected in portions of the network outside any partition may be optionally signaled into any designated partition.

The operating system notifies the system administrator of errors by sending a message to the system console processor. It also logs error information in a central system error log, from which it is available both to the administrator and to diagnostic utilities. System failures and transient hardware errors are also logged to central logging facilities on the system console.

18.6 Physical Monitoring Systems

The CM-5 system includes extensive power and temperature monitoring systems, designed for early detection of problems that might cause physical damage to the system. The monitoring system reports electronic danger signals, such as detection of an overheating cabinet, to the system console.

18.7 Fault Detection and Recovery

The CM-5 system is designed to provide high system availability. An important aspect of this design is rapid diagnosis and smooth degradation in the face of component failures. An integrated part of the administration system, the CM-5 diagnostic system is notable for its completeness, its speed, and the high degree of fault isolation it provides. If a failure should occur in a running partition, the administrator can interrogate all items in parallel, isolate the failing item, repartition around the failure, and have the partition up and running again quickly.

In addition, the CM-5 provides hardware and software support for checkpointing, either at specified time intervals or by explicit program request. The goal is to allow user applications to be restarted with full system capabilities, even in the presence of failed components.

The Diagnostic Network

The Diagnostic Network, which can probe and control the rest of the system, handles diagnostics. This network is designed to be simple and reliable. It is not particularly fast compared to the Control Network or Data Network, but testing and diagnostic procedures are nevertheless speedy because the Diagnostics Network can operate on all parts of the system in parallel.

The Diagnostic Library

The CM-5 diagnostic library includes a wide variety of tests. Particularly noteworthy among these are the JTAG diagnostics. Based on the IEEE Standard 1149.1, these scan-based vectors both test chips with a very high level of fault coverage and provide connectivity tests between chips (known as boundary scan checking). JTAG diagnostics exist for all CM-5 components; they provide extremely precise isolation of faults. This precision, in turn, allows rapid identification and replacement of failed components, and provides the data necessary for the administration database to exclude failed components when configuring partitions.

Diagnostics are run by partition: thus, one partition can be running diagnostics while others are running user programs. Within the partition, the administrator can choose to run diagnostics on

- the entire partition
- a single subsystem, such as the Control Network
- a single type of component, such as the nodes

Parallel processing provides speed. In the last example, all nodes are tested in parallel and report their status in parallel. In the first example, diagnostic tests on all components of the partition are run in parallel. The Diagnostics Network can address test vectors to the entire system or to any subsystem, such as a backplane, that is believed to be broken. The status of multiple chips or boards of the same type is read out in parallel, and components whose values differ from an expected value are quickly isolated.

Diagnostics and Components

All CM-5 components are designed to be testable when in place in the system. Nearly all data paths are protected by parity or full CRC. All dynamic memory is protected by full ECC that corrects single-bit errors and detects double-bit errors and DRAM chip failures. Transfers through the Control Network and Data Network are checked by hardware, not merely end-to-end but on every link, so that network component failures can be located precisely.

Failed components can be logically and electrically isolated from the rest of the system under control of the Diagnostics Network. Surrounding components are instructed to ignore any and all signals from failed components. The failed section of the system can then independently execute diagnostic tests or be powered down for repair or replacement, while the rest of the system continues normal operation.

All major CM-5 system components use either redundant or spare component schemes. If a processing node fails, then its local group of nodes is taken out of service and can be logically replaced by any other such group from anywhere in the system. All control processors are logically interchangeable; any control processor can manage any partition, and in a pinch can manage more than one at a time.

If a Control Network component fails, the consequences depend on the location of the failure within the network. It may be necessary to give up the use of 1/64 of the Network Interfaces in that partition and whatever they are connected to. In this case, spare processors may be logically mapped in to replace them. In other cases, the failure implies the loss of one partition. For example, if a CM-5 system supports up to 8 different partitions, then a Control Network failure might reduce the maximum number of partitions to 7—but the processing resources in the failed partition could be reallocated to other partitions.

