Chapter 7

Parallel Computation

7.1 PE Code Blocks

As described in Section 2.2.1, the compiler generates PE code blocks to perform parallel computation operations. These PE code blocks are simple routines executed simultaneously on all PNs. On CM-5s with vector units, the PE code blocks include DPEAC instructions that each PN broadcasts simultaneously to all four of its VUs.

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.

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

This low-level parallel computation is the most effective use of the CM hardware. If you're looking at the performance of your program under Prism, it's categorized as **PN cpu (user)**. The more parallel computation you can get into a code block, the better the observed performance you'll see.

Copyright © 1994 Thinking Machines Corporation

Another way of saying this is that anything that breaks a PE code block (causes its operations to be divided into two or more separate code blocks) will hurt the performance of your program, in two ways:

- by invoking the startup cost of the PE code block (function call, etc.) more than once
- by preventing the reuse of data (in VU registers, etc.) for parallel computations

Note that this doesn't mean you should spend your time worrying about making the parallel computation in your code as efficient as possible — merely that you should try to create large, unbroken sections of it that can be easily compiled into PE code blocks. You gain more by taking things away (that is, reducing the amount of communication and temporary copies of arrays in your program) than by manipulating the parallel computation in your code.

To give a rule of thumb, an improvement in the efficiency of the code in a PE code block might double the performance of a section of code. Reducing the need for parallel communication and other things that might break PE code blocks or reduce the amount of parallel computation in your code, however, can gain you a tenfold increase or more in performance.

This chapter discusses some of the ways in which you can improve or reduce the performance of the parallel computation sections of your program.

7.1.1 How PE Code Blocks Are Invoked

The scalar portion of the program, running on the partition manager, causes a PE code block to be executed simultaneously on all PNs by calling an internal run-time system function. Through this call, the following information needed to invoke the PE code block is broadcast to the PNs: the address of the code block itself, the memory addresses of the parallel variables used in the code block, and the values of any scalar variables used in the code block.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

7.1.2 Understanding Costs in PE Code Blocks

The amount of time spent executing a PE code block is equal to the startup time plus the time spent in the subgrid loop. The startup time includes

- the time spent packing up the information to be passed from the partition manager to the PNs
- the time spent broadcasting this information to the PNs (this typically dominates the startup time)
- the time spent in the prologue of the PE code block itself

The startup time is incurred once whenever the code block is invoked. It depends on the amount of information passed to the PE code block, but not upon the subgrid size.

Note: The FIFO connection between the PM and the nodes is buffered, but has a limited size — very long argument lists may force the PM to wait for the nodes to read part of the PE code block's argument list from the FIFO before the rest can be sent. This can have a significant impact on the startup time of a PE code block.

The loop time depends upon three things:

- the number of iterations spent in the loop (that is, the subgrid size divided by eight); this depends in turn upon the size of the arrays involved
- the amount of code in the loop body
- the amount of "tail" computation enforced by the subgrid size — a short inner loop and long outer loop reduces performance

The PE code block startup time is pure overhead, since useful computation is not performed during startup. When the startup time dominates your program's execution time, the program is running inefficiently. Two situations can cause this:

- Your subgrid sizes are too small.
- Your code is executed with many smaller code blocks rather than few larger ones.

Copyright © 1994 Thinking Machines Corporation

You can address the first problem by choosing an array size that is large relative to the machine size. Doing this may require reconsidering how your problem is mapped onto the machine. The subgrid size you need to amortize code block startup costs will vary, depending upon how much code is in your code blocks and how much information is passed to the code block. As a rough rule of thumb, aim for a subgrid size of at least 64 — that is, an array size that has at least 64 times as many positions as the number of PEs in the partition.

