Chapter 2

The CM Fortran Compiler

2.1 CM Fortran Execution Model

A global CM Fortran program executes in a data parallel manner on a partition of CM-5 processing nodes. The program's data is distributed across the memory banks of the CM-5's processing elements, and the PEs simultaneously execute each parallel operation of the program.

Note: For the most part, this document focuses on the global CM Fortran programming model. CM Fortran programs can also be compiled to execute in a nodal or global/local style, using message passing for communication between nodes, as discussed in Chapter 10.

2.1.1 Scalar/Parallel Data and Instructions

CM-5 programs and data are divided between the PM and the nodes:

- *Scalar instructions* and *scalar data* are stored in PM memory.
- *Parallel instructions* are replicated across PN scalar memory.
- *Parallel data* is distributed across PN parallel memory.

Notice the difference between parallel code and data. In general, each PN has a single copy of the parallel code in a program (since the microprocessor provides the vector units with the DPEAC instructions they are to execute, only the microprocessor needs a copy of the actual program code). However, each PE can have its own data. On CM-5s with vector units, this means that parallel data is distributed across the memory of the VUs.

Copyright © 1994 Thinking Machines Corporation

2.1.2 Parallel Data in CM Fortran

Parallel data is stored on the CM-5 in the form of distributed arrays, with a block of array elements stored in the memory of each PE (either PN or VU, depending on the available hardware). The part of the array stored in the memory of each PE is known as the array's *subgrid*. The subgrids of an array are the same size and shape for each PE.

Note: The CM Fortran compiler has a switch, **`-sparc`**, that causes the compiled program to execute as if the VUs are not present, even on CM-5s with VUs available.

Figure 4 shows the distribution of array subgrids on a CM-5 with and without vector units.

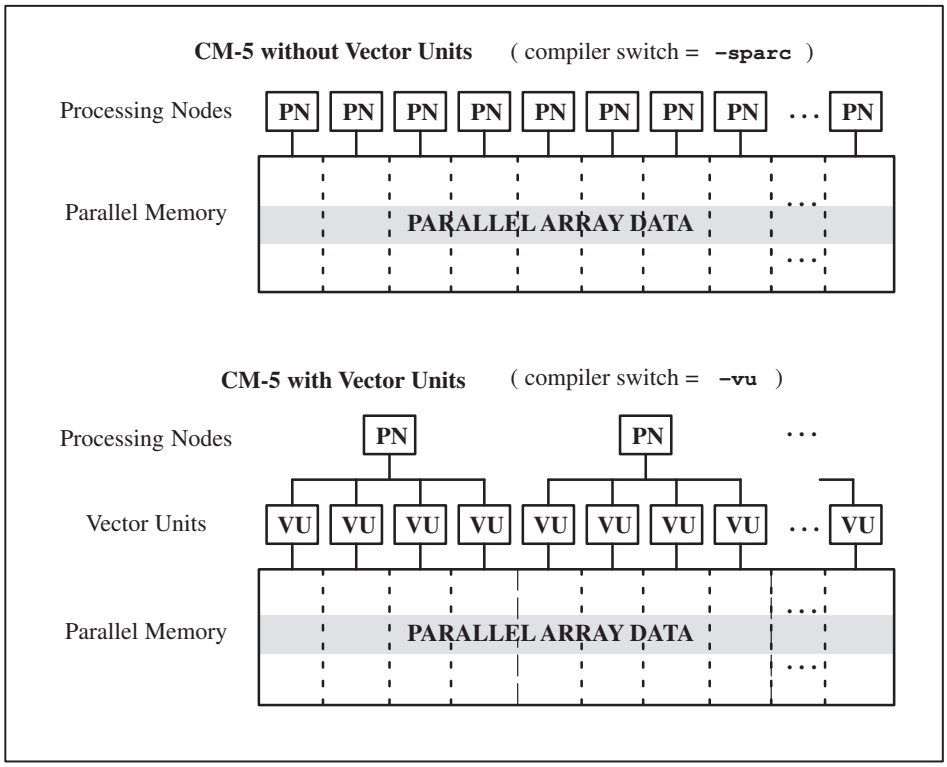

Figure 4. Parallel arrays on CM-5, with and without VUs.