If a Data Network component fails, the consequences similarly depend on the location of the failure. It may be necessary to give up the use of 1/64 of the Network Interfaces in that partition and whatever they are connected to. In other cases, no Network Interface need be abandoned; the total global bandwidth of the Data Network is diminished, but never by more than about 6 percent for each failure.

I/O devices are also designed to tolerate failures; disk arrays, for example, are designed to tolerate the failure of one or more disk units without loss of data. See the descriptions of individual I/O devices for details.

Chapter 19

Input/Output Subsystem

The initial CM-5 I/O implementation includes an interface to the family of CM-2 peripherals over a Thinking Machines proprietary CMIO bus. The CMIO bus peripherals are:

- **DataVault.** This is a high-performance disk-based mass storage system. It allows applications running on a CM-5 to access as much as 60 gigabytes of random access storage per DataVault at I/O bandwidths of up to 25 Mbytes/sec.

- **CM-HIPPI.** This is an I/O controller with 8 CMIO ports. It allows the CM-5 to exchange files with other high-performance systems over the ANSI X3T9.3 HIgh-Performance Parallel Interface (HIPPI). The multiple CMIO ports allow CM-5 I/O operations to take advantage of the HIPPI channel's exceptional bandwidth by transferring data over multiple CMIO buses in parallel.

- **CM-IOP.** This is a 16-port I/O controller for connection to SCSI-based devices, such as cartridge tape drives.

The CM-5 also supports I/O for standard VMEbus and SBus devices. The VMEbus and SBus interfaces allow the CM-5 to be connected to external control processors and peripheral device controllers that implement these popular buses. These links make available to CM-5 applications a variety of other forms of I/O, including framebuffers, tertiary storage devices, and FDDI networks.

To user applications, the CM-5 I/O subsystem consists of a collection of virtual I/O devices, any of which can be home to an accessible file system. All other details, such as file formatting and physical characteristics of the devices, are invisible to the application code. Implementation details for each I/O device are hidden by a combination of hardware and software interface modules.

19.1 I/O Architecture

Every I/O device is connected to the CM-5 through the Data Network, with each I/O interface occupying a block of Data Network address space. By convention, all I/O devices occupy the upper region of that address space. (See Figure 31.) An I/O interface attaches to the Data Network through one or more Network Interfaces — the same type of interface that connects processing nodes to the Data Network.

An important consideration in any I/O scheme is matching the system's internal bandwidth to the I/O rates of peripheral devices attached to the system. Here, the Data Network's intrinsic scalability plays a critical role. The number of Network Interfaces used to attach an I/O device to the Data Network determines how much of the network's bandwidth is made available to the device. The more ports an interface has into the Data Network, the greater its potential bandwidth.

An I/O interface with a single Network Interface provides a nominal bandwidth of 20 Mbytes/sec across the Data Network. This capacity can easily accommodate low- and medium-speed I/O devices. Interfaces for high-performance peripherals are implemented with as many Network Interfaces as are needed to support the transfer rates required by the particular device.

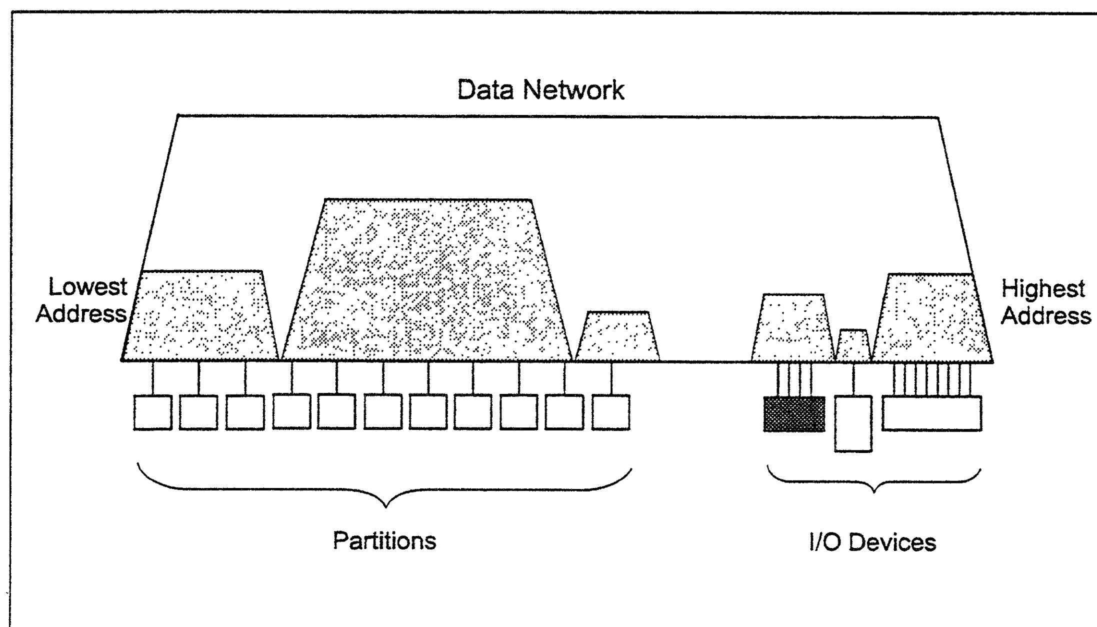


Figure 31. CM-5 I/O subsystem block diagram.

The diagram shown in Figure 32 illustrates this bandwidth scalability with two sample interfaces. The SBus device, for example, could be expected to send and receive files at about 5 Mbytes/sec. This is well within the capacity of a single Network Interface on the Data Network. The CMIO interface, however, which must support a transfer rate of up to 38 Mbytes/sec, is attached to two Network Interfaces.

Each I/O interface requires an I/O control processor (IOCP), which supervises all I/O operations for that interface. The IOCP maintains one or more file systems for the associated device and manages all requests for those file systems. The IOCP is a control processor of the same type used for other CM-5 control processing functions. Because it has the same capabilities as other control processors, the IOCP may be used to play other roles as well as its file server functions.

The VMEbus and SBus I/O interfaces are variants on this model. In VMEbus and SBus I/O subsystems, the file system processor and the other I/O interface functions are combined and reside within an external control processor. This control processor can also serve as a partition manager and as a file server for peripheral I/O devices.

19.2 File System Environment

I/O transactions execute within an enhanced UNIX environment, being modeled as reads and writes to files. Extensions have been added to handle parallel transfers, as well as to support much larger files than most UNIX implementations can accommodate.

Each file system attached to the CM-5 has a single I/O control processor (IOCP) that manages file system requests for its associated I/O device. All communication for the transfer takes place over the Data Network.

UNIX-style `open` and `close` requests go to the IOCP and so are independent of the file system data storage implementation. Requests for other file system operations, such as reading and writing files, go to the IOCP as well, which then directs the transfer of data. In these cases, data may be transferred in parallel directly from the source to the destination, without passing through the IOCP. (See Figure 33.)

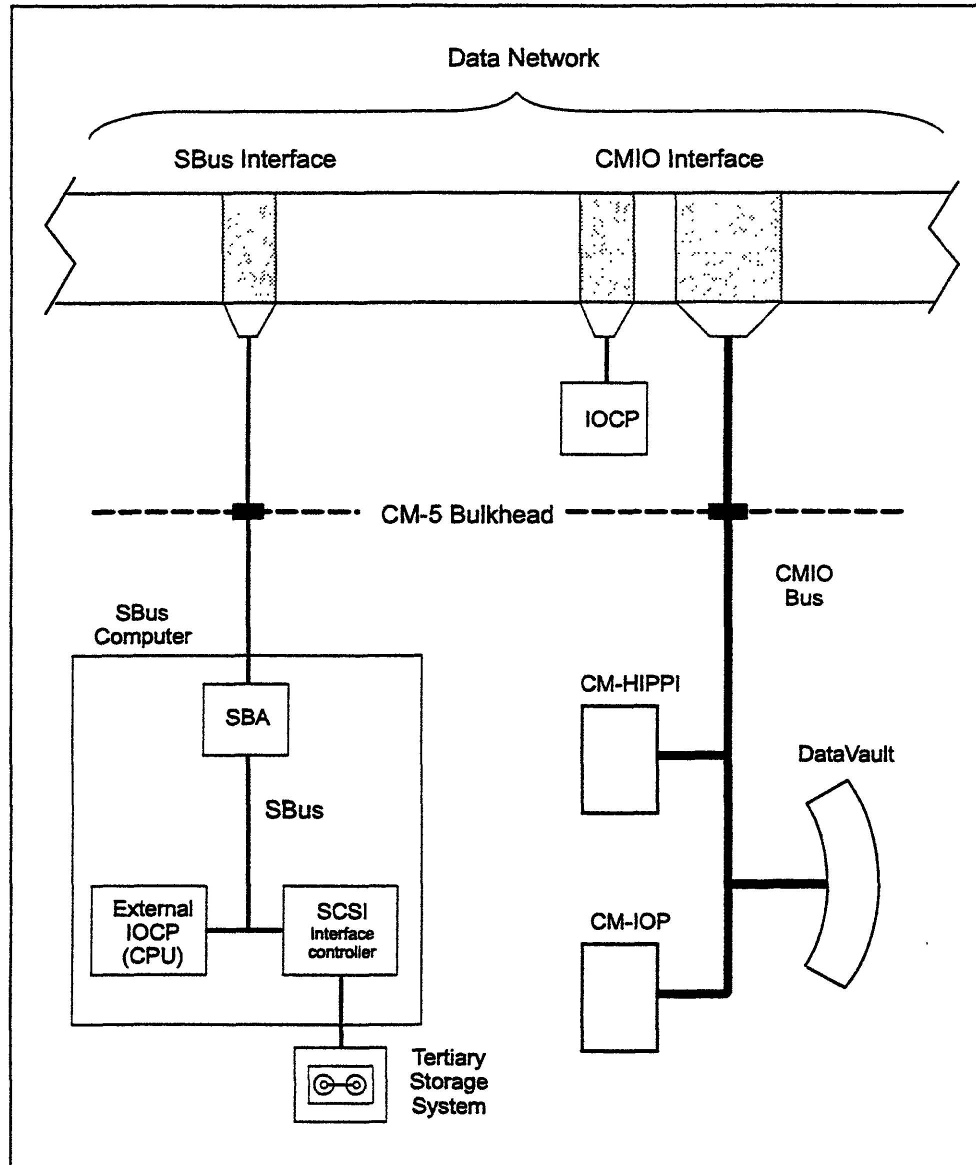


Figure 32. CM-5 I/O subsystem block diagram.

An I/O device interface requires an I/O control processor (IOCP), which maintains the I/O file system(s) for that device. For the CMIO interface, the IOCP resides in the CM-5 and is connected directly to the Data Network. In SBus and VMEbus interfaces, IOCP functions reside in the external SBus or VMEbus computer. Each I/O interface also includes device control and data buffering logic.

An I/O interface attaches to the Data Network via one or more Network Interfaces. Multiple Network Interfaces are used to increase I/O bandwidth. The drawing indicates this bandwidth difference by showing a wider attachment range for the CMIO interface than for the SBus interface.

When an application program requires I/O services, the partition on which the application is running initiates the file transfer with an appropriate read or write command. It directs the I/O request to the appropriate IOCP, which assumes control of the transfer.

For a **read** operation, the file is retrieved from the I/O device, encapsulated in message packets, and sent through the Data Network to the partition that requested the data. File order information embedded in the message packets enables the receiving partition to arrange the file data in correct sequence within each processor. A **write** operation is similar but the flow of data is reversed.

Different versions of **read** and **write** are used, depending on whether an application is running on a single processor or on a set of parallel processing nodes. A serial application uses the conventional UNIX **read** and **write** commands. Parallel applications use **CM_read** and **CM_write**.

Multiple I/O devices may be logically ganged for striped operation as a single file system. The CM-5 file system automatically routes data between requesting processors and individual I/O devices so that all striping is transparent. This same facility makes file structure independent of the number of computational processors that read or write the data. All files consistently appear to be stored as if in standard UNIX serial byte-stream order.

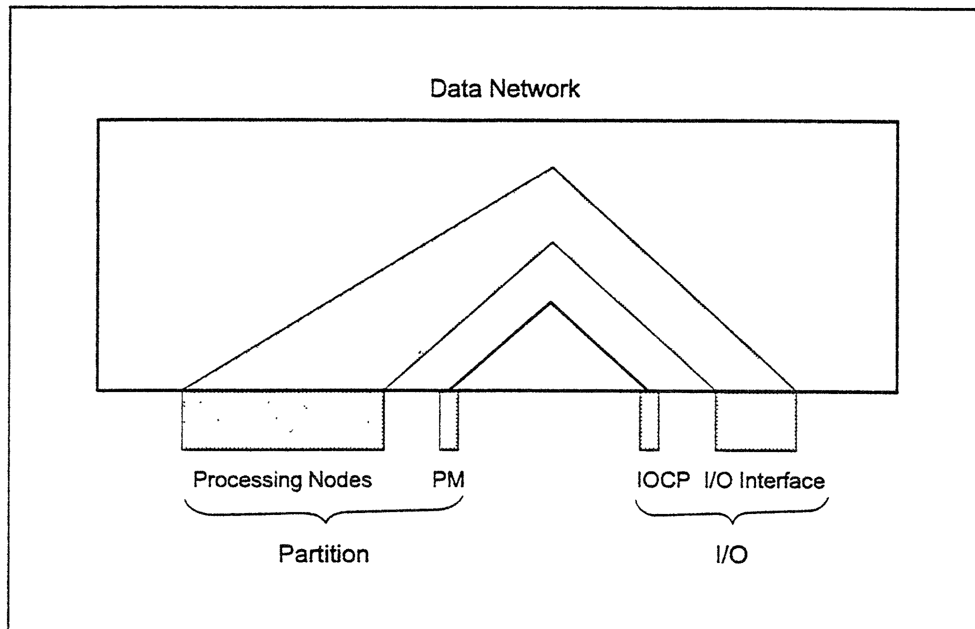


Figure 33. Independent control and data paths through the Data Network.

19.3 I/O Interfaces and Device Implementation

This section describes the key elements of the initial CM-5 I/O subsystem implementation: the CMIO bus peripherals (DataVault and CM-HIPPI), and the two standard bus interfaces, (SVME and SBA).

DataVault

The DataVault system is available in various storage capacities, ranging from 20 to 60 gigabytes. Each of these configurations is capable of transferring data at a sustained rate of 25 Mbytes/sec.

The basic DataVault storage configuration, used in the 20-gigabyte system for example, employs an array of 42 $5^1/_4$-inch Winchester disk drives, of which 39 are active and 3 are spares. (See Figure 34.) 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.

The double-capacity DataVault configuration (40 or 60 gigabytes) has 84 drives, of which 64 hold data, 14 hold ECC bits, and 6 are spares.

In all DataVault configurations, each 32-bit data word is spread across 39 data and ECC drives, one bit per drive. Each 64-bit data chunk received from the CMIO bus is first 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.

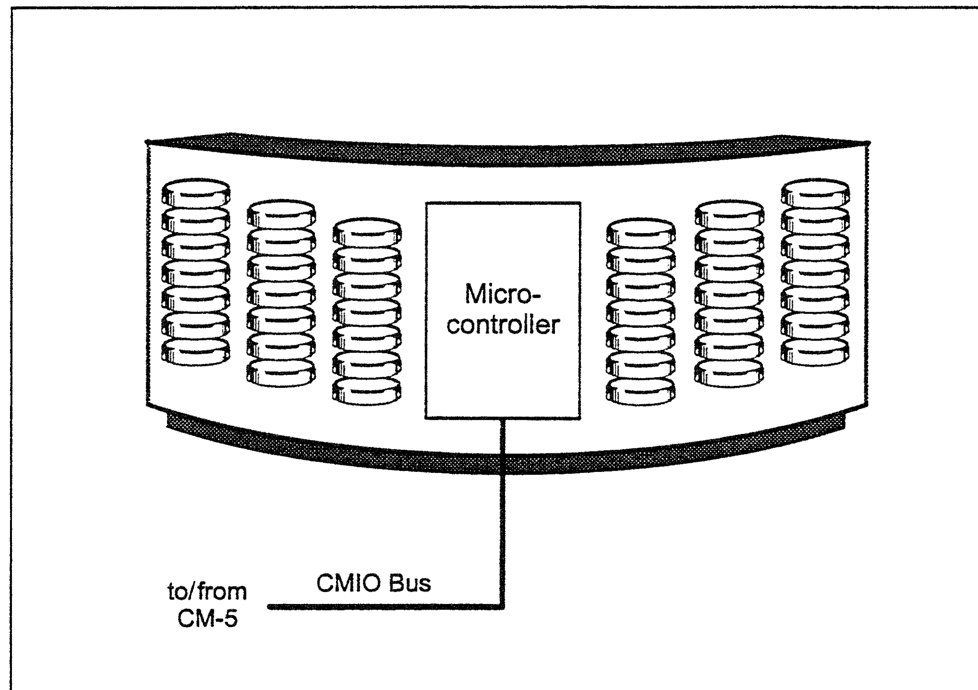


Figure 34. Inside the DataVault.

The File Server. All I/O transactions in a DataVault I/O interface are controlled by an IOCP running a file server process. The file server manages the DataVault's UNIX-based 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.

Internally, the file server represents a file as a series of *extents*, or areas of contiguous disk surface. Each extent starts at a logical offset within the file, has a physical disk address, and has 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 transfer.

Writing and Reading Data. Data transfers move information between a CM-5 partition and the DataVault. 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 but the flow of data is reversed.

A DataVault write operation is typically initiated by a partition manager, which issues a write command to the IOCP that is acting as the DataVault's file server. When the file server receives the logical file request, it translates the request into a series of physical disk addresses. Assuming that the request parameters satisfy the necessary validity checks (for example, that there is sufficient space), the file server returns a message to the requesting partition indicating the DataVault's availability. If the request cannot be fulfilled, the file server returns a failure report instead.

Data from the partition's memory is moved, via the Data Network, to I/O buffers in the CMIO interface and then across the CMIO bus to the DataVault. A microcontroller within the DataVault controls the distribution of data onto the disk array. State machines at each end of the CMIO bus ensure reliable transfer of large volumes of data across the bus. Parity checking is performed on all data as it is received from the CMIO bus to ensure data integrity.

Data being read from the DataVault follows the same path as for writing, but in reverse order: across the CMIO bus, through the CMIO interface, and across the Data Network. 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. As with write operations, parity checking is performed on data received from the CMIO bus.

Data Protection. The 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. 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 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.

CM-HIPPI 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 CM-5 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 Mbytes/sec.

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 Mbytes/sec.

The HIPPI controller CPU receives CM file system commands from a CM control processor over an Ethernet cable. A file server process running on the Sun CPU interprets these commands and controls the CM-HIPPI I/O operations 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 each for incoming data and 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 bandwidth of this configuration is 200 Mbytes/sec.

Standard Protocol I/O Interfaces

Two CM-5 standard bus interfaces, called SVME and SBA, enable the CM-5 operating system to access external VMEbus or SBus computers and their associated I/O resources using standard communications protocols. These I/O paths link the CM-5 to external networks of computing and I/O server resources.

They are connected by cable to a VMEbus- or SBus-based external control processor, which manages the file system and other I/O functions. An adapter board, installed in the control processor, provides an interface between the VMEbus and the CM-5 Data Network (and Control Network).

The SBus interface consists of an adapter board, called the SBA, that plugs into the SBus of an external control processor. (See Figure 32.) This adapter board is connected by cable to an interface module, called the control processor interface (CPI), that is plugged into the CM-5 Data Network and Control Network. The external control processor, running file server code, serves as the file system processor. This arrangement allows applications running on the CM-5 to exploit any I/O resources, such as a tape storage system, that are attached to the external control processor's SBus.

In a similar fashion, the VMEbus interface uses a VME adapter board, called the SVME, to connect a control processor's VMEbus to the CM-5 Data Network and Control Network. Apart from this difference, the VMEbus interface employs the same design features as the SBus interface.