## Chapter 12

# CM Array Layout

The CM-5's run-time system (RTS) distributes array elements among the processors (either nodes or vector units) that are available to the program. The particular distribution of an array is called its *layout*, and the layout chosen by the RTS without intervention from the user is the *canonical layout*.

CM Fortran normally spares the user the details of how arrays are laid out. However, you might improve program performance by using the compiler directive **LAYOUT** to control an array's layout yourself. Sophisticated use of this tool requires an understanding of how the RTS maps arrays onto the parallel memory and the implications of changing the canonical mapping.

You can also use the compiler switches **-nopadding** and **-noaxisreorder** to make global changes in the RTS's decision rules and thus in the canonical mapping for a particular program.

This chapter focuses on the **LAYOUT** directive and its impact on RTS decision making. Some related features will become clear once you understand how array layout is handled:

- the utility library procedures **CMF_ALLOCATE_LAYOUT_ARRAY** and **CMF_ALLOCATE_DETAILED_ARRAY**
- the utility library procedure **CMF_ALIAS_DETAILED**
- the compiler directive **ALIGN**

The **ALIGN** directive is described in more detail in the *CM Fortran Language Reference Manual* and the utility procedures in the *CM Fortran Libraries Reference Manual.* See also the *CM-5 CM Fortran Performance Guide* for additional information on the performance considerations in using all these features.


*Copyright © 1989–1994 Thinking Machines Corporation*

## 12.1 The Canonical Layout

This section describes how the current implementation of the run-time system lays out arrays. For convenience, it focuses on the CM-5 with vector units, but the same principles apply in the main to a CM-5 without vector units (see Section 12.2 for the differences).

Let's say you have an 8 x 12 array in a program running on a 16-vector-unit (4-node) partition. How does the RTS determine how to lay out the elements on the 16 vector units? Understanding how it does this requires understanding five concepts:

- physical grids
- garbage elements
- subgrids
- axis sequence
- subgrid sequence

### 12.1.1 Physical Grids

When laying out an array, the RTS configures the vector units into a grid whose rank is the same as the rank of the array. The total number of vector units in a partition is always a power of 2; therefore, the number of vector units along each axis of the physical grid must be a power of 2. Thus, for the 8 x 12 example, the RTS has these choices for arranging the 16 vector units into a 2-dimensional physical grid: 4 x 4, 8 x 2, 2 x 8, 16 x 1, and 1 x 16.

### 12.1.2 Garbage Elements

The RTS tries to divide the array's elements equally among the vector units. In doing so, it follows these rules:

- Each vector unit must receive the same number of elements.
- The number of elements per vector unit must be a multiple of the value returned by the utility library function **`CMF_ARRAY_SIZE_QUANTUM`**.


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

In the **-vu** execution model, the default value is 8 (the length of the vector units' vector register). It is 1 in the **-sparc** model.

However, it is not possible to follow these rules in laying out an 8 x 12 array on 16 vector units; each vector unit could receive the same number of elements, but the number of elements would not be a multiple of 8.

In such a case, the RTS internally uses a *machine array* with larger extents along one or more axes, so that the rules can be followed. It then lays out the actual array within this larger layout, leaving unused elements along the extended axes. These unused elements are referred to as *garbage elements* or *padding*.

The RTS always adds garbage elements to the high end of an axis, and adds as few garbage elements as possible. For our 8 x 12 array, the RTS would pad axis 2 to make a machine array of 8 x 16. Its layout on a 2 x 8 physical grid is shown in Figure 19.

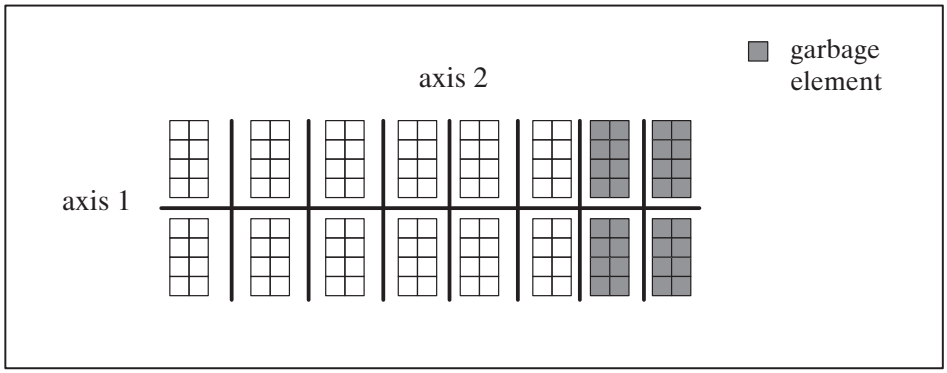


**Figure 19. An 8 X 12 array laid out on a 2 X 8 physical grid.**

You can think of garbage elements as coming from two distinct sources:

- *Global padding* is added, if necessary, to make a machine array's size an integer multiple of **CMF_NUMBER_OF_PROCESSORS**.
- *Vector-length padding* is added, if necessary, to make the subgrid an integer multiple of **CMF_ARRAY_SIZE_QUANTUM** (the vector length of 8 in VU model).

The compiler switch **-nopadding** removes the multiple-of-8 rule for subgrids under the **-vu** execution model. With this switch, the subgrid can be


Copyright © 1989–1994 Thinking Machines Corporation

any arbitrary length (**CMF_ARRAY_SIZE_QUANTUM = 1**). Note, however, that global padding may still be added if the array does not divide evenly among the vector units. The multiple-of-8 rule is in effect by default to support back compatibility.

With **-nopadding**, an 8 x 12 array would be laid out on a 4 x 4 physical grid, with 2 x 3 subgrids and no garbage elements.

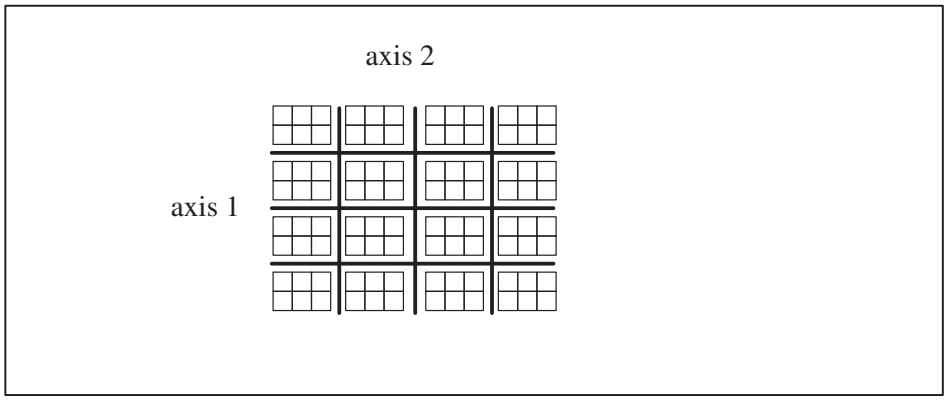


**Figure 20. An 8 X 12 array on 16 VUs with -nopadding.**

The RTS deals with padding transparently, guaranteeing that garbage data does not escape and corrupt user data. However, padding does have performance consequences, especially for communication. Also, you must take it into account when specifying detailed layouts yourself.

You can determine whether any given array is padded by calling the utility library subroutine **CMF_DESCRIBE_ARRAY**. If the output shows a machine array with a different number of elements from the user array, the difference is padding. For example, this array fits onto the current partition without padding:

```
DOUBLE PRECISION, DIMENSION(128,128,8,16) :: A
...
CALL CMF_DESCRIBE_ARRAY(A)
```



Copyright © 1989–1994 Thinking Machines Corporation

as shown by this output:

```
Array geometry id: 0xa3610
    Rank: 4
    Number of elements: 2097152
    Extents: [128 128 8 16]
    Machine geometry id: 0xa9e50, rank: 4
    Machine geometry elements:  2097152
...
```

### 12.1.3 Subgrids

Continuing its layout strategy, the RTS divides the (machine) array into a number of sections, each section corresponding to a vector unit. These sections are called *subgrids*.

Note the layout requirements discussed so far:

- The physical grid is of the same rank as the user array and must have a power-of-2 number of vector units along each axis.
- Each vector unit must contain the same number of elements — in other words, the subgrid must be the same size on each. The subgrid also has the same rank as the physical grid and the user array.
- With the default **-padding**, the subgrid must be a multiple of 8 elements; with **-nopadding**, the subgrid can be any arbitrary size.

Our sample 8 x 12 array, padded to 8 x 16 by default, will have 8 elements per vector unit. But, will they be laid out as a 4 x 2 subgrid, as shown in Figure 19, or, for example, as a 1 x 8 subgrid? The 1 x 8 layout implies a physical grid of 8 x 2, as shown in Figure 21.


Copyright © 1989–1994 Thinking Machines Corporation

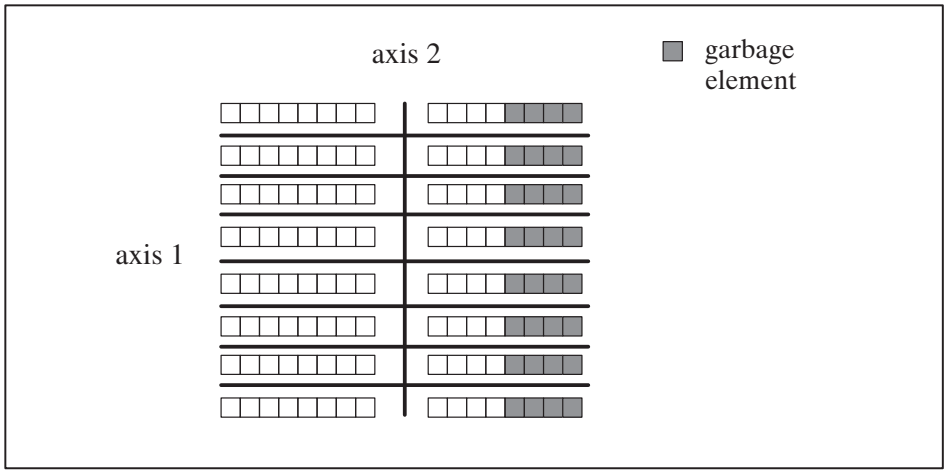


**Figure 21. An 8 x 12 array laid out on an 8 x 2 physical grid.**

The basic rule that the RTS follows is to minimize the size of the subgrid. In other words, it uses as few garbage elements as possible. This doesn't help it choose between the 2 x 4 and 1 x 8 subgrids, which both have the same number of garbage elements. It does, however, eliminate subgrids that use fewer than all 16 vector units. In practice, the RTS chooses a layout that uses fewer than all the vector units only when the number of elements in the array is small relative to the number of vector units on which the program is running.


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

### 12.1.4 Axis Sequence

One piece of information left out of the layouts shown in Figure 19 and Figure 21 is the numbering of the vector units within the physical grid. In the 2 x 8 physical grid layout, for example, Figure 22 shows two ways in which the vector units could be numbered.

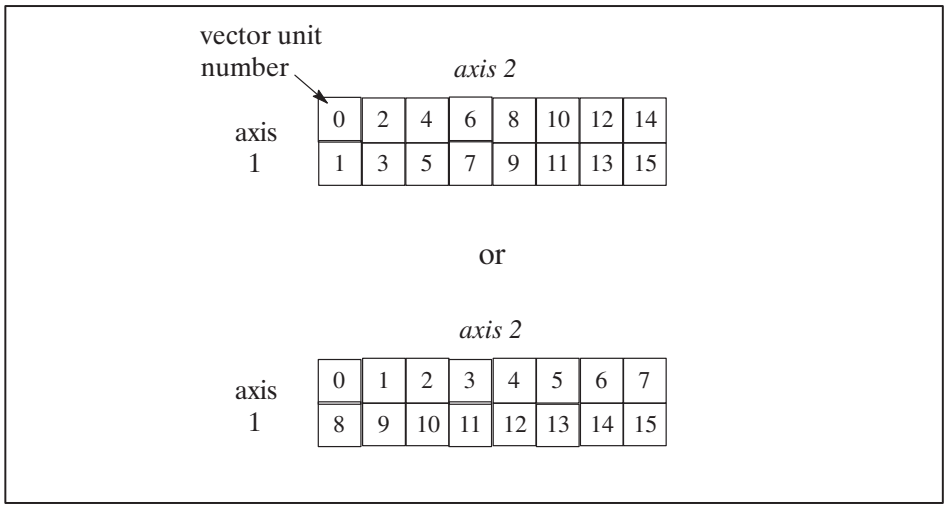


**Figure 22. Two possible vector-unit numberings.**

Vector units 0−3 are on node 0, vector units 4−7 are on node 1, etc.

In the example on the top in Figure 22, the vector-unit numbers increase fastest (that is, by the smallest interval) along axis 1; we call this an *axis sequence* of (1,2) — the first axis in the sequence is the one that varies fastest. In the example on the bottom in Figure 22, they increase fastest along axis 2; this corresponds to an axis sequence of (2,1).

By default, the current implementation of the RTS lays out multidimensional arrays so that the vector-unit numbers vary fastest along the highest-numbered axis — that is, it would choose the axis sequence of (2,1) in this example. The compiler switch **`-noaxisreorder`** causes it to choose axis sequence (1,2).


Copyright © 1989−1994 Thinking Machines Corporation

### 12.1.5 Subgrid Sequence

The final issue with regard to the canonical layout is how the elements in the subgrid are arranged into linear order in the memory of a vector unit. This is known as the *subgrid sequence*.

If you have a subgrid whose dimensions are 4 x 2, there are two possible layouts of the elements in vector-unit memory:

```
(1,1)                (1,1)
(1,2)                (2,1)
(2,1)                (3,1)
(2,2)       or       (4,1)
(3,1)                (1,2)
(3,2)                (2,2)
(4,1)                (3,2)
(4,2)                (4,2)
```

By default, the current implementation of the RTS chooses the layout on the left; it reorders the axes such that the highest-numbered axis varies fastest (that is, the adjacent subgrid elements along the highest-numbered axis are contiguous in memory). By specifying the non-default **`-noaxisreorder`**, you cause it to lay out elements in column-major order (leftmost is fastest).

**NOTE:** The default sequence becomes more complicated when serial axes are specified (as shown below).

### 12.1.6 Putting It All Together

It turns out that the RTS chooses a 2 x 8 physical grid for our 8 x 12 array. Given the information covered so far, we can now see exactly how the RTS would lay out the array using this physical grid (Figure 23).



Copyright © 1989–1994 Thinking Machines Corporation

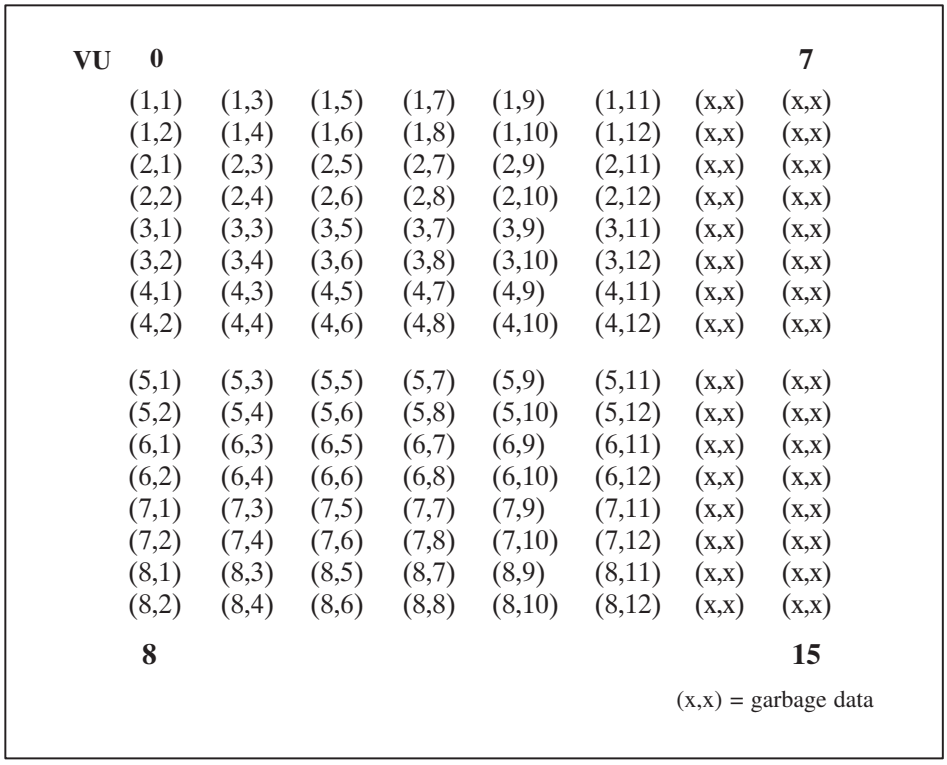


**Figure 23. Default layout of an 8 x 12 array on 16 vector units.**

Note that the padding occurs in vector units 6, 7, 14, and 15; other layouts would put the padding in other vector units.

Note these general performance rules for vector units:

- Data movement within a vector unit is faster than data movement between vector units. Thus, in the default, data movement would be faster along axis 2.
- Data movement between the two vector units on a chip is faster than data movement across the two chips of a node.
- Data movement between vector units on the two chips of a node is faster than data movement off-node.

To take advantage of the chip-versus-node distinction, you can specify layout in detail, as shown below.


*Copyright © 1989–1994 Thinking Machines Corporation*

## 12.2 Layout without Vector Units

The same general layout principles apply to all CM Fortran execution models. This section notes a few differences you will see if you are using the CM-5 global SPARC nodes model.

### 12.2.1 CM-5 SPARC Nodes Model

If your program is executing under the SPARC nodes model, the processing elements are the nodes instead of vector units. The layout rules are the same, with one exception — the rule that the subgrid must be a multiple of 8 elements does not apply. The RTS does not add vector-length padding in the SPARC model, although it may need to add global padding.

If the RTS were to lay out our sample 8 x 12 array on 4 nodes, it could choose among the physical grids and subgrids shown in Figure 24.

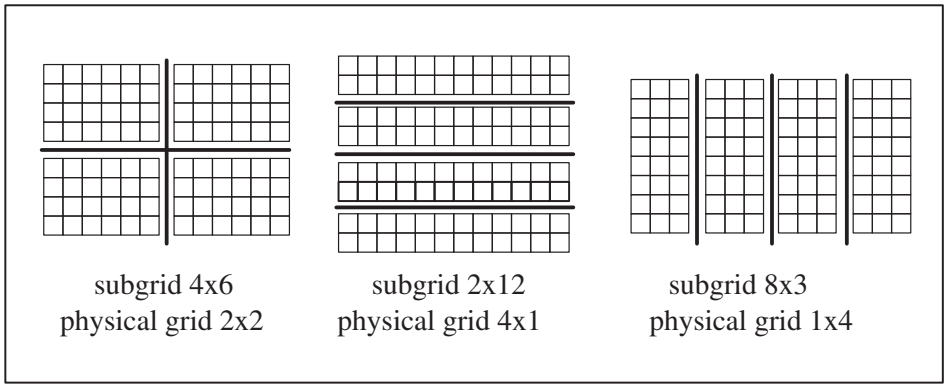


**Figure 24. Subgrids and physical grids for an 8 x 12 array on four nodes.**

No global padding is required in this example, because the 96 elements of the array divide evenly into 24 per subgrid.

The RTS in this case would choose the 4 x 6 subgrid. The actual default layout is shown in Figure 25.


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

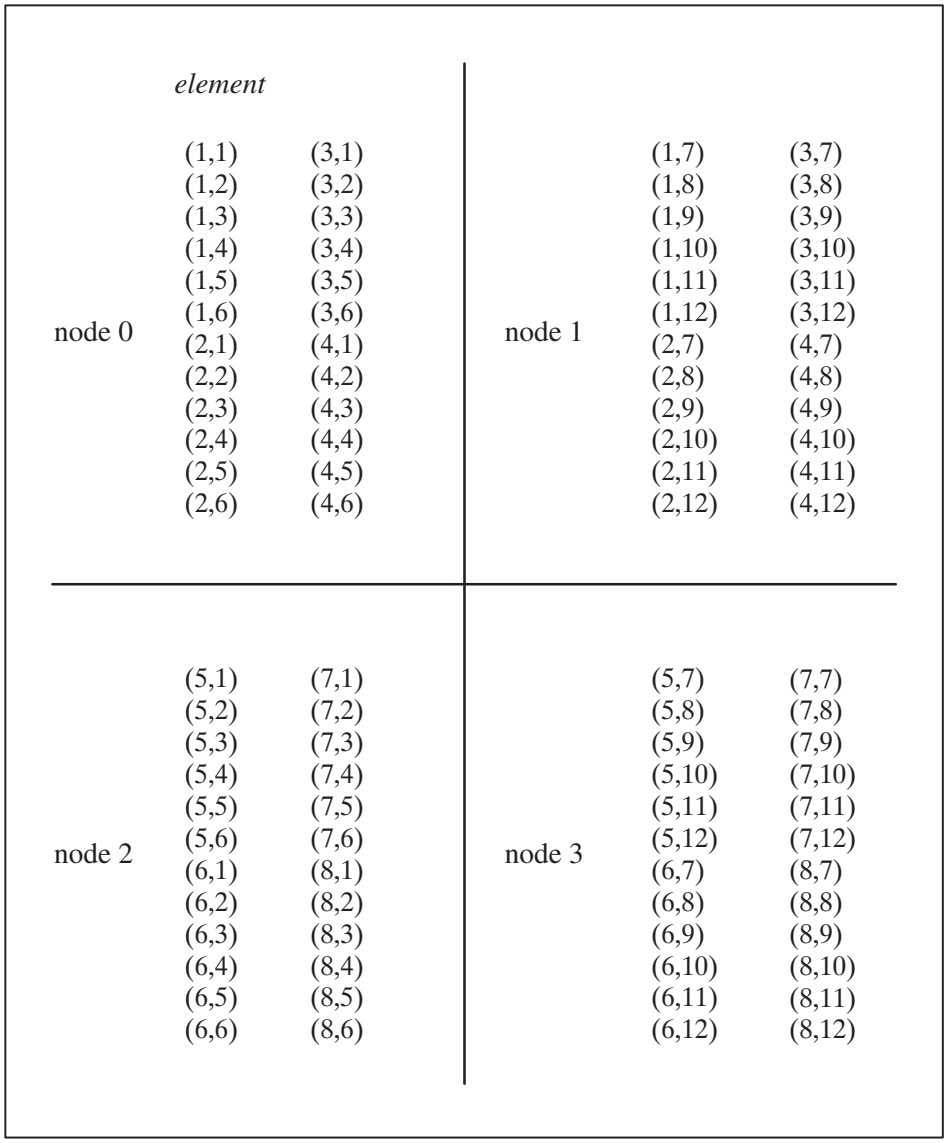


**Figure 25. Default layout of an 8 x 12 array on four nodes.**


*Copyright © 1989–1994 Thinking Machines Corporation*

## 12.3 Controlling Array Layout: Axis Ordering and Weights

The compiler directive **LAYOUT** enables you to guide the RTS in laying out an array. Two fairly straightforward characteristics you can specify are:

- serial axes
- weighting of axes

Some more detailed syntax is shown in Section 12.6.

### 12.3.1 Syntax

Only one **LAYOUT** directive can be applied to an array in a program, and the directive must be repeated in every program unit where that array is used. The directive has the form:

**CMF$ LAYOUT** *array-name* **(** *axis-1-spec*, *axis-2-spec*, . . . **)**

Each array dimension must have exactly one axis descriptor *(axis-spec)*, which specifies the ordering and weight of the dimension. In a simple **LAYOUT** directive, the ordering can be either **:SERIAL** or **:NEWS**, and each **:NEWS** keyword can be preceded by a literal or named integer constant that indicates the relative weight.

For example,

```
      DIMENSION A( 100,100,100 )
CMF$  LAYOUT A( :SERIAL, 2:NEWS, :NEWS )
```

This directive specifies that array **A** is to be laid out with one serial dimension and two NEWS-ordered dimensions, and that dimension 2, with a weight of 2, is to be favored for interprocessor communication over dimension 3, which gets the default weight 1.

Since NEWS is the default axis ordering, the keyword may be omitted and just a weight supplied; or, both ordering and weight may be omitted as long as placeholder commas remain. Thus, the following directive is equivalent to that above:

```
      DIMENSION   A( 100,100,100 )
CMF$  LAYOUT      A( :SERIAL, 2, )
```


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

Increasing the relative dimension weight does not guarantee a proportional increase in communication speed for a dimension, but it might help the system in selecting from among several possible layouts.

#### Note on :SEND Ordering

CM Fortran on CM-5 systems accepts the axis specification **:SEND** in **LAYOUT** directives to provide back compatibility with CM-2 applications. However, the RTS lays out a **:SEND** axis in the same way as a **:NEWS** axis.

You should avoid SEND ordering on the CM-5 whenever possible. Although the compiler accepts it without protest, it does not recognize that this ordering is identical to NEWS ordering. It therefore generates communication for an elemental operation between a **:NEWS**-ordered array and an otherwise identical **:SEND**-ordered array. Also, some CM-5 libraries do not accept **:SEND**-ordered arrays.

### 12.3.2 Serial Axes

Specifying an axis as **:SERIAL** in a **LAYOUT** directive has two effects:

- All elements along the axis will be located on the same vector unit.

  This layout is useful if communication tends to occur mostly along that one axis, since communication is faster within a vector unit than between vector units. It is also useful if the serial axis is used to simulate a "structure," whereby you specify a parallel "slice" of the array by supplying a scalar or triplet subscript for the serial axis.

  For example, if your array is 3 x 8 x 12 and you specify that axis 1 is to be serial, all elements along axis 1 would be located on the same vector unit; the physical grid of the remaining two axes is the same as it would be if axis 1 didn't exist — see Figure 23. One result of this is that the RTS never adds garbage elements to a serial axis; it always satisfies the requirements for garbage elements in the non-serial axes.

- By default (with the switch **-axisreorder**), the elements along a serial axis vary more slowly than elements along any non-serial axis — that is, they must be furthest away from each other in memory. For example, if you have a subgrid that is 2 x 3 x 4, by default the order of elements in memory is


Copyright © 1989–1994 Thinking Machines Corporation

```
(1,1,1)
(1,1,2)
(1,1,3)
(1,1,4)
(1,2,1)
(1,2,2)
(1,2,3)
(1,2,4)
(1,3,1)
(1,3,2)
```
etc.

If you were to specify that axis 2 is serial, the sequence would be

```
(1,1,1)
(1,1,2)
(1,1,3)
(1,1,4)
(2,1,1)
(2,1,2)
(2,1,3)
(2,1,4)
(1,2,1)
(1,2,2)
```
etc.

Axis 2 now varies most slowly. If you have more than one serial axis, by default the highest-numbered (rightmost) axis varies fastest, just as it does for non-serial axes. In effect, serial and non-serial axes are ordered independently. The rules are:

- All NEWS axes vary faster than any serial axis. (This guarantees that a section formed by scalar indexing into all serial axes is a contiguous block of memory.)
- The NEWS axis declared rightmost varies faster than other NEWS axes, the next rightmost NEWS axis varies next fastest, and so on.
- The serial axis declared rightmost varies faster than other serial axes, the next rightmost serial axis varies next fastest, and so on.

If you specify the switch **`-noaxisreorder`**, then axes are laid out in the order declared, with the leftmost varying fastest, the next leftmost next fastest, and so on, regardless of which are serial or parallel.

For example, if you have a 2 x 3 x 4 subgrid with the middle axis serial and **`-noaxisreorder`**, the sequence would be:


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

```
(1,1,1)
(2,1,1)
(1,2,1)
(2,2,1)
(1,3,1)
(2,3,1)
(1,1,2)
(2,1,2)
(1,2,2)
(2,2,2)
(1,3,2)
(2,3,2)
```
etc.

### 12.3.3 Weighting Axes

The layout tools enable you to influence the RTS's choice of layout by differentially weighting the array axes to indicate which will be most used in communication operations. By default, the RTS weights all axes evenly.

When an axis is weighted more highly than others, the RTS tries to localize more of the elements along that axis within a node or vector unit, thereby reducing the cost of the communication along that axis. In the example of laying out ax 8 x 12 array on 16 vector units, weighting axis 2 would tend to result in a 1 x 8 subgrid. If you have more than two dimensions, you can assign different weights to each dimension, or assign the same weight to two or more of the dimensions, causing the RTS to treat them similarly.

Note that the run-time system will not necessarily be able to lay out the array to completely reflect the weights you assign to the axes. In the current implementation, once it has chosen a subgrid with the fewest elements, it uses the weights to trade off factors of 2 in the lengths of subgrid axes against factors of 2 in the length of physical-grid axes, attempting to give axes with higher weights a longer subgrid length and a smaller physical-grid length.


Copyright © 1989–1994 Thinking Machines Corporation

**NOTE**

The use of axis weights is deprecated in favor of the detailed **LAYOUT** syntax described below. The RTS may find that layout is totally determined by other factors before weights are considered, and so it often cannot respond usefully to this directive.

## 12.4 Performance Issues

The main reason to use the layout tools is to improve performance over what you can obtain using the canonical layout. This section summarizes the performance issues you should consider (most of these issues have already been mentioned in previous sections).

Although the discussion here refers to vector units, it also applies to programs running on CM-5s without vector units.

### 12.4.1 Effect of Subgrid Length and Physical Grid

As we discussed above, the RTS by default chooses the smallest subgrid size. This is the most efficient size for operations that are local to a vector unit.

When you lengthen a subgrid axis, you improve the efficiency of communication along that axis at the expense of communication along other axes; you also decrease the overall efficiency of operations within the vector unit.

For example, assume that we're using a subgrid size of 8 x 4 and we want to do a shift of distance 1 along axis 2 — that is, each element sends its value to the element that is one coordinate higher along axis 2.

Figure 26 shows the subgrids on two vector units; each vector unit moves 8 values to the next vector unit, and 24 values within the vector unit.


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

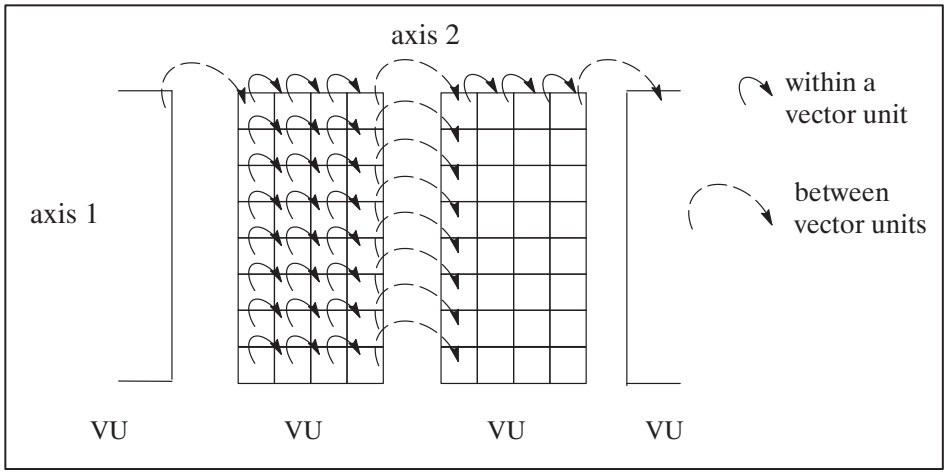


**Figure 26. On-VU and off-VU data movement for a NEWS operation along axis 2.**

In general, the number of off-VU moves in an operation like this is equal to the total number of elements in the subgrid divided by the subgrid length of the axis along which the communication is taking place (32/4 = 8); this value is referred to as the *subgrid-orthogonal-length*.

The number of moves within the vector unit is the total number of elements in the subgrid minus the subgrid-orthogonal-length (32−8 = 24). This number is roughly proportional to the subgrid size. As long as the subgrid size stays roughly constant, changing the layout does not greatly affect the cost of these on-VU moves. Decreasing the subgrid-orthogonal-length of a subgrid axis does, however, result in better communication performance along the axis.

### 12.4.2 Effect of Serial Axes

Making an axis serial guarantees that all elements along the axis are on the same vector unit— that is, no off-VU data movement is required. This means that a serial axis will have optimal within-VU performance, once again at the expense of communication along other axes.

But beware that, since all the other constraints on array layout must be satisfied by the non-serial axes, an array with only a small number of elements


Copyright © 1989−1994 Thinking Machines Corporation

along the non-serial axes can end up with an inefficient layout, because it will require a large number of garbage elements.

This is particularly true for the default **-padding** in **-vu** model. Since the RTS uses block layout — whereby successive elements are placed "down memory" rather than across processors — it is not possible to lay out an array with 1 element of a parallel axis per processor. Instead, 8 elements are laid out in the first few processors, along with all the serial axis elements, leaving the remaining processors with garbage elements only.

### 12.4.3 Effect of Garbage Elements

If an array has garbage elements, they are not actually part of the array's data, but must be taken into account by communication functions. Often this requires extra work to move data "over" the garbage locations, thus decreasing efficiency. You should therefore avoid choosing an array or a subgrid size that requires the creation of garbage elements along an axis that will be used heavily for communication.

For other communication operations, sometimes the existence of *any* garbage elements in the array will add overhead.


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

## 12.5 Querying an Array's Layout

CM Fortran provides several utility library procedures that report how the run-time system actually laid out an array. You can use this information to determine whether you should use the layout tools to specify a different layout.

Besides **CMF_DESCRIBE_ARRAY** mentioned earlier, the library also provides the following subroutines. These are particularly useful for deriving layout information for use in calls to **CMF_ALLOCATE_DETAILED_ARRAY** and **CMF_ALIAS_DETAILED**.

**CMF_GET_AXIS_ORDERS** (incompatible with **-noaxis** switch)
**CMF_GET_LOCAL_EXTENTS**
**CMF_GET_LOCAL_STRIDES**
**CMF_GET_PROCESSOR_MASKS**

The last item, processor masks, refers to integer bitmasks that describe the layout of axes across particular processors. They are reported by **CMF_DESCRIBE_ARRAY**, and you can specify them as part of a detailed layout directive or procedure, as described in the following section.


Copyright © 1989–1994 Thinking Machines Corporation

## 12.6 Specifying Detailed Array Layouts

The **LAYOUT** directive accepts additional axis specifications that describe array layout in detail. These forms of the directive are not meaningful at the language level or to the CM Fortran compiler, but are passed on as instructions to the run-time system. These detailed forms of the **LAYOUT** directive can be used with any array, including dynamically allocated arrays (although variable values used in the directive are established upon entry).

A comparable feature for allocating an array dynamically and specifying its layout in detail is the utility procedure **CMF_ALLOCATE_DETAILED_ARRAY**, where variable values are those at the time the utility executes. This procedure is described in the *CM Fortran Libraries Reference Manual*.

### 12.6.1 Syntax

The **LAYOUT** directive accepts the following forms of axis descriptor that pertain to detailed layouts:

**CMF$ LAYOUT** *array-name* ( *axis-descriptor*(*s*) )

An *axis-descriptor* is one of:

- **:SERIAL**
- **:BLOCK** = *axis-spec* **:PROCS** = *axis-spec*
- **:BLOCK** = *axis-spec* **:PDESC** = *proc-spec*

Each *axis-spec* or *proc-spec* may be a decimal integer constant, an integer-valued dummy argument, or an integer variable in **COMMON**. Integer expressions are not supported. The value of **:PROCS=** must be a power of 2.

The resulting axis layout can be either:

- the subgrid length (**:BLOCK=**) followed by the number of processors desired for that axis (**:PROCS=**)
- the subgrid length (**:BLOCK=**) followed by a bit-mask indicating which processors are desired (**:PDESC=**)


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

Keep in mind these restrictions on axis specifications:

- If one axis of an array is specified with **:BLOCK :PROCS**, then all axes of that array should be either **:BLOCK :PROCS** or **:SERIAL**. If one axis of an array is specified with **:BLOCK :PDESC**, then all axes of that array should be either :**BLOCK :PDESC** or **:SERIAL**.
- The compiler does not accept these detailed forms in an interface block when they use a dummy argument. Use only parameters, literal constants, or variables in **COMMON** in this situation.

### 12.6.2 Processor Masks

**NOTE:** This discussion assumes that you are compiling for the VU model. If you are not compiling for the SPARC nodes model, the use of masks is the same but they apply to nodes instead of vector units.

To understand processor masks, it is useful to review the concept of a *physical grid*. The physical grid of an array is the arrangement of vector units; it has one position for each VU, arranged in a grid whose rank is the same as that of the array (excluding serial axes, if any). The dimensions of the physical grid are each powers of 2, since there must be a power-of-2 number of VUs in the partition.

If axis $i$ of the physical grid has length $d_i$, then we need $\log_2(d_i)$ bits to represent a position in the physical grid along this axis. The processor mask for the axis is a mask with $\log_2(d_i)$ bits set. When we number all of the VUs linearly, these bits are the ones that determine each VU's position along axis $i$ in the physical grid. We call this linear numbering the *physical address* of the VU.

The vector units along the axis containing the least significant bit in the mask are contiguous. Also, in the current implementation, the bits for any one axis must be contiguous.

Let's assume we are going to run a program on 32 vector units, and we want a physical grid that is 4 x 8, as shown in Figure 27.


Copyright © 1989–1994 Thinking Machines Corporation

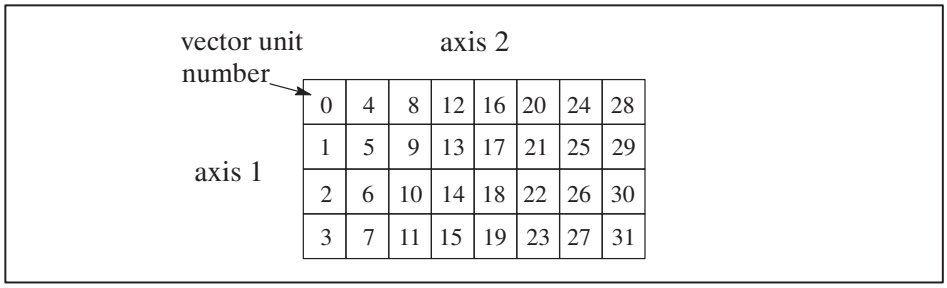


**Figure 27. A physical grid with processor masks of 3 and 28.**

We would represent the physical address of each of the 32 vector units with a number between 0 and 31; this requires 5 bits:

```
  bbbbb
MSB   LSB
```

To specify a 4 x 8 physical grid:

- Axis 1 (4 vector units) requires the lowest 2 bits of the mask because the vector units along this axis are contiguous; its processor mask is therefore 3:

```
  00011
MSB   LSB
```

- Axis 2 (8 vector units) requires bits 3, 4, and 5 of the mask; bits 0 and 1 are set to 0. Its processor mask is therefore 28:

```
  11100
MSB   LSB
```

If you want the vector units along axis 2 to be contiguous, the masks would be:

```
axis 1 = 11000     axis 2 = 00111
```

or 24 and 7.

Note that the least significant bits of the physical address denote the four vector units within a node. Since communication within a node is more efficient than communication between nodes, axes to which these two bits are assigned can be more efficient directions for communication.


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

What if we want to maximize the speed of communication along axis 1? To do that, we could allocate all the bits of the physical mask to axis 2. This would create these masks:

```
axis 1 = 00000     axis 2 = 11111
```

or 0 and 31. Elements along axis 1 would be on the same vector unit, and performance would be best along that axis.

If the program is running on 64 vector units and you want a physical grid of 8 x 8, both axes require three bits. But note that the bit mask lets us specify the axis sequence:

- To make vector units along axis 1 contiguous, specify axis 1 in the low-order bits:

  ```
  000111
  ```

  and axis 2 in the high-order bits:

  ```
  111000
  ```

  or 7 and 56.

- To make vector units along axis 2 contiguous, reverse the masks: 56 and 7.

Note these constraints in the current implementation for specifying the processor masks:

- Each mask must represent a contiguous set of bits. For example, a mask of 5 (101 in binary) is illegal.

- The mask for one axis must not use any bits used by another axis. For example, masks of 7 (binary 0111) and 12 (binary 1100) are illegal in combination, because both use bit 3.

- The sum of the masks must use all bits 0 through *n*, where *n* is less than or equal to the total number of bits that represent the vector units on which the program will run. For example, if you are going to run on 32 vector units, you can use all five bits, or the lowest four bits, or the lowest three bits, and so on. You can't use only the highest four bits. Typically the sum should be equal to the total number of bits, except for very small arrays, which don't use all the vector units.


Copyright © 1989–1994 Thinking Machines Corporation

If you use less than the total number of bits, the array will use less than the total number of vector units; this is usually inefficient.

- A serial axis has a processor mask of 0.

### 12.6.3 Examples of Detailed Layout Directives

- The array **A(16,16,4)** could be laid out on 8 processors as follows:

```
CMF$LAYOUT A(:SERIAL,:BLOCK=8:PROCS=2,:BLOCK=1:PROCS=4)
```

  In this mapping, the first axis is local (it could also have been specified as **:BLOCK=16:PROCS=1**); the 16 elements of the second axis are divided equally between 2 processors; and the third axis is spread across 4 processors, one element in each.

- The array **B(64,16)** could be laid out on 16 processors, using **:BLOCK :PDESC**, as follows:

```
CMF$  LAYOUT B(:BLOCK=16:PDESC=12,:BLOCK=4:PDESC=3)
```

  Since the mask **:PDESC=12** has 2 nonzero bits, it represents a physical axis of length 4. The axis extent (64) equals subgrid length (16) times physical axis length (4). Similarly, the mask of 3 for the second axis has 2 nonzero bits, and 4*4 = 16.

### 12.6.4 Restrictions on Array Size with Padding Enabled

Certain restrictions apply to detailed layouts when the program's execution environment pads user arrays such that the number of elements allocated per processor (for non-serial axes) is evenly divisible by the length of the vector registers in the CM-5 vector units.

These requirements vary by execution model; they apply only if the **cmf** switch **-padding** is enabled.

- For the vector units (**-vu**) model with **-nopadding**, there are no restrictions on the product of the **:BLOCK** values.


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

- For the vector units (**-vu**) model with **-padding**, the product of the **:BLOCK** values must be an integer multiple of 8. Axes labeled **:SERIAL** are excluded from the calculation.
- For the nodes (**-sparc**) model, there are no restrictions on the product of the **:BLOCK** values.

For example, assume a 32-vector-unit partition and this detailed layout:

```
      REAL B(100,32)
CMF$  LAYOUT B(:SERIAL, :BLOCK=1:PROCS=32)
```

With **-padding** enabled, this layout causes a run-time error:

```
*** RTS-FATAL-CMRTS:
*** Fatal RTS Error ***:
CMRT_intern_detailed_array_geometry: the product of the
specified subgrid lengths, 100, must be a multiple of 8
```

With **-nopadding**, this directive gives the intended layout of one 100-element serial axis per processor.


Copyright © 1989–1994 Thinking Machines Corporation

## 12.7 A Note on ALIGN

The **ALIGN** directive describes the layout of an array in terms of 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 elements of the target array.

### 12.7.1 Syntax

Like the **LAYOUT** directive, the **ALIGN** directive can be applied only once to an array in a program, and it must be repeated in every program unit where that array is used. Its format is:

**CMF$ALIGN** *source-array* **(** *axis-specs* **) WITH** *target-array* **(** *axis-specs* **)**

The *axis-specs* of the source array assign a symbolic name to each of its dimensions. The same symbolic names are then used in the *axis-specs* of the target array to indicate the pairs of source-and-target dimensions that are to be aligned in the same processors. Any target dimensions that are not related to the source array are identified with a scalar index value (or offset).

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)
```

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)
```


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

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
```

Other alignments, including offset alignments, are illustrated in the *CM Fortran Language Reference Manual.* Permuted alignments are not supported.

### 12.7.2 Caveats on ALIGN

The **ALIGN** directive can greatly enhance program performance when two arrays that are not conformable are often used together. By giving the arrays 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 is, however, a major cost in memory use when you align an array with another array of larger size or higher rank (which is normally the intention). The aligned array is given the same geometry as the target and is therefore allocated the same amount of memory, even though much of this memory may never be used. See the *CM-5 CM Fortran Performance Guide* for more information on the performance impact of **ALIGN**.


Copyright © 1989–1994 Thinking Machines Corporation