7.1.3 Example

The short CM Fortran program below performs both ordinary scalar Fortran operations and parallel array operations:

```
PROGRAM CODEBLOCK
DOUBLE PRECISION X, A(8,8), ASUM
X = 30
X = X * 17.2 + 14
PRINT *,'THE VALUE OF X IS:',X
A = 30
A = A * 17.2 + 14
ASUM = SUM(A)
PRINT *,'THE GLOBAL SUM OF A IS:',ASUM
PRINT *,'(SHOULD BE:',X*64,')'
END
```

The statements

```
X = 30
X = X * 17.2 + 14
```

as well as the **PRINT** statements, are generated as an ordinary scalar Fortran operations, executed by the PM.

The statements

```
A = 30
A = A * 17.2 + 14
```

perform parallel computation. The compiler generates a PE code block, and a call in the generated scalar code to execute that code block on all of the PNs after the first call to **PRINT**.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

The statement

```
ASUM = SUM(A)
```

performs parallel communication — in this case, all the values of **A** are summed to a single value. The compiler generates a call in the scalar code to a run-time routine that performs this reduction.

This program thus contains one PE code block. The startup time for the PE code block is about 48 µs, and the subgrid loop time is about 1.6 µs per 8-element subgrid loop iteration, or 0.2 µs per subgrid element. (Your timings may vary.)

Since the startup cost is significantly larger than the cost of one loop iteration in this code block, we will not be computing results efficiently unless the array size is large enough to require the loop to make several iterations.

Suppose the subgrid size is 32 elements per VU. (On a 32-PN CM-5 partition, this requires a shape with 4096 elements.) In this case, our loop executes four iterations. The total time spent executing the PE code block is about 54 µs, of which 48 are the overhead of starting the code block and 6 are spent doing actual computation in the loop body.

7.1.4 Timing PE Code Blocks

The values for the PE code block startup and loop times can be obtained for a specific PE code block by enclosing the code in a loop and executing it on arrays with very small and very large subgrid sizes. For the smallest subgrid sizes, the PE code block startup time should dominate the timing results; the time spent in the subgrid loop should be negligable by comparison. For arrays with very large subgrids, the loop time should predominate. Presented below is a sample timing program that you can use to determine the relative sizes of these timing factors.

Note: This sample program must be compiled with the compiler switch **-nopadding** for the arrays to be laid out correctly.

```
PROGRAM CODEBLOCK_TIME
INTEGER NP, CMF_NUMBER_OF_PROCESSORS
NP=CMF_NUMBER_OF_PROCESSORS
CALL DOIT(NP)
END
```


Copyright © 1994 Thinking Machines Corporation

```
      SUBROUTINE DOIT(NP)
      INCLUDE '/usr/include/cm/timer-fort.h'
      INTEGER NP,I,LOOPS
      DOUBLE PRECISION A(NP,1),B(NP,1000)
      DOUBLE PRECISION T1,T2
CMF$  LAYOUT A(:NEWS,:SERIAL)
CMF$  LAYOUT B(:NEWS,:SERIAL)
      LOOPS = 10000
      CALL CM_TIMER_CLEAR(0)
      CALL CM_TIMER_START(0)
      DO I=1,LOOPS
         A = A * 17.2 + 14
      END DO
      CALL CM_TIMER_STOP(0)
      T1 = (CM_TIMER_READ_ELAPSED(0)*1000000)/LOOPS
      PRINT *,'PE STARTUP TIME: APPROX. ',T1,' USECS.'

      CALL CM_TIMER_CLEAR(0)
      CALL CM_TIMER_START(0)
      DO I=1,LOOPS
         B = B * 17.2 + 14
      END DO
      CALL CM_TIMER_STOP(0)
      T2 = (CM_TIMER_READ_ELAPSED(0)*1000000)/LOOPS-T1
      PRINT *,'PE LOOP TIME: APPROX. ',T2,' USECS.'
      PRINT *,'PER ELEMENT: APPROX. ',T2/1000,' USECS.'
      END
```

The output from a sample run of this program on a 16-VU CM is:

```
% a.out
 PE STARTUP TIME: APPROX.   46.90943333333333 USECS.
 PE LOOP TIME: APPROX.   199.8088515151515 USECS.
 PER ELEMENT: APPROX.  0.1998088515151515 USECS.
```

Counting Flops

The expression we are timing performs two floating-point operations (an add and a multiply) for each element of the array subgrid. With a subgrid size of 1000 (as in the array **b** above), each PN performed the computation for 4000 subgrid positions (because each PN has four VUs) in approximately 199+46 = 245 μs.

