The Connection Machine (CM-2)
An Introduction

Carolyn J. C. Schauble

CU-CS-615-92 October 1992

The Connection Machine (CM-2)
An Introduction

Carolyn J. C. Schauble

CU-CS-615-92 October 1992

University of Colorado at Boulder

Technical Report CU-CS-615-92
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Abstract

This document is intended to introduce students to the Connection Machine (CM-2). Basic concepts of the architecture and of programming in *CM Fortran* are described. In addition, the particular characteristics of the machine located at the National Center for Atmospheric Research (NCAR) in Boulder, CO are given and are used in some examples.

The Connection Machine (CM-2) An Introduction*

Carolyn J. C. Schauble

October 1992

1 Overview

The CM-2[1] Connection Machine[2] is an SIMD (Single-Instruction, Multiple-Data) supercomputer that is manufactured by Thinking Machines Corporation (TMC). Data parallel programming is the natural paradigm for this machine, allowing each processor to handle one data element or set of data elements.

The initial concept of the machine was explained in a Ph.D. dissertation by W. Daniel Hillis [Hillis 85]. The first such commercial computer, called the CM-1[3], was manufactured in 1986. It contained up to 65,536 or 64K processors which could execute the same instruction concurrently. As shown in Fig. 1, sixteen one-bit processors, each allocated 4K bits of memory, were on one chip of the machine. These chips were arranged in a hypercube pattern. Thus, the machine was available in units of 2^d processors, where $d\ =\ 12$ through 16.

One of the original purposes of the computer was artificial intelligence; the eventual goal was a "thinking machine." Each processor was only a one-bit processor. The

*This work has been supported by the National Science Foundation under an Educational Infrastructure grant, CDA-9017953. It has been done by the HPSC Group, Department of Computer Science, University of Colorado, Boulder, CO 80309. Please direct comments or queries to Lloyd Fosdick at this address or e-mail lloyd@cs.colorado.edu.

[1]CM-2 is a trademark of Thinking Machines Corporation.

[2]The Connection Machine is a registered trademark of Thinking Machines Corporation.

[3]CM-1 is a trademark of Thinking Machines Corporation.

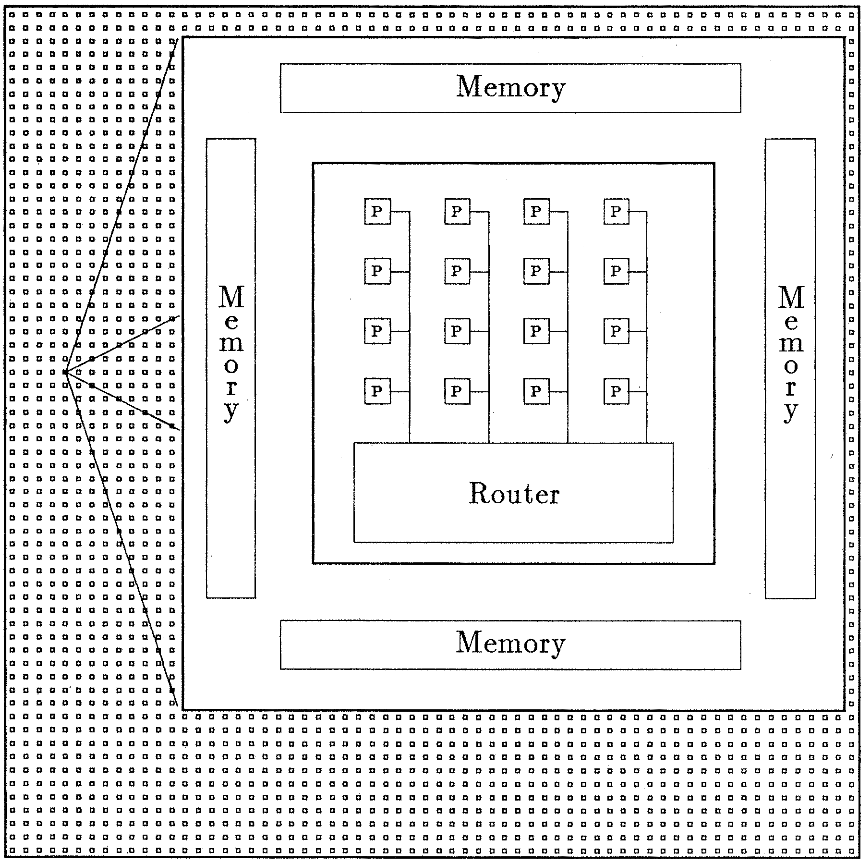


Figure 1: Each CM-1/CM-2 chip contains 16 one-bit processors. Memory chips are associated with each CM-1/CM-2 chip.

idea was to provide one processor per pixel for image processing, one processor per transistor for VLSI simulation, or one processor per concept for semantic networks.

The first high-level language implemented for the machine was **Lisp*[4], a parallel extension of *Lisp*. In fact, the design of portions of the **Lisp* language are discussed in the Hillis dissertation.

However, as the first version of this supercomputer came onto the market, TMC discovered that there was also significant interest (and money) for supercomputers that could be used for numerical and scientific computing. Hence, a faster version of the machine came out in 1987, called the CM-2; this was the first of the CM-200 series of computers. It included floating-point hardware, a faster clock, and increased the memory to 64K bits per processor. These models emphasised the use of data parallel programming. Both *C**[5] and *CM Fortran*[6] were available on this machine, in addition to **Lisp*.

A more recent machine, announced in November 1991, is the CM-5. This is an MIMD machine which embodies many of the earlier Connection Machine concepts with more powerful processors, routing techniques, and I/O.

The following sections will refer to the CM-2. For further information, see the *Connection Machine CM-200 Series Technical Summary* [TMC 91d], *Parallel Supercomputing in SIMD Architectures* [Hord 90], Chapter 7, or *Computer Architecture: Case Studies* [Baron & Higbie 92], Chapter 18.

1.1 Characteristics

A Connection Machine (CM[7]) may be considered as being in two main parts; these are depicted in Fig. 2. The *parallel processing unit* (PPU) is the SIMD portion of the machine and contains up to 64K single-bit processors; this part is the CM itself. The *front end* (FE) of the machine acts as a controller or host for the PPU and provides access to the rest of the world.

The FE is usually a small computer; this may be a *UNIX*[8] or a Symbolics[9] *Lisp* workstation or a VAX[10]. A CM may have up to four FE's; these do not need to all be the same type of machine. Programs are compiled, stored, and executed serially on the FE; any parallel operations in the program are recognized and pipelined to the

[4] **Lisp* is a trademark of Thinking Machines Corporation.
[5] *C** is a registered trademark of Thinking Machines Corporation.
[6] *CM Fortran* is a trademark of Thinking Machines Corporation.
[7] CM is a trademark of Thinking Machines Corporation.
[8] *UNIX* is a trademark of AT&T Bell Laboratories.
[9] Symbolics is trademark of Symbolics, Inc.
[10] VAX is a trademark of Digital Equipment Corporation.

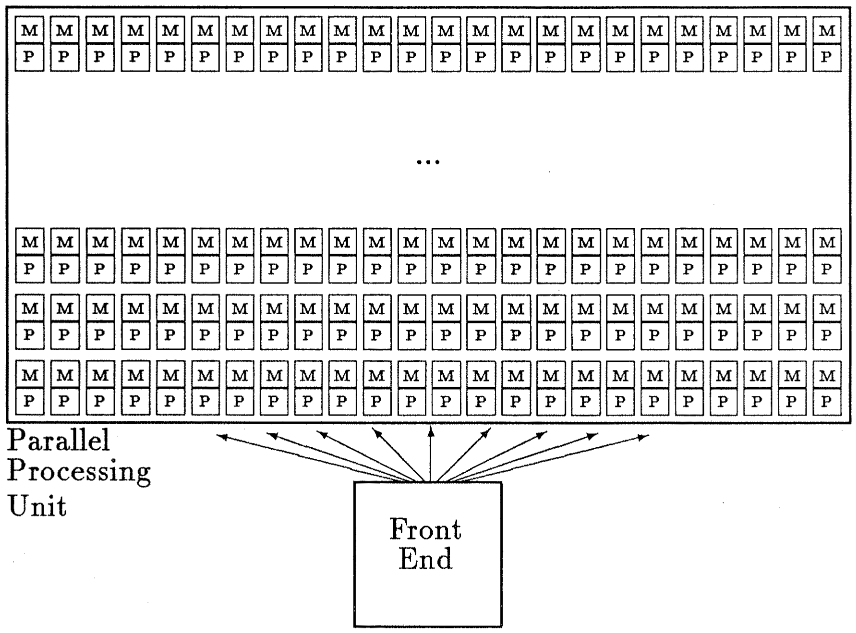


Figure 2: The two main parts of a Connection Machine System.

PPU for execution there, with the serial execution filling the pipeline and continuing until a response is needed back from the PPU. Thus, CM programs have the familiar sequential control flow and do not require additional synchronization primitives, as do other multiprocessors.