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

2.1.3 CM Fortran Program Execution

Global CM Fortran programs execute in a master/slave style between a partition manager and the processing nodes in its partition. See Figure 5. The PM executes the program, and handles all scalar instructions and data operations. The PNs wait in a *dispatch loop* for instructions from the PM.

When the PM comes to a program step that requires a parallel operation (that is, requires the PNs to execute some operation on their data), it broadcasts a call to a parallel routine, and the PNs immediately execute that routine. Once it is completed, the nodes return to the dispatch loop to await the next instruction from the PM.

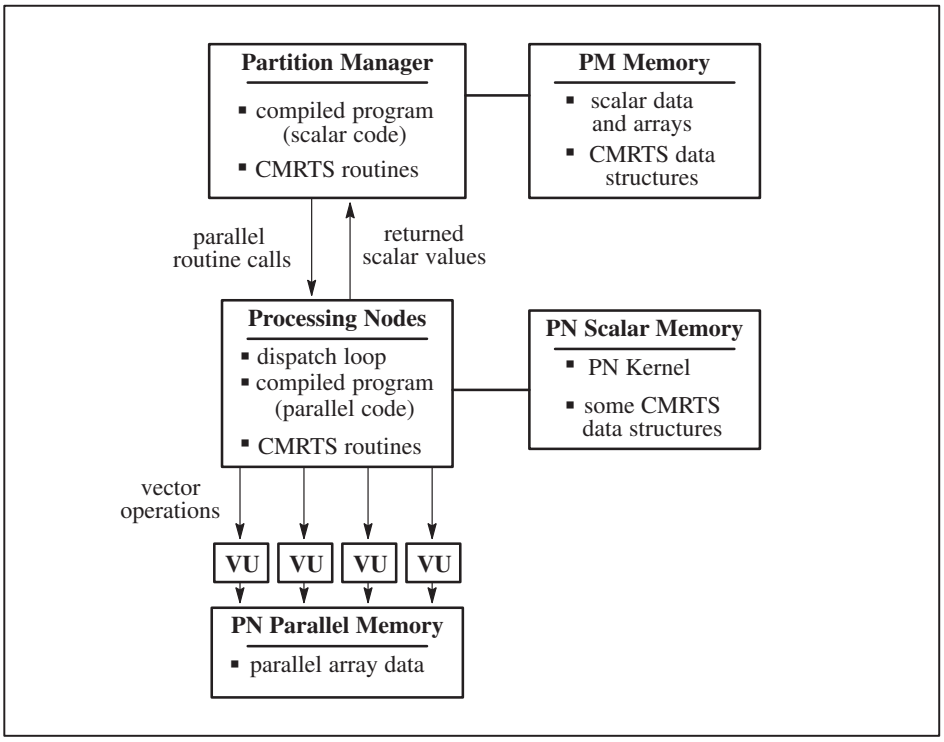

Figure 5. Distribution of code and data in a CM program.

For parallel operations that don't return a value to the PM (such as a parallel array addition, or a shift of data within a parallel array), the partition manager can simply broadcast the function call and then go on to the next step of the program. There is an instruction FIFO between the PM and the nodes that allows the PM to buffer many such function calls and let the PNs

Copyright © 1994 Thinking Machines Corporation

accept them at their own rate. However, when a parallel operation returns a value to the PM (for example, a global sum of a parallel array, or the copying of one or more of its elements to PM memory), the PM must broadcast the function call and then wait for the value to be returned to it by the PNs. This in itself causes a delay in program execution, and, depending on the number of instructions buffered in the instruction FIFO, there may be a small additional delay as the FIFO is cleared by the nodes to reach the value-returning operation.

2.2 The CM Fortran Compiler

The CM Fortran compiler (`cmf`) is like any standard compiler: it takes a source program written in a high-level programming language and converts it into a machine-language executable program. On the CM-5 the compiler-generated machine code consists of two parts:

- *Scalar code*: This is the main body of the program, executed in the partition manager. It is primarily SPARC assembler code, with embedded CMRTS library calls and calls to parallel node routines.
- *Parallel code*: This is the set of compiled code routines called by the PM. It is a mixture of SPARC assembler, intended for the node microprocessor, and DPEAC assembler, executed by the vector units.

The scalar part of a program includes:

- all scalar operations and procedure calls
- all allocation and deallocation of parallel memory
- all use (explicit or implicit) of parallel communication
- all calls to library routines (including the CMRTS)

2.2.1 PE Code Blocks

The parallel code of a program consists of everything else: that is, all the unbroken sections of pure parallel computation, not involving any of the components listed above for the scalar part of the program.

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

Such an unbroken region of parallel computation in a program is known as a *PE code block*. It is a block of code that can be executed purely by the PEs, using only the data each PE has in its own memory without reference to any data anywhere else in the machine.

PE code blocks are important to the performance of a CM program. They are simple routines executed simultaneously on all the processing nodes, and include DPEAC instructions that each PN broadcasts simultaneously to all four of its VUs. This pure parallel computation executes at the highest speed possible on the CM, so long code blocks generally make better use of the CM hardware than short code blocks.

When a program creates a parallel CM array, either through the declaration of a fully specified array variable or through dynamic allocation of an array, a certain number of array elements are stored in each PE's memory as the array's subgrid. The number of array elements on each PE is known as the array's *subgrid size*. This is the number of elemental operations a PE code block must perform on each PE to execute over all the data of the array.

An important part of any PE code block is its *subgrid loop*. This is the part of the code block that actually loops over array elements. The simplest type of subgrid loop is just a vectored iteration over array subgrids, one subgrid per array on each VU, eight subgrid elements at a time. More often, however, there is a nested set of loops in the code block, with the innermost being a vectored operation on subgrid elements, and outer loops iterating over other axes of the array.

For example, if this code example is compiled with the **`-nopadding`** switch and run on a 128-VU CM:

```
      INTEGER A(16,4,64), B(16,4,64)
CMF$  LAYOUT A(:SERIAL, :NEWS, :NEWS)
CMF$  LAYOUT B(:SERIAL, :NEWS, :NEWS)
      A=B*B
```

The innermost subgrid loop would iterate over the 16 subgrid elements on each node, eight at a time, and the outer subgrid loops would iterate over the 4*64=256 positions along the two **`:NEWS`** axes.

In each iteration, the innermost subgrid loop uses VU vector operations to perform the appropriate computations on eight subgrid elements per VU, or 32 elements per PN (since there are four VUs per PN). If the subgrid size is not a multiple of eight elements in length, the remaining "tail" of one to

Copyright © 1994 Thinking Machines Corporation

seven elements is handled by a separate step, in which the vector length of the operation is reduced appropriately.

Since a PE code block can include more than one CM Fortran statement, there can even be a series of independent nested subgrid loops. In general, this document uses the term *subgrid loop* to refer to any of these possibilities — in other words, the portion of the code block devoted to looping over the elements of one or more parallel arrays, as opposed the startup portion of the code block.

2.2.2 Parallel Computation and Parallel Communication

Parallel computation consists of the arithmetic computations and local rearrangements of data performed by individual PEs within PE code blocks, in which each PE does its own work on its own data, without reference to the operations of any other PE in the machine. This is usually produced by simple array arithmetic expressions like

```
INTEGER A(100),B(100),C(100)
C=A*A+B*B
```

Parallel communication involves the exchange of data between processors, via the CM's two networks. It is usually produced by array shifting or selection operations, such as

```
INTEGER A(200),B(100),C(100)
C=A(1:200:2)+B(100:1:-1)
```

It may at first seem odd that communication is treated as a scalar operation in terms of PE code block assignments. It is true that there is a parallel component to the communication operator; in fact, much of the communication operation executes on the nodes. However, parallel communication is initiated by a scalar procedure call by the PM; thus it is classed as a scalar operation and is not included in PE code blocks.

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

2.2.3 The Compilation Process

When you compile a CM Fortran program,

- The CM Fortran compiler processes the source file, **`program.fcm`**, producing two temporary files of assembly code:
  - **`program.s`**, a *scalar code stream* consisting of SPARC code
  - **`program.pe.dp`**, a *parallel code stream* consisting of SPARC and DPEAC code
- The scalar code stream is assembled by the UNIX assembler **`as`**, producing a scalar object file, **`program.o`**
- The parallel code stream is assembled by the DPEAC assembler **`dpas`**, producing a SPARC assembly code file, which is assembled by **`as`** to produce a parallel object file, **`program.pe.o`**
- The scalar and parallel object code files are linked by the CM linker program, **`cmld`**, along with any required code from CM libraries such as CMRTS, CMSSL, etc.