Copyright © 1994 Thinking Machines Corporation

We can compute the number of floating-point operations per second for each PN as follows:

$$\frac{2 \text{ ops/element} \cdot 4 \cdot 1000 \text{ elements/PN}}{245 \cdot 10^{-6} \text{ s.}} = 32 \text{ Mflops/PN}$$

Thus, on a 32-node partition, the computation with this subgrid size would be performed at about 1.0 Gflops.

Assessing Computation Performance

The CM-5 vector units can perform computation at a rate of up to 128 Mflops (million floating-point operations per second) per PN. They can also perform integer computation at this rate. This performance figure assumes that the VUs are continuously executing vector operations that do both a multiply/add and a memory operator as a single operation.

In practice, it is not possible for the compiler to emit mult/add/memory instructions for some expressions, and an application that is performing at 30 or 40 Mflops per PN may be doing as well as it can do, even though this figure is far below the theoretical peak Flops rate of the machine.

For example, in the timing example above, though the expressions being timed consist of a multiply, an add, and a memory operation, they are actually compiled as three instructions: a load/multuply, an add, and a store. From this, one might expect the operation to execute at a third of the peak rate, or 48 Mflops/PN. Other factors, however, come into play.

For example, if the array elements being referenced by an expression lie on different pages, the memory operations will generate page faults. A rule of thumb is that if an expression always generates page faults, then each 8 timer ticks of computation will incur 2.5 ticks of page faulting. Thus, one would expect the performance to be reduced by a factor of 8/10.5, or .76. This would suggest a performance rate of 48 * .76 = 36.5 Mflops/PN. But the example above can only incur a page-fault on two of three instructions, so page-faulting is not the entire reason for the 32 Mflops/PN result.

Copyright © 1994 Thinking Machines Corporation

7.2 Constructing PE Code Blocks Properly

In general, a single PE code block may perform computation for several program statements or even a fraction of a statement. The compiler merges parallel computation into a single PE code block only when the computation is not interrupted by an operation that "breaks code blocks." Since extra code blocks mean extra overhead, it is generally beneficial to write code that produces fewer code blocks.

To increase performance, you therefore need to understand what sorts of operations cause the compiler to break code blocks. Section 7.2.2 goes into detail about these operations. Here we consider a simple example:

```
PROGRAM CB1
INTEGER A(8,8),B(8,8)
A = 30
B = 30
A = A * 17.2 + 3
B = B / 4 + A
PRINT *,'THE SUM OF A IS ',SUM(A)
PRINT *,'THE SUM OF B IS ',SUM(B)
END
```

In this case, the two statements performing parallel computation on the arrays **A** and **B** are consecutive program statements. The compiler can produce a single PE code block for both of them. However, suppose we had written the code in a slightly different order:

```
PROGRAM CB1
INTEGER A(8,8),B(8,8)
A = 30
B = 30
A = A * 17.2 + 3
PRINT *,'THE SUM OF A IS ',SUM(A)
B = B / 4 + A
PRINT *,'THE SUM OF B IS ',SUM(B)
END
```

In this case, the compiler emits two PE code blocks, one for the statement

```
A = A * 17.2 + 3
```

and the other for the statement

```
B = B / 4 + A
```


Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

7.2.1 Compiler Optimization of Code Blocks

One of the compiler's optimization steps is a simple analysis of code to determine whether statements can be reordered so as to combine PE code blocks. (For example, if a communications routine in the midst of parallel computation can be performed before or after the computation, the code blocks can be combined.)

While this optimization step may catch and fix simple PE code block divisions that you haven't corrected in your code, you should not depend on it to find *all* possible source line optimizations in your code, especially for complex expressions involving both communication and computation.

7.2.2 Things That Break Code Blocks

This section discusses operations that break a code block when they appear in the middle of parallel computation. Note that future versions of the compiler may remove this limitation for many of these operations.

Flow Control

Any of the standard Fortran flow control statements cause code blocks to be broken. For example, parallel code in the body of an **`IF`** statement or **`DO`** loop is always compiled into a code block separate from surrounding code.

Function Calls

Any call to a subroutine or function causes code blocks to be broken. This includes calls to user-defined subroutines, functions in CM Fortran utility library or other libraries. Note that calls to the CM timer functions themselves break code blocks, and can affect timing results if placed improperly.

Contextualization

The body of a **`WHERE`** statement is compiled into a code block that is separate from surrounding code.

Copyright © 1994 Thinking Machines Corporation

Parallel Communication Operations

Statements that perform parallel communication operations, such as:

```
A(1:5)=B(10:14)
```

break code blocks. Basically, if data has to move between processing nodes, a call to a CMRTS communication operator is required, and it will form a wall between code blocks preceding it and code blocks following it.

Note that arrays having different **:NEWS** axis layouts will trigger communication, even if their subgrid sizes happen to be the same. For example,

```
      INTEGER A(8,8,8), B(8,8,8)
CMF$  LAYOUT A(:NEWS,:SERIAL,:NEWS)
CMF$  LAYOUT B(:SERIAL,:NEWS,:NEWS)
      A = A + B
```

Differing Array Geometries in Expressions

Consecutive array expressions that reference arrays or array subsections of different shapes will be put in separate code blocks. (The subgrid size and/or loop extents are different for different array geometries, and thus can't be combined in a single code block.)

For example, this code fragment will be compiled into two code blocks:

```
CMF$  LAYOUT A(:SERIAL, :NEWS )
CMF$  LAYOUT B(:SERIAL, :SERIAL, :NEWS )
      A(1:N,:) = B(1:N,5,:)
      A(1:M,:) = B(1:M,5,:)
```

But this fragment will compile as one code block with internal nested loops:

```
CMF$  LAYOUT A(:SERIAL, :NEWS )
CMF$  LAYOUT B(:SERIAL, :SERIAL, :NEWS )
      A(1:N,:) = B(1:N,5,:)
      B(1:N,5,:) = A(1:N,:)
```


Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

Parallel Reductions and Accessing Elements of Arrays

Accessing elements of parallel arrays (that is, causing a single array value to be copied from the nodes to the PM) breaks code blocks. For example:

```
INTEGER X,ARRAY(10,20)
X=ARRAY(5,9)
```

Similarly, reducing a parallel array to a single scalar value via the global reduction operators **SUM**, **MAXVAL**, **PRODUCT**, etc. breaks code blocks.

A future version of the compiler may remove this limitation.

Any Parallel I/O Operations

Parallel I/O operations require the assistance of the OS on the PM, and thus will break code blocks.

Compiling with the −g Debugging Option

Using the **-g** option to compile CM Fortran code forces separate statements to emit separate code blocks. This can dramatically affect your program's performance. You should be aware of this when examining the compiler output or timing code.

7.3 CM Fortran Compiler Serial Optimizations

An array dimension declared **:SERIAL** in a **LAYOUT** directive is guaranteed to be laid out in the memory of a single processor (node or vector unit, depending on the execution model). This section describes the ways in which the compiler now speeds up operations on serial dimensions.

7.3.1 Recognizing Serial Dimensions

The compiler applies the serial optimizations to axes that are declared **:SERIAL** in a **LAYOUT** directive, as long as the array is not deferred-shape (allocatable or pointer).

Copyright © 1994 Thinking Machines Corporation

The compiler also recognizes axes described with :**PROCS=1** or :**PDESC=0** in a detailed **LAYOUT** directive as local for the purpose of applying the serial optimizations, as long as the array is not deferred-shape and the program is compiled under the (nondefault) **-noaxisreorder**. Under the default **-axisreorder**, only **:SERIAL** axes are recognized as local. Note that every program unit in a program must be compiled with the same setting of **-[no]axisreorder**.

If a deferred-shape array is passed to a subprogram whose dummy array does not have the **POINTER** attribute and one or more axes are declared **:SERIAL** (or, under **-noaxisreorder**, **:PROCS=1** or **:PDESC=0**), the array benefits from serial optimizations in the subprogram. However, the optimizations are not applied to deferred-shape arrays in program units where such arrays have the attribute **ALLOCATABLE** or **POINTER**, because it is not possible to guarantee that the geometry will never change.

The compiler recognizes local axes in arrays laid out with **CMF_ALLOCATE_DETAILED_ARRAY** or with **CMF_ALIAS_DETAILED**, since these arrays must be passed to a subprogram that declares them as CM arrays whose layout is compatible with the descriptor created in the caller. As long as the **LAYOUT** directive in the subprogram meets the above requirements for recognizing local axes, the compiler applies the serial optimizations. If the subprogram uses an assumed **LAYOUT** directive (colons "**:**" for each axis) or uses detailed **LAYOUT** syntax under **-axisreorder**, the serial optimizations are not applied.

In addition, the compiler applies the serial optimizations to an array that is **ALIGN**ed with an array whose local dimensions are recognized as such.

7.3.2 No-Motion Detection

In previous versions, the compiler treated data motion along serial dimensions in the same way as data motion along parallel dimensions: by generating run-time communication routines. The compiler now recognizes array section assignments on serial dimensions as on-processor, and generates **DO** loops with local memory accesses instead. This change can increase performance by orders of magnitude.

For example, consider these two arrays:

```
      REAL A(100,100),B(100,100)
CMF$  LAYOUT A(:SERIAL,:NEWS),B(:SERIAL,:NEWS)
```


Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

The **LAYOUT** directive ensures that all the elements of the first dimensions of **A** and **B** are co-resident on a processor. Therefore, even if the elements referenced in triplets on this dimension are different, the operation is local:

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

In previous versions, this statement generated NEWS communication. Now it generates nested loops, one from 1 to 2 and one from 1 to the on-processor extent of the **:NEWS** axis. This should be several hundred times faster.

Similarly, a stencil expressed as the assignment of on-processor array sections is now much faster:

```
      REAL D(100,100,100),E(100,100,100)
CMF$  LAYOUT D(:SERIAL,:SERIAL,),E(:SERIAL,:SERIAL,)

      FORALL(I=1:N,J=1:M)
     $    D(I,J,:) =
     $    E(I+1,J,:)+E(I,J+1,:)+
     $    E(I-1,J,:)+E(I,J-1,:)+E(I,J,:)
```

An important consequence of this is that it is no longer necessary to write unrolled **DO** loops to express local assignments. You can use the more convenient array notation, as shown above.

Some forms of data motion on serial dimensions are not currently optimized in this way. These include

- some array transformations, such as reductions, whether expressed explicitly with the intrinsic functions or implicitly via **FORALL**
- references to the intrinsic functions **CSHIFT** and **EOSHIFT**

Any off-processor communication in a statement disables the serial optimizations for that statement, since the compiler does not fission its internal loops to isolate the local operations. Examples of off-processor operations include communication along a parallel dimension:

```
      REAL A(100,100),B(100,100)
CMF$  LAYOUT A(:SERIAL,:NEWS),B(:SERIAL,:NEWS)

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


Copyright © 1994 Thinking Machines Corporation

and communication between arrays that are not aligned in distributed memory:

```
      REAL A(100,100),C(100,500)
CMF$  LAYOUT A(:SERIAL,:NEWS),C(:SERIAL,:NEWS)

      A(1:50,:) = C(1:50,:)
```

7.3.3 Better Handling of Context

Another form of assignment that speeds up is

```
      REAL A(100,100)
CMF$  LAYOUT A(:SERIAL,)

      A(1:2,:) = 0
```

In previous versions, this statement looped over all the elements of **A**, masking out stores to elements that were not in the first two rows. Now, it generates nested loops, one from 1 to 2 and one from 1 to the on-processor extent of the **:NEWS** axis. This should be about 50 times faster.

Similarly, strided expressions on serial axes, such as

```
      A(L:U:S,:) = 0
```

now perform the indicated number of operations instead of operating on the whole array.

7.3.4 Nested and Flattened Loops

In previous releases, a local code block could be only a single, unit-stride loop over the entire subgrid. In 2.1, the compiler can generate code blocks with strided loops and can generate a sequence of perfectly nested loops.

For example, suppose **A(N,N,M)** is **(:SERIAL,:SERIAL,:NEWS)**. Then, the assignment

```
A(2:6,3:7,:) = 1
```

can be accomplished with one triply nested loop inside a code block. In earlier versions, this was done with a single unit-stride loop traversing all of

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

A and a masked store, or the user could write an explicit doubly nested **DO** loop:

```
DO I = 2,6
   DO J = 3,7
      A(I,J,:) = 1
   END DO
END DO
```

Of course, the user can still write the double **DO** loop, but it turns into a double loop of PN code blocks, and so incurs much higher overhead than the single code block containing a compiler-generated triply nested loop.

The innermost loop of a loop nest is always the one with shortest stride in memory. The compiler flattens the loop nest where possible, but does not yet take advantage of **(:BLOCK=1)** declarations. An example of flattening, assuming **A(:NEWS,:SERIAL), B(:SERIAL,:NEWS)** and the compiler switch **-noaxisreorder**:

`A(:,I:J) = 1` generates a single loop (the double loop is flattened)

`B(I:J,:) = 1` generates a double loop (which cannot be flattened)

The innermost loop is vectorized under the **-vu** execution model. The innermost loop usually corresponds to the innermost (unit-stride) axis in memory. If the loop is flattened, however, more than one axis is "the" vector axis.

7.3.5 Simple Dependence Analysis

The compiler now performs some very simple dependence analysis to determine whether it can avoid copying the evaluation of the right-hand side of an assignment to a temporary before performing the assignment. Dependence analysis is also necessary to permit several statements to be merged into a single nested loop, thereby saving memory references.

Suppose **A(M,N)** is **(:SERIAL,:NEWS)**. The analysis is sufficient so that the assignments

```
A(1:M-1,:) = A(2:M,:)
```

```
A(2:M,:) = A(1:M-1,:)
```


Copyright © 1994 Thinking Machines Corporation

can be done with in-place vector moves, the first running forward and the second running backward. In general, if the left-hand side array is different from all right-hand side arrays, then there is no dependence. (Users of array aliases must be careful.) If the left-hand side array and some array on the right-hand side are the same, then the offset between the references must be a compile-time constant for the compiler to resolve the dependence. If the offset is not a compile-time constant, then the compiler assumes that there is a dependence and will use a temporary. For example, the following assignment requires a temporary, since the compiler cannot determine whether **K** is greater or less than 2:

```
A(K:K+2,:) = A(2:4,:)
```

7.3.6 Contiguous Serial Sections Passed in Place

An array section that is specified by scalar indices or unit-stride triplets on one or more serial axes, and which defines a contiguous region of memory, is passed in place. It can also be operated upon with aligned portions of other arrays without communication. For example, with the default **-axisreorder**, the following operations involve no communication:

```
      REAL A(10,100,100), B(100,100), C(3,100,100)
CMF$  LAYOUT A(:SERIAL,,), C(:SERIAL,,)

      A(3,:,:) = B            ! no communication
      A(1:3,:,:) = C          ! no communication
      CALL FOO (A(4,:,:))     ! arg passed in place
```

The first example has always been recognized as not requiring communication, since the two arrays are aligned along their parallel axes. Previous versions required that all serial axes be referenced with scalar subscripts, but that restriction no longer applies as long as the section is contiguous.

The second example, with a triplet on the serial axis, is also recognized by this version of the compiler as a contiguous section.

The subroutine call would previously have required an expensive **SEND** of the array section to a canonical location before it was passed as an argument. This version of the compiler recognizes that the section is a contiguous region of memory, and no **SEND** is generated.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

Communication is generated when operating upon or passing noncontiguous array sections.

To determine whether a section is contiguous, you must recognize the axis whose subscripts vary fastest in memory, which axis varies next fastest, and so on. The simplest way to do this is to check the output of **CMF_DESCRIBE_ARRAY**. The field **subgrid axis increment** is the speed at which an axis varies: the axis with increment 1 varies fastest, and larger numbers mean slower varying.

You can also determine which axes are faster or slower varying by considering the setting of the **cmf** switch **-[no]axisreorder**. With **-noaxisreorder**, the leftmost declared axis is fastest varying, the next leftmost next fastest, and so on. Under **-axisreorder** (the default), all **:NEWS** axes vary faster than any **:SERIAL** axis, the rightmost **:NEWS** axis varies faster than other **:NEWS** axes, and the rightmost **:SERIAL** axis varies faster than other **:SERIAL** axes.

Once you know the relative speed with which axis subscripts vary in memory, apply the following rules to determine whether any given array section is contiguous. Complete axes in an array section are those referenced simply as **:**; incomplete axes are those referenced with a scalar or triplet subscript.

A CM array section is contiguous in memory when every incomplete axis is

1A Local to a processor, meaning described as **:SERIAL** or, in detailed **LAYOUT** under **-noaxisreorder**, as :**PROCS=1** or **:PDESC=0**.

and

1B Contiguous on that processor, meaning referenced with either a scalar index or a unit-stride triplet (not a strided triplet).

And in addition,

2 Starting from the axis whose subscripts vary slowest in memory and working toward the fastest varying, there may be an arbitrary number of scalar-indexed axes, followed by at most one incomplete unit-stride triplet axis, followed by all complete axes.

For example, consider the sections referenced in the code shown below.

Copyright © 1994 Thinking Machines Corporation

Array	Layout	Switch	Axes Speeds	Section	In-Place?	Why
(10,64,64)	(S,N,N)	-noaxisreorder	(1,2,3)	(2,:,:)	No	2
(10,64,64)	(S,N,N)	-axisreorder	(3,2,1)	(2,:,:)	Yes	
(64,10,64)	(N,S,N)	-noaxisreorder	(1,2,3)	(:,2,:)	No	2
(64,10,64)	(N,S,N)	-axisreorder	(2,3,1)	(:,2,:)	Yes	
(64,64,10)	(N,N,S)	-noaxisreorder	(1,2,3)	(:,:,2)	Yes	
(64,64,10)	(N,N,S)	-axisreorder	(2,1,3)	(:,:,2)	Yes	
(64,64,10)	(N,N,S)	-axisreorder	(2,1,3)	(:,:,1:4:2)	No	1B
(64,64,10)	(N,N,S)	-noaxisreorder	(1,2,3)	(:,2,:)	No	1A
(4,4,64,64)	(S,S,N,N)	-noaxisreorder	(1,2,3,4)	(1,:,:,:)	No	2
(4,4,64,64)	(S,S,N,N)	-axisreorder	(4,3,2,1)	(1,:,:,:)	Yes	
(4,4,64,64)	(S,S,N,N)	-noaxisreorder	(1,2,3,4)	(:,1,:,:)	No	2
(4,4,64,64)	(S,S,N,N)	-axisreorder	(4,3,2,1)	(:,1,:,:)	No	2
(64,4,64,4)	(N,S,N,S)	-noaxisreorder	(1,2,3,4)	(:,1:4,:,3)	No	2
(64,4,64,4)	(N,S,N,S)	-axisreorder	(2,4,1,3)	(:,1:4,:,3)	No	2
(64,4,64,4)	(N,S,N,S)	-axisreorder	(2,4,1,3)	(:,4,:,3:5)	Yes	

The program that references these sections is:

```
      SUBROUTINE THREE
      REAL A(10,64,64)
      REAL B(64,10,64)
      REAL C(64,64,10)

CMF$  LAYOUT A(:SERIAL,:NEWS,:NEWS)
CMF$  LAYOUT B(:NEWS,:SERIAL,:NEWS)
CMF$  LAYOUT C(:NEWS,:NEWS,:SERIAL)

      CALL X(A(2,:,:))
      CALL X(B(:,2,:))
      CALL X(C(:,:,2))

      CALL X(C(:,:,1:4:2))
      CALL X(C(:,2:,:))
      END
```


Copyright © 1994 Thinking Machines Corporation

```
      SUBROUTINE FOUR
      REAL A(4,4,64,64)
      REAL B(64,4,64,4)

CMF$  LAYOUT A(:SERIAL,:SERIAL,:NEWS,:NEWS)
CMF$  LAYOUT B(:NEWS,:SERIAL,:NEWS,:SERIAL)

      CALL V(A(1,:,:,:))
      CALL V(A(:,1,:,:))

      CALL Z(B(:,1:4,:,3))
      CALL Z(B(:,4,:,3:5))

      END
```

7.3.7 Gathers/Scatters with FORALL

When **FORALL** does a gather or scatter on a **:SERIAL** axis, the compiler can determine that the operation is entirely local and uses the fast indirect addressing hardware.

7.3.8 Serial Spreads with FORALL or SPREAD

When **FORALL** does a spread operation on a **:SERIAL** axis, the compiler can determine that the operation is entirely local and generates local memory accesses instead of communication routines.

```
      REAL A(10,10), B(10)
CMF$  LAYOUT A(:SERIAL,:NEWS), B(:NEWS)
      ...
      FORALL(I=1:10) A(I,:) = B(:)
```

The intrinsic function **SPREAD** has also been enhanced for serial axes, so that **SPREAD** and **FORALL** give comparable performance for this operation.

Note: Spreads expressed either with **SPREAD** or with **FORALL** generate run-time communication when used on **:NEWS** axes or in assignments of unaligned array elements.

Copyright © 1994 Thinking Machines Corporation

Note: **SPREAD** is the only nonelemental intrinsic function that has been optimized for serial dimensions. **CSHIFT**, **EOSHIFT**, and all the reduction and transformational intrinsics continue to generate communication routines in all cases.

7.4 Other Optimizations

7.4.1 Single-Precision Memory Accesses in Vector Unit Code

Conditional and unconditional stores of single-precision (32-bit) **REAL** data can often be compiled mostly or totally as double-precision (64-bit) operations. Since double-precision values are the "natural" data size of the vector units, this improves performance considerably. If the compiler can determine that **REAL** array data is doubleword aligned, it can replace single-precision stores with vector length 8 with faster double-precision stores at vector length 4.

Loads and stores of single-precision **COMPLEX** data can be similarly optimized. A single **COMPLEX** result can be loaded or stored at vector length 8 by performing a double-precision **REAL** memory access at vector length 8 (instead of two single-precision memory accesses at vector length 8). This requires interleaving the data into a double register.

For example,

```
COMPLEX, ARRAY(N1,N2,N3):: A, B, C
...
A = B+C
```

generates one double-precision store, instead of two single-precision stores.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation