Chapter 6

Array Alignment and Aliasing

6.1 Using Array Alignment

Even after you have selected an appropriate layout for each array in your program, there are cases where this is not enough. For example, say you want to combine the values of a one-dimensional vector with a specific row or column of a two-dimensional array. For unaligned arrays, the compiler will generate a temporary array and use communication operators to move the vector data into alignment with the matrix row or column.

LAYOUT directives can align the vector only with the first row or column of the array. The **ALIGN** directive, however, allows you to match the vector with any row or column of the array. It achieves this by causing the compiler to store the data of the vector in a two-dimensional array geometry, with the vector elements aligned with the selected row or column of the matrix so that no temporaries or communication operators are necessary.

6.1.1 WARNING — Use ALIGN With Caution!

The **ALIGN** directive is valuable for reducing communication and increasing the locality of array elements in expressions. However, it does have corresponding disadvantages, chief among them its spendthrift use of CM memory when aligning a small array with a large one. To align a vector with a 2D array, for example, **ALIGN** places the data of the vector within an array geometry that is the same shape as the 2D array, producing many unused "garbage" locations. In the examples in this chapter, be aware that all the array locations marked with dots (".") represent wasted parallel memory.

Copyright © 1994 Thinking Machines Corporation

There are many cases where arrays are automatically aligned without the use of the **ALIGN** directive. Thus, using array alignment correctly is as much a matter of knowing when *not* to use it, as when to use it. Use of **ALIGN** can also result in severe degradation of performance if you don't use it properly.

As a rule, you should try to avoid using **ALIGN** if, by restructuring your arrays and/or using **LAYOUT** statements, you can otherwise produce the same alignment of arrays.

Note: The **ALIGN** directive is primarily useful with allocatable arrays, where the compiler doesn't automatically have information about the natural alignment of the array. For example, even if two allocated arrays have the same layout, there's no way for the compiler to determine this.

6.1.2 Default Alignments

In some cases, the compiler can tell automatically that two arrays are aligned in CM memory. Thus, no **ALIGN** statements are necessary to make sure that the compiler does not generate communication to align the arrays before operating upon them. Here are the guidelines for determining whether two arrays are automatically aligned.

Conformable Arrays

It follows from the discussion of layouts in the previous chapter that, for statically allocated arrays,

- Conformable arrays with canonical layouts are aligned.
- Conformable arrays with *identical* noncanonical layouts are aligned.

Dynamic and Assumed-Shape Arrays

Dynamically allocated arrays and assumed-shape dummy arrays are not aligned with static arrays of identical shape and layout, since their geometries are not known at compile time. You can, however, use the compiler directive **ALIGN**, described in this chapter, to avoid unnecessary communication with these arrays.

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

Nonconformable Arrays

In most cases, nonconformable arrays are treated by the compiler as unaligned. For example, two vectors **P(127)** and **R(128)** are very likely to have their first 127 elements aligned in distributed memory, but the compiler does not make this assumption. The expression **P+R(1:127)** would therefore generate communication routines to assure alignment at run time. To avoid communication, you should declare similar-shaped arrays in exactly the same shape, or supply an **ALIGN** directive.

Nonconformable Arrays with Serial Dimensions

There is one exception to the rule that nonconformable arrays are not aligned: an array with mixed serial and parallel dimensions is given the same geometry as another array with the same *parallel* dimensions (assuming equal axis weights).

For example, consider two arrays declared as follows:

```
      DIMENSION A( 10, 256, 256 )
      DIMENSION B( 256, 256 )
CMF$  LAYOUT A( :SERIAL, :NEWS, :NEWS )
CMF$  LAYOUT B( :NEWS, :NEWS )
```

Because of the **LAYOUT** directive, the compiler places these two arrays in the same geometry. Each processor contains corresponding elements of the parallel axes of the two arrays *and* the whole first axis of array **A**. As a result, an operation that involves the parallel planes of array **A** (referenced with a scalar subscript on the serial axis) and the corresponding elements of array **B** requires no interprocessor communication. For example,

```
      B = A(3,:,:)        ! No communication
```

or,

```
      DO (I = 1,10)
        A(I,:,:) = B + I  ! No communication
      END DO
```

On the other hand, an array that is the same shape as **A** but subject to a different **LAYOUT** directive does not share **A**'s geometry. In this case, operations on corresponding elements of the arrays take place across geometries, and the compiler generates interprocessor communication for them.

Copyright © 1994 Thinking Machines Corporation

For example, because the arrays **A** and **D** have different axis weights, the assignment statements below involves interprocessor communication:

```
      DIMENSION A( 10, 256, 256 )
      DIMENSION D( 10, 256, 256 )
CMF$  LAYOUT     A( :SERIAL, :NEWS, :NEWS )
CMF$  LAYOUT     D( :SERIAL, 2:NEWS, :NEWS )
      . . .
      A = D              ! Interprocessor communication
      A(3,:,:) = D(1,:,:)! Interprocessor communication
```

6.2 The ALIGN Directive

The **ALIGN** directive describes the layout of an array in terms of specified axes of another array whose layout is already determined. This directive causes the elements of the source array to be placed in the same processors as certain sections of the target array.

Note: In certain situations, the **ALIGN** directive can be extremely wasteful of parallel memory, and can actually reduce the performance of your code if incorrectly applied. See Section 6.3 for a discussion of the potential risks and benefits.

The **ALIGN** directive can speed up an application program by eliminating unnecessary communication of data when operations are performed on multiple arrays that would not normally be aligned. This section describes its effects on statically allocated and assumed-shape arrays. Its effect on dynamically allocated arrays is treated in Section 6.4.

For example, to align a 5-element vector **V** with the first row of the 5 x 5 matrix **A**,

```
      DIMENSION V(5), A(5,5)
CMF$  ALIGN V(I) WITH A(1,I)
```


Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

The effect is to place **V** in the same geometry as **A**, in the same processors as **A**'s first row:

```
VVVVV          AAAAA
.....          AAAAA
.....          AAAAA
.....          AAAAA
.....          AAAAA
```

Similarly, to align **V** with the last row of **A**:

```
CMF$  ALIGN V(I) WITH A(5,I)
```

as illustrated by

```
.....          AAAAA
.....          AAAAA
.....          AAAAA
.....          AAAAA
VVVVV          AAAAA
```

To align **V** with the first column of **A**:

```
CMF$  ALIGN V(I) WITH A(I,1)
```

as illustrated by

```
V....          AAAAA
V....          AAAAA
V....          AAAAA
V....          AAAAA
V....          AAAAA
```

6.2.1 Multiple Source Arrays

More than one source array can be aligned (in separate directive lines) with a single target. For example, to align **V** and another 5-element vector **Q** with different sections of **A**:

```
CMF$  ALIGN V(I) WITH A(I,1)
CMF$  ALIGN Q(I) WITH A(I,3)
```


Copyright © 1994 Thinking Machines Corporation

as illustrated by

```
V....       ..Q..       AAAAA
V....       ..Q..       AAAAA
V....       ..Q..       AAAAA
V....       ..Q..       AAAAA
V....       ..Q..       AAAAA
```

No interprocessor communication is required when **V** is used together with the first column of **A** or when **Q** is used together with the third column of **A**. However, **V** and **Q**, although in the same geometry, are not necessarily in the same processors. The compiler therefore generates communication routines for operations between the two vectors.

6.2.2 Multidimensional Source Arrays

The source array can of course be multidimensional. The target array must be of the same or higher rank. For example, to align a 2-dimensional array with the first two dimensions of a target 3-dimensional array on the target's third "row" (actually a "plane"),

```
      DIMENSION B(5,5), C(5,5,10)
CMF$  ALIGN B(I,J) WITH C(I,J,3)
```

The CM Fortran **ALIGN** directive does not support permutation of array dimensions. The dimension names in the source axis-specs must appear in the same order in the target axis-specs (although the target's axis-specs may intersperse dimension subscripts among the symbolic names).

```
      DIMENSION B(5,5), D(5,5)
CMF$  ALIGN B(I,J) WITH D(J,I)    [ not supported ]
```

6.2.3 Aligning Different-Sized Dimensions

A source array dimension must fit entirely within the corresponding dimension of the target array. Unless otherwise specified, the lower bound of the source dimension is aligned with the lower bound of the target dimension. For example, the directive

```
      DIMENSION  R(5), Q(20)
CMF$  ALIGN R(I) WITH Q(I)
```


Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

aligns the lower bounds of **R** and **Q**:

```
RRRRRR..............
QQQQQQQQQQQQQQQQQQQQ
```

You can supply a (positive) offset in the target's axis-spec to indicate another placement. For example, to align **R** with the section **Q(6:10)**, which is offset by five elements from **Q**'s lower bound:

```
CMF$  ALIGN R(I) WITH Q(I+5)
```

as illustrated by

```
.....RRRRRR..........
QQQQQQQQQQQQQQQQQQQQ
```

Extending this example to a 2-dimensional target array is straightforward. For example,

```
      DIMENSION V(5), B(10,10)
CMF$  ALIGN V(I) WITH B(2,I+3)
```

causes **V** to be allocated in the same processors that contain the section **B(2,4:8)**:

```
..........  BBBBBBBBBB
...VVVVV..  BBBBBBBBBB
..........  BBBBBBBBBB
..........  BBBBBBBBBB
..........  BBBBBBBBBB
..........  BBBBBBBBBB
..........  BBBBBBBBBB
..........  BBBBBBBBBB
..........  BBBBBBBBBB
```

Similarly, for a 2-dimensional source array,

```
      DIMENSION A(5,5), B(10,10)
CMF$  ALIGN A(I,J) WITH B(I+5,J+2)
```


Copyright © 1994 Thinking Machines Corporation

causes **A** to be allocated in the same processors that contain **B(6:10,3:7)**, as illustrated by

```
..........  BBBBBBBBBB
..........  BBBBBBBBBB
..........  BBBBBBBBBB
..........  BBBBBBBBBB
..........  BBBBBBBBBB
..AAAAA...  BBBBBBBBBB
..AAAAA...  BBBBBBBBBB
..AAAAA...  BBBBBBBBBB
..AAAAA...  BBBBBBBBBB
..AAAAA...  BBBBBBBBBB
```

6.2.4 Alignment Limitations

One current limitation of **ALIGN** is that the compiler doesn't know whether the upper bounds of arrays are the same. So even with **ALIGN**, all expressions involving **ALLOCATABLE** arrays require contextualized (masked) operations, which makes them slower than context-free computations.

In addition, there is currently no way to **ALIGN** only some dimensions of arrays. Thus, for instance, it is not possible to tell the compiler that **ALLOCATABLE** arrays having the same **NEWS** layout but different serial dimensions can be handled using memory copy operations rather than **SEND** communication.

6.3 Benefits and Costs of Explicit Alignment

The **ALIGN** directive can greatly enhance program performance when two arrays that are not conformable are often used together. By placing the arrays in the same geometry and aligning specified elements in the same processors, the compiler can use local memory accesses instead of communication to operate on corresponding elements of the two arrays.

There are, however, certain potential trade-offs that you should consider before using the **ALIGN** directive in an application program. These trade-offs are greater memory use, more subgrid looping, and interprocessor communication in cases where it would not normally be needed.

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

6.3.1 Memory Use

A potential penalty for the increase in speed that **ALIGN** gives is greater memory use. The layout of the source, or "aligned," array is determined by the target array. In cases where the source array is much smaller or of lower rank than the target array, a certain amount of the memory allocated for it in their shared virtual grid is never used. (Note all the "empty" PE dots in the above examples.) If the target array has serial dimensions, the potential for wasted memory is larger.

6.3.2 Subgrid Looping

The source array shares the subgrid size of the target array. If subgrid looping is required to cover all the memory allocated for the target array, the same amount of looping is required for the source array, even when the source array is used alone. For example:

```
      DIMENSION C(128), D(128, 512)
CMF$  LAYOUT D( :NEWS, :NEWS )
CMF$  ALIGN  C(I) WITH D(I,1)
```

Array **D** requires a virtual grid of 64K. If the program executes on a 512-processor partition, each processor loops over 128 memory locations. The same amount of subgrid looping, and consequent execution time, is required for array **C** in the statement **C=0**. In other words, clearing **C** takes about 128 times as long as it does without the **ALIGN** directive. However, triplet assignments do not need to loop over all elements of the array (see Section 7.3.3), thus **C(1:128)=0** does not take 128 times as long.

6.3.3 Communication Costs

An aligned array may no longer be in the same geometries as other arrays of its own shape. Array **C** in the example above is conformable with, but not in the same geometry as, another 128-element vector that is canonically allocated.

```
      DIMENSION C(128), D(128, 512), E(128)
CMF$  LAYOUT D( :NEWS, :NEWS )
CMF$  ALIGN  C(I) WITH D(I,1)
```


Copyright © 1994 Thinking Machines Corporation

If you use vectors **C** and **E** together, the operation will require interprocessor communication and thus greater execution time than if **C** had been allocated canonically.

6.4 Dynamic Array Alignment

Judicious use of the **ALIGN** directive can help avoid needless communication between arrays of different shapes. Dynamic arrays of the same shape are sometimes recognized as such by the compiler (as described below). If not, the compiler generates a new routine, reported by **-list** as **BLOCKMOVE**, which does geometry comparison at run time and chooses the most efficient code for an assignment where at least one operand is a dynamic array.

Note: The bounds expressions used in an **ALLOCATE** statement may enable the compiler to determine whether a deferred-shape array is conformable with other arrays. Then you won't need to explicitly **ALIGN** such arrays. See Section 6.4.4.

6.4.1 The Compiler's View of Dynamic Arrays

For static (compile-time constant) arrays and assumed-shape (function argument) arrays, the compiler generates code for the allocation of memory upon entry to the program unit. This code assigns the geometry that will be defined and used at run time. For dynamic arrays, however, space is not actually allocated until the **ALLOCATE** statement executes. The reason for this is that dynamic arrays may be allocated and deallocated more than once in each program unit, and the control flow of the code may dictate that one **ALLOCATE** may be executed at a different point in the execution than another **ALLOCATE**. For example,

```
SUBROUTINE FOO(N)
REAL, ALLOCATABLE, DIMENSION(:) :: A
IF (N .GT. 100) THEN
  ALLOCATE(A(100))
  A = [1:100]
ELSE
  ALLOCATE(A(N))
  A = [1:N]
END IF
```


Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

```
      PRINT *, A
      DEALLOCATE(A)
      END
```

Because of this, the compiler must generate the call to uniquely determine the geometry at the point that the array is allocated. However, the compiler must be able to refer to the geometry at any place that the array is referenced in the routine, even though the array has not yet been assigned a geometry. In order to do this, at the beginning of the routine the compiler allocates a temporary that will hold the result of the call to the run-time routine **intern_geometry**; when the geometry is needed at compile time, code that represents a fetch of the value stored in that temporary is generated.

Since the dynamic array's size may be different at different points in the program, no assumptions about the geometry of the array can be made with respect to other arrays that may be the same size at a given point in the execution. In previous releases, the compiler would therefore generate a communication operator when it encountered references to dynamic arrays:

```
      SUBROUTINE FOO(N)
      REAL,ALLOCATABLE,DIMENSION(:) :: A
      REAL B(N)

100   ALLOCATE(A(N))

      B = [1:N]
200   A = B
300   PRINT *, A
      DEALLOCATE(A)
      END
```

In this example, the assignment of array **B** to array **A** required a **SEND** of **B**, even though the two arrays are aligned by default.

Copyright © 1994 Thinking Machines Corporation

6.4.2 ALIGN Directives and Dynamic Arrays

The **ALIGN** directive allows the compiler to make assumptions about how a dynamic array will be used. The compiler can then attempt to generate more efficient code.

The **ALIGN** directive may be used with dynamic arrays in one of two ways:

- Align a dynamic array with a non-dynamic array:

```
      SUBROUTINE FOO(N)
      INTEGER N
      REAL*8, ALLOCATABLE :: A(:), B(:)
      REAL*8 X(N)
CMF$  ALIGN A(I) WITH X(I)
```

- Align a dynamic array with another dynamic array:

```
      SUBROUTINE FOO(N)
      INTEGER N
      REAL*8, ALLOCATABLE :: A(:), B(:)
      REAL*8 X(N)
CMF$  ALIGN A(I) WITH B(I)
```

Aligning Dynamic Arrays with Non-Dynamic Arrays

You can **ALIGN** an allocatable array or array pointer with an explicit-shape or assumed-shape array. In either situation, the compiler assigns to the dynamic array the same geometry that it assigns to the target array. For example, in this case,

```
      SUBROUTINE FOO(N)
      INTEGER N
      REAL*8, ALLOCATABLE :: A(:), B(:)
      REAL*8 X(N)
CMF$  ALIGN A(I) WITH X(I)

      ALLOCATE(A(N))
 100  X = A
```

the array **A** will be assigned the same geometry as the array **X**. Therefore, the compiler should not generate a **SEND** for the assignment at the statement labeled 100.

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

Similarly, in the case,

```
      SUBROUTINE FOO(N)
      INTEGER N
      REAL*8, ALLOCATABLE :: A(:), B(:)
      REAL*8 X(1000)
CMF$  ALIGN A(I) WITH X(I)
      ALLOCATE(A(1000))
 100  X = A
```

the compiler should not generate a **SEND** for the assignment labeled 100.

Aligning Dynamic Arrays with Other Dynamic Arrays

You can **ALIGN** one allocatable array or array pointer with another dynamic array:

```
      SUBROUTINE FOO(N)
      INTEGER N
      REAL*8, ALLOCATABLE :: A(:), B(:)
      REAL*8 X(N)
CMF$  ALIGN A(I) WITH B(I)
      ALLOCATE(A(N))
      ALLOCATE(B(N))
 100  B = A
```

In this case, there should not be a **SEND** generated for the assignment labeled 100.

6.4.3 Using ALIGN Effectively with Dynamic Arrays

Align Multiple Arrays

If the desired effect is to have dynamic arrays aligned with both static and dynamic arrays, use one static array as the base destination of the **ALIGN** directive:

```
      SUBROUTINE FOO(X,N)
      INTEGER N
      REAL *8, ALLOCATABLE :: A(:), B(:)
      REAL*8 X(N+5)
      REAL*8 C(N+5)
      REAL*8 ANS
```


Copyright © 1994 Thinking Machines Corporation

```
CMF$  ALIGN A(I) WITH X(I)
CMF$  ALIGN B(I) WITH X(I)
CMF$  ALIGN C(I) WITH X(I)
```

In this example, the arrays **A**, **B**, and **C** are all aligned with the base destination array **X**, which is a dummy argument array. The compiler then views all four arrays as having the same geometry.

Avoid Using Allocatables Unnecessarily

In many cases, an assumed-shape dummy or local array may be just as useful as a dynamic array. For example, if the array shape is determined by an argument passed into a subprogram, then an assumed-shape array will work just as well.

Instead of this:

```
      SUBROUTINE FOO(X,N)
      INTEGER N
      REAL*8, ALLOCATABLE :: A(:)
      ALLOCATE(A(N))
```

Do this:

```
      SUBROUTINE FOO(X,N)
      INTEGER N
      REAL*8 A(N)
```

Be Careful!

As always when you use directives, make sure that an **ALIGN** directive with dynamic arrays does not conflict with the code. Ill-defined code sequences result in undefined results. For example,

```
      SUBROUTINE FOO(X,N)
      INTEGER N
      REAL*8, ALLOCATABLE :: A(:)
      REAL*8 X(N+5)
      REAL*8 C(N+5)
      REAL*8 ANS
CMF$  ALIGN A(I) WITH X(I)
 100  ALLOCATE(A(-200:0))
      A(-100:0) = X(1:101)
```


Copyright © 1994 Thinking Machines Corporation

Here, the dummy array **X**'s lower bound is 1, while the allocatable array **A**'s lower bound is −200 at the time of allocation. In this situation, the results are unpredictable — the code may work or it may not.

6.4.4 Compile-Time Shape Checking

The bounds expressions used in an **ALLOCATE** statement may enable the compiler to determine whether a deferred-shape array is conformable with other arrays. In the absence of directives, conformable arrays are known to have the same geometry, and the compiler can avoid generating communication for operations between them.

The compiler checks the bounds expressions in **ALLOCATE** statements for

- **ALLOCATABLE** arrays
- array pointers that are not in **COMMON**

If the bounds are specified with

- constant expressions
- invariant dummy arguments
- **UBOUND** or **LBOUND** expressions referencing other arrays (the compiler also accepts the older **DUBOUND**/**DLBOUND**)

the compiler can determine whether the deferred-shape array is conformable with other arrays used with it, including other deferred-shape arrays, as well as static, automatic, and dummy arrays.

The compiler cannot make assumptions about shape in the following cases (and so does not check bounds):

- array names that appear in more than one **ALLOCATE** statement in the program unit (even if the statements are within a conditional construct and only one will be executed)
- array pointers in **COMMON** (since they may have been reallocated in another program unit)

Copyright © 1994 Thinking Machines Corporation

For example, the compiler can determine that these two arrays are conformable because the **ALLOCATABLE**'s upper bound is a constant:

```
PARAMETER (N=1000)
INTEGER DIMENSION(N) :: A
INTEGER, DIMENSION(:), ALLOCATABLE :: B
ALLOCATE (B(N))
```

The compiler can determine that these two pointers are conformable because **B**'s upper bound is derived from **A**'s with **UBOUND**:

```
INTEGER, DIMENSION(:), POINTER :: A,B
READ *,N
...
ALLOCATE (A(N))
ALLOCATE (B(UBOUND(A,1)))
```

If the second example had been written

```
READ *,N
...
ALLOCATE (A(N))
ALLOCATE (B(N))
B=A
```

the compiler would not try to prove that **N** had not changed between the **ALLOCATE** statements. Instead, it would generate **BLOCKMOVE** code before the assignment to check for alignment at run time. (See the release notes for 2.1 Beta 2.0 for information on run-time alignment checking of dynamic arrays.)

In addition, the compiler does not check for multiple uses of a bounds expression within a single **ALLOCATE** statement unless the expression meets the criteria stated above. For example, it would generate **BLOCKMOVE** code for operations between these two arrays:

```
READ *,N
...
ALLOCATE (A(N), B(N))
```


Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

Run-Time Geometry Checking

When the compiler cannot compare the shapes of dynamic arrays (as described in the previous section), it generates code reported by **-list** as **BLOCKMOVE**. This item indicates code that compares the geometries of source and destination arrays at run time when either operand is a deferred-shape array. The code then generates memory accesses if possible; if not, **NEWS** communication if possible; or **SEND** routines as a last resort.

Note, however, that there may still be overhead even when **BLOCKMOVE** translates to local memory accesses. This is because **BLOCKMOVE** causes the compiler to break the current PN code block and move data to a temporary (against the possibility that communication is required at run time).

6.5 Array Aliasing

Another way of affecting the compiler's view of a parallel array is by the use of *array aliasing*. You can instruct the compiler to view part or all of an array's data as an array of a different shape, size, or data type. This is done by the use of *alias arrays*. Aliasing is similar in effect to the **EQUIVALENCE** statement for scalar variables; it causes the compiler to view a single region of CM parallel memory as being more than one type of array.

However, the effect of aliasing is more complex than that of **EQUIVALENCE** statements. Equivalencing two variables simply gives the same memory location two names. Aliasing an array changes the way the compiler treats it with regard to communication operators and subgrid looping in PE code blocks. Array aliasing can provide you with greatly increased performance if you use it correctly, but can easily produce hard-to-read, hard-to-debug code if you're careless with it.

Copyright © 1994 Thinking Machines Corporation

IMPORTANT

Unless you clearly understand the layouts of the array(s) you are aliasing, the array aliasing tools can easily produce subtle, invisible errors in your program. Most common is the treatment of garbage locations in an array as actual data — this is caused by an incorrect aliasing operation that fails to take into account the location and layout of the garbage data. If you use aliasing in your code, refer to the list of restrictions and caveats in Section 6.5.2 and memorize them — ignoring these restrictions is a sure-fire method of increasing your frustration level in debugging your code.

6.5.1 Array Aliasing: A Quick Overview

Array aliasing is also described in the *CM Fortran Libraries Reference Manual*, but here is a brief overview. A CM array can be viewed as having two major parts (see Figure 25):

- An *array descriptor* (including a geometry object) stored in the memory of the PM. The array descriptor describes the layout of the array data in CM memory, and provides information needed for executing subgrid loops in PE code blocks.
- A *memory region* in each PE on the CM, containing array data and possibly garbage locations. Since the memory region for each PE begins at the same address in PE memory, the total array memory region can be viewed as a stripe across the memory of the entire CM.

Among other things, the array descriptor contains the starting address of the parallel memory region on the CM. (This is the thick line in Figure 25). If the array data is laid out correctly in CM memory, the data can be viewed differently by simply changing the information in the array descriptor and/or altering the starting point of the array.

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

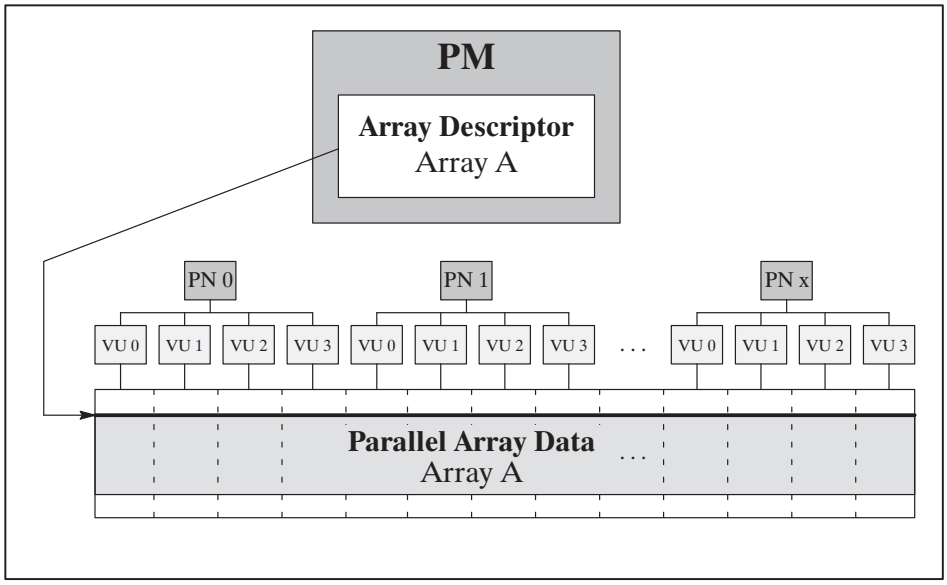

Figure 25. A CM Fortran array: array descriptor and parallel memory region.

That's all that an array alias is: a copy of an existing array descriptor with changes made to the data type, geometry, or parallel memory starting address. An array alias involves no actual movement or copying of array data. It's simply creative misinformation — you're telling the CM Fortran compiler to look at existing array data differently.

Here's an example of how you might use array aliasing to alter the data type of an array. Suppose you have an array defined to be of type **COMPLEX**, and laid out so that there are 8 complex values stored on each PE. You can use array aliasing to access the complex data as an array of type **REAL** with twice as many elements on each PE. For example,

```
      PROGRAM ALIASTYPE
      INCLUDE 'CMF_defs.h'
      INTEGER NPES
      NPES = CMF_NUMBER_OF_PROCESSORS()
      CALL SETUP(NPES)
      END

      SUBROUTINE SETUP(N)
      INCLUDE 'CMF_defs.h'
      INTEGER N
C***** DEFINE ORIGINAL ARRAY, AXIS 1 IN PE MEMORY
      COMPLEX C(N,8)
```


Copyright © 1994 Thinking Machines Corporation

```
CMF$  LAYOUT C(:NEWS,100:NEWS)
      INTEGER A(CMF_SIZEOF_DESCRIPTOR), ASTAT
CMF$  LAYOUT A(:SERIAL)
C***** TYPE ALIAS, CHANGE LENGTH OF AXIS 2 (0 BASED)
      ASTAT=1
      CALL CMF_ALIAS_TYPE(A,CMF_FLOAT,C,
     $                    CMF_MUTATE_SELECT,ASTAT)
      CALL RUNIT(NPES,C,A)
      END

      SUBROUTINE RUNIT(N,C,A)
      COMPLEX C(N,8)
CMF$  LAYOUT C(:NEWS,100:NEWS)
      REAL A(N,16)
CMF$  LAYOUT A(:NEWS,100:NEWS)
C***** INITIALIZE COMPLEX ARRAY
      C=(1,2)
C***** SET REAL AND IMAGINARY COMPONENTS VIA ALIAS
      A(:,1:15:2)=8
      A(:,2:16:2)=4

C***** DISPLAY ORIGINAL ARRAY TO SHOW EFFECT:
      DO I=1,10
         PRINT 10,(C(I,J),J=1,8)
 10      FORMAT (100('(',F2.0,',',F2.0,') '))
      END DO
      END
```

As you can probably guess, you have to be pretty good at choosing and defining array layouts in order to use array aliasing effectively. After all, you have to know what's already there (the arrangement of data, its type, etc.) before you can decide what you want to tell the compiler about it.

6.5.2 Restrictions and Caveats

Take note of the following restrictions and warnings, and you'll avoid many of the most common pitfalls related to array aliasing:

- The aliasing procedures behave incorrectly if the **ARRAY** argument — the target of an alias constructor or the array queried by an inquiry procedure — is aligned in a strided or offset fashion with another array or aligned with the less rapidly varying axes of an array of higher rank.

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

- Beware of padding in either a target array or its alias. If the target array and the alias have different geometries, the internal garbage-masking software will mistake the location of garbage elements in the alias. The result can be that real data is masked and garbage data is computed upon and stored. Recall that the **cmf** switch **-nopadding** removes vector-length padding only; global padding, if any, is not affected by this switch.

- It is not advisable to pass to a subprogram two (or more) array arguments that address the same or overlapping memory if the subprogram changes either argument. The compiler assumes that argument memory does not overlap and conducts its dependence analysis and optimizations on this assumption. (Moreover, the compiler is justified in this assumption because aliasing of this kind is forbidden by the ANSI Fortran standard.)

- An example of aliases confounding dependence analysis is seen in the compiler's internal strip-mining. For example, given vectors **A**, **B**, and **A1**, suppose that **A1** is aliased to the location offset by one element from the base of **A**. Then, consider a program executing under the CM-5 vector units model that contains the statements:

```
A = EXPRESSION_1
B = A1 + EXPRESSION_2
```

  If both these source lines are included in one local code block, the procedure is as follows: The code block turns into a loop where each iteration of the loop works on a vector of length 8. In the first iteration of the loop, elements 1:8 of **A** are updated. Note that these are elements 2:9 of **A1**. So the following assignment to elements 1:8 of **B** is using new values from elements 1:7 of **A1**, and an old value for element 8 of **A1**. This is probably not what you, as a programmer, would intend by these statements.

Copyright © 1994 Thinking Machines Corporation