Connection Machine® Model CM-2 Technical Summary

Thinking Machines Corporation
Technical Report Series

HA87-4
5/88

Thinking Machines Technical Report HA87-4

Connection Machine®
Model CM-2
Technical Summary

April 1987

© 1987 Thinking Machines Corporation

Connection Machine is a registered trademark of Thinking Machines Corporation.
DataVault, C*, *Lisp, CM-Lisp, and Paris are trademarks of
Thinking Machines Corporation.
Symbolics 3600 is a trademark of Symbolics, Inc.
UNIX is a trademark of AT&T Bell Laboratories.
VAX, VAXBI, and ULTRIX are registered trademarks of Digital Equipment Corporation.

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 reserves the right to make changes to any products described herein to improve functioning or design. Although the information in this document has been reviewed and is believed to be reliable, Thinking Machines Corporation does not assume responsibility or liability for any errors that may appear in this document. Thinking Machines Corporation does not assume any liability arising from the application or use of any information or product described herein.

Contents

1 Introduction 1
1.1 The Connection Machine System 1
1.2 Data Parallel Hardware 2
1.3 Data Parallel Software 3

2 System Organization 4

3 The Paris Language 8
3.1 Overview 8
3.2 Virtual Machine Model 9
3.3 Organization of the Instruction Set 10
3.4 Instruction Set Summary 11

4 Processor Architecture 19
4.1 Data Processors 19
4.2 The Router 20
4.3 The Floating Point Accelerator 22

5 The Role of the Front End 23
5.1 Applications Development 23
5.2 Running Connection Machine Applications 24
5.3 Maintenance and Operations Utilities 24
5.4 The Digital Equipment Corporation VAX As a Front End 25
5.5 The Symbolics Lisp Machine As a Front End 25

6 Connection Machine I/O Structure 27

7 The Connection Machine DataVault 28
7.1 The File Server 28
7.2 Off-line Loading and Backup 29
7.3 Writing and Reading Data 29
7.4 Drive Failure and Data Base Healing 30

8 High-Resolution Graphics Display 31

9 Languages 33

10 The C* Language 35
10.1 Data Parallel Machine Model 35
10.2 Parallel Expressions 37
10.3 Parallel Statements 39
10.4 Compiler Implementation 41

11 Fortran 42
11.1 The Environment 42
11.2 The Array Extensions of Fortran 43
11.3 Fortran Statements for Controlling Context 43
11.4 Interprocessor Communication in Fortran 44
11.5 Fortran and the Data Parallel Approach 45

12 The *Lisp Language 46
12.1 Pvars: The Basic *Lisp Data Object 46
12.2 Processor Addressing 47
12.3 Reading and Writing Data from and to Pvars 47
12.4 Basic Parallel Operations 47
12.5 Selection of Active Sets of Processors 48
12.6 Communication between Processors 48
12.7 Global Reduction Operations 49
12.8 Summary 49

13 The CM-Lisp Language 50
13.1 Xappings, Xets, and Xectors 50
13.2 Parallel Computation: α Syntax 51
13.3 Interprocessor Communication: β Syntax 54
13.4 Library Functions 55

14 An Example Program 56
14.1 The Example Program in C* 56
14.2 The Example Program in Fortran 57
14.3 The Example Program in *Lisp 57
14.4 The Example Program in CM-Lisp 57

15 Performance Specifications 58
15.1 General Specifications 58
15.2 Input/Output Channels 58
15.3 Typical Application Performance (Fixed Point) 58
15.4 Interprocessor Communication 59
15.5 Variable Precision Fixed Point 59
15.6 Double Precision Floating Point 59
15.7 Single Precision Floating Point 60
15.8 Parallel Processing Unit Physical Dimensions 60
15.9 Parallel Processing Unit Environmental Requirements 60

1 Introduction

The Connection Machine Model CM-2 is a data parallel computing system. Data parallel computing associates one processor with each data element. This computing style exploits the natural computational parallelism inherent in many data-intensive problems. It can significantly decrease the execution time of a problem, as well as simplify its programming. In the best cases, execution time can be reduced in proportion to the number of data elements in the computation; programming effort can be reduced in proportion to the complexity of expressing a naturally parallel problem statement in a serial manner. In order to fully exploit these potential benefits, a computing system consisting of both hardware and software that support this model of computing is required.

The Connection Machine Model CM-2 is an integrated system of hardware and software. The hardware elements of the system include front-end computers that provide the development and execution environments for the system software, a parallel processing unit of 64K processors that execute the data parallel operations, and a high-performance data parallel I/O system. The system software is based upon the operating system or environment of the front-end computer. The visible software extensions are minimal. Users can program using familiar languages and programming constructs, with all the development tools provided by the front end. Programs have normal sequential control flow; new synchronization structures are not needed. Thus, users can easily develop programs that exploit the power of the Connection Machine hardware.

1.1 The Connection Machine System

At the heart of any large computational problem is the data set: some combination of interconnected data objects, such as numbers, characters, records, structures, and arrays. In any application this data must be selected, combined, and operated upon. Data level parallelism takes advantage of the parallelism inherent in large data sets.

At the heart of the Connection Machine Model CM-2 system is the parallel processing unit, which consists of thousands of processors, each with thousands of bits of memory. These processors can not only process the data stored in their memory, but also can be logically interconnected so that information can be exchanged among the processors. All these operations happen in parallel on all processors. Thus, the Connection Machine hardware directly supports the data parallel problem model.

One way to view the relationship of the CM-2 parallel processing unit to the other parts of the system is to consider it as an intelligent extension to the memory of the front-end computer. The data parallel data objects are stored by assigning each one to the memory of a processor. Then the operations on these objects can be specified to operate simultaneously on any or all data objects in this memory.

The Connection Machine processors are used whenever an operation can be performed simultaneously on many data objects. Data objects are left in the Connection

Machine memory during execution of the program and are operated upon in parallel at the command of the front end. This model differs from the serial model of processing data objects from a computer's memory one at a time, by reading each one in turn, operating on it, and then storing the result back in memory before processing the next object.

The flow of control is handled entirely by the front end, including storage and execution of the program and all interaction with the user and/or programmer. The data set, for the most part, is stored in the Connection Machine memory. In this way, the entire data set can be operated upon in parallel through commands sent to the Connection Machine processors by the front end. The front end can also operate upon data stored in individual processors in the Connection Machine, treating them logically as memory locations in its virtual memory.

There are several direct benefits to maintaining program control only on the front end. First, programmers can work in an environment that is familiar. The front end interacts with the Connection Machine parallel processing unit using an integrated command set, and so the programming languages, debugging environment, and operating system of the front end remain relatively unchanged. Second, a large part of the program code for any application pertains to the interfaces between the program, the user, and the operating system. Since the control of the program remains on the front end, code developed for these purposes is useful with or without the Connection Machine parallel processing unit, and only the code that pertains specifically to the data residing on the Connection Machine processors needs to use the data parallel language extensions. Finally, parts of the program that are especially suited for the front end, such as file manipulation, user interface, and low-bandwidth I/O, can be done on the front end, while the parts of the program that run efficiently in parallel, namely the "inner loops" that operate on the data set, can be done on the Connection Machine. In this way, the individual strengths of both the serial front end and the Connection Machine processors can be exploited.

In general, the Connection Machine system appears to be a very powerful extension of the front-end system. The data parallel hardware looks like intelligent memory; the data parallel software extends the front end's capabilities to allow the direct execution of parallel operations.

1.2 Data Parallel Hardware

The Connection Machine system implements data parallel programming constructs directly in hardware. The system includes 65,536 physical processors, each with its own memory. Parallel data structures are spread across the data processors, with a single element stored in each processor's memory. When parallel data structures have more than 65,536 data elements (the normal case), the hardware operates in virtual processor mode, presenting the user with a larger number of processors, each with a correspondingly smaller memory.

Communication among elements of a parallel data structure is implemented by

a high-speed routing network. Processors that hold interrelated data elements store pointers to one another. When data is needed, it is passed over the routing network to the appropriate processors.

Scalar data is held in a front-end processor. The front end also controls execution of the overall data parallel program. Program steps that involve parallel data are passed over an interface to the Connection Machine parallel processing unit, where they are broadcast for execution by all the processors at once.

The Connection Machine front end provides the programming environment for the system. Programs can be stored on front-end disks. Network communications links are most effectively implemented on the front end as well.

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

1.3 Data Parallel Software

The Connection Machine system software is designed to utilize existing programming languages and environments as much as possible. The languages are based on well-known standards; the extensions to support data parallel constructs are minimal so that a new programming style is not required. The CM-2 front-end operating system (either UNIX or Lisp) remains largely unchanged.

Fortran on the Connection Machine system uses the array extensions in the draft Fortran 8x standard (proposed by ANSI technical committee X3J3) to express data parallel operations. The remainder of the language is the standard Fortran 77. No extension is specific to the Connection Machine; the Fortran 8x array extensions map naturally onto the underlying data parallel hardware.

The *Lisp and CM-Lisp languages are data parallel dialects of Common Lisp (a version of Lisp currently being standardized by ANSI technical committee X3J13). *Lisp gives programmers fine control over the CM-2 hardware while maintaining the flexibility of Lisp. CM-Lisp is a higher-level language that adds small syntactic changes to the language interface and creates a very powerful data parallel programming language

The C* language is a data parallel extension of the C programming language (as described in the draft C standard proposed by ANSI technical committee X3J11). C* programs can be read and written like serial C programs; the extensions are unobtrusive and easy to learn.

The assembly language of the CM-2 is Paris. This is the target language of the high-level language compilers. This language logically extends the instruction set of the front end and masks the physical implementation of the CM-2 processing unit.

2 System Organization

The Connection Machine Model CM-2 is a complete computing system that provides both development and execution facilities for data parallel programs. Its hardware consists of a parallel processing unit containing thousands of data processors, from one to four front-end computers, and an I/O system that supports mass storage and graphic display devices (see Figure 1). The user interacts with the front-end computer; all program development and execution takes place within the front end. Because the front-end computer runs standard serial software, the user sees a familar system environment with additional languages and utilities and some very powerful hardware.

The central element in the system is the CM-2 parallel processing unit, which contains:

- thousands of *data processors*
- an interprocessor communications network
- one or more *sequencers*
- an interface to one or more front-end computers
- zero or more *I/O controllers* and/or *framebuffers*

A parallel processing unit may contain 64K, 32K, or 16K data processors. (Here, and throughout this document, "K" stands for 1024. Thus 64K means 65,536; 32K means 32,768; 16K means 16,384; 8K means 8,192; and so on.) Each data processor has 64K bits (8 kilobytes) of bit-addressable local memory and an arithmetic-logic unit (ALU) that can operate on variable-length operands. Each data processor can access its memory at a rate of at least 5 megabits per second. A fully configured CM-2 thus has 512 megabytes of memory that can be read or written at about 300 gigabits per second. When 64K processors are operating in parallel, each performing a 32-bit integer addition, the CM-2 parallel processing unit operates at about 2500 Mips[1]. (This figure includes all overhead for instruction issuing and decoding.) In addition to the standard ALU, the CM-2 parallel processing unit has an optional parallel floating point accelerator that performs at 3500 MFlops[2] (single precision) or 2500 MFlops (double precision).

One of the most important requirements of general purpose data parallel computing is the ability of the data elements to communicate information among themselves in patterns that vary according to the problem and with time. The CM-2 system provides two forms of communication within the parallel processing unit. The more general mechanism is known as the *router*, which allows any processor to communicate with any other processor. One may think of the router as allowing every processor to send a message to any other processor, with all messages being sent and delivered at the same time. Alternatively, one may think of the router as allowing every processor to access

[1] Mips = Millions of instructions per second

[2] MFlops = Millions of floating point operations per second

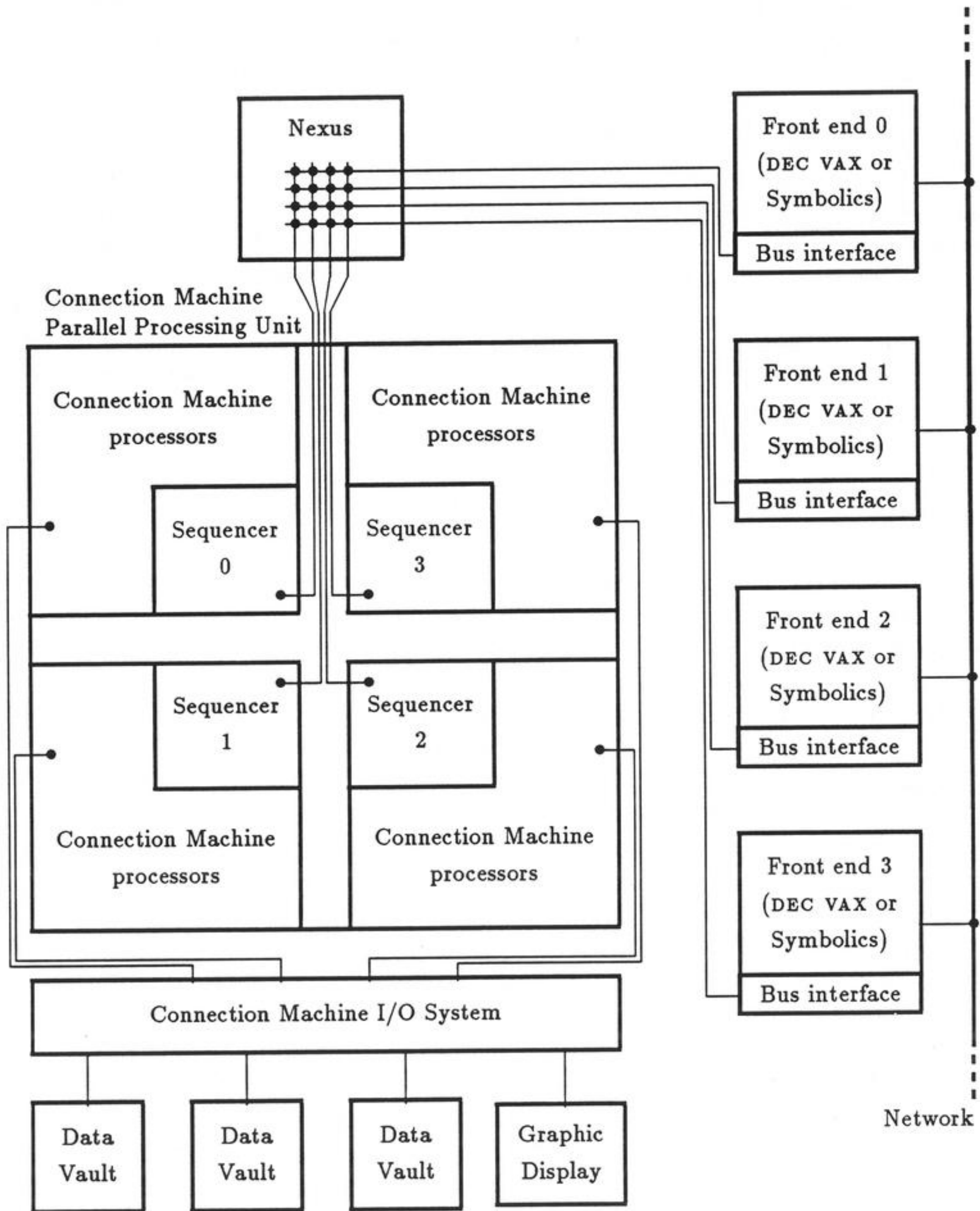

Figure 1: Connection Machine Model CM-2 System Organization

any memory location within the parallel processing unit, with all processors making memory accesses at the same time; in effect, the router allows the local memories of the data processors to be treated as a single large shared memory. The messages (or accessed fields, if you will) may be of any length. The throughput of the router depends on the message length and on the pattern of accesses; typical values are 80 million to 250 million 32-bit accesses per second.

The CM-2 parallel processing unit also has a more structured, somewhat faster communication mechanism called the *NEWS grid.* In the CM-1 and some other fine grained parallel systems, communication can take place over a fixed two-dimensional grid. The CM-2, however, supports programmable grids with arbitrarily many dimensions. Possible grid configurations for 64K processors include 256×256, 1024×64, 8×8192, $64 \times 32 \times 32$, $16 \times 16 \times 16 \times 16$, and $8 \times 8 \times 4 \times 8 \times 8 \times 4$. The NEWS grid allows processors to pass data according to a regular rectangular pattern. For example, in a two-dimensional grid each processor could receive a data item from its neighbor to the east, thereby shifting the grid of data items one position to the left. The advantage of this mechanism over the router is merely that the overhead of explicitly specifying destination addresses is eliminated; for many applications this is a worthwhile optimization.

The parallel processing unit is designed to operate under the programmed control of a front-end computer, which may be either a Symbolics 3600 Lisp machine or a DEC VAX 8000 series computer with a BI bus. The front end provides the program development and execution environment. All Connection Machine programs execute on a front end; during the course of execution the front end issues instructions to the CM-2 parallel processing unit. In effect, the CM-2 parallel processing unit extends the instruction set and I/O capabilities of the front-end computer. The set of instructions that the front end may issue to the parallel processing unit is called Paris. It is designed for convenient use by front-end programs, and includes not only such operations as integer arithmetic, floating point arithmetic, and interprocessor communication, but also such powerful operations as vector summation, matrix multiplication, and sorting. The Paris instruction set is described further in Chapter 3.

The data processors do not handle Paris instructions directly. Instead, Paris instructions from the front end are processed by a *sequencer* in the parallel processing unit. The task of the sequencer is to break down each Paris instruction into a sequence of low-level data processor and memory operations. The sequencer broadcasts these low-level operations to the data processors, which execute them at a rate of several million per second. The low-level operations are described further in section 4.1.

To increase the flexibility of program development and execution, the CM-2 processing unit may be divided into as many as four sections. Depending on the configuration, a section will have either 8K or 16K data processors. For example, a parallel processing unit with 64K data processors will be divided into four sections of 16K data processors; a processing unit with 32K data processors could consist of either two 16K sections or four 8K sections.

Each section can be treated as a complete parallel processing unit in itself; in particular, each section contains its own sequencer, router, and NEWS grid. Sections may also be ganged; when this is done, their sequencers are also ganged and behave as a single sequencer, their routers cooperate as a single router, and their NEWS grids cooperate to form a single grid. A programmable, bidirectional switch called the *Nexus* allows up to four front-end computers to be attached to a single parallel processing unit. The front ends need not all be of the same type. Under front-end software control, the Nexus can connect any front end to any section or valid combination of sections in the CM-2 parallel processing unit. For example, in a CM-2 system with 32K data processors (in four 8K sections) and four front ends, one could assign one section to each front end for software testing; or one could gang all four sections to be controlled by any one front end for a production run; or one could assign 8K sections to each of two front ends, gang the other two sections to give 16K data processors to a third front end, and use the fourth front end for purposes unrelated to the parallel processing unit. The Nexus can be reconfigured in seconds; once this is done, data and instructions flow between the front end and the sequencers without visible intervention by the Nexus.

For every group of 8K data processors there is one I/O channel. (A section with 8K processors therefore has one channel; a section with 16K processors has two channels.) To each I/O channel may be connected either one high-resolution graphics display framebuffer module or one general I/O controller supporting an I/O bus to which several DataVault mass storage devices may be connected. The front end controls I/O transfers in exactly the same manner that it controls the data processors, by issuing Paris instructions to the sequencer. The sequencer can then send low-level commands to the I/O channels and interrogate channel status. Data is transferred directly and in parallel between the I/O devices and the data processors, without being funneled through the sequencers.

3 The Paris Language

The instructions that the front end may issue to the parallel processing unit constitute a language called Paris (from the phrase "parallel instruction set"). It is the lowest-level protocol by which the front-end computer directs the actions of Connection Machine processors.

3.1 Overview

Paris is intended primarily as a base upon which to build higher-level languages for the Connection Machine system. It provides a large number of operations similar to the machine-level instruction set of an ordinary computer. Paris supports primitive operations on signed and unsigned integers and floating point numbers, as well as message-passing operations, I/O commands, and facilities for transferring data between the Connection Machine processors and the front-end computer.

Paris instructions direct the handling of data by the Connection Machine processors. Control instructions, such as subroutine calls, **if-then-else** conditionals, and **while** loops are not a part of the Paris instruction set. The control structure for an application is provided by the front-end computer. A program that is "written in Paris" must actually be written in some ordinary sequential language for the front end, such as C, Fortran, Pascal, or Lisp.

The Paris user interface consists of a set of functions, subroutines, and global variables. The functions and subroutines direct the actions of the Connection Machine processors, and the variables allow the user program to find out such information about the Connection Machine system as the number of processors available and the amount of memory per processor.

As a simple example, here is a bit of C code that repeatedly causes every processor whose floating point `z` field is greater than `1.0` to be divided by two; the loop is terminated when no processor has a `z` value greater than one.

```
while (CM_f_gt_constant(z, 1.0, 23, 8),
       CM_global_logior(CM_test_flag, 1)) {
    CM_f_divide_constant_2(z, 2.0, 23, 8);
}
```

The functions whose names begin with "CM_" are Paris operations: `CM_f_gt_constant` causes every processor to compare a field to a common, broadcast constant, storing a bit reflecting the result in its "test" flag; `CM_f_divide_constant` similarly causes every processor to divide a floating point field by a common constant; and `CM_global_logior` takes a bit field (in this example, a one-bit field, namely the test flag) from every processor, and returns to the front end the result of a many-way bitwise inclusive-OR operation. The `while` construct is an ordinary C `while` loop, and is not a part of the Paris language proper.

Several different versions of the user interface are provided, one for each front-end programming language in which Paris is to be embedded. These interfaces are functionally identical; they differ only in conforming to the syntax and data types of one language or the other. Here is what the preceding example would look like if embedded in the Lisp language:

```
(do ()
    ((progn (CM:f-gt-constant z 1.0 23 8)
            (zerop (CM:global-logior CM:test-flag 1))))
  (CM:f-divide-constant-2 z 2.0 23 8))
```

This example of Lisp code uses a Lisp control structure, `do`, that is nearest in function to the C `while` statement. (It is actually a do-until statement, and the Lisp function `zerop` is used here to invert the sense of the result of `CM:global-logior`.) However, it would be appropriate to Lisp programming style to use a recursive function instead to express such a loop:

```
(defun loop ()
  (CM:f-gt-constant z 1.0 23 8)
  (unless (zerop (CM:global-logior CM:test-flag 1))
    (CM:f-divide-constant-2 z 2.0 23 8)
    (loop)))
```

This example underscores the point that the control structure of the program may be written in any programming language (even the assembly language of the front-end computer, if necessary), and in any style suitable to that programming language. Paris merely extends that language by providing for the parallel processing of data.

3.2 Virtual Machine Model

Paris presents to the user an abstract machine architecture that is very much like the physical Connection Machine hardware architecture, but with two important extensions: the virtual processor abstraction and a much richer instruction set.

The virtual processor abstraction (on which most higher-level software depends) is supported at the Paris level. When the Connection Machine system is initialized for a particular application, the number of virtual processors required by the application may be specified. If this number exceeds the number of available physical processors, then the local memory of each processor is split up into as many regions as necessary, and for every Paris instruction the processors are automatically time-sliced among the regions. For example, if an application should need to process a million pieces of data, it might request $V = 2^{20}$ virtual processors. Assume the available hardware to have $P = 2^{16}$ physical processors each with $M = 2^{16}$ bits of memory. Then each physical processor would support $V/P = 16$ virtual processors; this ratio V/P, usually denoted N, is called the *virtual processor ratio*, or *VP-ratio*. In this example each virtual processor

would have $M/N = 2^{12}$ bits of memory and would appear to execute code at about $1/N = 1/16$ the speed of a physical processor.

The time taken to perform a `move` depends on the length of the field to be moved and also on the number of virtual processors in use. If each physical processor is simulating N virtual processors, then issuing a single `move` instruction causes each physical processor to execute N `move` operations. This will take N times as long as if virtual processors were not in use, but also does N times as much work, so the Mips measurement is about the same. Indeed, the use of virtual processors usually increases the measured Mips rate, because the instruction needs to be decoded by the sequencer only once for N executions, and so the decoding overhead may be amortized.

Each virtual processor has some local memory and also a number of 1-bit flags. Most of the flags are condition codes such as `overflow` and `float-inexact`. The `context` flag, however, controls conditional execution: for most Paris operations a processor executes the operation if its context flag is 1, but does not participate if its context flag is 0. Processors whose context flag is 1 are said to be *active*; the set of active processors is called the *current context*. A few operations are unconditional, being executed by all processors regardless of the values of their context flags. (It is important, for example, that there be a way to set all context flags to 1 unconditionally!)

3.3 Organization of the Instruction Set

Most Paris operations deal with *fields* in the local memories of the Connection Machine processors. A field is specified by two quantities: the address of its first bit, and its length in bits. Uninterpreted bit fields (as processed by such operations as `move`, `send`, and `logand`) may be of any length. The length of an unsigned integer may range from 0 to 128 bits, and the length of a signed integer may range from 2 to 128 bits. (Some very simple arithmetic operations, such as addition, subtraction, and comparisons, are not limited to 128 bits.) Floating point operations are available in a variety of precisions, including 32-bit, 64-bit, and 80-bit formats.

Nearly all operations are memory-to-memory; for example, the signed integer addition operation can add the value of one memory field into another memory field (two-address mode) or can replace a memory field with the sum of two other fields (three-address mode). The flags are addressed as if they were 1-bit memory fields.

Many operations come in several forms, differing from each other in up to three categories:

- *Addressing modes.* The operations `s-add-2` and `s-add-3` both perform signed integer addition, but the one takes two addresses and a length and the other takes three addresses and a length. The operation `s-add` takes three addresses and three lengths, allowing the three fields involved to be of different sizes. Anything `s-add-2` can do, `s-add-3` can do by duplicating one address operand; anything `s-add-3` can do, `s-add` can do by triplicating the length operand. The concise addressing modes improve performance by reducing total instruction size; the

front end has fewer operands to send to the sequencer, and the sequencer has fewer operands to decode.

- *Conditionalization.* Most operations are executed only by active processors, but some are executed unconditionally by all processors. For example, the operation **move** copies one memory field to another for processors in the current context, but the operation **move-always** copies one memory field to another in all processors, regardless of the current context.

- *Immediate operands.* The operation **s-add-2** adds one memory field into another in all active processors; the operation **s-add-constant-2** adds an immediate quantity, sent from the front end as part of the instruction, into a memory field in all active processors. Note that the word "constant" in the instruction name is a relative term. The immediate operand is constant in being the same for all the data processors, but need not be constant within the front-end program; the front end may calculate the value to be sent to the sequencer.

3.4 Instruction Set Summary

The following sections list groups of related Paris instructions, with commentary, to illustrate the expressive power of the instruction set. This is not a complete list of Paris operations.

The names of the Paris operations are listed here in a compromise format. The name to be used in a Lisp program is derived by prefixing a name given below with "CM:"; the name to be used in a C program is derived by prefixing a name given below with "CM_" and converting all hyphens to underscores. Thus the operation **s-add-2** would be called CM:**s-add-2** in Lisp code and CM_s_add_2 in C code.

3.4.1 Operations on Bit Fields

move	move-constant	move-always
logand	logand-constant	logand-always
logior	logior-constant	logior-always
logxor	logxor-constant	logxor-always
logeqv	logeqv-constant	logeqv-always
lognand	lognand-constant	lognand-always
lognor	lognor-constant	lognor-always
logandc1	logandc1-constant	logandc1-always
logandc2	logandc2-constant	logandc2-always
logorc1	logorc1-constant	logorc1-always
logorc2	logorc2-constant	logorc2-always
lognot	array-fetch	array-store
load-context	store-context	set-context
move-reversed	move-zero	latch-leds

Every instruction in this group is executed by each data processor independently of the other data processors.

The `move` operations copy data from one memory field to another. Assuming only one virtual processor per physical processor and 32-bit fields, a `move` instruction, including all decoding overhead, takes about 21 microseconds; with 64K processors, this represents an aggregate execution rate of 3000 million individual 32-bit `move` operations per second.

All ten nontrivial binary bitwise boolean operations are provided. The `array-fetch` and `array-store` perform indexed load and store operations; every data processor has a small array of items within it, and each data processor may have a different index into its array. The `load-context`, `store-context`, and `set-context` operations are special cases of `move` optimized for use on the context flag.

The red lights on the CM-2 cabinet may be turned off and on by the `latch-leds` instruction; there is one light for every 16 processors.

3.4.2 Operations on Signed Integers

```
s-add          s-add-constant        s-add-carry
s-subtract     s-subtract-constant   s-subtract-borrow
s-multiply     s-multiply-constant   s-add-flags
s-divide       s-divide-constant     s-mod
s-max          s-max-constant        s-rem
s-min          s-min-constant        s-random
s-eq           s-eq-constant         s-eq-zero
s-ne           s-ne-constant         s-ne-zero
s-gt           s-gt-constant         s-gt-zero
s-ge           s-ge-constant         s-ge-zero
s-lt           s-lt-constant         s-lt-zero
s-le           s-le-constant         s-le-zero
s-shift        s-shift-constant      s-integer-length
s-abs          s-signum              s-new-size
s-negate       s-count-bits          s-isqrt
```

Every instruction in this group is executed by each active data processor independently of the other data processors. Most of these are operations familiar to any assembly language programmer: arithmetic operations, comparisons, absolute value, negate, and shift. The `s-new-size` operation copies a signed integer from one field to another of different size, performing sign extension or overflow checking as appropriate.

Assuming only one virtual processor per physical processor and 32-bit fields, an `s-add` instruction, including all decoding overhead, takes about 26 microseconds; with 64K processors, this represents an aggregate execution rate of 2500 million 32-bit additions per second.

3.4.3 Operations on Unsigned Integers

`u-add`	`u-add-constant`	`u-add-carry`
`u-subtract`	`u-subtract-constant`	`u-subtract-borrow`
`u-multiply`	`u-multiply-constant`	`u-add-flags`
`u-divide`	`u-divide-constant`	`u-mod`
`u-max`	`u-max-constant`	`u-rem`
`u-min`	`u-min-constant`	`u-random`
`u-eq`	`u-eq-constant`	`u-eq-zero`
`u-ne`	`u-ne-constant`	`u-ne-zero`
`u-gt`	`u-gt-constant`	`u-gt-zero`
`u-ge`	`u-ge-constant`	`u-ge-zero`
`u-lt`	`u-lt-constant`	`u-integer-from-gray-code`
`u-le`	`u-le-constant`	`u-gray-code-from-integer`
`u-shift`	`u-shift-constant`	`u-integer-length`
`u-abs`	`u-signum`	`u-new-size`
`u-negate`	`u-count-bits`	`u-isqrt`

Every instruction in this group is executed by each active data processor independently of the other data processors. Most of these operations correspond to those listed in the preceding section, but operate on unsigned integers rather than signed integers. Unusual are two instructions that convert values between unsigned binary representation and a binary reflected Gray code representation; these have some utility in the Connection Machine architecture in performing low-level addressing calculations, because the processor addresses used by the router and those used by the NEWS grid are related by a Gray encoding.

3.4.4 Operations on Floating Point Numbers

`f-move`	`f-move-constant`	`f-move-decoded-constant`
`f-add`	`f-add-constant`	`f-square`
`f-subtract`	`f-subtract-constant`	`f-integer-power`
`f-multiply`	`f-multiply-constant`	`f-integer-power-constant`
`f-divide`	`f-divide-constant`	`f-mod`
`f-max`	`f-max-constant`	`f-rem`
`f-min`	`f-min-constant`	`f-random`
`f-eq`	`f-eq-constant`	`f-eq-zero`
`f-ne`	`f-ne-constant`	`f-ne-zero`
`f-gt`	`f-gt-constant`	`f-gt-zero`
`f-ge`	`f-ge-constant`	`f-ge-zero`
`f-lt`	`f-lt-constant`	`f-lt-zero`
`f-le`	`f-le-constant`	`f-le-zero`
`f-scale`	`f-scale-constant`	`f-logb`
`f-abs`	`f-signum`	`f-new-size`

`f-negate`	`f-float-signum`	`f-sqrt`
`f-sin`	`f-asin`	`float-exp`
`f-cos`	`f-acos`	`float-log`
`f-tan`	`f-atan`	`float-atan2`
`f-sinh`	`f-asinh`	`float-power`
`f-cosh`	`f-acosh`	`float-square`
`f-tanh`	`f-atanh`	`float-polynomial`

Every instruction in this group is executed by each active data processor independently of the other data processors. Most of these are floating point operations familiar to any assembly language programmer: arithmetic operations, comparisons, absolute value, negate, scale, and the usual exponential, logarithm, and trigonometric functions.

3.4.5 Type Conversions

`s-floor`	`u-floor`	`s-float`
`s-ceiling`	`u-ceiling`	`u-float`
`s-truncate`	`u-truncate`	
`s-round`	`u-round`	

Every instruction in this group is executed by each active data processor independently of the other data processors. These operations convert between integer (signed or unsigned) and floating point representations.

3.4.6 Intraprocessor Vector Operations

```
f-vector-dot-product
f-vector-3d-cross-product
f-vector-norm
f-matrix-multiply
```

There are two ways to represent vectors and matrices within the Connection Machine memory: one may represent a large vector or matrix by placing one element within each data processor, or one may represent many small vectors or matrices by placing an entire vector or matrix within each data processor.

The operations in this section assume the latter representation. As an example, `f-matrix-multiply` could be used to direct every active processor to multiply two 4×4 matrices. These operations could be expressed in terms of the simple floating point instructions listed in the previous section; they are provided purely for reasons of convenience and performance.

Computing the single-precision dot product of two vectors of length n with the floating point accelerator takes approximately $13n - 5$ microseconds. Assuming that this operation requires $2n - 1$ "flops" (n multiplications and $n - 1$ additions), then the aggregate execution rate for 64K processors is 10,000 MFlops (that is, 10 gigaflops).

3.4.7 Interprocessor Vector Operations

`global-count`	`copy-scan`	`segmented-copy-scan`
`global-logand`	`logand-scan`	`segmented-logand-scan`
`global-logior`	`logior-scan`	`segmented-logior-scan`
`global-s-add`	`s-add-scan`	`segmented-s-add-scan`
`global-s-multiply`	`s-multiply-scan`	`segmented-s-multiply-scan`
`global-s-max`	`s-max-scan`	`segmented-s-max-scan`
`global-s-min`	`s-min-scan`	`segmented-s-min-scan`
`global-u-add`	`u-add-scan`	`segmented-u-add-scan`
`global-u-multiply`	`u-multiply-scan`	`segmented-u-multiply-scan`
`global-u-max`	`u-max-scan`	`segmented-u-max-scan`
`global-u-min`	`u-min-scan`	`segmented-u-min-scan`
`global-f-add`	`f-add-scan`	`segmented-f-add-scan`
`global-f-multiply`	`f-multiply-scan`	`segmented-f-multiply-scan`
`global-f-max`	`f-max-scan`	`segmented-f-max-scan`
`global-f-min`	`f-min-scan`	`segmented-f-min-scan`

Each of these operations takes one datum from each active processor and combines them in some way.

The `global-` operations perform reduction; the set of values, one from each processor, is reduced to a single value through application of a binary combining function. This value is then returned to the front end. For example, `global-s-add` returns to the front end the signed integer sum of all the values, and `global-f-max` treats the items as floating point values and returns the largest one.

The `-scan` operations perform a scan (also called "parallel prefix"). This takes an array of values, one per virtual processor, and replaces each item with the reduction of all items occurring before (and possibly including) that item. For example, if there were eight processors, the argument and result fields might look like this for various operations:

Argument	3	2	6	4	5	11	0	9
Result of exclusive `u-add-scan`	0	3	5	11	15	20	31	31
Result of inclusive `u-add-scan`	3	5	11	15	20	31	31	40
Result of exclusive `u-multiply-scan`	1	3	6	36	144	720	7920	0
Result of inclusive `u-multiply-scan`	3	6	36	144	720	7920	0	0
Result of exclusive `u-max-scan`	0	3	3	6	6	6	11	11
Result of inclusive `u-max-scan`	3	3	6	6	6	11	11	11

On a CM-2 system with 64K physical processors, a `u-add-scan` operation on 64K 32-bit fields takes on the order of 300 microseconds.

The `-scan` operations come in many varieties. One set operates along the NEWS grid, so as to perform many scan operations, one for each row or column in the grid. Another set allows the processors to be segmented into subarrays of differing length,

performing a scan independently within each subarray. The **copy-scan** operation is particularly useful in these cases; within each row, column, or segment it copies a value from the first processor into all the other processors.

3.4.8 General Interprocessor Communication

send	**store**
send-with-overwrite	**store-with-overwrite**
send-with-logior	**store-with-logior**
send-with-logand	**store-with-logand**
send-with-s-add	**store-with-s-add**
send-with-s-multiply	**store-with-s-multiply**
send-with-s-max	**store-with-s-max**
send-with-s-min	**store-with-s-min**
send-with-u-add	**store-with-u-add**
send-with-u-multiply	**store-with-u-multiply**
send-with-u-max	**store-with-u-max**
send-with-u-min	**store-with-u-min**
send-with-f-add	**store-with-f-add**
send-with-f-multiply	**store-with-f-multiply**
send-with-f-max	**store-with-f-max**
send-with-f-min	**store-with-f-min**
get	**fetch**

Each of the **send-** operations takes two fields from each active processor, one containing message data and the other containing the address of a destination processor; each message is deposited into a third field within the memory of the processor specified as the destination for that message.

The plain **send** operation assumes that no processor will receive more than one message. The other **send-** operations cause multiple messages for the same destination to be combined in a specified way; they differ only in the combining operation to be used. Thus **send-with-overwrite** causes one message to be retained and the rest discarded; **send-with-s-add** causes the destination processor to receive the sum of all messages sent to it; and so on.

The **send** operation can process messages at rates varying typically from 80 million to 250 million per second, depending on the communication pattern. For example, if each of 64K processors sends a message to some other processor, the entire operation will take somewhere between 260 and 820 microseconds.

If **send** is viewed as a write into a global shared memory, then **get** is the corresponding read operation.

The **store** operation is like **send**, but the processor sending a message specifies not only which processor is to be the destination but also the memory location into which to deposit the message. This allows a processor to receive more than one message

without combining them; it also supports the abstraction of having completely general pointers into a global shared memory. The **fetch** operation is to **store** as **get** is to **send**.

3.4.9 Communication within a Cartesian Grid

send-to-news	**get-from-news**
send-to-news-bounded	**get-from-news-bounded**

The **send-to-news** operation takes operands that specify a Cartesian coordinate system and a direction within that system, and causes every active processor to send a message to its neighbor in that direction. In the case of a two-dimensional grid the choices are North, East, West, or South, whence the name "NEWS grid." The **get-from-news** operation is complementary: each active processor fetches data from its neighbor. (There is no difference between sending to the West and getting from the East if all processors are active.)

The ordinary NEWS operations actually organize the grid as a hypertorus: the edges "wrap around" so that the West neighbor of a processor on the West edge of the grid is the processor at the East edge of the same row. The **-bounded** versions of the operations do not wrap around; data sent past the boundary of the grid is discarded, and a specified immediate operand is sent in from the opposite boundary. In other words, the plain operations perform a one-place circular shift of each row or column, while the bounded operations perform a one-place end-off shift with a specified value shifted in.

3.4.10 Sorting

s-rank	**u-rank**	**f-rank**

A ranking operation takes one value from each active processor and calculates for each processor the rank of that processor's value in a sorted ordering of all the values. For example, if there were eight processors, the argument and result fields might look like this:

Argument	3	2	6	4	5	11	0	9
Result of u-rank	2	1	5	3	4	7	0	6

If it is then desired to rearrange the values within the processors according to the sorted order, the result of the rank operation may be used as a processor address (or to calculate an address, say within the NEWS grid) for the **send** operation. An advantage of separating the ranking process from the actual rearrangement of the data is that one may perform the ranking step on a small key field and then use the result to reorder a much larger record. This is usually much faster than simply sorting the large records in one step.

On a CM-2 system with 64K physical processors, sorting 64K 32-bit fields (ranking them and then rearranging them) takes about 30 milliseconds.

3.4.11 Data Transfer between Processors and Front End

`s-read-from-processor`	`s-write-to-processor`
`u-read-from-processor`	`u-write-to-processor`
`f-read-from-processor`	`f-write-to-processor`
`s-read-news-array`	`s-write-news-array`
`u-read-news-array`	`u-write-news-array`
`f-read-news-array`	`f-write-news-array`
`s-read-send-array`	`s-write-send-array`
`u-read-send-array`	`u-write-send-array`
`f-read-send-array`	`f-write-send-array`

The `-read-from-processor` and `-write-to-processor` commands allow the front end to read or write a single field within a single data processor. The `-array` commands provide a fast block transfer of many data items, stored one per data proessor in either NEWS-address order or **send**-address order, either to or from a block of memory in the front end.

3.4.12 Housekeeping Operations

`get-stack-pointer`	`get-stack-limit`	`get-stack-upper-bound`
`set-stack-pointer`	`set-stack-limit`	`set-stack-upper-bound`
`push-space`	`pop-and-discard`	`initialize-random`
`cold-boot`	`attach`	`power-up`
`warm-boot`	`detach`	`set-system-leds-mode`

A single global stack pointer is maintained that allows part of the local memory of each data processor to be treated as a stack, typically for the run-time allocation of automatic variables for a compiled high-level language. The operation `push-space` allocates stack space by adjusting the common stack pointer and performs a stack overflow check; the operation `pop-and-discard` deallocates stack space.

The `initialize-random` initializes the pseudo-random number generator used by the operations `s-random`, `u-random`, and `f-random`.

The operations `cold-boot`, `warm-boot`, `attach`, `detach`, and `power-up` are used to initialize the parallel processing unit and to assign sections for use by particular front-end computers.

The `set-system-leds-mode` operation determines whether the red lights on the CM-2 cabinet are to display internal status information or are to be controlled by the user program through the `latch-leds` instruction.

4 Processor Architecture

This chapter describes details of the hardware in the CM-2 parallel processing unit. Most of these details are hidden from the user by the Paris interface and usually are of no concern to the Connection Machine application programmer. However, an understanding of these details is helpful in predicting program performance.

The Connection Machine Model CM-2 parallel processing unit contains thousands of data processors. Each data processor contains:

- an arithmetic-logic unit (ALU) and associated latches
- 64K bits of bit-addressable memory
- four 1-bit flag registers
- optional floating point accelerator
- router interface
- NEWS grid interface
- I/O interface

The data processors are implemented using four chip types. A proprietary custom chip contains the ALU, flag bits, router interface, NEWS grid interface, and I/O interface for 16 data processors, and also contains proportionate pieces of the router and NEWS grid network controllers. The memory consists of commercial RAM chips. The floating point accelerator consists of a custom floating point interface chip and a floating point execution chip; one of each is required for every 32 data processors. A fully configured parallel processing unit contains 64K data processors, and therefore contains 4096 processor chips, 2048 floating point interface chips, and 2048 floating point execution chips, and half a gigabyte of RAM.

4.1 Data Processors

A CM-2 ALU consists of a 3-input, 2-output logic element and associated latches and memory interface. The basic conceptual ALU cycle first reads two data bits from memory and one data bit from a flag; the logic element then computes two result bits from the three input bits; finally, one of the two results is stored back into memory and the other result into a flag. One additional feature is that the entire operation is conditional on the value of a third flag; if the flag is zero, then the results for that data processor are not stored after all.

The logic element can compute *any* two boolean functions on three inputs; these functions are simply specified (by the sequencer) as two 8-bit bytes representing the truth tables for the two functions.

This simple ALU suffices to carry out, under control of the sequencer, all the operations of the Paris instruction set. Consider, for example, addition of two k-bit signed integers. First the virtual processor context flag is loaded into a hardware flag register

(which is then used as the condition flag for all remaining ALU operations). Next a second hardware flag is cleared for use as a carry bit. Next come k iterations of an ALU cycle that reads one bit of each operand from memory and also the carry bit, computes the sum (a three-way exclusive OR) and carry-out (a three-input majority function), and stores the sum back into memory and the carry-out back into the carry flag. These cycles start with the least significant bits of the operands and proceed toward the most significant bits. The last of the k cycles stores the carry-out into a different hardware flag, so that the last two carry-outs may be compared to determine whether overflow has occurred. Arithmetic is therefore carried out in a bit-serial fashion; at about half a microsecond per bit, plus instruction decoding and other overhead, a 32-bit add takes about 21 microseconds. With 64K processors all computing in parallel, this produces an aggregate rate of 2500 Mips (that is, 2.5 billion 32-bit adds per second). All other Paris operations are carried out in like fashion.

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

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

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

4.2 The Router

Interprocessor communication is accomplished in the CM-2 parallel processing unit by special-purpose hardware. Message passing happens in a data parallel fashion; all processors can simultaneously send data into the local memories of other processors, or fetch data from the local memories of other processors into their own. The hardware supports certain message-combining operations: that is, the communication circuitry may be operated in such a way that processors to which multiple messages are sent receive the bitwise logical OR of all the messages, or the numerically largest, or the integer sum.

Each CM-2 processor chip contains one router node, which serves the 16 data processors on the chip. The router nodes on all the processor chips are wired together to form the complete router network. The topology of this network happens to be a boolean n-cube, but this fact is not apparent at the Paris level. For a fully configured CM-2 system, the network is a 12-cube connecting 4096 processor chips. Each router node is connected to 12 other router nodes; specifically, router node i (serving data processors $16i$ through $16i+15$) is connected to router node j if and only if $|i-j| = 2^k$ for some integer k, in which case we say that routers i and j are connected along dimension k.

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

The algorithm used by the router can be broken into stages called *petit cycles*. The delivery of all the messages for a Paris **send** operation might require only one petit cycle if only a few processors are active, but if every processor is active then typically many petit cycles are required. It is possible for a message to traverse many dimensions, possibly all 12, in a single petit cycle, provided that congestion does not cause it to be blocked; the message data is forwarded through multiple router nodes in a pipelined fashion. A message that cannot be delivered by the end of a petit cycle is buffered in whatever router node it happens to have reached, and continues its journey during the next petit cycle. If petit cycles are regarded as atomic operations, then the router may be viewed as a store-and-forward packet-switched network. Within a petit cycle, however, the router is better regarded as a circuit-switched network, where dimension wires are assigned to particular messages whose contents are then pumped through the reserved circuits.

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

Each router node also contains specialized logic to support virtual processors. When a message is to be delivered by a router node, it is placed not only within the correct physical processor, but in the correct region of memory for the virtual processor originally specified as the message's destination.

4.3 The Floating Point Accelerator

In addition to the bit-serial data processors described above, the CM-2 parallel processing unit has an optional floating point accelerator that is closely integrated with the processing unit. There are two possible options for this accelerator: Single Precision or Double Precision. Both options support IEEE standard floating point formats and operations. They each increase the rate of floating point calculations by more than a factor of 20 (see Chapter 15). Taking advantage of this speed increase requires no change in user software.

The hardware associated with each of these options consists of two special purpose VLSI chips, a memory interface unit and a floating point execution unit, for each pair of CM-2 processor chips.

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

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

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

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

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

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

If the virtual processor ratio is N, this process is pipelined so as to require only $3N + 2$ stages instead of $5N$ stages.

5 The Role of the Front End

A front-end computer is a gateway to the Connection Machine system. It provides software development tools, software debugging tools, and a program execution environment familiar to the user. From the point of view of the user, the Connection Machine environment appears to be an extended version of the normal front-end environment. In addition to the usual suite of tools and languages provided by the front end, the environment includes at least one resident compiler or interpreter for a Connection Machine language. The front end also contains specialized hardware, called a Front-End Bus Interface (or FEBI), which allows communication with the Connection Machine.

A front end can be any computer system for which a FEBI exists. At the present time, a FEBI is available for most Digital Equipment Corporation VAX 8000 series minicomputers and for Symbolics 3600 series Lisp machines. The choice of which computer to use as a Connection Machine system front end depends on the nature of the application and on the preferences of the intended users. For example, an artificial intelligence application such as visual object recognition may be most naturally implemented in CM-Lisp, and would therefore work best with a Symbolics front end, whereas scientific applications normally implemented in Fortran would require a VAX front-end computer. Different types of front-end computers may be attached to the same Connection Machine and be running applications simultaneously. In addition, a single front-end computer may contain more than one FEBI to support up to four time-sharing users running Connection Machine applications simultaneously.

The front-end computer serves three primary functions in the Connection Machine system:

- It provides an applications development and debugging environment.
- It runs applications and transmits instructions and associated data to the Connection Machine parallel processing unit.
- It provides maintenance and operations utilities for controlling the Connection Machine and diagnosing problems.

5.1 Applications Development

Users create Connection Machine programs in the development environment provided by the front end. The editors, file systems, and debugging tools are those that are part of the front end's normal environment. The resident Connection Machine language, which contains parallel extensions to a language already familiar to the user, is used to express algorithms exploiting the data parallel structure of a problem. Thus, users with very little experience in data parallel programming may begin to use the Connection Machine immediately.

The native debugging facilities of the front end are augmented by simulators provided as part of the Connection Machine software system. The use of simulators can enhance productivity of users by allowing them to debug application programs, at least in part, without tying up the Connection Machine hardware.

5.2 Running Connection Machine Applications

Once a Connection Machine program has been written, it is executed on the front-end computer. Most statements are translated directly to the native machine code of the front end. Those source-level constructs that correspond to Connection Machine (data parallel) operations are translated to a mix of native machine code and memory operations addressing the FEBI. These are totally transparent to the user.

Data that resides in the Connection Machine need not be returned to the front end immediately. In typical programs, data structures are created in the Connection Machine memory and are used in precisely the same manner as structures in front-end memory. The difference is that operations on the Connection Machine structures can be carried out on many data items in parallel.

Facilities are provided for users to run their programs in interactive or batched mode. Typically the interactive mode will be used during initial program debug, where the user will run the same program repeatedly under control of a debugger, or when the program requires user intervention. Programs that do not require interaction may be placed on a batch queue and run in the background.

5.3 Maintenance and Operations Utilities

The front-end computer also provides utilities to support these functions:

- Allocating and deallocating Connection Machine resources
- Querying Connection Machine system status
- Diagnosing hardware problems

These tools are designed to be compatible with the style and operation of similar tools in the front-end environment.

Information on what segments of the Connection Machine system are in use is made available through status-querying functions. "Attach" and "detach" utilities are provided to allocate and deallocate all or a legal subset of Connection Machine processors to a user logged into a front-end computer. The minimum unit of allocation is whatever is attached to a single sequencer. See Chapter 2 for a description of hardware associated with a sequencer. The following table lists permitted configurations.

Total number of processors	Number of sequencers	Processors per sequencer	Permitted attachable subsets
16K	2	8K	8K, 16K
32K	2	16K	16K, 32K
32K	4	8K	8K, 16K, 32K
64K	4	16K	16K, 32K, 64K

Tools are provided for initializing an allocated sequencer, a procedure known as "booting" the Connection Machine system. There are two levels of initialization provided. The more drastic is "cold boot," which initializes the state of the attached sequencer (including downloading fresh microcode to the sequencer's writable control store) and also initializes the associated Connection Machine data processors (including clearing all memory and initializing per-processor memory-resident global data). The milder form of initialization is called "warm boot," which resets only the state of the *sequencer* without touching Connection Machine processor memory. When debugging programs, "warm boot" can be used to get the sequencer to a known state in order to be able to examine Connection Machine memory after a program crash. Note that neither of these procedures will affect other users running at the same time on other segments of the Connection Machine, nor will they affect unallocated processors.

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

5.4 The Digital Equipment Corporation VAX As a Front End

Currently any Digital Equipment Corporation VAX that contains a VAXBI I/O bus and runs the ULTRIX operating system may be used as a Connection Machine system front end. The VAXBI bus FEBI board provided by Thinking Machines Corporation is designed to allow the user program access to the Connection Machine system sequencer and Nexus registers with minimum system overhead. To accomplish this, the ULTRIX device driver for the FEBI maps the FEBI registers into the address space of the Connection Machine applications program, which then reads and writes the registers as if they were VAX processor memory. Thus, no system overhead at all is incurred in performing Connection Machine I/O. This scheme works especially well with the two-processor VAX computers in the 8000 series, as one processor can be dedicated to running the Connection Machine while the other performs normal time-sharing duties.

All Connection Machine languages are supported in the VAX environment. A VAX front end may contain more than one FEBI (up to four).

5.5 The Symbolics Lisp Machine As a Front End

Any Symbolics 3600 series Lisp machine can be used as a Connection Machine system front end. The FEBI board provided by Thinking Machines Corporation is designed to

allow the user program access to the Connection Machine system sequencer and Nexus registers with minimum system overhead. To accomplish this, the FEBI registers are mapped into into the Lisp address space; a Connection Machine applications program can then read and write the registers as if they were 3600 processor memory. Since Lisp machines are single user workstations, only one FEBI per front end is supported.

The languages currently supported for the Symbolics Lisp machine front end are CM-Lisp, *Lisp, and Paris.

6 Connection Machine I/O Structure

The Connection Machine I/O structure allows data to be moved into or out of the parallel processing unit at aggregate peak rates as high as 320 megabytes per second for a system with multiple I/O controllers. Input/output is done in parallel, with as many as 2K data processors able to send or receive data at a time. All transfers are parity checked on a byte-by-byte basis.

The data processors send and receive data via I/O controllers, which interface through an I/O channel to Connection Machine data lines. These I/O controllers, in turn, operate under the control of the parallel processing unit sequencers. There may be as many as four sequencers in a fully configured system. A maximum I/O configuration for a 64K processor Connection Machine system includes eight I/O channels, each of which permits input and output operations for a set of 8K data physical processors.

An I/O controller treats its 8K physical processors as two banks of 4K. Each CM-2 processor chip contains 16 data processors and has one I/O line, so each bank of 4K processors is implemented on 256 chips and has 256 I/O lines. A bank can therefore pass 256 bits in parallel at a time to its associated I/O controller. Each sequencer controls a bank switch that determines which bank is active.

I/O controllers store data internally in 288-bit chunks (256 data bits plus 32 parity bits). Parity is checked each time data is transferred between a controller and the data processors. Each controller has the ability to store 512 of these 288-bit chunks in its own internal memory. Data transfers between I/O controllers and data processors proceed under control of a Connection Machine sequencer. Two I/O controllers may be active simultaneously on each sequencer.

A Connection Machine I/O bus runs from each I/O controller to the devices it controls. This bus is 80 bits wide (64 data bits, 8 parity bits, and 8 control bits). The I/O controller multiplexes and demultiplexes between 256-bit processor chunks and 64-bit I/O bus chunks. The controller also acts as arbitrator, allocating bus access to the various devices on the bus.

Since standard peripheral devices do not operate at the speeds that the Connection Machine system itself can sustain, it is often desirable to place multiple devices on multiple buses. For example, each of eight disk units could interface to several sections of data processors via several I/O controllers, each disk reading and writing data in parallel with the others. In this way, up to eight times the aggregate transfer rate of a single disk unit is achieved. Alternatively, devices may be interfaced to a single bus, interfaced in turn to I/O controllers in all sections of the parallel processing unit, allowing data to be moved directly between that device and any part of the processing unit. Typical configurations use a mix of these techniques. Some devices are connected to multiple controllers. Others connect to just one controller, and the Conection Machine router is used as necessary to move data to its final destination in the parallel processing unit.

7 The Connection Machine DataVault

The DataVault is the Connection Machine mass storage system. It combines very high reliability with very fast transfer rates for large blocks of data. The DataVault holds five gigabytes of data, expandable to ten gigabytes. It transfers data at a rate of 40 megabytes per second. Eight DataVaults, operating in parallel, offer a combined data transfer rate of 320 megabytes per second and hold up to 80 gigabytes of data.

Each DataVault unit stores its data in an array of 39 individual disk drives. Data is spread across the drives. Each 64-bit data chunk received from the Connection Machine I/O bus is split into two 32-bit words. After verifying parity, the DataVault controller adds 7 bits of Error Correcting Code (ECC) 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 code allows any single bit error to be detected *and* corrected. Although operation is possible with a single failed drive, three spare drives are available to replace failed units until they are repaired. The ECC codes permit 100% recovery of the data on the failed disk, allowing a new copy of this data to be reconstructed and written onto the replacement disk. Once this recovery is complete, the data base is considered to be healed.

The DataVault supports job staging and data base storage. New jobs may be loaded onto the DataVault from external devices such as magnetic tape drives. Once in the DataVault, a maximum-size 512-megabyte memory image may be loaded in under 15 seconds. This same 512-megabyte memory image may be loaded in less than 2 seconds on a system with eight DataVaults operating in parallel. Running jobs may use the DataVault for file storage, opening and accessing files as needed.

7.1 The File Server

All DataVault operations take place under the control of a file server, which is a standard minicomputer. File server commands include creating files, as well as opening, reading, writing, and determining status. Commands to be executed by the file server are passed to it over the Connection Machine I/O bus. Commands such as "open" or "status" that do not involve data transfers are completed by the file server, and a completion message is returned via the I/O bus to the front end.

The file server supports "read" and "write" commands that can specify a field of any size. The data in this field is then transferred between each Connection Machine processor and the DataVault.

In systems with multiple DataVaults, a single master file server controls the file creation and deletion process, although the file itself may be spread across multiple units. Each file server that has a portion of the file maintains a file of disk block locations that allows files to be mapped into disk blocks. These files are not stored on the DataVault itself. They are stored redundantly on two independent file server disks to prevent a single medium failure from blocking access to the file. Two write operations are performed each time the information is changed. When a file is opened,

the block location information is moved to the file server's main memory for faster access during subsequent reads and writes. File space is allocated in blocks of 32K bytes.

7.2 Off-line Loading and Backup

Off-line storage devices (such as magnetic tape) interface directly to the file server minicomputer. New data may be loaded into the DataVault without involvement of the rest of the Connection Machine system. Dumping of DataVault information to magnetic tape for backup also occurs without involving the rest of the system.

7.3 Writing and Reading Data

Data transfers move information between parallel variables in Connection Machine memory and DataVault files. A single read or write moves a specified number of bits (which could correspond to a single parallel variable or to a series of parallel variables that are contiguous in memory) into or out of each Connection Machine virtual processor.

Reading and writing of data are very similar operations. Here, the process of writing data will be described under the assumption that no errors occur.

A write operation is initiated by the front end. The front end issues a write instruction to the appropriate sequencer, which in turn activates the necessary I/O controllers. The request is received by the DataVault file server, which translates the logical file request into a series of physical disk addresses.

Data from Connection Machine memory is moved to the I/O controllers, with parity checked for each byte, and stored in the 288-bit$\times$512 buffer memories on those controllers. When the buffers are sufficiently full, the I/O controller signals its readiness to send data to the DataVault. At this point, the Connection Machine processors are free to proceed with other tasks.

Data in the I/O controllers is split into 64-bit units. Eight parity bits are added and the resulting 72-bit unit is sent on a Connection Machine I/O bus to the DataVault.

Parity of data arriving at the DataVault is checked twice, by two independent sets of logic. If both parity checkers agree that the data is valid, the 64 bits of data are split into two 32-bit words. For each 32-bit word, two independent ECC circuits generate 7 ECC bits for the data. As long as both units generate the same code, the resulting 39 bits are split up and each bit is sent to one of 39 disk buffers. As these buffers fill up, the data is written out to the individual disks.

When all data has been moved from Connection Machine memory through the I/O controllers and the disk buffers and physically written on the disks, a signal is returned to the front end that the transfer is complete.

Data being read into the Connection Machine memory from the DataVault follows the same path as for writing, but in reverse order), through the disk buffers, the I/O bus, and the I/O controller buffers. The data coming off the disks is checked by two

independent ECC circuits. Errors are checked for, corrected, and logged, and the data is written to the I/O bus.

7.4 Drive Failure and Data Base Healing

A transfer status may indicate that a single disk drive is failing and that the ECC has been required to correct the data. At this point, system operation should be interrupted to verify that, in fact, a drive has failed. If it has, it must be switched out of the array and a spare drive switched in. Switching and sparing is done automatically by the DataVault. To assure integrity of the data, the information on the failed disk is reconstructed and written onto the spare. The ECC information stored along with each 32 bits of data allows this reconstruction. Regeneration of this data takes about ten minutes, after which the data is again protected against the failure of another drive.

Repair or replacement of the failed drive allows it to return to active use. Restoration of data at this point is very straightforward. It is only necessary to copy the contents of the spare drive that has been used in the interim. Once this transfer is completed, the repaired drive may be returned to active status. The spare drive is again marked as unused, and the data base is fully healed.

8 High-Resolution Graphics Display

The Connection Machine graphics system consists of a framebuffer module and a high-resolution 19-inch color monitor. The framebuffer, unlike the DataVault, is not connected to a Connection Machine I/O bus; instead it is a single module that resides in the Connection Machine backplane in place of an I/O controller. This direct backplane connection allows the framebuffer to receive data from the Connection Machine processors at rates up to 1 gigabit per second.

The framebuffer contains a large video memory, which holds the actual raster image data. There are 28 planes of memory, divided into 4 buffer areas: red, green, and blue areas having 8 planes each, and an "overlay" area with 4 planes. Each plane provides one bit per pixel, and contains enough memory for 2^{21} (over two million) pixels. There are also three color lookup tables (red, green, and blue). Each color lookup table is 8 bits wide and has 259 entries; the first 256 entries handle data from the red, green, or blue area, and the last 3 entries are used for overlay processing.

The region displayed from the video memory planes is software configurable. Pan and zoom logic allows a specified subrectangle of the video memory to be displayed, magnified by an integral zoom factor. The subrectangle displayed at zoom factor 1 (no magnification) is typically 1280×1024 pixels.

The framebuffer uses 24 bits of data per pixel to produce an analog video signal to be supplied to the monitor. The 24 bits for a pixel may be computed in one of two ways, depending on a software selectable mode. (To simplify the discussion, the effects of the overlay planes are ignored for the moment.)

In 24-bit mode, 24 bits are read from the red, green, and blue planes. For each of the three colors, the 8 bits from the video memory for that color are used as an index into the corresponding color lookup table. The 8 bits read from the color lookup table are then used to produce the analog signal for that color. In the simplest case, entry j of each color lookup table can be initialized to contain the value j ($0 \leq j \leq 255$), so that the values in the video memory in effect drive the digital-to-analog converters directly; but the color lookup tables can be initialized in other ways so as to perform gamma correction for the particular monitor being used.

In 8-bit mode, only 8 bits are read from the video memory for each pixel. The same 8-bit value is used as an index into all three color lookup tables (red, green, and blue); the three table values are then used to produce the analog red, green, and blue signals as for 24-bit mode. In this mode the color lookup tables provide a palette of up to 256 distinct colors. The 8 bits for each pixel may be taken from any one of the color areas (red, green, or blue) of the video memory, depending on a software-controlled submode. (Of course, an image read from from the "red" color area in 8-bit mode is not restricted to shades of red, because each 8-bit pixel value is used to index all three color lookup tables and thereby produce signals for all three colors. In this context the three areas of video memory are called "red" and "green" and "blue" merely for purposes of identifying areas of the video memory within which three distinct images

may be stored.)

Only the 24-bit mode can display images containing more than 256 different pixel color values. The 8-bit display mode does offer two advantages, however. One is that only one-third as much data must be transferred from the Connection Machine processors for each displayed image. The other is that in the 8-bit display mode the framebuffer supports double buffering of output data. While data is being displayed on the monitor from, say, the "red" video memory planes, the "green" planes may be loaded from the Connection Machine processors. Once the data has been completely loaded, a software command can cause the rôles of the "red" and "green" memories to be reversed. The reversal does not occur immediately, but rather during the next vertical retrace. The image in the "green" planes is then displayed, and the Connection Machine processors can begin to load the next image into the "red" planes. In this manner the user never sees parts of two different images on the screen at the same time; the change is synchronized and appears to be instantaneous.

In either 8-bit mode or 24-bit mode, the region of video memory to be displayed on the monitor is defined by a table of address pointers that indicate the starting point within the video memory of each scan line. The framebuffer can hold several of these scan line tables at one time, and can switch between scan line tables during vertical retrace in the same way that it can switch between buffers in double buffered 8-bit mode.

The overlay planes make it easy to overlay a full color image with independent or temporary images such as text labels and cursors. Overlay information may be white, black, or one of the three overlay colors specified in the last three entries of the color lookup tables.

The usual way to organize an image within the Connection Machine memory is one pixel per virtual processor, with the virtual processors organized into a two-dimensional NEWS grid. Any subrectangle of such a NEWS grid may be transferred to the framebuffer for display.

The framebuffer supports a number of output formats under software control. The two principal formats are 1280 × 1024 pixels for a 60 Hz non-interlaced high-resolution monitor, and NTSC format, which is of broadcast resolution and is suitable for use with a standard television monitor or videotape recorder.

9 Languages

The data parallel style of programming associates a processor with every element of a program's data. There are very few differences between a data parallel program and a conventional serial program. In both cases, a single sequence of instructions is used, with a serial control structure. the Connection Machine system provides parallel processing without requiring the applications programmer to indicate synchronization explicitly in programs.

Because the data parallel and serial programming styles are similar, they utilize the same languages. The languages currently supported for the Connection Machine system are C*(pronounced "see-star"), Fortran, *Lisp (pronounced "star-lisp"), and CM-Lisp (pronounced "see-em-lisp"). The Fortran 8x array extensions to Fortran 77 are implemented directly, with no changes to the standard language definition. Each of the other three languages is very close to the corresponding serial language specification, but in each case extends it by adding a new data type; very little new syntax is added, the power of parallelism arising instead from extending the meaning of existing program syntax when applied to parallel data.

There are some broad themes common to any data parallel programming language that are useful to keep in mind when examining a language description:

Establishing Parallel Data Structures. Data parallel programs can be expressed in terms of the same data structures used in serial programs. The difference is that the individual elements of a composite data structure, such as an array, are spread across processing elements, so that each data element has an associated processor. Since each processing element has its own dedicated memory, the task of associating data elements with processing elements is simply the task of assigning memory locations across processors. This assignment is done by the compilers when the array is first declared or created. In C*, the data types in a declaration implicitly specify whether a data structure is parallel. In Fortran, the compiler determines whether an array should be considered serial or parallel according to how it is used in the program. In *Lisp and CM-Lisp, data structures are created dynamically, and different creation operations are used by the programmer to indicate creation of serial or parallel data structures.

Establishing Linkages among Data Elements. During the execution of a program, data from different problem elements are used together. Data parallel programs use pointers or array subscripts to establish connections between processors and hence between their data elements. An array of pointers, itself a parallel data structure, establishes an arbitrary pattern of intercommunication. If the required patterns are regular and local, such as processors sharing data with their nearest neighbors, then no explicit array of pointers is needed because each processor can easily calculate the address of its neighbors as needed.

Establishing Scalar Data. Some data is not parallel. eFor example, it is wasteful to place a copy of a constant in every processor's memory since the constant can be efficiently broadcast as needed from a central point. For this reason, scalar data

(whether constant or variable) is declared as such and stored in the front end.

Operations on Parallel Data Structures. In a data parallel program, a single operation can affect all the elements of a parallel data structure at once, since each data element has its own processor. The same operation in a serial program must be expressed as a loop, with the basic operation applied sequentially to all the elements of the array. Some parallel operations are totally local to individual processing elements. The required data elements are all in the processing element's memory and the result is to be stored there. Other parallel operations have implicit communications cycles imbedded in them since some or all of the required data resides in other processors' memories.

Operations on Mixed Data. Operations that use both scalar and parallel data typically involve replication or reduction. If a scalar value participates in an operation that yields a parallel result (such as adding a constant to every element of an array), the scalar value is replicated by broadcasting it to all processors at once. If parallel data participates in an operation that yields a scalar result (such as finding the sum of all of the elements of an array), a reduction operation is used; given one processor for each data element, such an operation can be completed in time logarithmic in the number of data elements, by organizing the operations on the data into a balanced binary tree. (This organization is carried out by the underlying language implementation.) As with the sequential programming style, data parallel programmers do not need to do anything special when mixing scalar and parallel data.

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

Chapters 10–13 describe the four high-level programming languages for the Connection Machine system. Fortran and C* are the most commonly used languages for numeric applications. CM-Lisp is commonly used for artificial intelligence and other symbolic processing applications. The *Lisp language is significantly "closer to the hardware" than the other three languages; it allows the programmer access to nearly all the hardware features of the Connection Machine system within a framework offering all the convenience and power of the Lisp language. Chapter 14 compares the four languages by presenting versions of the same small program in each of the languages.

All of the CM-2 languages are upward compatible extensions of existing industry standard languages. CM-Lisp and *Lisp extend Common Lisp; C* extends the proposed ANSI standard C language; and Fortran extends Fortran 77 with the proposed ANSI Fortran 8x array extensions. Each language supports all the serial programming constructs defined by the industry standard. The design goal in each case was to maintain the normal programming style of the serial language even for data parallel operations.

10 The C* Language

C* is an extension of the C programming language designed to support programming in the data parallel style, in which the programmer writes code as if a processor were associated with every data element. C* features a single new data type (based on classes in C++), a synchronous execution model, and a minimal number of extensions to C statement and expression syntax. Rather than introducing a plethora of new language constructs to express parallelism, C* relies on existing C operators, applied to parallel data, to express such notions as broadcasting, reduction, and interprocessor communication in both regular and irregular patterns. While C* effectively allows the processing of large arrays of data, it preserves the interchangeability of arrays with pointers, a feature central to the C language. C* relies on pointers for interprocessor communication.

10.1 Data Parallel Machine Model

Just as the C language assumes an abstract machine model with certain interesting abstract properties (sequential execution, uniform address space, meaningful pointer arithmetic), so C* assumes a certain abstract machine model. The C* model is an extension of the plain C model. They share such important features as a uniform address space and meaningful pointer arithmetic. C* extends C by having many processors instead of just one, all executing the same instruction stream. The C* model may be summarized as providing the programmer with *lots of processors* of an otherwise *conventional* nature, operating within a *uniform address space* in a *synchronous execution* mode.

C* assumes a synchronous model of computation, in which all instructions are issued from a single source, a distinguished processor called the *front end*. All the other processors are called *data processors*. At any time, the data processors that are executing the instruction stream sent out from the front end are called the "active set." The local memory of an idle processor does not change, unless another processor writes it.

The layout of memory within each data processor is conventional. Except for the fact that no code is stored in the memory of a data processor, memory is laid out exactly as for a C program in a conventional sequential computer. One end of memory is used to hold statically allocated variables (storage classes `static` and `extern`), and the other end is used as a stack area for the allocation of automatic variables (storage class `auto`).

Processor memory layout can be informally described as a record structure, that is, a C `struct`. (The C language is very good at describing arbitrary memory layouts.) When there are many processors, as in the C* machine model, different processors may have different memory layouts because they may hold different kinds of data for different purposes. If we think of a data processor's memory layout as being a record structure, then we might as well say that a processor's memory really *does* belong to

such a structure type, and we can distinguish groups of processors by that type. In C* a structure type that describes the memory of a data processor is called a domain. The layouts of 26 different processors might be described as follows:

```
domain employee {
    double salary;
    employee_type type;
    char *name;
    int knowledge;
};

domain part {
    int part_number;
    double price;
    vendor *supplier;
    char *description;
};

domain book {
    char *title, *ISBN;
    int content;
    employee *owner;
};

domain employee Fred;                /* Processor 0 */
domain employee Sally;               /* Processor 1 */
domain part grommet;                 /* Processor 2 */
domain employee George;              /* Processor 3 */
domain part wing_nut;                /* Processor 4 */
domain book my_novel;                /* Processor 5 */
domain employee programmer[20];      /* Processors 6-25 */
```

In C*, all code is divided into two kinds: serial and parallel. Code that belongs to a domain is parallel, and may be executed by many data processors at once. Other code is serial, and is executed by the front end as if it were ordinary sequential C code. The two types of code are distinguished by syntactic context: code may belong to a domain (and therefore be parallel) only as the body of a member function of the domain or as the substatement of a selection statement (discussed in section 10.3) that selects the domain. Once the context is established, however, the two types of code are written using the same syntax; parallel code, taken out of context, looks exactly like ordinary sequential C code.

In C*, all data is also divided into two kinds: scalar and parallel. These are described in the language using two new keywords, `mono` and `poly`; they are used

somewhat like the storage class keywords **extern**, **static**, and **auto**, but describe an independent attribute. In certain situations they may sensibly be used in the same way as the **const** and **volatile** keywords of proposed ANSI standard C. Some example declarations:

```
mono int total;
poly int salary;
poly extern float coefficients[10];
mono int *poly x;    /* A poly pointer
                        to a mono integer. */
poly static struct foo x[20];
poly auto double all_the_day;
```

The `mono` or `poly` attribute may be omitted, and usually is, just as the storage class is often omitted in ordinary C code. Within parallel code, the default is `poly`; within serial code, the default is `mono`. (The declaration of `poly` variables is in fact forbidden within serial code, and so the keyword `poly` is required only in pointer-declaration contexts and casts.)

Scalar (`mono`) data resides in the memory of the front end, and parallel (`poly`) data resides in the memory of the data processors. Note that `poly` data is only *potentially* parallel; it is processed in parallel only if referred to by parallel code. It is possible for the front end, executing serial code, to access `poly` data in a sequential manner. Similarly, serial data may be processed by many data processors at once if the front end will first broadcast copies.

Domains differ from classes in that member declarations for domains can use the storage class keywords **auto**, **register**, **static**, and **extern**. In particular, different files can declare different members of a domain, and the **extern** keyword can mark members that are defined in one file but referenced in another. (In contrast, a C++ class may not be declared in such a piecemeal fashion.) Note that **auto** variables in member functions are allocated within each instance, on per-processor stacks. This is all consistent with the fact that the memory of each data processor is organized in the same way as for a sequential C program.

10.2 Parallel Expressions

For convenience, the C* language includes maximum and minimum operators, which are really arithmetic operators. The minimum operator `<?` and the maximum operator `>?` may be applied to pointers as well as to numeric data. By themselves these operators are relatively unimportant, but the assignment operators `<?=` and `>?=` have great utility in C*.

In C* most assignment operators may be used as unary operators. This unary use of existing binary operators is introduced purely for convenience, as an abbreviation for a frequently used and otherwise rather awkward idiom.

Instead of adding new operators for parallel computation, C* takes advantage of the compile-time type distinction between scalar (`mono`) and parallel (`poly`) data, and extends existing operators, through overloading, to operate on parallel data. These extended interpretations allow us to express various interesting patterns of communication:

reading: fetching one value from a particular data processor to the front end
writing: storing a value from the front end into a particular data processor
replication: broadcasting a value from the front end to all data processors
reduction: combining values from all data processors to produce one result
permutation: interprocessor communication (in both regular and irregular patterns)

All these patterns of communication are achieved by using the standard C operators and by adding two rules to the usual rules of C evaluation:

Replication Rule: A scalar value is automatically replicated where necessary to form a parallel value.

As-If-Serial Rule: A parallel operator is executed for all active processors *as if* in some serial order.

The Replication Rule requires that when a binary (or ternary) operator combines `mono` and `poly` data, the `mono` value is replicated before you do the operation. A `mono` value is also replicated if passed as an argument to a function whose corresponding formal parameter is `poly`. In other words, *replication* occurs automatically wherever necessary.

The As-If-Serial Rule is more subtle; it facilitates parallelism by imposing a sequential semantics (while permitting a parallel implementation). The following code segment illustrates the point:

```
double total_salary;
...
{
    total_salary = 0;
    [domain employee].{
        total_salary += salary;
    }
}
```

The second assignment (the one within the selection of `domain employee`) will first replicate the variable `total_salary` as an lvalue; then the processor for every employee will attempt to perform the `+=` operation on its own salary and that same lvalue. The As-If-Serial Rule is a simple way of stating the guarantee that, from the programmer's point of view, the processors do not interfere with each other. The net effect is that every employee's `salary` value has been added into `total_salary` exactly once. This, then, is how *reduction* is expressed in C*. The other C assignment operators may be used in a similar manner.

The other three patterns of communication, namely *reading*, *writing*, and *permutation*, arise naturally from the fact that addressing and the use of pointers in C* is perfectly as general as in C. To put it another way, the language restrictions that one might fear would be imposed because of implementation considerations are *not* imposed after all.

The communication pattern of *reading* is expressed quite simply. Within serial code one might write, for example,

```
strcmp(programmer[2].name, "Jane Jetson");
```

As in the example introduced earlier, `programmer` is an array of twenty employees, and so the elements of this array are instances of the class `employee`, residing in the memories of the data processors. The front end can refer to the `name` component of programmer number 2 simply by referring to `programmer[2].name` in the natural way.

Writing is expressed in exactly the same manner; for example, because the `name` component is public and writable, one can change an employee's name in the obvious way:

```
programmer[2].name = "Jane Eyre";
```

Permutation is also achieved through the natural use of C pointers. Any parallel processor can have pointers into the memory of any other processor. Therefore if `x` is some `poly` variable of type T, and `p` is a `poly` variable of type "pointer to T," then the statement

```
*p = x;
```

means "send message `x` to processor `p`" (or more precisely to a specific variable within a processor, both being indicated by `p`); all active processors do this in parallel.

The use of an explicit pointer variable `p` allows any topological communications pattern to be expressed. The space required for such a pointer may be eliminated in cases where the pattern is sufficiently regular that it may easily be computed "on the fly." Here the ability of the C language to express address arithmetic is valuable; every processor can obtain a pointer to itself (by referring to the variable `this`) and then perform arithmetic on that pointer, allowing all kinds of relative addressing. For example,

```
x = (this+1)->x;
```

causes all `x` values to be shifted downward by one processor (every processor fetched the `x` value from the processor one above it).

10.3 Parallel Statements

C* adds only one new type of statement to C, the selection statement, which is used to activate multiple processors.

All of the standard C statement types may be used in C* in both serial and parallel code. The treatment of control flow in parallel code satisfies the following design goals:

- As long as processors do not interact, the program behaves as if each processor were executing its own code independently. It is as if each of the parallel processes were executing ordinary serial C code.
- When processors do interact, the interactions are completely predictable, deterministic, and repeatable. This is achieved without ever requiring the programmer to write explicit synchronization code.

10.3.1 Selection Statement

The format of a selection statement is as follows:
`[domain` *tag*`] .` *statement*

A selection statement activates all instances of a specified domain and then executes a substatement. (As with the `switch` statement, the substatement may be any statement but in practice it is typically a block.) On completion of the substatement, the instances activated by the selection statement are deactivated.

Within the substatement, the keyword `this` is bound to the primal parallel value: for each active instance, `this` is a pointer to that very instance. Because writing the name of a member variable `memvar` is equivalent to writing `this->memvar`, all references to such a variable also constitute parallel values.

The selection statement is the means by which serial code initiates parallel execution. The selection statement is also used within parallel code; in this case all instances active just before execution of this statement become inactive, and on completion of the statement the same instances become active again.

10.3.2 If Statement

In parallel code, the expression in an `if` statement is treated as a `poly` value, so that each active domain instance has its own value for the test. (If the expression is not `poly`, then one may regard the parallel `if` statement either as behaving like an ordinary serial `if` statement or as first casting the value of the expression to be `poly`, thereby replicating it. These two points of view are equivalent.)

For the statement
`if (` *expression* `)` *statement*

the *statement* is executed with only those instances active whose test value was non-zero.

For the statement

`if (` *expression* `)` *statement* `else` *statement*

the first substatement is executed with only those instances active whose test value was non-zero, and then the second substatement is executed with only those instances active whose test value was zero.

10.3.3 While Statement

On each iteration of the statement

`while` (*expression*) `do` *statement*

the *expression* is calculated as for an `if` statement. Instances that calculate the value zero become inactive; instances that calculate a non-zero value execute the substatement and then loop. The `while` loop completes if and when the active set becomes empty. At that time each individual processor has executed the substatement some number of times, and each may have executed it a different number of times, depending on the data being processed.

If the processors do not interact during the course of the loop, then it is *as if* each processor executes the `while` statement independently, each iterating the appropriate number of times, and then all processors become resynchronized when all have completed.

If the processors do interact, then their interactions are predictable; for example, all processors that execute the substatement as many as three times will all be executing it for the third time together.

10.4 Compiler Implementation

The C* compiler for the Connection Machine computer system is implemented as a translator to ordinary C code that is then compiled by an ordinary C compiler for the front-end computer. The C* compiler parses the C* source code, performs type and data flow analyses, and then translates parallel code into a series of function calls that invoke Connection Machine Paris operations. The use of the front end's usual C compiler allows all the programming tools associated with the front-end programming environment to be applied to C* programs.

11 Fortran

Fortran for the Connection Machine system is a complete implementation of ANSI Fortran 77 as defined by ANSI X3.9-1978, incorporating two sets of extensions: those defined by MIL-STD-1753, and a subset of those proposed in the draft ANSI Fortran 8x standard (draft S8, version 103). Fortran 8x is a data parallel language. The array extensions treat whole arrays as single entities. An array in Fortran 8x is a parallel data structure. The Connection Machine system associates a processor with each element of data in an array.

Newly written Fortran programs can take advantage from the start of the array features from Fortran 8x to implement data parallel algorithms that run efficiently on the Connection Machine. These same programs will run on any other computing system that supports the array handling features described in the draft Fortran 8x standard. Since the Fortran 8x standard is still in a state of flux, future versions of the Fortran implementation for the Connection Machine system may change to reflect the evolving standard.

In the current draft of the Fortran 8x standard, some of the most powerful array handling features have been moved to an appendix and labeled "removed extensions." The removed extensions are not formally part of the standard, but it is intended that any Fortran implementation providing the functionality of these features will follow the definitions given in the appendix. Many of these removed extensions are implemented on the Connection Machine system, among them vector-valued subscripts; the **FORALL** statement, which performs element array assignment; and the instrinsic functions **DIAGONAL**, **REPLICATE**, **RANK**, **PROJECT**, **FIRSTLOC**, and **LASTLOC**.

11.1 The Environment

Fortran runs on a system consisting of a Digital Equipment Corporation VAX computer equipped with a VAXBI bus attached to a Connection Machine. Fortran currently runs under the ULTRIX operating system. A future version will run under the VMS operating system.

Object modules generated by the Digital Equipment Corporation VAX Fortran compiler may be linked with modules produced by the Thinking Machines Fortran compiler without recompilation. This facility is very useful for incorporating existing library routines into a Fortran application, as well as supporting the incremental conversion of an application from serial code to parallel array operations. Routines compiled by the Digital Equipment Corporation VAX Fortran compiler will of course not take advantage of the Connection Machine processors.

A Fortran program may call C routines. In addition, Paris operations may be invoked directly from a Fortran program, through the same interface that is used from C code.

11.2 The Array Extensions of Fortran

The most important difference between Fortran 77 and Fortran 8x is that expressions in Fortran 8x treat entire arrays as atomic objects. The expression

```
A = B + C
```

adds every element of `C` to the corresponding elements of `B` and store the results in `A`. `B` and `C` may be scalars, vectors, matrices, or many-dimensional arrays.

Arrays are stored in the Connection Machine with one element per processor. The array axes map directly onto the multidimensional communications grid of the Connection Machine system. To perform an operation on whole array arguments, the context flags are set so that every processor that contains an element of the arrays in question is enabled. The operation is executed simultaneously in all selected processors.

Most Fortran 77 intrinsic functions are extended to arrays in this element-by-element fashion. Where an elemental function takes two arguments, they must be conforming. Two arrays are conforming if they are the same rank and shape. The Fortran compiler allocates conformable arrays in the same processors, eliminating unnecessary data movement. Scalars may be freely used in array valued expressions. They are automatically replicated to conform with the other arrays in the expression.

In addition to the elemental functions, Fortran includes many functions that inquire about array attributes, perform data reduction, or perform other complicated array operations. Examples of reduction operations are `SUM`, `PRODUCT`, `MAXVAL`, `MINVAL`, `ANY`, and `ALL`. Examples of complex operations are `DOTPRODUCT` (vector dot product) and `MATMUL` (matrix multiplication).

For example, suppose that `M` is a matrix with 30 rows and 56 columns. `SUM(M, DIM=1)` yields a 56-element vector containing the sum of each column. `MAXVAL(M, DIM=2)` yields a 30-element vector containing the largest value from each row.

A reduction operation may take a `MASK` argument, a boolean vector indicating which elements of the array argument are to be included. On the Connection Machine system, the mask is used to subselect processors before the operation is performed.

The transformational intrinsics of Fortran 8x facilitate the treatment of arrays as single data objects. The following are examples of Fortran transformational intrinsics.

`MERGE`	Merge of two arrays according to a mask
`SPREAD`	Replication of an array along a new dimension
`TRANSPOSE`	Transposition of a two-dimensional matrix
`CSHIFT`	Circular shift of an array
`EOSHIFT`	End-off shift of an array

11.3 Fortran Statements for Controlling Context

The `WHERE` statement uses a boolean array as a mask on the elements of a conforming array expression. The expression:

```
WHERE(A .GE. 0) A = SQRT(A)
```

replaces the nonnegative elements of A with their square roots while leaving the negative elements untouched. An alternate form of the WHERE statement specifies what is to happen in the processors that fail the test. For example, in the code:

```
WHERE(B .NE. 0)
  C = A / B
ELSEWHERE
  C = INFINITY
END WHERE
```

where A, B, C, and INFINITY are all conforming arrays, the result of A/B is assigned to C in each processor containing a non-zero element of B, and INFINITY is assigned to C in each processor containing a zero element of B.

The FORALL statement is similar to the WHERE statement except that index expressions may be used instead of (or in addition to) a mask to select the active processors. The following code initializes the matrix H to contain a Hermitian matrix of size N:

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

This code clears the part of the matrix H below the diagonal:

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

Note, in this last example, the use of a mask expression in addition to the index variables I and J.

11.4 Interprocessor Communication in Fortran

The Fortran 8x standard defines the concept of array sections. Array sections may be used anywhere that whole arrays can be used. An array section is the result of extracting selected elements from another array as specified by a subscript expression for each dimension of the array. Suppose that A is a 10×10 matrix. Then A(1:5,1:5) is the upper left quadrant of A, and A(1:5,6:10) is the upper right quadrant; each of these sections is a 5×5 array. The statement

```
A(1:5,1:5) = A(1:5,6:10)
```

therefore copies the upper right quadrant into the upper left quadrant. The section A(1:10:4,1:10:3) is a 3×4 array of elements of A as follows:

```
A(1,1)  A(1,4)  A(1,7)  A(1,10)
A(5,1)  A(5,4)  A(5,7)  A(5,10)
A(9,1)  A(9,4)  A(9,7)  A(9,10)
```

A particularly useful way of describing an array section is by the use of vector-valued subscripts, currently a "removed extension" of Fortran 8x. Vector-valued subscripts may be used to describe general interprocessor communication. If the same index appears more than once in a subscript vector, the result is to communicate the same source value to more than one destination. For example, after the statements

```
S = [1, 2, 2, 3, 3, 3, 2, 2, 1, 7, 1, 1, 3, 3, 3, 2]
V = [15, 20, 25, 30, 35, 40, 45]
B = V(S)
```

the value of `B` will be

```
[15, 20, 20, 25, 25, 25, 20, 20, 15, 45, 15, 15, 25, 25, 25, 20]
```

When this code is executed on the Connection Machine hardware, the values of `V` at the processor addresses specified by `S` are sent to the processors in which the array `B` resides.

11.5 Fortran and the Data Parallel Approach

Fortran has traditionally been a very scalar oriented language. In Fortran 77, operations are defined only on individual scalars; `DO` loops are required to step through a collection of elements performing a given operation on each one. The array extensions in Fortran 8x express program sequences that operate on all the data at once. These extensions are implemented directly in the Connection Machine hardware.

12 The *Lisp Language

The *Lisp language is an extension of Common Lisp for programming the Connection Machine in a data parallel style. It is intended for people who wish to write Connection Machine programs in Lisp that map simply onto Connection Machine hardware features. It supports primitives that correspond directly to the operation of the hardware, and also allows the users to build their own abstractions on top of those primitives. The language is a fully compatible extension of the Common Lisp standard.

Because the primitives of the language correspond very closely to the instruction set of the Connection Machine, is is possible to write code that executes very efficiently.

The parallel primitives of *Lisp support a model of the Connection Machine in which each processor executes a subset of Common Lisp, with a single thread of control residing on the front-end computer. For most Common Lisp functions, *Lisp provides a corresponding parallel function that can operate on all processors, or some selected subsets, simultaneously. In addition, the language provides Lisp-level operators for communicating between processors, both through pointers and in regular patterns. Sequential Common Lisp code, running on the front end, can be freely intermixed with the parallel code executed on the Connection Machine.

Most *Lisp functionality corresponds directly to underlying Paris instructions (see Chapter 3). As a result, the execution speed of a *Lisp program is predictable and easily computed by hand, and direct calls to Paris instructions and special purpose microcode blend in naturally with *Lisp code.

*Lisp provides a safe programming environment. The run time system will signal an error when the user causes a computation to overflow or when a pointer is used illegally. All user type declarations are continuously verified during the execution of the application. This error checking may be turned off for better performance.

The *Lisp implementation consists of an interpreter and a compiler. Both are written in Common Lisp and are transportable to any front-end computer that supports Common Lisp.

12.1 Pvars: The Basic *Lisp Data Object

*Lisp supports all of the standard Common Lisp data types, including symbols, fixed and floating point number, and arrays. It also supports an additional parallel data type called a *pvar* (parallel variable). A *pvar* is a first-class Lisp data type that has value for each processor in the machine. It is similar to an array, except that it is also possible to access and update its elements in parallel.

There are two ways of viewing a pvar. In one model, each processor is simultaneously running the same Common Lisp program, and the pvar represents a variable that exists in all processors and gets operated upon simultaneously in all processors. In the other model, the pvar represents an array whose size is the same as the number of processors in the machine. The elements of the array are located in consecutive processors.

The individual elements of a pvar may contain different types of data. *Lisp supports data of type integer, float, boolean (`t` and `nil`), and pointer. The integer and float types may be of any size supported by Paris. Although integers of any size from 1 to 128 bits are guaranteed to work, most operations work for sizes up to the amount of memory available in a processor.

Like Common Lisp, *Lisp supports run-time type checking, so a *Lisp program requires no declarations. If desired, programmers may insert type declarations to improve performance. *Lisp adheres to the standard Common Lisp declaration syntax.

12.2 Processor Addressing

The Connection Machine supports two different types of communication between processors. One is general pointer-based addressing, and the other is local communication on an n-dimensional grid. For general addressing, each processor is assigned a single number between zero and the number of processors in the machine. For grid addressing, each processor is assigned a vector address. *Lisp provides functions for communication in both modes.

12.3 Reading and Writing Data from and to Pvars

The standard functions for reading or writing the contents of a pvar in a single processor are `pref` and `pref-grid`. The Common Lisp macro `setf` is used in combination with `pref` and `pref-grid` to store data from the front end into a pvar of a processor.

For example, `(setf (pref my-pvar 10) 123.4)` will store the quantity `123.4` into processor `10` of the pvar `my-pvar`. Thereafter `(pref my-pvar 10)` will return `123.4`.

Similarly, `(setf (pref-grid my-pvar 5 7) 111)` will store `111` into pvar `my-pvar` at grid location `(5,7)` (assuming of course that the processors are configured as a two-dimensional grid). Thereafter `(pref-grid my-pvar 5 7)` will return `111`.

12.4 Basic Parallel Operations

All the functions in this section operate only on active processors.

The assignment operator is called `*set`. It takes a destination pvar and a pvar expression whose value is to be stored into that destination.

For example, `(*set pvar1 pvar2)` will store the contents of `pvar2` into `pvar1` in all active processors.

The function `!!` accepts a scalar and returns a pvar that contains the scalar in all active processors.

The statement `(*set pvar1 (!! 5))` will store the quantity `5` into `pvar1` for all active processors.

The functions `+!!`, `-!!`, `*!!`, and `/!!` perform the same operations as the Common Lisp functions `+`, `-`, `*`, and `/`, but in all active processors.

The statement `(*set pvar1 (+!! pvar1 (!! 1)))` will increment the value of `pvar1` in all active processors.

There are many other *Lisp functions for manipulating other types of data. For example, the functions `and!!`, `not!!`, and `or!!` return boolean (`t` or `nil`) quantities in active processors. After the statement

```
(*set boolean-pvar (and!! (=!! pvar1 (!! 5))
                          (>!! pvar2 (!! 100))))
```

is executed, `boolean-pvar` will be `t` in all processors where `pvar1` contains `5` and `pvar2` is greater than `100`.

Most Common Lisp functions have parallel equivalents in *Lisp. Typically, the user thinks of the name of a Common Lisp function and appends the characters "`!!`" to the function name to arrive at the parallel version. The characters "`!!`" are meant to represent the mathematical symbol $||$, which means "parallel." Some of these functions are:

```
mod!!    ash!!    round!!    integerp!!
max!!    min!!    if!!       eql!!
ldb!!    dpb!!    byte!!     numberp!!
```

There are also parallel functions in *Lisp that do not have a corresponding Common Lisp equivalent. For example, `(*set pvar1 (self-address!!))` will set `pvar1` to the send address of each processor, and `(*set pvar1 (self-address-grid!! (!! 1)))` will set `pvar1` to the first component of each processor's vector grid address.

12.5 Selection of Active Sets of Processors

All basic *Lisp functions will compute values only in active processors. Pvars in inactive processors are always left unmodified. Some of the *Lisp macros for manipulating the current set of active processors are `*all` and `*when`. The `*all` construct activates all processors for the block of *Lisp code in its body; the `*when` construct subselects, for the duration of the block of code in its body, all already active processors that satisfy a predicate.

```
(*all (*set pvar1 (!! 10)))     ;store 10 in all processors pvar1
(setf (pref pvar1 100) 0)       ;set processor 100 pvar1 to 0
(*when (/=!! pvar1 (!! 0))
  (*set pvar1 (1-!! pvar1))     ;decrement non-zero values
```

One may nest `*when` and `*all` statements to any depth.

12.6 Communication between Processors

One of the primary strengths of the Connection Machine lies in its communication abilities. The basic functions for using the communication system are `pref!!` and `pref-grid!!`. Whereas `pref` and `pref-grid` allow the front-end computer to serially

read or write the data in a pvar in a single processor, the `!!` versions allow each active processor to simultaneously read/write the value of a pvar in any processor. Even if two or more processors attempt to read the data of a single processor, they all receive the same correct data. (This is supported by the Connection Machine router hardware.)

The following two pieces of code have equivalent effects, although they achieve these effects in different ways:

```
(*all (*set pvar2 (pref!! pvar1 (!! 23))))

(*all (*set pvar2 (!! (pref pvar1 23))))
```

Note that the second form freely mixes serial and parallel code.

Although the previous example used **pref!!** to access the data of a single processor, it may also be used to access data in any processor. For example, the statement

```
(*all (*set pvar2 (pref!! pvar1 (random!! (!! 100)))))
```

causes every processor to make a pseudo-random choice from the first 100 elements of `pvar1` and store the fetched value in `pvar2`.

Some other standard *Lisp routines that use the communication network especially efficiently are `sort!!` and `enumerate!!`.

12.7 Global Reduction Operations

Some *Lisp functions reduce the contents of a pvar in all active processors to a single value, which is then returned to the front-end computer. Examples of this class of functions are `*min`, `*sum`, and `*logior`. For example, `(*all (*sum (!! 1)))` will sum together the quantity `1` in all processors in the Connection Machine. The result will be the number of processors in the particular Connection Machine system being used (actually, the number of virtual processors into which the system has been configured).

12.8 Summary

Each of the categories of functions described above contain many more functions not mentioned here. *The Essential *Lisp Manual* documents them all, as well as memory management, machine initialization and operation, and other related topics.

*Lisp has been successfully used by many Connection Machine programmers. Its simplicity leads to a quick understanding of how to program the Connection Machine efficiently. As an extension of Common Lisp, it is easily learned by those who already know Lisp. Though it provides a small number of abstractions, users frequently build their own with the help of the excellent tools provided by Common Lisp. The result is a productive programming environment for easily exploiting the massive power of the Connection Machine.

13 The CM-Lisp Language

CM-Lisp is a dialect of Common Lisp extended to allow a fine-grained, data-oriented style of parallel execution. This parallelism is organized around objects called *xappings*, which are similar to arrays or hash tables. Two syntactic constructs are introduced: one allows existing Lisp functions to operate on elements of xappings in parallel, and the other provides a means of expressing general interprocessor communication.

CM-Lisp differs from *Lisp in providing higher-level data abstractions, an implementation based on garbage-collected heap storage, and a somewhat more rigorous semantic theory. CM-Lisp is concerned less with providing access to details of the Connection Machine system than with providing convenient programming support for parallel processing of symbolic data structures. *Lisp is therefore the Lisp dialect of choice for the CM-2 when speed is important, whereas CM-Lisp may be more convenient for experimentation with algorithms and rapid prototyping.

13.1 Xappings, Xets, and Xectors

All parallelism in CM-Lisp is organized around a data structure known as a *xapping* (pronounced "zapping," and derived from "mapping"). Xappings are data objects similar in structure to arrays or hash tables, but they have one essential characteristic: operations on the entries of xappings may be performed in parallel. A xapping, like any other Lisp object, may be dynamically allocated, and its associated storage is automatically reclaimed when all references to it are deleted. Xappings may be of arbitrary size (up to some reasonably high implementation-dependent limit), and may contain pointers to any other Lisp objects (including other xappings).

A xapping is an unordered set of ordered pairs. The first item of each pair is called an *index*, and the second item is called a *value*. Pairs are written as *index*→*value*, and all the pairs in a xapping are written surrounded by braces. All the indices in a given xapping must be distinct, but their values need not be (the Common Lisp function `eql` determines sameness). A xapping that maps `yertle` to `turtle` and `horton` to `elephant` would be written like this:

```
{yertle→turtle horton→elephant}
```

The order in which the pairs are written makes no difference. One may think of indices as abstract names for processors in a parallel computer, and of their corresponding values as data stored within those processors. Later we will see how these "abstract processor names" are used to combine data within processors and route data between processors.

The set of all indices of a xapping is called the *domain* of the xapping. The corresponding set of values is called the *range* of the xapping. When a xapping maps each index to itself, the xapping is known as a *xet* (pronounced "zet," and derived from "set"). Pairs in which index and value are the same may be written in an abbreviated notation as just the value, without the index or separating arrow. So, the xet

`{constantinople→constantinople timbuktu→timbuktu}`

can be written as simply

`{constantinople timbuktu}`

If the domain of a xapping consists entirely of consecutive integers starting with zero, then the xapping is known as a *xector* (pronounced "zector," and derived from "vector"). A xector may be abbreviated by writing its values in order surrounded by brackets. So, the xector

`{0→hop 1→on 2→pop}`

can be written as

`[hop on pop]`

Observe that the use of brackets is merely a notational convenience; a xector is the same data structure whether written with braces (and explicit indices) or with brackets.

The number of elements in a xapping may range from zero to infinitely many. The smallest xapping has zero elements; it is called an empty xapping, and is written as {}. At the other extreme are two sorts of infinite xappings:

- A *constant xapping* has the same value for every index. A constant xapping with value v is written as $\{\rightarrow v\}$. The xapping `{→5}` has the value `5` for every index.
- A *universal xapping* maps every index to itself; it is the xet of all Lisp objects. A universal xapping is written as {→}.

Infinite xappings may have a finite number of explicit exceptions, where values for particular indices are specified. The infinite part of the xapping is written after all of the explicit pairs. For example,

`{boy→blue girl→pink →green}`

specifies that `boy` maps to `blue`, `girl` maps to `pink`, and all other objects map to `green`.

13.2 Parallel Computation: α Syntax

The function call mechanism of CM-Lisp allows xappings of functions to be called *as* functions, provided that all arguments to a xapping of functions are themselves xappings. The result of such a function call is a xapping whose domain is the intersection of the domains of the function xapping and argument xappings, and whose range is made up of the results of applying each function to the values from the argument xappings at the corresponding indices. When a xapping of functions is called in this way, the individual function calls may be performed in parallel. Resynchronization occurs, at latest, when all of the parallel computations have completed.

One could perform several additions in parallel by doing this:

```
(funcall '{→+} '[10 20 30 40] '[8 7 6 5 4 3 2])
```

The result of this call is a xapping whose domain consists of the integers 0 through 3 (i.e., the intersection of the infinite domain, the domain of a xector of length 4, and the domain of a xector of length 7), and whose range is formed by calling elements from the function xapping (always +) with corresponding elements from the argument xappings:

```
[18 27 36 45]
```

A special syntax using the alpha character, α, allows us to write parallel function calls more concisely than we did above. The expression αx constructs a constant xapping with the value x. Using α, we can now rewrite the parallel function call shown above like this:

```
(α+ '[10 20 30 40] '[8 7 6 5 4 3 2])
```

Note that a xapping of xappings of functions may be called as a function, too, as long as its arguments are xappings of xappings, and so on.

```
(αα+ '[[1 2 3] [4 5 6] [7 8 9]]
     '[[9 8 7] [6 5 4] [3 2 1]])
   ⇒ [[10 10 10] [10 10 10] [10 10 10]]
```

Consider two forms: `(α+ α2 α3)` and `α(+ 2 3)`. The first evaluates the function and argument forms to produce {→+}, {→2}, and {→3}. An "infinite number" of function calls are set up, all of the form `(+ 2 3)`. All of these calls produce the result 5, and so the result is {→5}. `α(+ 2 3)` simply constructs a constant xapping from the result of the form `(+ 2 3)`, and so the result here is also {→5}. This leads to an important syntactic property: α distributes over function calls.

Suppose there is a need to add 32 to every element of a xapping `c`; one may write `(α+ c α32)`. Now suppose instead that one wishes to multiply each element of `c` by `9/5` before adding `32`; the appropriate code is `(α+ (α* c α9/5) α32)`. Or perhaps the real need is for a xapping of lists pairing each such computed value with the original element of `c`: `(αlist c (α+ (α* c α9/5) α32))`. More complicated expressions contain more and more α operators. The distribution rule can be used to "factor out" these operators if *every* subform of a function call has a preceding α, but that is not the case in the above example.

This problem is solved by using the bullet operator, •, which is an "inverse" to α. By definition, $\alpha \bullet x \equiv x$. Thus, it is possible to apply the distribution law by introducing occurrences of "α•" first. To continue the example, one can begin with the expression `(αlist c (α+ (α* c α9/5) α32))` and make successive transformations:

```
 (αlist   c  (α+  (α*   c α9/5) α32)) ≡
 (αlist α•c  (α+  (α* α•c α9/5) α32)) ≡
 (αlist α•c  (α+ α( *  •c  9/5) α32)) ≡
 (αlist α•c α( +  ( *  •c  9/5)  32)) ≡
α( list  •c  ( +  ( *  •c  9/5)  32))
```

and derive the result `α(list •c (+ (* •c 9/5) 32))`.

This notation is powerful because it allows two simultaneous points of view. On the one hand, it can be understood as a computation with a single thread of control, operating on arrays of data, thereby allowing a global understanding of how the data is transformed. On the other hand, it can be understood as an array of processes, with each process executing the same code that follows the "α" and with "•" flagging data values that may differ among processes. This view allows one to take a piece of code written for a single processor and trivially change it to operate on many processors by annotating it with "α" and "•" in a few places. Thus the notation simultaneously supports both macroscopic and microscopic views of a parallel computation.

Lisp control structure does not consist entirely of function calls—special forms and macros are used very frequently to express variable binding, control flow, and other operations. In general, it is an error to precede the name of a special form or macro with α; however, there are a number of special forms and macros for which parallel execution is both meaningful and useful.

Conditional execution is accomplished with `αif`, a parallel version of the `if` special form. To evaluate the expression (`αif` *condition then else*), one first evaluates the *condition*, which must return a xapping. The next step is to evaluate the *then* expression, but only at the indices for which the *condition* is true. The final step evaluates the *else* expression at indices for which the *condition* is false. The result of `αif` is the union of the *then* and *else* xappings. For example:

```
α(if (oddp •'[0 1 2 3 4 5 6 7 8 9]) 'odd 'even)
   ⇒ [even odd even odd even odd even odd even odd]
```

Note that both consequents of an `αif` are always evaluated, but at disjoint sets of indices (i.e., in disjoint sets of processors).

Local variable bindings may be established in each processor with `αlet`, a parallel version of the `let` special form. Variables are bound to values specified in an initial value xapping for the duration of the `αlet` body, in which each form is evaluated as if it were preceded by α. The result of `αlet` is the result of the last form in the body.

```
α(let ((x •'[0 1 2 3 4 5 6 7 8 9]))
   (* x x x))
   ⇒ [0 1 8 27 64 125 216 343 512 729]
```

The values at a set of indices in a xapping may be altered in parallel with `αsetf`, a parallel version of the `setf` macro. (`αsetf` *old new*) sets the value of each index appearing in both *old* and *new* to the value at that index in *new*.

```
(setq x '{a→1 b→2 c→3})
(αsetf x '{b→5 c→7 d→9})
x ⇒ {a→1 b→5 c→7}
```

13.3 Interprocessor Communication: β Syntax

The α syntax is a way of broadcasting data and programs to different indices (i.e., processors). Another syntax, using the beta character, β, is used to gather data and route it between processors.

The simplest use of β is called *reduction*. The expression (βf x) takes a two-argument function f and a xapping x and returns the result of combining all the values of x using f. For example,

```
(β+ '[0 1 2 3 4 5])
```

returns the sum of all the values in the xector, namely 15. Any two-argument combining function may be used, but the result is unpredictable if the function is not associative and commutative, because the order in which the values are combined is not predictable.

The more complex use of β is called *combination*. (βf d x) takes a binary function f and two xappings d and x and returns a new xapping z whose indices are specified by the values of d and whose values are specified by the values of x. The value of (βf d x) is:

$$\{q \rightarrow s \mid S = \{p \rightarrow r \mid (p \rightarrow r \in x) \wedge (p \rightarrow q \in d)\} \wedge |S| > 0 \wedge s = (\beta f\ S)\}$$

For every distinct value q in d there will be a pair $q \rightarrow s$ in the result. If that value q occurs in more than one pair of d, then s is the result of combining all of the corresponding values from x. For example:

```
(β+ '{toyota→japan gm→usa ford→usa fiat→italy}
     {toyota→135   gm→125 ford→103 vw→164})
   ⇒ {japan→135 usa→228}
```

The pair `usa→228` appears because the values `125` from `gm` and `103` from `ford` were summed by the combining function `+`. The result has no pair with index `italy` or value `164` because neither `fiat` nor `vw` appears as an index in *both* operand xappings.

Reduction may be viewed as a communications operation that sends values from each index (i.e., each processor) of a xapping to some neutral ground, where the values are combined. Combination may be viewed as a distributed form of reduction, in which different values may be sent to different indices (processors) and combined there. This functional similarity between reduction and combination was our motivation for using one character, β, in expressing both operations.

13.4 Library Functions

Common Lisp has a large library of functions for performing useful operations on arrays and lists. CM-Lisp extends the functionality of this library to xappings as well. These "generic sequence functions" allow one to extract subsequences of a sequence, reverse the ordering of elements in a sequence, concatenate sequences, search sequences for particular items or subsequences, sort sequences, and so on.

Common Lisp provides operations on character strings for case conversion, string comparison, and so forth. These functions have also been extended to operate on xappings.

A number of other functions are introduced that provide useful high-level operations on xappings. Perhaps the most interesting of these is **scan**, which is analogous to the $f\backslash x$ operator of APL. The **scan** function takes a combining function and a xector, and returns a new xector in which the value at each index i is the result of reducing the values at indices 0 through i of the original xector with the given function. The scan may be broken up into several separate scans by using the **:segment** keyword argument. The scan will "start from scratch" at each index in the **:segment** xector that has a non-`nil` value. For example:

```
(scan #'max '[1 6 2 7 3 4 2])
   ⇒ [1 6 6 7 7 7 7]
(scan #'max '[1 6 2 7 3 4 2]
      :segment '[t nil nil nil t nil nil])
   ⇒ [1 6 6 7 3 4 4]
```

A number of very simple functions are used idiomatically in reduction, combination, and scanning. These have been given canonical names to save typing: **arg1** always returns its first argument, **arg2** always returns its second argument, **arb** returns either argument unpredictably, and **@** signals an error if called (it is usually used in combination when no collisions are expected).

14 An Example Program

The four high-level programming languages for the Connection Machine system may be compared by examining the four subroutines given below. Each identifies all prime integers below 100,000 by the sieve method, with minor variations dictated by the natural style of the programming language.

The algorithm uses two parallel arrays of boolean (true/false) values called **prime** and **candidate**. At every step element k if **candidate** is true if k has not yet been ruled out as a possible prime. At the beginning of each iteration, the smallest value of k for which **candidate** has a true entry is in fact always a prime; the corresponding element of **prime** may be set to true, and multiples of this value are then eliminated as candidates. The algorithm terminates when no more candidates remain, at which point element j of **prime** is true if j is prime and false if j is not prime.

This example does not show off all the capabilities of the languages or of the Connection Machine system. It is intended merely to illustrate the stylistic differences among the languages. For example, iteration in the C* example is performed by a **while** loop; in Fortran, by a logical **IF** statement with a **GO TO**; and in both the *Lisp and CM-Lisp examples by a Lisp **do** loop, though with different termination tests.

One of the more interesting differences is the way in which each processor calculates its own position within an array. C* has all the facilities of C for performing address calculations; in the example, every processor takes a pointer to itself (represented by the reserved word **this**) and subtracts from it the address **&sieve[0]** of the start of the array of processors, thereby computing its own index within the array. In Fortran, the **FORALL** statement provides each processor with a different value for an index variable. In *Lisp, the built-in function **self-address!!** behaves like the keyword **this** in C*, returning within each processor the address of that processor. In CM-Lisp, the built-in function **iota** takes a number n and generates a xector of n integers from 0 to $n - 1$.

14.1 The Example Program in C*

```
#define N 100000
typedef int bit:1;
domain SIEVE { bit prime; } sieve[N];
void sieve::find_primes() {
    int value = this - &sieve[0];
    bit candidate = (value >= 2);
    prime = 0;
    while (candidate) {
        mono int next_prime = (<?= value);
        sieve[next_prime].prime = 1;
        if (value % next_prime == 0) candidate = 0;
    }
}
```

14.2 The Example Program in Fortran

```
      SUBROUTINE FINDPRIMES(PRIME)
      PARAMETER (N = 99999)
      LOGICAL PRIME(N),CANDIDATE(N)
      PRIME = .FALSE.
      CANDIDATE = .TRUE.
      CANDIDATE(1) = .FALSE.
 20   NEXTPRIME = MINLOC([1,N],CANDIDATE)
      PRIME(NEXTPRIME) = .TRUE.
      FORALL (I = 1:N, MOD(I,NEXTPRIME) .EQ. 0) CANDIDATE(I) = .FALSE.
      IF (ANY(CANDIDATE)) GO TO 20
      RETURN
      END
```

14.3 The Example Program in *Lisp

```
(*defun find-primes ()
  (*all
    (*let ((prime (!! nil)) (candidate (!! t)))
        (*if (<!! (self-address!!) (!! 2))
          (*set candidate nil))
        (do () ((*or candidate))
          (*when candidate
            (let ((next-prime (*min (self-address!!))))
              (setf (pref prime next-prime) t)
              (*when (zerop!! (mod!! (self-address!!) (!! next-prime)))
                (*set candidate (!! nil))))))
        prime))
```

14.4 The Example Program in CM-Lisp

```
(defun primes (n)
  (let ((candidate (make-xector n :initial-element t))
        (primes (make-xector n :initial-element nil)))
    (αsetf candidate '[nil nil])
    (do ((next-prime (position t candidate) (position t candidate)))
        ((null next-prime) primes)
      (setf (xref primes next-prime) t)
      α(setf •candidate
             (and •candidate
                  (not (zerop (mod •(iota n) next-prime)))))))
```

15 Performance Specifications

The specifications in this chapter assume a fully configured Connection Machine Model CM-2 system with 64K data processors and eight I/O channels. Specifications for floating point performance assume the use of a floating point accelerator.

Thinking Machines Corporation believes all specifications are accurate as of the date of publication. Thinking Machines Corporation cannot, however, be responsible for inadvertent errors. Product specifications are subject to change without notice.

15.1 General Specifications

Processors	65,536
Memory	512 megabytes
Memory bandwidth	300 gigabits per second

The memory bandwidth is the maximum sustained transfer rate of data to or from memory.

15.2 Input/Output Channels

Number of channels	8
Capacity per I/O controller	40 megabytes per second
Total I/O controller transfer rate	320 megabytes per second
Capacity per framebuffer	1 gigabit per second

Each I/O channel may support either one general-purpose I/O controller or one framebuffer module. The total I/O controller transfer rate assumes simultaneous use of eight I/O controllers.

15.3 Typical Application Performance (Fixed Point)

General computing	2500 Mips
Terrain mapping	1000 Mips
Document search	6000 Mips

These numbers indicate the averaged performance of the machine on applications for which it is well matched. The numbers are based on actual measurements that include all overhead in the sequencer, the operating system, the front-end user code, and inefficiencies of I/O transfers and algorithm design. The terrain mapping application, for example, cited as running at 1000 Mips, does indeed run approximately 1000 times faster than the same application running on a serial computer rated at 1 Mips.

Mips = Millions of instructions per second

15.4 Interprocessor Communication

Regular pattern of 32-bit messages	250 million per second
Random pattern of 32-bit messages	80 million per second
Sort 65,536 32-bit keys	30 milliseconds

The amount of time required to deliver messages depends on the pattern. A fully loaded random pattern is the worst case that has currently been measured. Sparse message patterns are faster, as are patterns with regular structure, such as grids, trees, or shuffles. The sort time is given here because it is a communication-intensive benchmark.

15.5 Variable Precision Fixed Point

64-bit integer add	1500 Mips
32-bit integer add	2500 Mips
16-bit integer add	3300 Mips
8-bit integer add	4000 Mips
64-bit move	2000 Mips
32-bit move	3000 Mips
16-bit move	3800 Mips
8-bit move	4500 Mips

These numbers indicate the performance of the machine running repeated cycles of the same instructions. The rates include the worst case for all overhead associated with virtual processors. For applications using large numbers of virtual processors per physical processor, the performance will be higher, especially when operating on small fields.

15.6 Double Precision Floating Point

4K×4K matrix multiply benchmark	2500 MFlops
Dot product	5000 MFlops

The 4K×4K matrix multiply benchmark starts with two matrices; approximately 16,000,000 elements each are distributed to the machine. The result is the matrix product. The number includes all communications overhead. The dot product rate is for multiplying two vectors, approximately a hundred elements each, stored within each processor in the optimal format, using the Paris `f-vector-dot-product` operation. This gives an indication of high rates that can be achieved for short periods of time. It is unusual to sustain such rates over the course of a computation. All double precision rates assume the machine is equipped with a double precision floating point accelerator.

MFlops = Millions of floating point operations per second

15.7 Single Precision Floating Point

Addition	4000 MFlops
Subtraction	4000 MFlops
Multiplication	4000 MFlops
Division	1500 MFlops
4K×4K matrix multiply benchmark	3500 MFlops
Dot product	10,000 MFlops

Single precision rates are for a CM-2 equipped with either a double precision or a single precision floating point accelerator. The rates for addition, subtraction, multiplication, and division assume the use of two-address, unconditional Paris instructions with a virtual processor ratio of 32 (2048K virtual processors), and include all instruction issuing and decoding overhead.

See the comments in section 15.6 concerning the 4K×4K matrix multiply benchmark and dot product.

15.8 Parallel Processing Unit Physical Dimensions

Size	56" × 56" × 62"
Weight	2600 lbs.

These dimensions are for the parallel processing unit only and do not include the front-end computer(s), the high-resolution graphics display monitor, or the DataVault mass storage system.

15.9 Parallel Processing Unit Environmental Requirements

Power Dissipation	28 kW
Power Input	Four 30-amp 3-phase 110/208V
Operating Temperature	70°F ± 5°F
Operating Relative Humidity	50% ± 10%

These figures are for the parallel processing unit only and do not include the front-end computer(s), the high-resolution graphics display monitor, or the DataVault mass storage system.