Depending on the configuration of the machine, each one-bit, PPU processor has 64K, 256K, or 1024K bits of memory, an arithmetic-logic unit (ALU), four one-bit registers, and interfaces for two forms of communication and I/O. These processors all work in parallel on simple instructions. In fact, the PPU can be thought of as a large, synchronized drill team while the FE is the sargent who yells out the commands. For instance, all the PPU processors fetch something from their individual memories to their ALU's at one command; they all add something to that at the next command; and they all store their individual results into their own memories at the next command.

The language, *Paris*[11] *(PARallel Instruction Set)*, is used to express the parallel operations that are to be run on the PPU. All **Lisp*, *CM Fortran*, or *C** parallel commands are compiled into *Paris* instructions. Such operations include parallel

[11] *Paris* is a trademark of Thinking Machines Corporation.

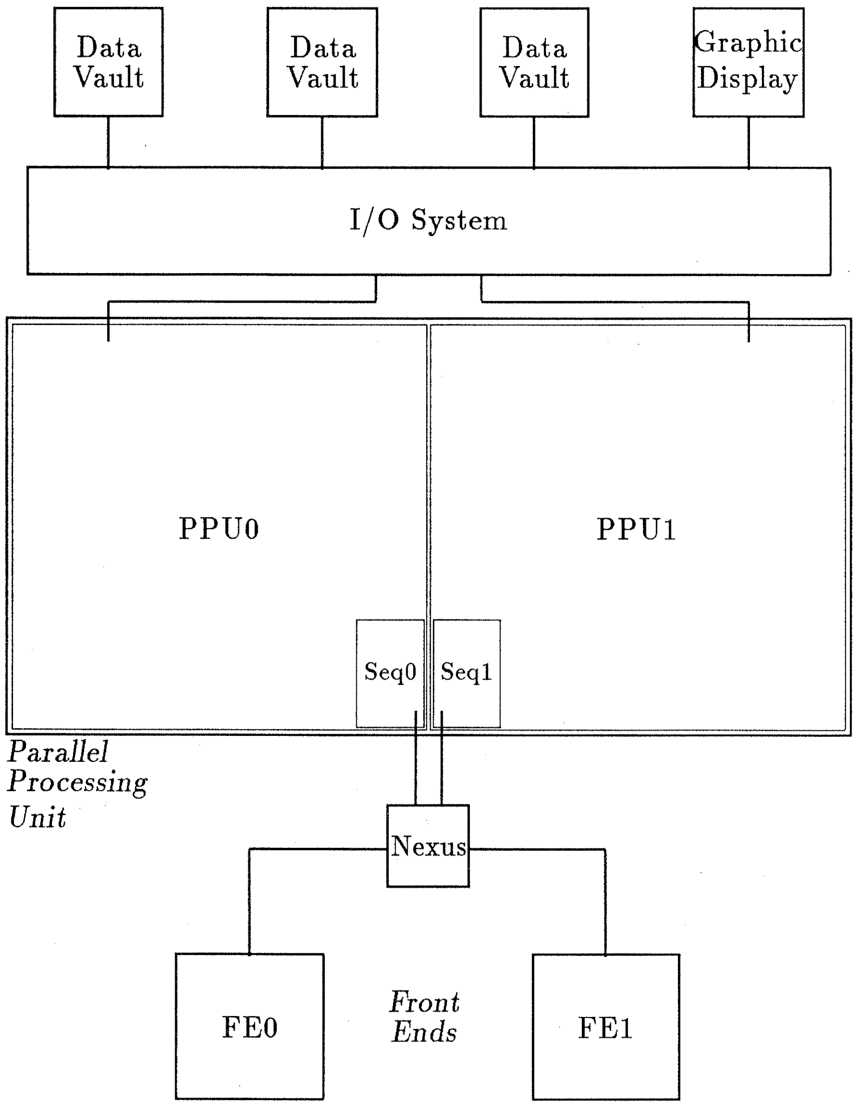


Figure 3: Breakdown of CM with 2 sections and I/O system.

arithmetic operations (both floating-point and fixed), vector summation (and other reduction operations), sorting, and matrix multiplication.

An alternate run-time system exists for machines with 64-bit FPA's. This is called the *Slicewise* model and provides a different viewpoint of the CM than the *Paris* model. This can be used only for *CM Fortran* programs to obtain higher efficiency.

The PPU may be broken up into two or four sections as shown in Fig. 3. Each section has its own *sequencer* and can be used as a sub-PPU by itself or be grouped with other sections. The *Nexus* switch provides the pathway between a given FE and its current section(s) of the PPU.

A sequencer receives *Paris* instructions from the FE and breaks them down into a sequence of low-level instructions which can be handled by the one-bit processors. When that is done, the sequencer broadcasts these instructions to all the processors in its section. Each processor then executes the instructions in parallel with the other processors. When the execution of low-level instructions is completed, control is returned to the FE. Used independently, each section sets up its own grid layout for computation and communication for each array; if the sections are grouped together, one grid per array is laid over all the processors. These grids may be altered dynamically during the execution of the program.

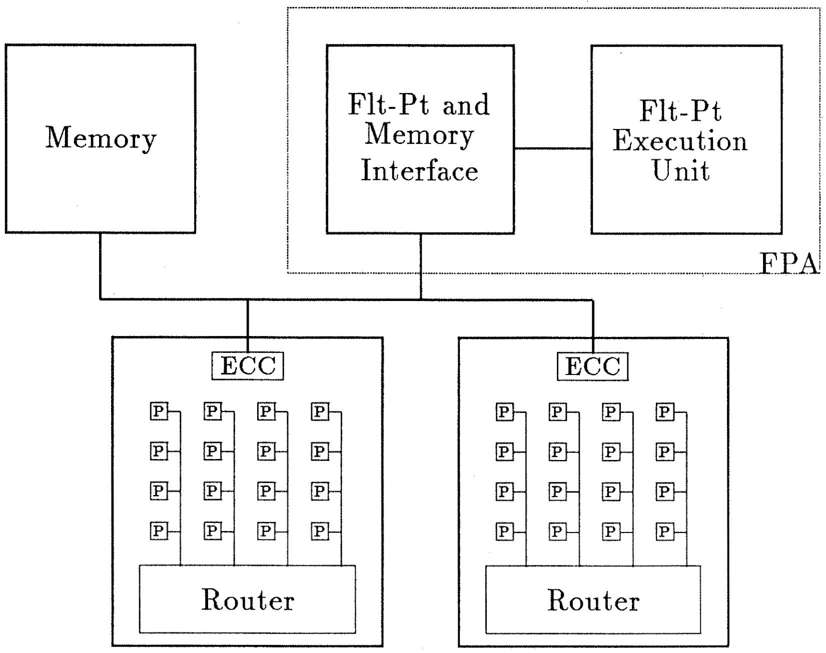


Figure 4: A pair of CM chips communicate with a single FPA.

Virtual processors are another feature of the CM. If the number of elements for the data of a given program is larger than the number of processors, the machine will act as if there were enough processors, providing virtual processors by assigning the data elements across the PPU in as efficient a manner as possible. In other words, each physical processor may act as one or more virtual processors.

Floating-point accelerators are optional and come in two types: 32-bit or 64-bit. A 32-bit FPA interprets the bits from two chips of sixteen one-bit processors as a single floating-point number, allowing single-precision arithmetic computations. A 64-bit FPA also works with the contents of the processors contained on two chips, and provides double-precision arithmetic. For this reason, the processor chips are grouped in pairs, with a floating-point and memory interface attached to each pair, as shown in Fig. 4; it is common to think of each pair of chips as being equivalent to a floating-point processor. The FPA's speed up the processing of floating-point computations on the CM by more than a factor of twenty.

The *Paris* model of computation assigns full 32-bit or 64-bit words to the memory of each processor. These are passed to the FPA bit by bit, when needed. The *Slicewise* model spreads a 32-bit or 64-bit word over all the 32 processors on the two chips connected to a single FPA. When the FPA needs one of these words, each processor can send its portion concurrently with the other processors. Hence, the amount of time spent passing data to and from the FPA is much smaller for the *Slicewise* model.

Data may be read and written serially through the FE to the PPU, but this is far from efficient. An optional input/output system allows peak rates up to 320 Mbytes per second for transfers of data between the PPU processors and the I/O system buffers in parallel. One solution for speeding up peripherals is to attach multiple devices (i.e., disks) to the CM I/O system in parallel. Each device is connected to a separate CMIO bus and may transfer data in parallel with the other devices. Thus, every 64 bits of data (the bandwidth of the bus) is transferred to a different device; this is called *file* or *disk striping*. Alternatively, these I/O buffers may attach to *DataVaults*[12] which contain 5 to 60 Gbytes of data per vault. A maximum of eight data vaults can be on a given system, each transferring data at a peak rate of 40 Mbytes per second or an average rate of 20 Mbytes per second.

Another peripheral available through the I/O system is a *graphical display* for output generated by the PPU; this uses the CM *framebuffer* which can handle up to one Gbyte per second of data. This high-resolution graphical display is a 19" color monitor. The framebuffer for the display attaches directly to the CM backplane, as if it were an I/O controller, and holds actual raster image data. Alternatively, the

[12] *DataVault* is a trademark of Thinking Machines Corporation.

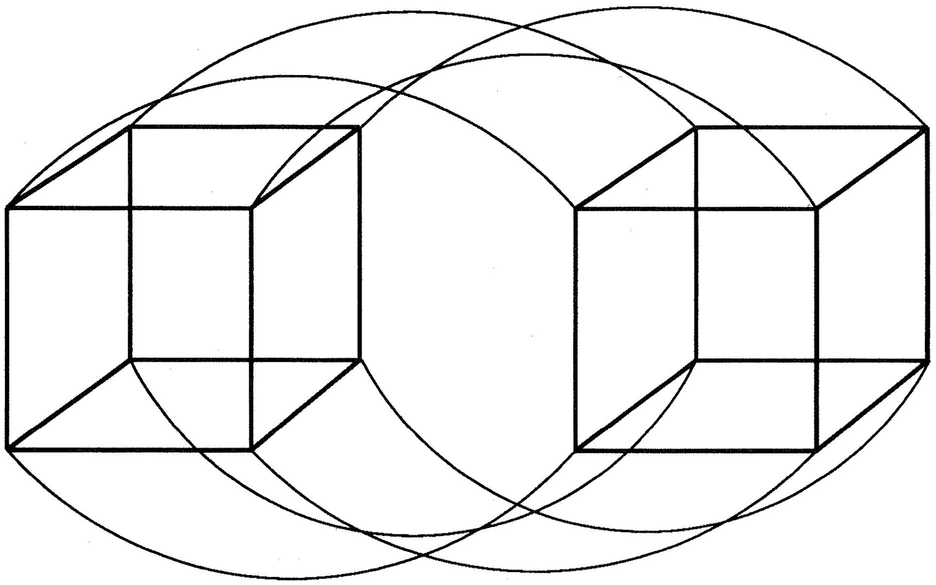


Figure 5: Part of a hypercube network.

image can be displayed on a chosen *X-window*[13]. The size of the window determines the number of pixels; each PPU processor generates data for one pixel. Either 8-bit mode or 24-bit mode can be used.

In addition to the serial communication available between the FE and the PPU, the PPU processors may communicate with each other in different ways. The general form of communication is via the *router* which is present on each chip of sixteen processors. This allows each processor to communicate with any other processor on the PPU and is based on a hypercube network, as shown in Fig. 5. The router also contains the logic to handle virtual processors.

However at times, the data needs to be considered as elements on a mesh or grid spread across the PPU; this second and faster method of communication is called *NEWS* (*N*orth-*E*ast-*W*est-*S*outh) and allows each processor to communicate easily with its nearest neighbors on that mesh. However, it is limited to nearest neighbor communications and cannot be used for broadcasting information to all the processors. An example of a two-dimensional NEWS grid is shown in Fig. 6. While the CM-1 only allowed a fixed, two-dimensional grid, the CM-2 permits NEWS grids with up to 31 dimensions. Each chip on the PPU has an interface to the NEWS grid; processors that have data elements which will be used by or provided by their neighbors store pointers to those neighbors. When two or more sections of the PPU are being used independently, each section may have its own NEWS grid setup. When the sections

[13]X-Window System is a trademark of M.I.T.

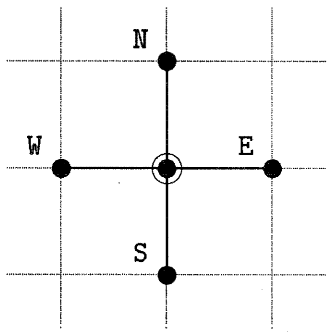


Figure 6: A two-dimensional NEWS grid.

are all ganged together; there is a single NEWS grid across the sections for each array or parallel structure.

Communication of processors on the same chip is called *local* or *on-chip* communication. Clearly, this is faster than the two methods of *off-chip* communication discussed above.

1.2 Performance

A full CM-2 with 64K processors and double-precision FPA's can perform arithmetic operations at the following speeds, under the *Paris* model of execution[14]:

Operation (*Paris* Model)	Single-Precision MFlops
Fl. Pt. Addition	4,000
Fl. Pt. Multiplication	4,000
Fl. Pt. Division	1,500
Dot Product	10,000
4K×4K Matrix Mult	3,500

The speed of memory reads and writes between each processor and its memory is greater than or equal to 5 Mbits/second. Each processor has 64K to 1024K bits or 8K to 128K bytes of memory and a full machine has 64K processors providing a total memory of 512 Mbytes to 8 Gbytes. Hence, given the minimum memory access speed

[14]These figures were taken from the *Connection Machine Model CM-2 Technical Summary* [TMC 88], pp. 59-60.

of a single processor above, the CM-2 can perform memory read/writes at about 300 Gbytes/second.

Other performance tests were done at the University of Colorado using the CM-2 at the National Center for Atmospheric Research (NCAR). This machine is just one-eighth the size of a full CM-2 (having merely 8192 processors) and only contains single-precision FPA's; it also uses the *Paris* model of execution. Standard *CM Fortran* routines were used exclusively for these tests. The results are summarized in the following table:

Operation (*Paris* Model)	Single-Precision MFlops
Fl. Pt. Addition	180
Fl. Pt. Multiplication	200
Fl. Pt. Division	70
Dot Product	56
Cosine	65
Exponential	60
Square Root	83

2 Programming for the CM-2

2.1 Organization of the CM-2

When programming for the CM, it is best to think of the machine in two parts, as shown in Fig. 2:

1. the front end which handles all the scalar operations, and
2. the mesh of 2^d $(12 \leq d \leq 16)$ processors which executes all parallel operations.

For usual scientific computing applications, the front end (FE) is simply a sequential *UNIX* computer or workstation[15]. This is the machine to which the user logs in, the machine which compiles programs for the CM, and the machine which controls the execution of programs on the CM. All program variables which are used only in a sequential or scalar fashion are stored on the front end. The parallel processing unit (PPU), with its SIMD architecture, handles all parallel or array operations, each processor taking care of one piece in unison with the other processors.

Since the editing and compiling of programs are done on the FE, programming on the CM is much like programming on any UNIX machine. In fact, if a program does not use the CM parallel constructs, it will execute entirely on the FE, completely ignoring the PPU.

2.2 Compiling and Executing Programs on the CM-2

In addition to the usual UNIX commands, there are a few special commands relating to the operation of the PPU of the CM. An explanation of a few of these follow:

- `cmf` - this command produces compiled, executable code for both *C* and *Fortran* programs on the CM. The options for this command are much the same as for the UNIX `f77` command; for instance, `-O` is used for optimization and `-o pname` places the executable code in the file `pname` instead of `a.out`. There are additional flags as well; for example, `-list` will produce a program listing in the file named `pname.lis`.

  Normal *C* program files have the file extension, `.c`, and *Fortran*, `.f`. To use the CM programming language extensions, *C* and *Fortran* program files should end with `.peac` and `.fcm`, respectively. The `cmf` compiler is able to recognize the programming language to be compiled from the file extension.

[15]Some installations use Symbolics *Lisp* machines as the FE to the CM.

```
% cmusers
CM           Seqs  Size     Front end  I/F      User     Idle    Command
-------------------------------------------------------------------------
CAPP-CM      ---   ---      capitol     0       (nobody)
CAPP-CM      ---   ---      lizard      0       (nobody)
             cm2 with 64K memory, 32-bit floating point
             framebuffer on sequencer 0 (seq 0 is free)
             CMIOC on sequencer 0 (seq 0 is free)
             2 free seqs on CAPP-CM -- 0 1 -- totalling 8K procs

No processes waiting for an interface or sequencer(s)
% cmfinger
CM           Seqs  Size     Front end  I/F      User     Idle    Command
-------------------------------------------------------------------------
CAPP-CM      0-1   8K       capitol     0       jadoe    0h 00m "starlisp"
CAPP-CM      ---   ---      lizard      0       (nobody)
             cm2 with 64K memory, 32-bit floating point
             framebuffer on sequencer 0
             CMIOC on sequencer 0
             no free sequencers on CAPP-CM

No processes waiting for an interface or sequencer(s)
% cmfinger lizard
CM           Seqs  Size     Front end  I/F      User     Idle    Command
-------------------------------------------------------------------------
CAPP-CM      0-1   8K       capitol     0       jadoe    0h 00m "starlisp"
CAPP-CM      ---   ---      lizard      0       (nobody)
             cm2 with 64K memory, 32-bit floating point
             framebuffer on sequencer 0
             CMIOC on sequencer 0
             no free sequencers on CAPP-CM

No processes waiting for an interface or sequencer(s)
% cmfinger CAPP-CM
CM           Seqs  Size     Front end  I/F      User     Idle    Command
-------------------------------------------------------------------------
CAPP-CM      0-1   8K       capitol     0       jadoe    0h 00m "starlisp"
CAPP-CM      ---   ---      lizard      0       (nobody)
             cm2 with 64K memory, 32-bit floating point
             framebuffer on sequencer 0
             CMIOC on sequencer 0
             no free sequencers on CAPP-CM

1 process waiting:
   schauble [14471] waiting since 1:18:42 PM; wants i/f 0 ucc(s) 0 or 1.
% ...
```

Figure 7: Sample responses for CM commands **cmusers** and **cmfinger**.

The target language of the `cmf` compiler is the assembly language of the FE extended by *Paris* or *Slicewise* commands. All serial or scalar operations will be compiled to run on the FE, while all parallel or array operations[16] will be translated into *Paris* commands to run on the PPU. All scalar or serial data will be stored on the FE and all parallel data (for instance, arrays) will be placed on the PPU.

- `cmattach` - this command connects your executable file to the PPU of the CM, allocates which CM processors are to be used, then executes the program and, finally, releases the processors. It can be thought of as being equivalent to the sequence of three Intel iPSC[17]/2 commands: `getcube`, execute a program, and `relcube`.

  There are a few options on this command. Some common ones are shown in the following statement:

  ```
  cmattach -w -p 8k pname > pname.out
  ```

  In this example, `pname` is the name of the executable file and `pname.out` is the output file. The `-w` flag tells the CM to wait until the needed processors are available; otherwise, if the CM PPU was already occupied, the command would not be executed. The `-p 8k` portion of the command tells the CM how many processors are desired; this could also have been expressed as `-p 8192`.

  When used without an executable file name,

  ```
  cmattach
  ```

  this command starts a subshell with all the resources of the PPU attached. The user can then work interactively.

- `cmusers` – this command provides a list of the users currently using the CM, as shown in Fig. 7. Not only are the names of the users given, but also other information about the machine itself, such as, the size, the type of FPU on the machine (if any), the memory per PPU processor, the number of sequencers (including the number of available sequencers), the number and id of processes waiting for the PPU (if any), and the names of the FE's and the PPU.

- `cmfinger` – this command is also shown in Fig. 7 and returns the same information as the command, `cmusers`. However, the information can be restricted to a particular machine name by using that name as the optional argument to the command.

[16]Section 2.3 will discuss parallel *CM Fortran* operations on the CM-2 in more detail.
[17]iPSC is a registered trademark of Intel Corporation.

For more information on these and other commands, consult the **man** pages on the FE.

2.3 *CM Fortran*

The version of *Fortran* used on the CM is called *CM Fortran.* It is based on *Fortran-77* and extended by parallel constructs, most of which are contained in a subset of *Fortran-90.* The control flow of a *CM Fortran* program is handled by the FE, as are all *Fortran-77* statements. All *Fortran-90* statements are executed on the PPU.

The following subsections will introduce a few elements of *CM Fortran* to get you started. For further reference, see the CM manuals: [TMC 91a], [TMC 91b], and [TMC 91c].

2.3.1 Arrays

CM Fortran arrays may be considered as *data-parallel* objects. In fact, the only *CM Fortran* variables which may be stored on the PPU are those arrays which are used in parallel operations; all scalars and all arrays not involved with parallel operations are stored on the FE of the CM.

The properties of an array are *rank*[18] and *shape.* The rank of an array is the number of its dimensions; e.g., the array declared as `S(5,10)` has rank 2. The shape of an array is its dimensions; so the shape of `S(5,10)` is 5×10. Two arrays with the same shape are said to be *conformable.* Most parallel operations require that the arrays involved be conformable.

Once an array has been declared, the use of the name of the array by itself (not subscripted) denotes the entire array, with all its elements. Such usage implies a parallel operation is to be applied to the array. For instance, the statement

```
S = 0.0
```

will set all the elements of `S` to zero in parallel on the PPU.

Subsections of an array can be specified by *triples.* The general form of a triple is

firstvalue : lastvalue : increment[19]

For example, if `S` is declared as above, then `S(1:5:2,1:10)` refers to the odd rows of `S`. The triple, `1:5:2`, specifies that rows 1, $1+2=3$, and $3+2=5$ are to be used;

[18]In this context, *rank* has a different meaning than its customary mathematical definition.

[19]This differs from the triple form used by Matlab: *firstvalue : increment : lastvalue.*

the triple, 1:10, has an implied increment of 1 and so specifies all ten columns. This second triple, 1:10, could have been replaced by a single colon, as in S(1:5:2,:), to imply that all the columns be used for the chosen rows.

As in *Fortran-77*, *CM Fortran* arrays may be declared by DIMENSION, COMMON, or *type* statements. They may also be declared using *array attribute* statements. For example, assuming the array S is a *real* array, it could have been defined by the following array attribute statement:

```
REAL, ARRAY(5,10) ::  S, T
```

This simply says that both S and T are *real* arrays with 5 rows and 10 columns. Notice the comma after the type indicator, REAL; this is how the array attribute statement is recognized by the compiler. The double colon, ::, is also a requirement; it must be placed between the array definition and the array names. You should recall that blank spaces in *Fortran* are traditionally ignored; hence, any number of spaces can be added to this statement (even between the colons) or deleted from the statement and it will still compile correctly.

Array constructors can be used to initialize the elements of an array in parallel. For instance, if Z has been declared by the statement

```
REAL, ARRAY(N) ::  Z
```

then the statement

```
Z = REAL( [1:N] )
```

will assign 1.0 to Z(1), 2.0 to Z(2), and REAL(N) to Z(N). Array constructors can also be included in array attribute statements, by adding a DATA parameter to the statement; thus, the following has the effect of defining and initializing Z at once:

```
REAL, ARRAY(N), DATA ::  Z = [1:N]
```

This is an efficient way of assigning initial values to an array, as it is effectively done at load time. A limitation on the array constructor is that is can only be used for one-dimensional arrays.

2.3.2 Homes

All program variables in *CM Fortran* programs are assigned a *home*. This is simply where the variable is stored. Since scalar variables can only be on the FE, that is

```
ARRAYS
  Offset     Size  Type         Block/Class      Home      Name
       0     2048  REAL4        local            CM        A
    2048     2048  REAL4        local            CM        B
    4096     2048  REAL4        local            CM        C
```

Figure 8: ARRAYS Section of a *CM Fortran* Listing.

their home. However, arrays may be stored on the FE or on the PPU, depending on whether or not they are used in parallel operations. If they are used in both serial and parallel operations, they will be stored on the PPU and copied to and from the FE for the serial operations. The exception to the rules above is for arrays of type CHARACTER; these are *always* stored on the FE.

To see where homes have been assigned for your variables, check the last part of your program listing. The two sections, VARIABLES and ARRAYS, provide the name, type, and size of the scalar variables and the arrays used in your program. The ARRAYS section also lists the Home for each array. Under this heading, the term CM refers to the PPU and FE to the FE. It is wise to double check this part of the program listing to make sure the arrays have been assigned as expected. A portion of a sample listing showing the ARRAYS section is given in Fig. 8.

Homes for variables in every program unit are assigned individually. That is, the array Z in SUBROUTINE MYSUB may not be assigned the same home as Z in FUNCTION MYFTN. Each program module is treated as a unit.

Homes of actual and dummy arguments must match. If you pass an array to a function or argument, the dummy array argument must have the same home as the incoming array parameter. Otherwise, unpredictable results are possible. There are a number of ways to force arrays to be assigned homes on the PPU:

- Declare the array in COMMON, as all COMMON arrays are placed on the PPU;
- Put parallel operations in every program module for the appropriate arrays (even if they do nothing useful);
- Use the LAYOUT compiler directive:

  ```
  CMF$ LAYOUT Z(:NEWS).
  ```

  With :NEWS as an argument, Z *MUST* be in the PPU. (Other possible arguments are :SERIAL and :SEND. More on the LAYOUT compiler directive will be found in Section 2.4.1.)

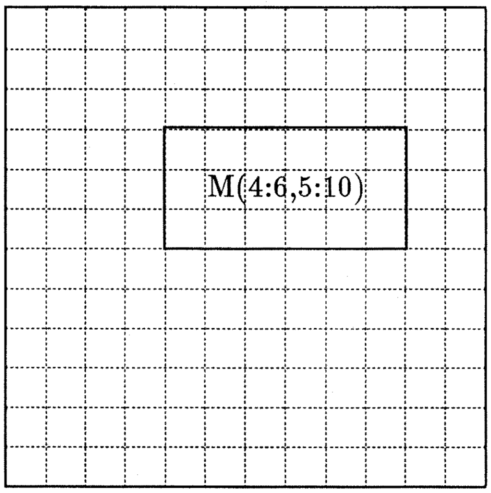


Figure 9: Subsection of array `M(12,12)`.

2.3.3 More *Fortran-90* Extensions

Most of the parallel array facilities of *Fortran-90* are part of *CM Fortran*. This includes the treatment of arrays. As mentioned in Section 2.3.1, the ability to work with all the elements of an array or subsections of an array in parallel is provided; however, all of the arrays involved in this type of parallel expression must be parallel arrays with homes on the PPU.

Array Sections: Using the name of an array implies using the entire array in parallel. For instance, the statement

```
Y = Z**2
```

will cause `Y(1)` to be set to the value of `Z(1)**2`, `Y(2)` to `Z(2)**2`, etc. This also works for assignment statements; the statement

```
Y = -1.0
```

means that all the elements of `Y` will be set to −1.0.

Subsections of arrays can be used in assignment statements as well. In the following statement

```
Y(1:10) = Z(11:20)
```

the first ten elements of Y are set to the second ten elements of Z. Such statements execute in parallel.

An example of a subsection of a two-dimensional array is shown in Fig. 9. Here the array, M has been declared as

```
REAL M(12,12)
```

and the 3×6 subsection is defined by

```
M(4:6,5:10)
```

Alternate DO Loops: Additional control constructs exist in *CM Fortran*; these are similar to those in *Fortran-90*. The first of these are alternate forms of DO loops, as demonstrated below:

```
N = 4096
DOWHILE (N .GT. 0)
   Z(1:N) = ...
   N = N/2
ENDDO

KK = 1
DO (N) TIMES
   KK = KK * K
ENDDO
```

The first of these loops assigns values to the first N elements of the array Z, for N equal to decreasing powers of two. The second loop terminates when KK is equal to K^N, where N is a non-negative integer. This form of the DO loop is useful when the loop index is not needed within the body of the loop. Note that the DO WHILE construct is a legal *Fortran-90* construct; the second form of the DO loop, DO (N) TIMES, is not.

WHERE Statements: The WHERE statements provide a means for working with a subset of a full array, still as a parallel operation:

```
WHERE (Z .GT. 0.0)  Y = SQRT(Z)
```

Here, all processors actually compute Y = SQRT(Z). However, only those processors with a value for Z greater than zero will store the result. Notice that the intrinsic function, SQRT, is used on the entire array. If we wished to set the other (negative and zero) elements to zero at the same time, this operation could be programmed as in the following statements:

```
WHERE (Z .GT. 0.0)
   Y = SQRT(Z)
ELSEWHERE
   Y = 0.0
ENDWHERE
```

In this set of statements, the elements of Y are set to zero when the corresponding element of Z is not greater than zero. Note that this construct acts in two steps. First, all the processors compute Y = SQRT(Z), but only the values for elements of Y corresponding to non-zero elements of Z are stored. Then, all the processors compute Y = 0, but only the values corresponding to zero elements of Z are stored. In other words, the construct appears similar to the IF..THEN..ELSE..ENDIF statement, but behaves a little differently.

FORALL Statements: First, please note that the FORALL construct is not a part of *Fortran-90*; but such statements are very convenient for the CM-2.

The FORALL statements, as in

```
FORALL (I=1:N)   Y(I) = I
```

can only contain one assignment. This statement is equivalent to the following DO loop:

```
DO I = 1,N
   Y(I) = I
ENDDO
```

Notice that there is a colon between the start and stop values of the FORALL index; this is in a *triple* format much like the subscripts discussed earlier. An increment may be used as well, after another colon, as the third element of the triple.

The FORALL statements may execute on either the FE or the PPU, depending on the home of the involved arrays and the nature of the operations. An expression which contains arrays on both units, some of which are on the FE and some of which are on the PPU, will be executed on the FE. A FORALL statement with dependencies that cannot be resolved will also execute on the FE, instead of the PPU.

More than one index can be used within the FORALL statement; for instance, consider the following:

```
FORALL (I=1:N, J=1:N)   S(I,J) = I
```

which sets all the elements of the Ith row of S to I.

Often individual elements of an array need to be initialized to specific values. If the home of the array is on the PPU, it is best to try to use a FORALL statement for this purpose. For instance, the statement

```
S(6,1) = S(6,2) + S(6,3)
```

will be executed on the FE, since it is essentially a scalar operation. However, if we rewrite this statement as a FORALL statement,

```
FORALL (I=6:6, J=1:1)   S(I,J) = S(I,J+1) + S(I,J+2)
```

it will be executed on the PPU. In effect, the sum of S(I,J+1) and S(I,J+2), for each I, and J will be computed by all the elements in the array, but only the processor containing the element in the first column of the sixth row of S will store this value into its array element.

2.3.4 CM Built-In Functions

The *CM Fortran* intrinsic functions are, for the most part, the same as the intrinsic functions described for *Fortran-90*. Some of these are described below.

To aid in the explanation of some of the CM built-in functions, assume the following arrays have the values given below:

$$\mathtt{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

$$\mathtt{B} = \begin{bmatrix} 2 & 4 & 5 \\ 3 & 8 & 5 \end{bmatrix}$$

$$\mathtt{C} = \begin{bmatrix} 1 & -2 & 3 & -4 & 5 & -6 \end{bmatrix}$$

Intrinsic Functions: The usual *Fortran* intrinsic functions are available in *CM Fortran*. Moreover, most of them can be used in a parallel fashion. For instance, if A has been declared as above, then

```
MOD(A,5)
```

will return a matrix of the same type and shape as `A`, containing the values of the elements of `A` *mod(5)*:

$$\texttt{MOD(A,5)} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 0 & 1 \end{bmatrix}$$

Similarly, the `SQRT` function can handle a whole array at once.

$$\texttt{SQRT(A)} = \begin{bmatrix} 1.000 & 1.414 \\ 1.732 & 2.000 \\ 2.236 & 2.449 \end{bmatrix}$$

Masks: *Masks* are logical arrays created by performing a relational operation on all the elements of a given array. Both the mask and the original array have the same shape. Consider the following examples.

$$\texttt{A.GT.0} = \begin{bmatrix} T & T \\ T & T \\ T & T \end{bmatrix}$$

$$\texttt{B.EQ.5} = \begin{bmatrix} F & F & T \\ F & F & T \end{bmatrix}$$

$$\texttt{C.LT.0} = \begin{bmatrix} F & T & F & T & F & T \end{bmatrix}$$

Special Functions: In addition to the normal *Fortran* intrinsic functions, *CM Fortran* provides several special functions to aid in the parallel operation of the machine.

For use in examples of the functions described below, assume the following: `ARRAY` is the name of any array of type real, integer, or logical; `DIM` is an integer denoting which particular dimension of the array (if any) the function is to be applied to; `MASK` is a logical array with the same shape as `ARRAY` telling which particular elements the function is to use; `V`, `V1` and `V2` are singly-dimensioned arrays or vectors; `M1` and `M2` are two-dimensional arrays or matrices; and `SHIFT` is an integer or integer array describing the shift to be made. For some of the following functions, `DIM` and `MASK` may be used as keyword parameters.

Reduction Operations: The following functions perform commonly used reduction operations:

- `SUM (ARRAY [, DIM] [, MASK])`: This function computes the sum of all the elements of `ARRAY`, according to the values of `DIM` and `MASK`. (Note: in the last example, `MASK` is used as a keyword parameter, since the second parameter, `DIM`, is missing.)

```
SUM(A)                = 21
SUM(B, 1)             = [ 5, 12, 10]
SUM(B, 2)             = [ 11, 16]
SUM(C, MASK=C.GT.0)   = 9
```

- `PRODUCT (ARRAY [, DIM] [, MASK])`: This function computes the product of all the elements of `ARRAY`, according to the values of `DIM` and `MASK`. (Note: in the last example, `MASK` is used as a keyword parameter, since the second parameter, `DIM`, is missing.)

```
PRODUCT(A)                = 720
PRODUCT(B, 1)             = [ 6, 32, 25]
PRODUCT(B, 2)             = [ 40, 120]
PRODUCT(C, MASK=C.GT.0)   = 15
```

- `DOTPRODUCT (V1, V2)`: This function computes the dot product of the two vectors or one-dimensional arrays, `V1` and `V2`.

```
DOTPRODUCT(A(1,:),B(:,2))   = 20
DOTPRODUCT(A(:,1),B(2,:))   = 52
DOTPRODUCT(C,C)             = 91
```

- `MAXVAL (ARRAY [, DIM] [, MASK])`: This function finds the maximum value of all the elements of `ARRAY`, according to the values of `DIM` and `MASK`.

```
MAXVAL(A)            = 6
MAXVAL(B(:,1))       = 3
MAXVAL(C)            = 5
MAXVAL(C,1,C.LT.0)   = -2
```

- `MINVAL (ARRAY [, DIM] [, MASK])`: This function finds the minimum value of all the elements of `ARRAY`, according to the values of `DIM` and `MASK`.

```
MINVAL(A)            = 1
MINVAL(B(:,1))       = 2
MINVAL(C)            = -6
MINVAL(C,1,C.GT.0)   = 1
```

- `MAXLOC (ARRAY [, MASK])`: This function returns an integer value or integer array which represents subscripts of the maximum values of all the elements of `ARRAY`, according to the values of `MASK`. If more than one such location exists, which subscript is returned is non-deterministic.

`MAXLOC(A)`	= [3 2]
`MAXLOC(B(:,3))`	= 1 (could also be 2)
`MAXLOC(C)`	= 5
`MAXLOC(C,C.LT.0)`	= 2

- `MINLOC (ARRAY [, MASK])`: This function returns an integer value or integer array which represents subscripts of the minimum values of all the elements of `ARRAY`, according to the values of `MASK`. If more than one such location exists, which subscript is returned is non-deterministic.

`MINLOC(A)`	= [1 1]
`MINLOC(B(:,3))`	= 1 (could also be 2)
`MINLOC(C)`	= 6
`MINLOC(C,C.GT.0)`	= 1

- `COUNT (MASK [, DIM])`: This function returns the number of elements for which the `MASK` held true.

`COUNT(A.GT.0)`	= 6
`COUNT(A.GT.0,1)`	= [3 3]
`COUNT(B.EQ.5)`	= 2
`COUNT(C.LE.0)`	= 3

- `ANY (MASK [, DIM])`: This function returns True if the `MASK` held true for any of the elements.

`ANY(A.GT.0)`	= T
`ANY(A.GT.0,1)`	= [T T]
`ANY(B.EQ.5)`	= T
`ANY(C.LE.0)`	= T

- `ALL (MASK [, DIM])`: This function returns True if the `MASK` held true for all of the elements.

`ALL(A.GT.0)`	= T
`ALL(A.GT.0,1)`	= [T T]
`ALL(B.EQ.5)`	= F
`ALL(C.LE.0)`	= F

Functions for matrices:

- `TRANSPOSE (M1)`: This function returns the transpose of the matrix or two-dimensional array, `M1`.

$$\texttt{TRANSPOSE(A)} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

- `MATMUL (M1, M2)`: This function returns the result of the matrix multiplication of `M1` by `M2`. (Note that the expression, `M1*M2`, does *not* perform matrix multiplication. Instead, it produces element-by-element multiplication; that is, for all i, j $(M1 * M2)_{i,j} = M1_{i,j} * M2_{i,j}$.)

$$\texttt{MATMUL(A,B)} = \begin{bmatrix} 8 & 20 & 15 \\ 18 & 44 & 35 \\ 28 & 68 & 55 \end{bmatrix}$$

$$\texttt{MATMUL(B,A)} = \begin{bmatrix} 39 & 50 \\ 52 & 68 \end{bmatrix}$$

- `DIAGONAL (ARRAY [, FILL])`: This function creates a diagonal matrix from the vector `ARRAY`. The elements of the vector are placed on the diagonal and the value of `FILL` (if any) is placed in the other elements of the matrix. If there is no `FILL` value, the value of 0 (or `.FALSE.`, if logical) is used.

$$\texttt{DIAGONAL(C)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & -6 \end{bmatrix}$$

$$\texttt{DIAGONAL(C,99.0)} = \begin{bmatrix} 1 & 99 & 99 & 99 & 99 & 99 \\ 99 & -2 & 99 & 99 & 99 & 99 \\ 99 & 99 & 3 & 99 & 99 & 99 \\ 99 & 99 & 99 & -4 & 99 & 99 \\ 99 & 99 & 99 & 99 & 5 & 99 \\ 99 & 99 & 99 & 99 & 99 & -6 \end{bmatrix}$$

Other useful functions:

- `RANK (ARRAY)`: This function returns the rank of the given scalar or `ARRAY`.

$$\begin{array}{ll}
\texttt{RANK(100)} & = 0 \\
\texttt{RANK(A)} & = 2 \\
\texttt{RANK(B)} & = 2 \\
\texttt{RANK(C)} & = 1
\end{array}$$

- `DSHAPE (ARRAY)`: This function returns the shape of the given scalar or `ARRAY`.

$$\begin{array}{ll}
\texttt{DSHAPE(-1)} & = [\,] \\
\texttt{DSHAPE(C)} & = 6 \\
\texttt{DSHAPE(A)} & = [\ 3\ 2\ ] \\
\texttt{DSHAPE(B)} & = [\ 2\ 3\ ]
\end{array}$$

- `REPLICATE (ARRAY, DIM, NCOPIES)`: This function adds `NCOPIES` of the `ARRAY` along the given `DIM`ension. The resultant array has the same rank as the original `ARRAY`, but the shape in greater in the given `DIMENSION`.

$$\texttt{REPLICATE(A,1,2)} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

$$\texttt{REPLICATE(A,2,3)} = \begin{bmatrix} 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \\ 5 & 6 & 5 & 6 & 5 & 6 \end{bmatrix}$$

$$\begin{array}{ll}
\texttt{REPLICATE(A, 1, 0)} & = [\,] \\
\texttt{REPLICATE(C, 1, 2)} & = [\ 1\ \text{-}2\ 3\ \text{-}4\ 5\ \text{-}6\ 1\ \text{-}2\ 3\ \text{-}4\ 5\ \text{-}6]
\end{array}$$

- `SPREAD (ARRAY, DIM, NCOPIES)`: This function makes `NCOPIES` of the `ARRAY` along `DIM`. The resultant array has rank one greater than that of the original `ARRAY`. This can also be used to make a vector from a scalar.

$$\begin{array}{ll}
\texttt{SPREAD(-1, 1, 6)} & = [\ \text{-}1\ \text{-}1\ \text{-}1\ \text{-}1\ \text{-}1\ \text{-}1\ ] \\
\texttt{SPREAD(-1, 1, 0)} & = [\,]
\end{array}$$

$$\texttt{SPREAD(A,1,2)} = \begin{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \\ \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \end{bmatrix}$$

$$\texttt{SPREAD(A,2,3)} = \begin{bmatrix} \begin{bmatrix} 1 & 2 \\ 1 & 2 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 3 & 4 \\ 3 & 4 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 5 & 6 \\ 5 & 6 \end{bmatrix} \end{bmatrix}$$

$$\texttt{SPREAD(C,1,2)} = \begin{bmatrix} 1 & -2 & 3 & -4 & 5 & -6 \\ 1 & -2 & 3 & -4 & 5 & -6 \end{bmatrix}$$

$$\texttt{SPREAD(C,2,3)} = \begin{bmatrix} 1 & 1 & 1 \\ -2 & -2 & -2 \\ 3 & 3 & 3 \\ -4 & -4 & -4 \\ 5 & 5 & 5 \\ -6 & -6 & -6 \end{bmatrix}$$

- **PACK (ARRAY, MASK [, V]):** This function gathers elements from the ARRAY, under the control of the MASK. If the vector, V, is specified, the result is placed on top of the values already there.

```
PACK(B, B.GT.4)       = [ 8 5 5 ]
PACK(A, A.GT.3, C)    = [ 5 4 6 -4 5 -6 ]
PACK(A, A.GT.0, C)    = [ 1 3 5 2 4 6 ]
```

- **UNPACK (V, MASK, ARRAY):** This function scatters elements from the vector, V, under the control of the MASK, into the ARRAY.

```
UNPACK(C, C.LT.0, [6[0.0]])    = [ 0 1 0 -2 0 3 ]
UNPACK(C, C.LT.0, [6[-1.0]])   = [ -1 1 -1 -2 -1 3 ]
UNPACK(C, C.GT.0, C)           = [ 1 -2 -2 -4 3 -6 ]
```

- **CSHIFT (ARRAY, DIM, SHIFT):** This function does a *Circular SHIFT* on ARRAY, returning a result which has the same type and shape as ARRAY.

```
CSHIFT(C, 1, 1)    = [ -2 3 -4 5 -6 1 ]
CSHIFT(C, 1, -1)   = [ -6 1 -2 3 -4 5 ]
CSHIFT(C, 1, 2)    = [ 3 -4 5 -6 1 -2 ]
CSHIFT(C, 1, -3)   = [ -4 5 -6 1 -2 3 ]
```

$$\texttt{CSHIFT(A,1,2)} = \begin{bmatrix} 5 & 6 \\ 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$\texttt{CSHIFT(A,2,-1)} = \begin{bmatrix} 2 & 1 \\ 4 & 3 \\ 6 & 5 \end{bmatrix}$$

$$\texttt{CSHIFT(A,1,[0,1])} = \begin{bmatrix} 1 & 4 \\ 3 & 6 \\ 5 & 2 \end{bmatrix}$$

- `EOSHIFT (ARRAY, DIM, SHIFT [, BOUNDARY])`: This function does an *End-Off SHIFT* on `ARRAY`, returning a result which has the same type and shape as `ARRAY`. The value of `BOUNDARY` (if any) is used to fill up the spaces made by shifting away from the edges; otherwise, zero (or `.FALSE.`) is used.

$$\begin{array}{ll} \texttt{EOSHIFT(C, 1, 1)} & = [\,\text{-2 3 -4 5 -6 0}\,] \\ \texttt{EOSHIFT(C, 1, -1)} & = [\,\text{0 1 -2 3 -4 5}\,] \\ \texttt{EOSHIFT(C, 1, 3, 9.0)} & = [\,\text{-4 5 -6 9 9 9}\,] \end{array}$$

$$\texttt{EOSHIFT}(\texttt{A}, 2, -1) = \begin{bmatrix} 0 & 1 \\ 0 & 3 \\ 0 & 5 \end{bmatrix}$$

$$\texttt{EOSHIFT}(\texttt{A}, 1, [-1, 0]) = \begin{bmatrix} 0 & 2 \\ 1 & 4 \\ 3 & 6 \end{bmatrix}$$

$$\texttt{EOSHIFT}(\texttt{A}, 1, 2, 99.0) = \begin{bmatrix} 5 & 6 \\ 99 & 99 \\ 99 & 99 \end{bmatrix}$$

$$\texttt{EOSHIFT}(\texttt{A}, 1, 2, \texttt{REAL}([1:2])) = \begin{bmatrix} 5 & 6 \\ 1 & 2 \\ 1 & 2 \end{bmatrix}$$

There are many additional intrinsic functions which can be used to manipulate arrays in parallel, such as, `RESHAPE` and `PROJECT`. Refer to the *CM Fortran Reference Manual* [TMC 91b] for more information on these and other functions. A reference book on *Fortran-90*, such as the *Fortran 90 Handbook* [Adams et al. 92] or the *Programmers Guide to Fortran 90* [Brainerd et al. 90], may be helpful as well.

2.4 Compiler Directives

The compiler directives in *CM Fortran* are used in two ways. The first is to specify the location of the elements of arrays with respect to each other. The second is to specify the use and homes of arrays in the common blocks of the program.

All the *CM Fortran* compiler directives begin with the characters, `CMF$`; they will appear as comment lines to other *Fortran* compilers. The continuation of such lines, if needed, differs from the normal *Fortran* statement continuation; since the compiler directives look like comments, an ampersand, `&`, needs to be placed on the end of the line which is to be continued, instead of the continuation character in column six,

There are three compiler directives in *CM Fortran*:

```
CMF$ LAYOUT args
CMF$ ALIGN args
CMF$ COMMON args
```

We will briefly discuss the use and purpose of each here.

2.4.1 LAYOUT

In this section and the next, a reference to a processor will mean a virtual processor. If an array has more elements than the number of physical processors, each processor will act as a virtual processor to more than one element. The ordering of the assignment of elements to virtual processors depends on the layout or ordering chosen for the given dimension of the array.

In the normal or default allocation of arrays on the PPU, each element will be placed on a single processor. When there are more elements than processors, each element is mapped to a virtual processor. The intent is to allow each processor to have its own piece of the action. Such a mapping for the array, V(0:N), is shown in Fig. 10.

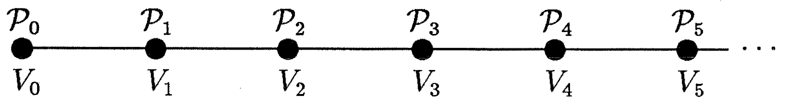


Figure 10: Normal layout for the array, V(0:N).

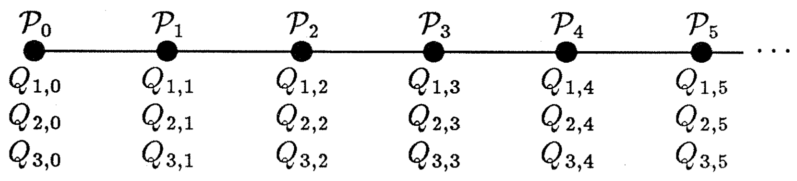
However, there are algorithms where it is more efficient to allow each processor to have a set of elements of a given array, as those elements will be used together in the same computation. For instance, in the case of arrays of two or more dimensions, it may be best to have all the elements of one of the dimensions be placed on the same processor. For instance, suppose the array, Q, is declared by the following DIMENSION statement

```
DIMENSION  Q(3,0:N)
```

Further, suppose that

$$\mathtt{Q}_{3,i} = \mathcal{F}(\mathtt{Q}_{1,i}, \mathtt{Q}_{2,i})$$

Then it might be desirable to have all three elements of each column of the array on the same processor. This type of layout is shown in Fig. 11. The compiler directive which assigns this layout is



$\mathcal{P}_0$ $\mathcal{P}_1$ $\mathcal{P}_2$ $\mathcal{P}_3$ $\mathcal{P}_4$ $\mathcal{P}_5$ ...
$Q_{1,0}$ $Q_{1,1}$ $Q_{1,2}$ $Q_{1,3}$ $Q_{1,4}$ $Q_{1,5}$
$Q_{2,0}$ $Q_{2,1}$ $Q_{2,2}$ $Q_{2,3}$ $Q_{2,4}$ $Q_{2,5}$
$Q_{3,0}$ $Q_{3,1}$ $Q_{3,2}$ $Q_{3,3}$ $Q_{3,4}$ $Q_{3,5}$

Figure 11: Layout for the array, `Q(:SERIAL,:NEWS)`.

```
CMF$ LAYOUT  Q(:SERIAL,:NEWS)
```

Here the term, `SERIAL`, means that the elements on the array in this dimension should all be on the same processor. The term, `NEWS`, implies that the elements of the second dimension (each column) should be spread across the processors in an order to allow efficient communication between nearest neighbors.

The general format of this directive is

CMF$ LAYOUT ARRAY(*weight*1:*order*1,*weight*2:*order*2,..., *weight*N:*order*N)

This specifies an order and a weight to that order for each dimension of the given `ARRAY`. The order is as defined above; the weight is any constant expression that indicates the importance of that ordering for the given dimension in relation to the other dimensional orders. If the weight is missing, it is assumed to be `1`; if the order is missing, it is assumed to be `NEWS`. The sample statement above,

```
CMF$ LAYOUT  Q(:SERIAL,:NEWS)
```

provides no weights. Weights have no meaning for serial ordering since the elements in that dimension should all be on the same processor.

There are three different types of *orders*. As mentioned in the last paragraph, the `SERIAL` order tells that compiler that all elements of the given dimension should be arranged sequentially on the same processor.

The `NEWS` order is the default ordering; it specifies that elements of the given dimension of the array should be stored on processors in such a way as to provide quick nearest neighbor communication. This is most often used in grid computations, where each element of the array needs to be updated using the values of its neighboring elements, as in the following two-dimensional expression:

$$X_{i,j} \;=\; \mathcal{F}(X_{i,j}, X_{i-1,j}, X_{i+1,j}, X_{i,j-1}, X_{i,j+1})$$

This two-dimensional form of a grid is shown in Fig. 12. So, the default layout

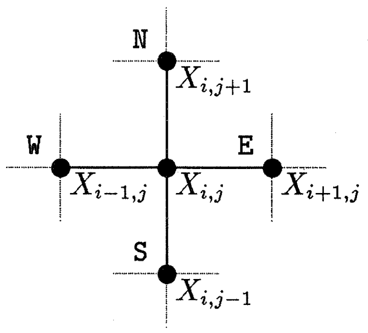


Figure 12: A two-dimensional `NEWS` grid.

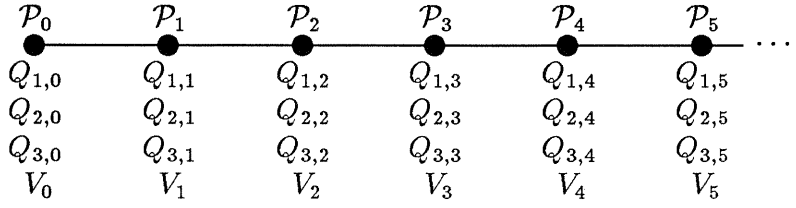

$\mathcal{P}_0$ $\mathcal{P}_1$ $\mathcal{P}_2$ $\mathcal{P}_3$ $\mathcal{P}_4$ $\mathcal{P}_5$ ...
$Q_{1,0}$ $Q_{1,1}$ $Q_{1,2}$ $Q_{1,3}$ $Q_{1,4}$ $Q_{1,5}$
$Q_{2,0}$ $Q_{2,1}$ $Q_{2,2}$ $Q_{2,3}$ $Q_{2,4}$ $Q_{2,5}$
$Q_{3,0}$ $Q_{3,1}$ $Q_{3,2}$ $Q_{3,3}$ $Q_{3,4}$ $Q_{3,5}$
V_0 V_1 V_2 V_3 V_4 V_5

Figure 13: Layout for the arrays, `Q(:SERIAL,:NEWS)` and `V(:NEWS)`.

```
CMF$ LAYOUT  X(1.0:NEWS,1.0:NEWS)
```

would be best for this computation. Were the layout defined as

```
CMF$ LAYOUT  X(1000:NEWS,1:NEWS)
```

the compiler will assume that elements along the first axis of the array will communicate far more often than those of the second axis. Thus, it may assign elements of the first array on the same chip, if possible, to make the communication local.

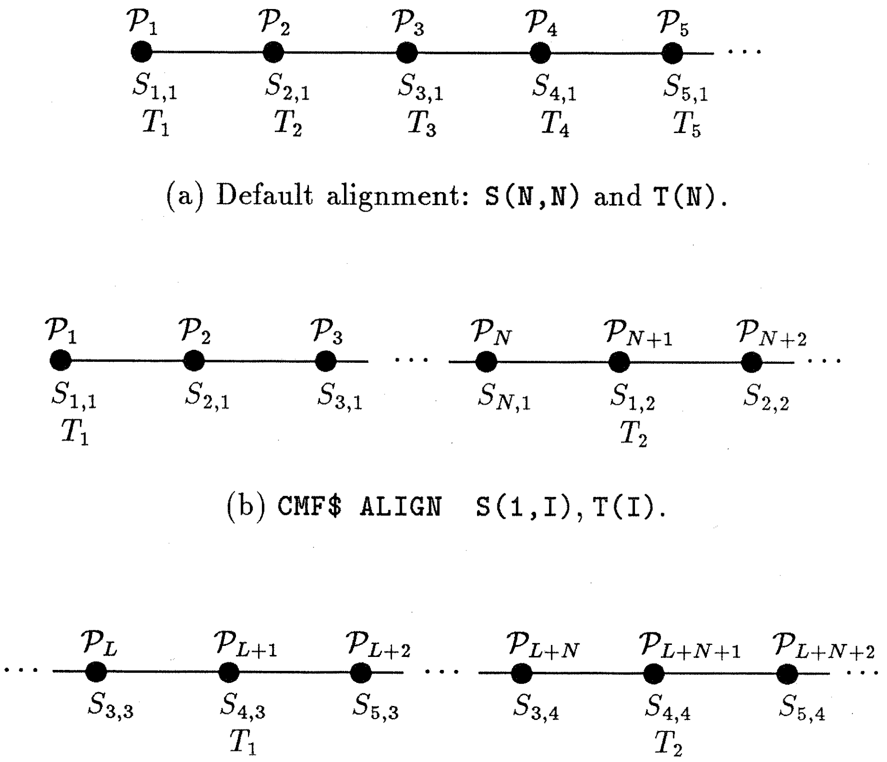
The last ordering, `SEND`, arranges the elements of the array so that each element can be quickly communicated to any other element in the array. This means they are laid out on the underlying hypercube of the CM. This ordering is best if there will be communication of local processors which are not also nearest neighbors. An example of this is an FFT computation.

It is assumed that arrays declared in the same `DIMENSION` statement should have their first elements assigned to the same processor. Hence, if the arrays, `Q` and `V`, have been declared together, as in the statement

```
DIMENSION  Q(3,0:N), V(0:N)
```

and `Q` has the same `LAYOUT` directive as above, then the elements of both arrays will be assigned to the processors in a manner similar to that shown in Fig. 13.

2.4.2 ALIGN



$\mathcal{P}_1$ $\mathcal{P}_2$ $\mathcal{P}_3$ $\mathcal{P}_4$ $\mathcal{P}_5$
$S_{1,1}$ T_1 $S_{2,1}$ T_2 $S_{3,1}$ T_3 $S_{4,1}$ T_4 $S_{5,1}$ T_5
(a) Default alignment: `S(N,N)` and `T(N)`.
$\mathcal{P}_1$ $\mathcal{P}_2$ $\mathcal{P}_3$ $\mathcal{P}_N$ $\mathcal{P}_{N+1}$ $\mathcal{P}_{N+2}$
$S_{1,1}$ T_1 $S_{2,1}$ $S_{3,1}$ $S_{N,1}$ $S_{1,2}$ T_2 $S_{2,2}$
(b) `CMF$ ALIGN  S(1,I),T(I).`
$\mathcal{P}_L$ $\mathcal{P}_{L+1}$ $\mathcal{P}_{L+2}$ $\mathcal{P}_{L+N}$ $\mathcal{P}_{L+N+1}$ $\mathcal{P}_{L+N+2}$
$S_{3,3}$ $S_{4,3}$ T_1 $S_{5,3}$ $S_{3,4}$ $S_{4,4}$ T_2 $S_{5,4}$
(c) `CMF$ ALIGN  S(4,I+2),T(I).`

Figure 14: Alignments of arrays `S` and `T`.

The `ALIGN` compiler directive is used to align the elements of two arrays with each other. For example, suppose the following arrays are declared:

```
DIMENSION  S(N,N), T(N)
```

By default, the elements of the arrays would be laid out so that `S(1,1)` and `T(1)` are on the same processor, `S(2,1)` and `T(2)` would be on the next processor, etc. This is the ordering shown in Fig. 14(a). However, it might be desirable to have `T(2)` on the same processor as S(1,2), as shown in Fig. 14(b). This can be accomplished by the following ALIGN statement.

```
CMF$ ALIGN  S(1,I), T(I)
```

Similarly, the statement

```
CMF$ ALIGN  S(4,I+2), T(I)
```

would produce a layout as shown in Fig. 14(c).

2.4.3 COMMON

The COMMON compiler directive is used to define a default home for the arrays in a given common block. The three possible forms of this directive are as follows:

```
CMF$ COMMON [, CMONLY] /blkname/
CMF$ COMMON FEONLY /blkname/
CMF$ COMMON INITIALIZE /blkname/
```

The first form of the directive tells the compiler to put the arrays contained in the given common block, *blkname*, on the PPU. The term, CMONLY, is optional and is used for clarity.

The middle or second form of the directive informs the compiler that the arrays should be placed on the FE. Otherwise, the normal default for arrays in COMMON blocks would be on the PPU.

The third and final form of the directive also instructs the compiler to make the PPU be the home of the arrays in the common block, *blkname*. But in addition, it allocates space on the FE for the common block as well to allow for the static initialization of the arrays. Such arrays may be initialized by DATA statements.

3 Using the CM-2 at NCAR

The CM-2 located at the National Center for Atmospheric Research (NCAR) in Boulder, Colorado belongs to the Center of Applied Parallel Processing (CAPP) of the University of Colorado. It has 8K processors, single-precision floating-point processors, and two sequencers which break up the 8K processors into two sections of 4K processors. One of the front ends of this machine is a VAX named `capitol.ucar.edu` with an IP-address of 128.117.64.8.

To log into this machine, type the following command:

```
rlogin capitol.ucar.edu
```

The CM will respond by asking for your password.

You will now be logged into the front end of the CM. This is a VAX running *ULTRIX*[20]. Here you may edit, compile, and store programs and data. However, this front end computer must take care of all the other CM users as well as the execution of the serial portions of any running CM programs. Please try to use this resource sparingly.

Data and program files can be created on your home machine and copied over to the CM using `ftp` or `rcp` (with an appropriate `.rhosts` file). Similarly, output files and program listings can be copied back to your home machine for further processing or debugging.

The two main CM commands you will be using on `capitol` are `cmf` and `cmattach`. The full pathnames for these and other commands on `capitol` are as follows:

```
/usr/local/bin/cmf
/usr/local/bin/cmattach
/usr/local/bin/cmusers
/usr/local/bin/cmfinger
```

Be sure that this directory is in your path or that you have aliases with the correct pathnames for these commands.

A library of graphical routines is also available; it is called the *Generic Display Interface*. Using these routines, one can display the dynamic output of the PPU on the CM graphical display or remotely, using an *X-window*. To include these routines, add the options, `-lcmsr` and `-lX11`, to your `cmf` command to get access to both the `CMSR` and `X11` libraries. You will also need to add the statement

[20] *ULTRIX* is a trademark of Digital Equipment Corporation.

```
      INCLUDE '/USR/INCLUDE/CM/DISPLAY-CMF.H'
```

at the top of the program which uses such routines. For more information, see the *CM Graphics Programming* manual [TMC 89].

4 Acknowledgements

The author would like to thank Dr. Steven Goldhaber of Thinking Machines Corporation at the National Center for Atmospheric Research for all his assistance and patient explanations of the operation and architecture of the CM-2.

References

[Adams et al. 92] ADAMS, JEANNE C., WALTER S. BRAINERD, JEANNE T. MARTIN, BRIAN T. SMITH, AND JERROLD L. WAGENER. [1992]. *Fortran 90 Handbook.* Intertext Publications. McGraw-Hill Book Company, New York, NY.

[Baron & Higbie 92] BARON, ROBERT J. AND LEE HIGBIE. [1992]. *Computer Architecture: Case Studies.* Electrical and Computer Engineering. Addison-Wesley Publishing Company, New York, NY.

[Brainerd et al. 90] BRAINERD, WALTER S., CHARLES GOLDBERG, AND JEANNE C. ADAMS. [1990]. *Programmers Guide to Fortran 90.* McGraw-Hill Book Company, New York, NY.

[Hillis 85] HILLIS, W. DANIEL. [1985]. *The Connection Machine.* The MIT Press, Cambridge, MA.

[Hord 90] HORD, R. MICHAEL. [1990]. *Parallel Supercomputing in SIMD Architectures.* CRC Press, Boston, MA.

[TMC 88] [May 1988]. Connection Machine Model CM-2 Technical Summary. Technical Report HA87-4, Thinking Machines Corporation, Cambridge, MA.

[TMC 89] Thinking Machines Corporation, Cambridge, MA. [Oct 1989]. *Connection Machine: Graphics Programming Manual.* Version 5.2.

[TMC 91a] Thinking Machines Corporation, Cambridge, MA. [Jan 1991]. *Connection Machine: Fortran Programming Guide.* Version 1.0.

[TMC 91b] Thinking Machines Corporation, Cambridge, MA. [Jul 1991]. *Connection Machine: Fortran Reference Manual.* Version 1.0 and 1.1.

[TMC 91c] Thinking Machines Corporation, Cambridge, MA. [Jul 1991]. *Connection Machine: Fortran Users's Guide.* Version 1.0 and 1.1.

[TMC 91d] [Jun 1991]. Connection Machine CM-200 Series Technical Summary. Technical report, Thinking Machines Corporation, Cambridge, MA.