The result is a single executable code file, containing both the scalar code and the parallel code for the program. When this file is executed on a CM-5 partition manager, the scalar and parallel portions of the program are separated. The scalar code is kept on the PM, and the parallel code is broadcast to the PNs and stored in scalar PN memory for the duration of execution of the program.

Figure 6 outlines the CM Fortran compilation process.

Copyright © 1994 Thinking Machines Corporation

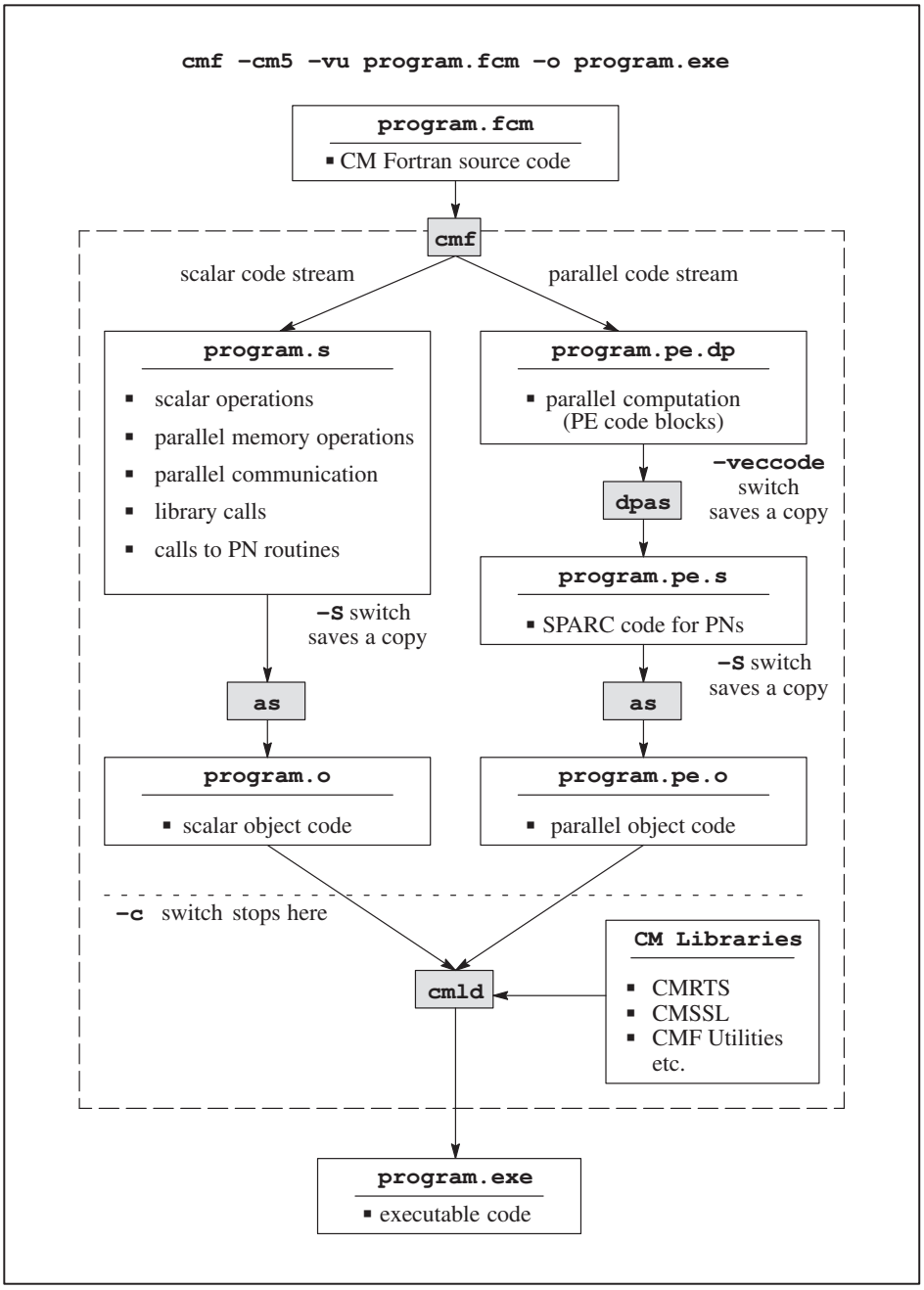

Figure 6. What happens when you compile a CM Fortran program.

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

2.2.4 Useful Compiler Switches

The CM Fortran compiler provides a number of switches that you can use while compiling a program. The basic switches are:

-cm5	Sets CM-5 as target machine for compiled program.
-vu,-sparc	Select whether or not to use the VUs.
-o *output*	Name output executable file (default is **a.out**).

Other switches that affect the debugging, performance, and performance analysis of a program are:

-c	Compile to produce object files, but don't link.
-S	Save (don't delete) intermediate assembly code files.
-veccode	Leave PN object code as DPEAC (**.dp**) file.
-list	Produce listing file (**.lis**) of variables, subroutines, and CM communication calls used in the program.
-forall_print	Show translations of **FORALL** statements (unsupported).
-O	Enable full code optimization in all compilers.
-g	Include full symbol table for the Prism debugger.
-cmdebug	Include minimal information for debugging purposes.
-cmprofile	Include information for Prism performance analysis.
-nopadding	Don't use vector length padding (default is to use vector length padding).
-noaxisreorder	Make array layouts be lowest-axis-varies-fastest (default layout is highest-axis-varies-fastest), except for serial axes, which are treated separately.

The use and effects of these switches are discussed below.

Copyright © 1994 Thinking Machines Corporation

2.2.5 Controlling the Compilation Process

The CM Fortran compiler switches allow you to stop the compilation process at any stage so that you can examine the intermediate code files being produced. By default, the CM Fortran compiler produces an executable file and deletes all intermediate assembly code files:

`cmf -cm5 -vu program.fcm` produces:

`a.out` (executable file)

The **`-c`** switch instructs the compiler to omit the **`cmld`** linking step, with the result that only the object code files are produced. You can use this switch to generate object code files that you'll later link by hand with **`cmld`**.

`cmf -cm5 -vu -c program.fcm` produces:

`program.o` (scalar object code file)
`program.pe.o` (parallel object code file)

The **`-S`** switch instructs the compiler to save (not delete) the intermediate SPARC assembly code files:

`cmf -cm5 -vu -S program.fcm` produces:

`program.s` (scalar SPARC assembly code file)
`program.pe.s` (parallel SPARC assembly code file)

Finally, the **`-veccode`** switch instructs the compiler to save (not delete) the intermediate DPEAC assembly code file.

`cmf -cm5 -vu -veccode program.fcm` produces:

`program.pe.dp` (parallel DPEAC assembly code file)

You can use the **`-c`**, **`-S`**, and **`-veccode`** switches in any combination. If all you want is the assembly code for a program, use the **`-c`** switch with either or both of the **`-S`** and **`-veccode`** switches to avoid the long linking step:

`cmf -cm5 -vu -S -c -veccode program.fcm` produces:

`program.s` (scalar SPARC assembly code file)
`program.pe.dp` (parallel DPEAC assembly code file)
`program.pe.s` (parallel SPARC assembly code file)
`program.o` (scalar object code file)
`program.pe.o` (parallel object code file)

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

2.2.6 Generating a Listing File

The switch **-list** causes the CM Fortran compiler to generate an additional *listing file*, with a **.lis** extension. This file contains a complete line-numbered source-code listing of the program, as well as detailed information for each segment (main program or subroutine) of the code. Among the information provided is:

- lists of variables, arrays, and parameters (constants) defined or used
- lists of subroutines and functions defined or called
- a list of all CM interprocessor communication operators used by the program, with line numbers showing where they are used

This last item is particularly useful: the listing file shows you "hidden" or implicit uses of CM communication that might not be immediately obvious from inspection of the original source code. For example, here's an abbreviated sample from the listing generated by a short CM Fortran program:

```
Connection Machine Fortran Version: 2.2 ...
Source Listing File: program.fcm
    1     1  0           PROGRAM SIMPLE
    2     2  0           INTEGER I,X(200)
    3     3  0
    4     4  0           X=0
    5     5  0           FORALL(I=1:200) X(I)=I
    6     6  0           FORALL(I=1:199) X(I)=X(I+1)
    7     7  0           FORALL(I=1:200) X(I)=X(201-I)
    8     8  0
    9     9  0           END
...
ARRAYS
Offset Size  Type   Block/Class  Home   Name
   4   800   INT4    local        CM     X
...
COMMUNICATION ROUTINES
  Name     Line Number (number of times)
  NEWS          6
  SEND          7
```

Notice that the listing file reports the definition of a CM array named **X**, 800 bytes (200 integers) long, defined on line 2. It also flags the implicit use of CM communication operations by the **FORALL** statements on lines 6 and 7.

Copyright © 1994 Thinking Machines Corporation

2.2.7 Generating a FORALL Transformation Printout

The CM Fortran indexed array assignment statement **FORALL** can often be transformed by the compiler into an equivalent, more efficient CM Fortran expression. You can use the switch **-forall_print** to see the transformations applied by the compiler — these are printed out as part of the compiler's output. For example, here's the **-forall_print** output for the **simple** program shown above:

```
% cmf -cm5 -vu -forall_print -c simple.fcm

Entering Forall #1:     FORALL(I=1:200) X(I) = I
Processed Forall:       X(1:200) = [1:200]
Lowered Processed Forall:X(1:200) = [1:200]
Optimized Forall:       X = [1:200]
Exiting Forall

Entering Forall #2:     FORALL(I=1:199) X(I) = X(I+1)
Processed Forall:       X(1:199) = X([1:199]+1)
Lowered Processed Forall:X(1:199) = X([1:199]+1)
Optimized Forall:       X(1:199) = X(2:200)
Exiting Forall

Entering Forall #3:     FORALL(I=1:200) X(I) = X(201-I)
Processed Forall:       X(1:200) = X(201-[1:200])
Lowered Processed Forall:X(1:200) = X(201-[1:200])
Optimized Forall:       X = X(200:1:-1)
Exiting Forall
```

As the `Optimized Forall` lines show, the first **FORALL** is transformed into a simple array copy that involves no communication, while the other two **FORALL** statements are transformed into array reorderings that do involve communication between processors.

Note: The **-forall_print** keyword is "officially" considered to be an internal compiler feature, and thus is not supported for customer use, but it is described here because it has practical uses for performance analysis.

2.2.8 Enabling the CM Fortran Code Optimizer

The **-O** switch enables the CM Fortran code optimizer, which uses a number of different methods to improve the execution efficiency of your code.

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

2.2.9 Compiling Code for Debugging and Performance Analysis

The CM Fortran compiler has a trio of switches that trigger the inclusion of symbol table information for debuggers and performance analysis tools, such as the Prism debugger and profiler.

The **-g** switch is the most comprehensive. It includes full symbol table information, and prevents optimization of code (that is, turns off the effects of **-O**, if present) so that the code generated by each program source line can be clearly distinguished. The **-g** switch is best used only for major debugging of code, and for line-by-line analysis of performance, because it prevents full optimization of your code and thus makes it difficult for you to gauge the overall performance of the entire program.

The **-cmdebug** switch is less comprehensive, but better for performance analysis purposes. It includes the minimal amount of symbol table information required by debuggers, and does not obstruct optimization as thoroughly as does the **-g** switch.

If you plan to use the Prism performance analysis tools, you must include the **-cmprofile** switch. The **-cmprofile** switch causes the compiler to insert calls to Prism timing routines at entry and exit points of PE code blocks, communication routines, and other major operations of your program. Within Prism, you run the compiled program with Prism's data collection feature turned on. When the program has finished executing, you can instruct Prism to display the collected performance data.

2.2.10 Array Padding and Default Axis Ordering

The **-nopadding** switch allows the CM Fortran compiler to store arrays on the CM in the minimum amount of space allowable, without vector-length padding — that is, without rearranging the array layout to put a multiple of eight array elements in each PE subgrid. The default (in the **-vu** model) is to include this extra vector-length padding. You can also force it by supplying the **-padding** switch. Vector-length padding is described in more detail in Section 5.3.5 — for now, just think of this switch as a way to allow the compiler to pack a given array in a smaller space than the default.

The **-noaxisreorder** switch similarly changes the default way that the compiler stores arrays on the CM. It changes the order in which elements

Copyright © 1994 Thinking Machines Corporation

along various axes of the array are stored in memory. This is also described in more detail in Section 5.3.7.

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation