Chapter 5

Parallel Array Layout

The arrangement in CM memory of the elements of your program's parallel arrays is critical for high program performance. The goal you want to achieve is maximal locality of operations. Array operations that take place within the memory space of a single processor (either PE code blocks or CM communication operations) execute at the highest rate possible. Thus, you want to arrange your array data in CM memory so that most or all of the array elements to be combined in an array operation are located together in the memory of the same PE.

5.1 Array Dimensions and Array Layouts

The CM Fortran compiler accepts array expressions such as **A+B** or **A=B** when the arrays **A** and **B** have the same dimensions — that is, the same rank and axis extents. For example, these two arrays have the same dimensions:

```
REAL A(100,200,300), B(100,200,300)
```

However, two arrays may have the same dimensions without having the same *layout* — that is, the same arrangement of array elements in PE memory. For example, the following use of the CM Fortran **LAYOUT** compiler directive (described later in this chapter) gives these two arrays entirely different CM layouts:

```
      REAL A(100,200,300), B(100,200,300)
CMF$ LAYOUT A(:NEWS,   :NEWS, :SERIAL)
CMF$ LAYOUT B(:SERIAL, :NEWS, :NEWS)
```

The CM Fortran compiler uses the available information to determine whether two arrays in an array expression (such as **A+B**) have the same lay-

Copyright © 1994 Thinking Machines Corporation

out. If it can tell that two arrays have the same layout (either through a **LAYOUT** directive, or by other methods), it will compile the expression as simple parallel computation. If the compiler can tell that the two arrays do *not* have the same layout (or if it can't decide), the compiler will insert parallel communication calls that force one or both of the arrays to be copied into an internal temporary array for parallel computation. This implicit array copy takes extra memory space and execution time, so it's best to avoid it by using identical array layouts as much as possible, and by making sure the compiler has as much information about the layout of the arrays as possible.

5.2 Controlling Array Layouts

CM Fortran usually spares the programmer the details of how arrays are laid out. The layout of a given array on the CM is determined at run time by the array allocation routines of the CM Run-Time System. (Only at run time can the CMRTS determine the actual number of available PEs.) By default, the CMRTS uses a *canonical layout* algorithm that provides reasonable performance for most parallel programs. Thus, if you do nothing, your parallel arrays are automatically defined by the canonical layout algorithm. There are, however, a number of ways you can influence or control the layout of parallel arrays in your program:

- CM Fortran compiler switches **-nopadding** and **-noaxisreorder** cause the compiler to make general, global changes in the way the canonical algorithm lays out arrays.
- Compiler directives, such as **LAYOUT**, provide specific instructions to the compiler on how an array is to be laid out on the CM.
- Dynamic array allocation routines, such as the utility library procedures **CMF_ALLOCATE_LAYOUT_ARRAY** or **CMF_ALLOCATE_DETAILED_ARRAY**, let you specify at run time the layout of an array, much as you would at compile time with the **LAYOUT** directive.

To use these tools effectively, you need an understanding of how the CMRTS lays out parallel array elements in the memory of the CM — in particular, how the canonical array layout algorithm works and what aspects of it can be altered or changed by the use of the above-mentioned switches, directives, and library routines.

Copyright © 1994 Thinking Machines Corporation

5.3 The Canonical Array Layout

5.3.1 The Array Subgrid

When laying out a parallel array on the CM, the CMRTS divides the array into contiguous blocks of array elements called *subgrids*. Each subgrid has the same rank as the original array, and all subgrids have the same shape and number of elements. One such subgrid is stored in the memory of each PE.

Technical Note: The CMRTS uses a *block layout* of elements (contiguous blocks of elements distributed one block per PE), as opposed to a *cyclic layout* (individual elements distributed in sequence across all PEs). The block layout affords better performance for regular communication, which is often used by CM programmers for efficiency.

5.3.2 The Physical Grid

The division of an array into subgrids implicitly defines an arrangement of PEs, known as a *physical grid*, based on which subgrids contain neighboring array elements. This physical grid also has the same rank as the original array. Since the total number of available PEs is always a power of 2, each axis of the physical grid is also a power of two.

As an example, given an 8 x 16 parallel array and a 4-node (16-VU) CM, the RTS can use any one of five different physical grids to lay out the array: 4 x 4, 8 x 2, 2 x 8, 16 x 1, and 1 x 16. Which one does it actually use? Here's where the CMRTS layout algorithm begins to make some choices, and where you can begin using methods like the **`LAYOUT`** directive to control the choices it makes.

By default, the RTS tries to choose a subgrid size that is as small as possible. This provides a good distribution of array elements across processors, and tends to select subgrids with axis lengths that are in the same proportion as the axis lengths of the entire array. (There are other factors that can affect this, however, so the subgrid may have different proportions from the main array.)

For an 8 x 16 array, the RTS would by default choose an array layout as shown in Figure 19.

Copyright © 1994 Thinking Machines Corporation

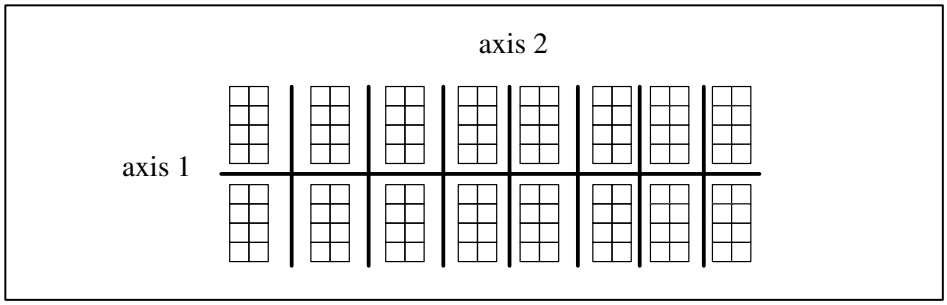

Figure 19. An 8 x 16 array laid out as a 2 x 8 physical grid.

5.3.3 Viewing Array Geometries

When the CMRTS defines an array, it internally creates two data structures:

- an *array descriptor* that defines the rank, extents, and data type of the array, as well as its starting location in CM memory (that is, the address in PE memory of the start of each PE's subgrid)
- a *geometry object* that defines the layout of the array (size and shape of the physical grid, and subgrids, etc.)

You can use the utility library procedure **`CMF_describe_array`** to look at the contents of the CMRTS array descriptor and geometry object:

```
PROGRAM TWODARR
INCLUDE '/USR/INCLUDE/CM/CMF_DEFS.H'
INTEGER NPES,X,Y
NPES=CMF_NUMBER_OF_PROCESSORS()
PRINT *,'NUMBER OF PES = ',NPES
PRINT *,'AXIS SIZES FOR 2D ARRAY?'
READ *,X,Y
CALL DOIT(NPES,X,Y)
END

SUBROUTINE DOIT(NP,X,Y)
INTEGER NP, X,Y, A(X,Y)
CALL CMF_DESCRIBE_ARRAY(A)
END
```


Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

Below is a sample run of this program for the 8 x 16 array size we've been using, on a CM with 4 nodes (16 VUs).

Note: The output has been edited to remove some array descriptor fields that are of no importance to this discussion, and to boldface key items.

```
% 2darr
 Number of PEs =           16
 Axis sizes for 2D array?
8,16

Descriptor address    : d8784
  desc_or_obj_kind     :  array argument
  element_type         :  signed integer
  home                 :  cm
  cm_location          :  1342178408
  user_rank            :  2
  axes_extents         :  8 16
  array_geometry       :  159d80
  geometry_rank        :  2
  element_size         :  4

Array geometry id: 0x159d80
  Rank: 2
  Number of elements: 128
  Extents: [8 16]
  Machine geometry id: 0x159cf0, rank: 2, column major
   Machine geometry elements:  128
   Overall subgrid size:       8
  Axis 0:
   Extent:    8 (2 physical x 4 subgrid)
   Off-chip:  1 bits, mask = 0x8
   Subgrid:   length = 4, axis-increment = 2
  Axis 1:
   Extent:    16 (8 physical x 2 subgrid)
   Off-chip:  3 bits, mask = 0x7
   Subgrid:   length = 2, axis-increment = 1
```

As you can see, the array descriptor specifies the **`element_type`**, **`user_rank`**, and **`axes_extents`**, while the geometry object gets into details of the **`physical`** and **`subgrid`** sizes of each axis.

For the Puzzled: CMRTS data structures use a zero-based axis numbering instead of the one-based axis numbering of CM Fortran. Thus, "Axis 0" in the output above actually refers to Axis 1 of the CM Fortran array.

Copyright © 1994 Thinking Machines Corporation

5.3.4 Garbage Element Padding

It is not always possible to divide a given parallel array into equal-sized subgrids, one subgrid for each PE in the CM. For example, while an 8 x 16 array can be neatly distributed across the PEs of a 16-VU machine, a 7 x 15 array cannot be so easily divided. If the CMRTS cannot equally divide the elements of an array among the PEs, it extends the length of one or more of the array axes until it has a number of array elements that *will* divide equally. It then defines a geometry object for this expanded array size, and adds to it an internal *garbage mask* that defines which elements of the array actually contain real data.

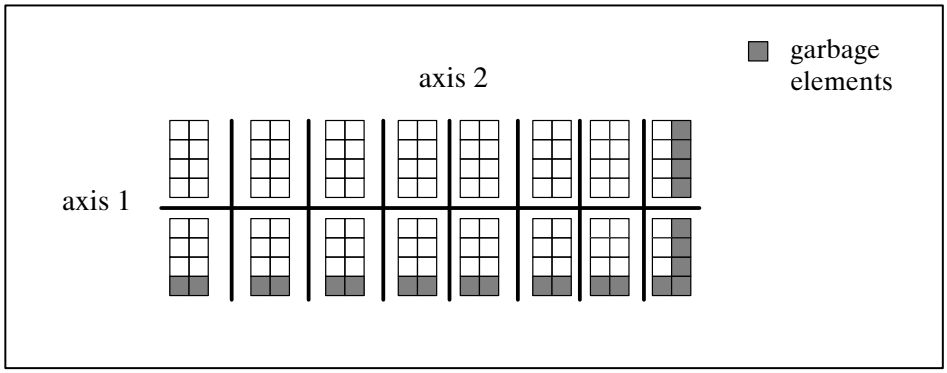

Figure 20. A 7 x 15 array laid out as an 8 x 16 array with garbage elements.

The masked-out elements are referred to as *garbage elements*, or more simply as *padding*. These garbage elements are always added at the high end of array axes, and the number of garbage elements added is always as small as possible. Garbage elements play no part in array operations. The CMRTS is designed to ignore them transparently; from the point of view of a CM Fortran programmer, they are not even visible.

However, they do have a significant effect on communications performance, particularly on regular shifts of array data, because the CMRTS routines must do extra work to "skip over" the hidden garbage elements. Thus, from a performance point of view it is better to choose an array shape that does not generate garbage elements, or at least puts the padding along an array axis where it will have the least impact on communications performance. In doing so, you'll probably be better off choosing the array layout and padding yourself rather than relying on the canonical layout algorithm to do it for you.

Copyright © 1994 Thinking Machines Corporation

When you use the **LAYOUT** directive to choose your own array layouts, you must consider garbage padding, and decide which axes will be padded.

Technical Note: Internally, the CMRTS uses two kinds of geometry objects to represent the layout of an array:

- a *machine geometry*, which describes the layout of an array that divides neatly across the available PEs of the CM
- an *array geometry*, which combines a machine geometry and a garbage mask to define the layout of an actual Fortran array

You can see the distinction between the two kinds of geometry objects in the output of **CMF_describe_array**, which includes a description of the machine geometry under the array geometry heading.

5.3.5 VU Vector Length Padding

Note: This section applies only to CM-5 systems with vector units.

There is one other kind of padding that may be applied to an array, in particular an array with a small number of elements relative to the number of PEs in the CM. This is referred to as *vector length padding*. CM Fortran defines the *vector length* of the VUs (that is, the number of array elements that can be handled as a single vector operation) to be 8 elements. This means that the subgrid loop of a PE code block operates on eight array elements at a time. It also means that the "natural" subgrid size for CM arrays is a multiple of 8.

By default, the CM Fortran compiler directs the CMRTS to lay out arrays so that each PE's subgrid contains an exact multiple of 8 elements. For small arrays, or arrays with small subgrid sizes, this may cause additional garbage padding space to be allocated. (Note that a very small array — with fewer elements than the number of PEs will be subject to garbage padding to even out the number of elements in each subgrid. Vector length padding generally affects arrays that are larger than the number of PEs.)

For example, an 8 x 12 array will be laid out with an 8 x 16 array geometry having 32 garbage elements, as shown in Figure 21.

The subgrid size does not have to be a multiple of 8, however. The CM Fortran compiler can include additional DPEAC code to handle any "tail" of

Copyright © 1994 Thinking Machines Corporation

remaining elements for subgrids that are not a multiple of 8, by temporarily changing the VU vector length to the length of the tail of elements and performing an extra vector operation.

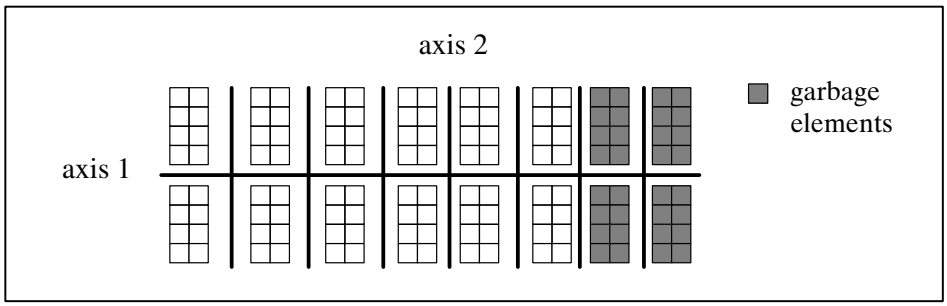

Figure 21. A 8 x 12 array laid out as an 8 x 16 array with garbage elements.

You can instruct the compiler to ignore the multiple-of-eight rule and include this extra tail-handling code by supplying the compiler switch **`-nopadding`**. With this switch, the subgrid can be any arbitrary length. (Note, however, that garbage padding may still be added if the array does not divide evenly among the PEs.)

With **`-nopadding`**, an 8 x 12 array can be laid out on a 4 x 4 physical grid, with 2 x 3 subgrids and no garbage elements, as shown in Figure 22.

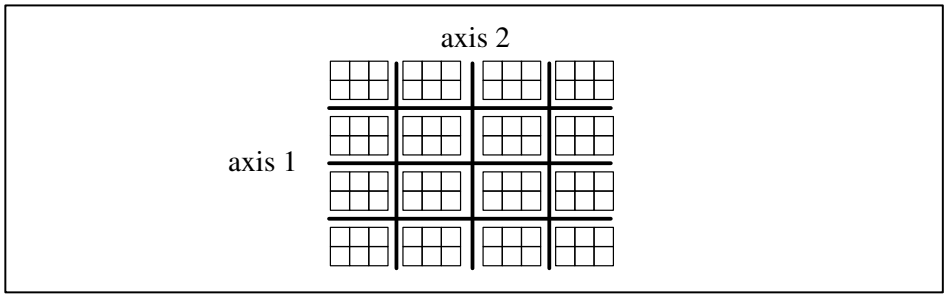

Figure 22. An 8 x 12 array on 16 VUs with `-nopadding`.

There is a minute amount of overhead associated with the extra “tail” code. In most cases, this overhead will be insignificant compared to the time taken to handle the rest of the array via the normal 8-element vector operations. In rare cases, however, it can become significant.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

For example:

```
      INTEGER A(100,8,16)
CMF$  LAYOUT A(:SERIAL,:SERIAL,:NEWS)
      A(:,2:4,:)=28
```

Here the axis of length 8 is the serial axis that varies most rapidly. If this is compiled with the **-nopadding** switch and executed on a 16-VU CM, then the array assignment will consist of 100 iterations of a three-element "tail" operation, which will have much less optimal performance than the similar example below in which the length-100 axis varies fastest:

```
      INTEGER A(8,100,16)
CMF$  LAYOUT A(:SERIAL,:SERIAL,:NEWS)
      A(2:4,:,:)=28
```

For the Curious: Vector length padding is an historical artifact; an earlier version of the CM Fortran compiler required subgrid sizes to be an exact multiple of eight. Since then, the compiler has been extended to handle non-multiple-of-eight subgrid sizes. The **-nopadding** switch is intended to allow programmers access to arbitrary subgrid sizes, while providing compatibility for programmers who depend on the older, restricted subgrid size layouts.

5.3.6 Seeing Array Padding

You can determine whether any given array has garbage padding by looking at the output of **CMF_describe_array**. If the output shows a machine array with a different number of elements than the user array size, the difference is padding. For example, here's the relevant portion of the **CMF_describe_array** output for the 8 x 12 array from Figure 21:

```
 Descriptor address    : d8784
  desc_or_obj_kind     :  array argument
  debug info: <nil>
...
Array geometry id: 0x159d80
  Rank: 2
  Number of elements: 96
  Extents: [8 12]
  Machine geometry id: 0x159cf0, rank: 2, column major
   Machine geometry elements:  128
   Overall subgrid size:       8
...
```


Copyright © 1994 Thinking Machines Corporation

The boldfaced lines show the difference between machine and array geometry sizes: 32 elements, which are garbage padding.

5.3.7 Physical and Subgrid Axis Ordering

Once an array has been divided into subgrids, the CMRTS must next determine two orderings:

- The order in which the subgrids are distributed across the available PEs — that is, how elements of the physical grid are assigned to PEs. This is the *physical axis ordering*.
- The order in which individual subgrid elements are stored in the memory of the PEs. This is the *subgrid axis ordering*.

Physical Axis Ordering

The natural ordering of the PEs in the CM is linear, across the entire machine (Figure 23 shows this ordering for a CM-5 with VUs). This means that the *n*-dimensional arrangement of subgrids must be translated into a one-dimensional sequence of PEs.

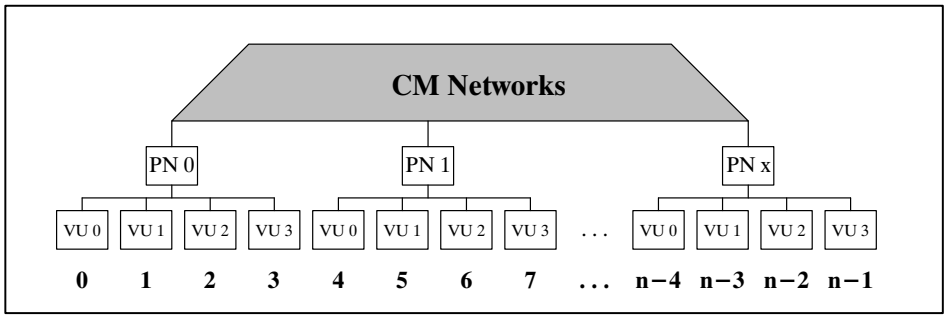

Figure 23. Linear ordering of CM processing elements.

This is done by choosing a *physical axis sequence*, that is, an ordering of physical grid axes such that the distance between PEs along each axis steadily increases (or decreases) from the first to last axis in the sequence.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

For example, Figure 24 shows the two possible axis sequences for the 2 x 8 physical grid of the 8 x 12 array example used in the preceding sections. The numbers indicate the assignment of PEs to array subgrids. The first axis sequence (1,2) has the subgrids along axis 1 closest together in the CM hardware (this is the lowest-axis-varies-fastest ordering). The second sequence (2,1) has subgrids along axis 2 closest together (this is the highest-axis-varies-fastest ordering).

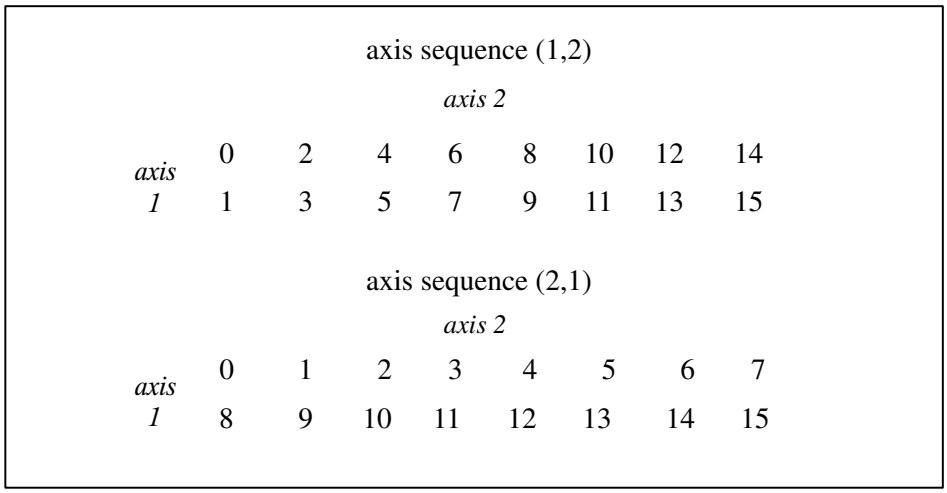

Figure 24. Two possible axis orderings for a 2 x 8 physical grid.

By default, CM Fortran arrays are laid out using the (2,1) physical axis ordering (highest axis varies fastest). This is closely related to the way the subgrid axis ordering is chosen, as described below.

Subgrid Axis Ordering

Array subgrids may be of rank 2, 3, or more. Nevertheless, the elements of a subgrid must be stored in a linear order, since PE memory addressing is linear. A *subgrid axis sequence* is chosen: an ordering of subgrid axes such that the distance in PE memory between elements along each axis steadily increases (or decreases) from the first to last axis in the sequence.

For example, Figure 25 shows the two possible subgrid axis sequences for the 4 x 2 subgrid from the array examples in the preceding sections.

Copyright © 1994 Thinking Machines Corporation

Here the numbers indicate the ordering of array elements in PE memory. The first axis sequence (1,2) has the elements along axis 1 closest together in PE memory. The second sequence (2,1) has elements along axis 2 closest together.

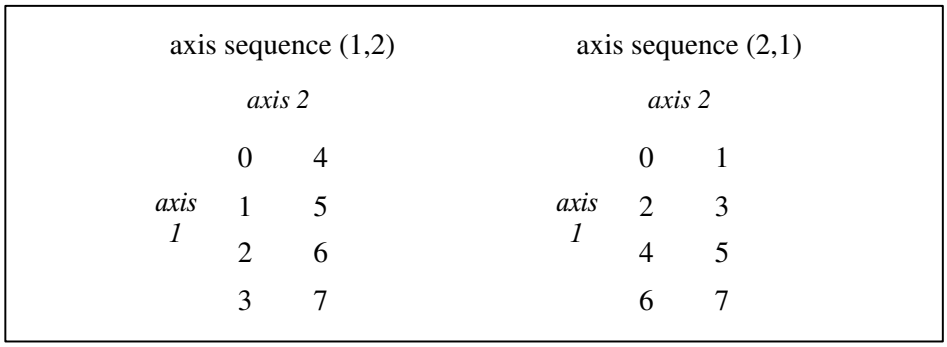

Figure 25. Two possible axis orderings for an 8 x 2 subgrid

Another way of looking at this is to consider how the indices of the subgrid elements change as you move from element to element in PE memory:

- In the (1,2) axis sequence, the index of the lowest-numbered axis varies the fastest. This is known as *lowest-axis-varies-fastest* ordering. For a 5-dimensional array, this sequence would be (1,2,3,4,5).
- In the (2,1) axis sequence, the index of the highest-numbered axis varies the fastest. This is known as *highest-axis-varies-fastest* ordering. For a 5-dimensional array, this sequence would be (5,4,3,2,1).

By default, the CMRTS currently lays out multidimensional arrays using *highest-axis-varies-fastest* ordering for both the physical grid and the subgrids. That is, PE numbers and array indexes vary fastest along the highest-numbered axis, and slowest along the lowest-numbered axis. (This is contrary to the memory ordering used in standard Fortran 77; there, array elements are stored in memory in lowest-axis-varies-fastest ordering.)

You can instruct the compiler to use the lowest-axis-varies-fastest ordering for all arrays. The compiler switch **`-noaxisreorder`** causes the compiler to always order the axes by increasing order of axis number: (1,2,3,4,...,*n*).

Note: The subgrid axis sequence is made more complicated when you use the **`LAYOUT`** compiler directive to declare one or more array axes as being *serial axes*. For more information, see Section 5.4.2.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

5.3.8 The Complete Canonical Array Layout

When the physical grid ordering and subgrid ordering of an array are put together, the result is the canonical layout of array elements across PEs and within PE memory. Figure 26 shows the canonical layout and indexes of the elements in an 8 x 12 array on a 4-node, 16-VU CM:

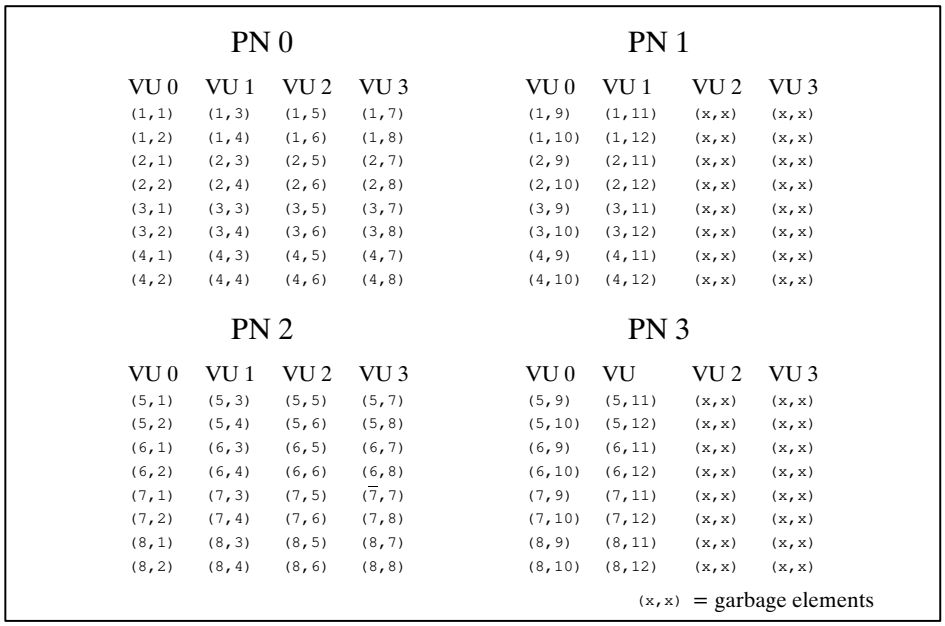

Figure 26. Canonical layout of 8 x 12 Fortran array on 16-VU CM

5.3.9 Send Addresses

Once the array's layout is defined, each element of the array is assigned a *send address* based on its location in the entire machine. Figure 27 shows the send addresses of an 8 x 12 array on a 4-node, 16-VU CM. Note that garbage elements have send addresses, even though these elements are "invisible."

Technical Note: The data structures for send addresses are different at the CM Fortran and CMRTS level. CM Fortran encodes send addresses as 32-bit integers. CMRTS send addresses are 64-bit quantities. CM Fortran utility functions transparently convert from the one to the other, but if you do any low-level CMRTS coding, be aware that the data structures change.

Copyright © 1994 Thinking Machines Corporation

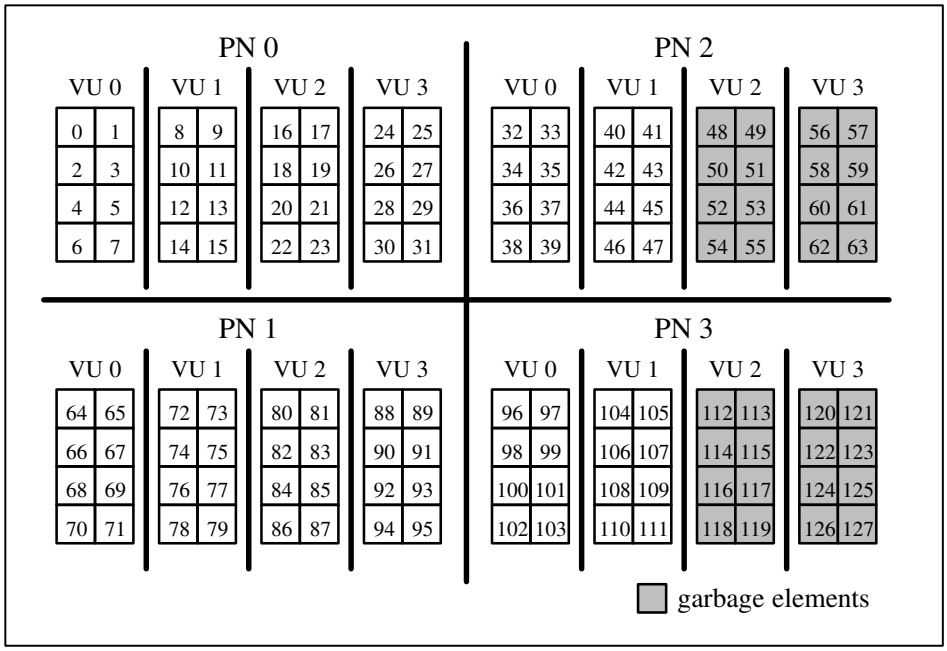

Figure 27. Assignment of send addresses to elements of 8 x 12 parallel array.

You can use the utility library routine **`CMF_my_send_address`** to load an array with the send addresses of its elements. For example, here's the output of a short CMF program that displays the send addresses of a 2D array:

```
% selfaddr
 Axis sizes for 2D array?
8,12
Send Addresses:
  0   1   8   9  16  17  24  25  32  33  40  41
  2   3  10  11  18  19  26  27  34  35  42  43
  4   5  12  13  20  21  28  29  36  37  44  45
  6   7  14  15  22  23  30  31  38  39  46  47
 64  65  72  73  80  81  88  89  96  97 104 105
 66  67  74  75  82  83  90  91  98  99 106 107
 68  69  76  77  84  85  92  93 100 101 108 109
 70  71  78  79  86  87  94  95 102 103 110 111
```

Compare this with the send addresses shown in Figure 27. The program that generates this output is shown below.

Copyright © 1994 Thinking Machines Corporation

```
      PROGRAM SELFADDRESS
      INCLUDE '/usr/include/cm/CMF_defs.h'
      INTEGER X,Y
      PRINT *,'Axis sizes for 2D array?'
      READ *,X,Y
      CALL DOIT(X,Y)
      END

      SUBROUTINE DOIT(X,Y)
      INTEGER X,Y, A(X,Y)
      PRINT *,'Send Addresses:'
      CALL CMF_MY_SEND_ADDRESS(A)
      DO I=1,X
          PRINT 10,(A(I,J),J=1,Y)
10       FORMAT (1000(I3,' '))
      END DO
      END
```

5.3.10 Canonical Layout on CMs without Vector Units

If your program is compiled for a CM without vector units, the PEs are the processing nodes themselves. In this case, the layout rules are basically the same as above, except that vector unit length padding does not apply.

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 28.

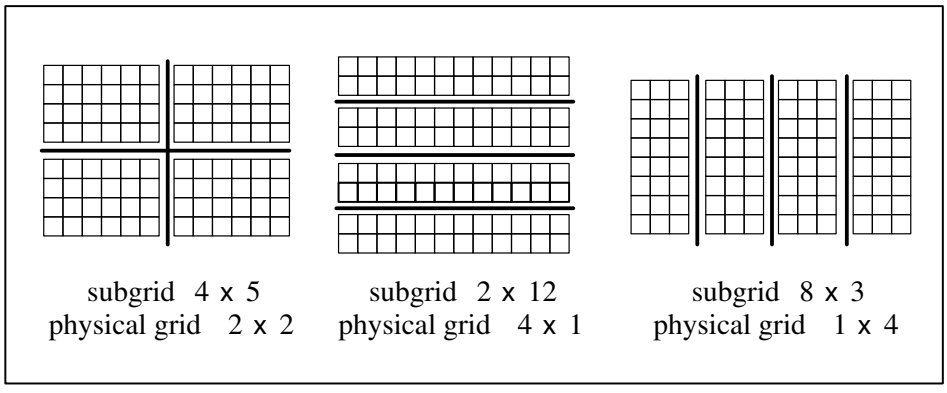

Figure 28. Subgrids and physical grids for an 8 x 12 array on four nodes.

Copyright © 1994 Thinking Machines Corporation

No global padding is required in this example, because the 96 elements of the array divide up 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 29.

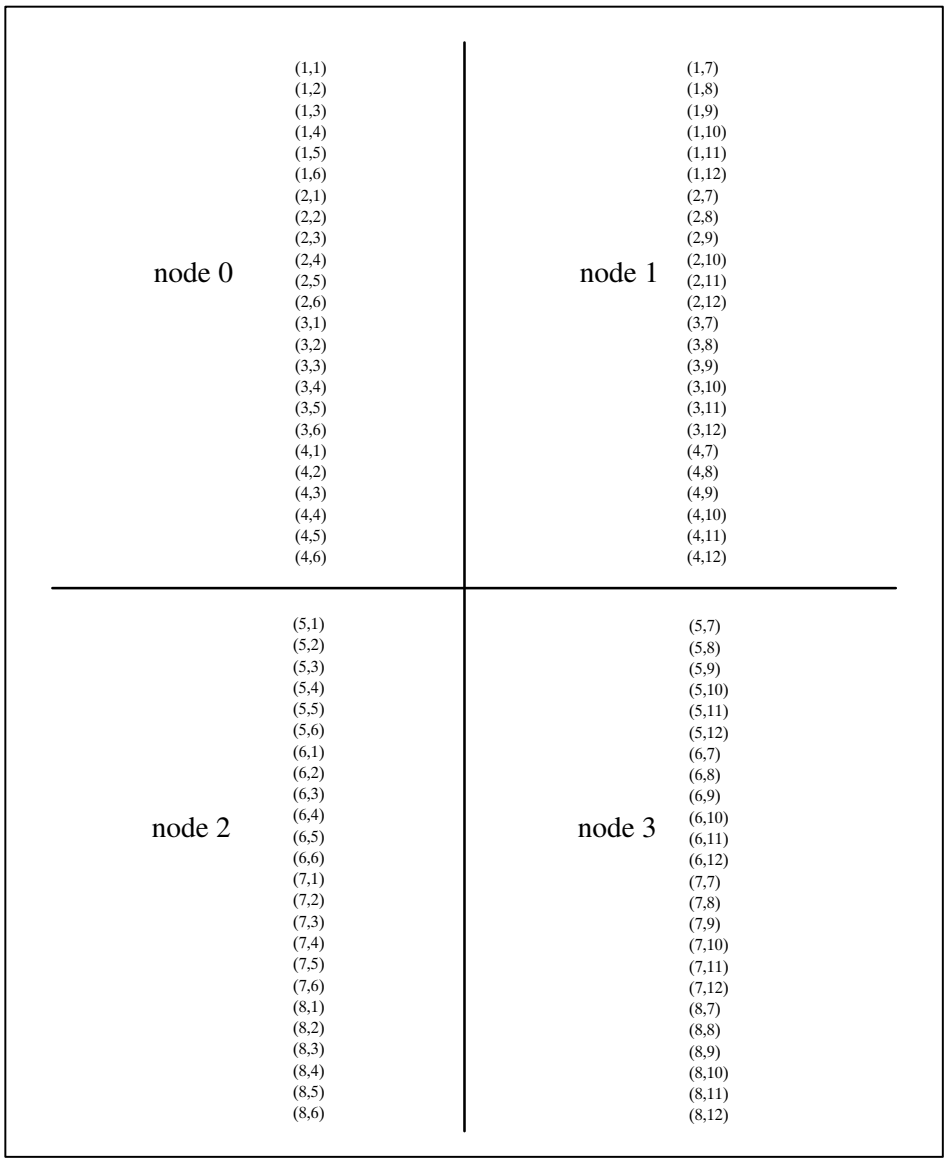

Figure 29. Default layout of an 8 x 12 array on four nodes.

Copyright © 1994 Thinking Machines Corporation

5.4 Controlling Array Layout: Specifying Axis Ordering and Weights

The compiler directive **LAYOUT** and the dynamic array allocation procedure **CMF_ALLOCATE_LAYOUT_ARRAY** enable you to guide the RTS in laying out an array.

The **LAYOUT** syntax is discussed in the *CM Fortran Language Reference Manual,* Version 2.1, and the **CMF_ALLOCATE_LAYOUT_ARRAY** procedure is described in the *CM Fortran Libraries Reference Manual,* Version 2.1.

Two fairly straightforward layout aspects you can specify are:

- serial axes
- weighting of axes

We discuss serial axes below, and weighting of axes in Section 5.4.3.

Note: The use of axis weights is deprecated in favor of the detailed layout tools described later in this chapter. See the note in Section 5.4.3.

5.4.1 Serial Axes

You can specify an axis as **:serial** in a **LAYOUT** directive, or as **CMF_SERIAL** in an allocation procedure. Specifying an axis as **:serial** has these effects:

- All elements along that axis are stored in the memory of the same PE.

  This layout is useful if communication tends to occur mostly along 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. 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.

- The compiler can apply serial axis optimizations, which may greatly increase the performance of your code. See Section 7.3.

Copyright © 1994 Thinking Machines Corporation

- 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

```
(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 axis 2 as serial, the sequence would change to

```
(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.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

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:

```
(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.
```

5.4.2 Serial Axes and Array Layouts

Axes declared **:SERIAL** are not taken into account when the CMRTS layout algorithm is dividing up an array among the available PEs and determining whether garbage elements or vector length padding are needed. In effect, the **:SERIAL** axes are ignored and not added in until the after the assignment of subgrids to PEs. This can cause unexpected layout results from apparently simple layout requests.

For example, the following layout directive (used without the **-nopadding** compiler option), contrary to what you might expect, will not put 8 elements on each PE of a 16-VU system:

```
      integer A(16,8)
cmf$  LAYOUT A(:news, :serial)
```

Instead, the array is laid out as if it were (16,1); the **:NEWS** axis is divided between two PEs to put at least 8 elements on each PE. Then the 8 elements along the **:SERIAL** axis are added in, making a subgrid of 8 x 8 on each of two VUs. The result is that 14 VUs contain no actual data, only padding.

The simplest way to force a layout that puts an 8 x 1 subgrid on all 16 VUs is to use the **-nopadding** compiler switch to avoid the use of vector length

Copyright © 1994 Thinking Machines Corporation

padding. Alternatively, you can declare the second axis to be a **:NEWS** axis but give it a weight value sufficiently great to ensure that it will be laid out as a serial axis. With this method, the sample **LAYOUT** directive above becomes:

```
      INTEGER A(16,8)
CMF$  LAYOUT A(:NEWS, 100:NEWS)
```

With this directive, the pseudo-"serial" axis (the heavily-weighted **:NEWS** axis) is taken into account for the purpose of determining array division and padding, and therefore the array is laid out as an 8 x 1 subgrid on each of 16 VUs. Note, however, that this second method prevents any potential serial-axis compiler optimizations from having the 8-element axis declared **:SERIAL**. In some cases, you may find it more advantageous to use the **:SERIAL** declaration and apply the **-nopadding** compiler switch to prevent padding from altering your intended array layout.

5.4.3 Weighting Axes

As shown above, you can 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 one 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.

NOTE

The use of axis weights is deprecated in favor of the detailed layout tools described in Section 5.5, below. The CMRTS algorithm may in some cases determine that an array's layout is totally determined by other parameters before weights are considered, so it often cannot respond usefully to axis weights in a **LAYOUT** directive.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

In the example of laying out an 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.

5.5 Specifying Detailed Array Layouts

The **LAYOUT** directive and the utility library procedure **CMF_ALLOCATE_DETAILED_ARRAY** allow you to define more precisely the array layout you want. Although you must work within the same constraints as the CMRTS software itself in choosing an array layout, you have a large latitude in the choices you can make to shape an array for best performance.

There are three main ways to specify detailed array layouts:

- using **:block :procs** axis descriptors in a **LAYOUT** directive
- using **:block :pdesc** axis descriptors in a **LAYOUT** directive
- using the **CMF_ALLOCATE_DETAILED_ARRAY** procedure

5.5.1 IMPORTANT: Detailed Layouts Can Restrict Layout Flexibility

Note that when you define the layout of an array using detailed layout directives, you are to some extent making the layout directly dependent on the number of PEs available in your CM, losing some of the machine-independent flexibility of the canonical CMRTS layout algorithm. Although you can write your code to be usable on machines of differing sizes (for example, by using **CMF_number_of_processors** to determine the machine size), you must assume more of the responsibility for assuring that the array layout you select is compatible with the CM on which you are running your program.

Copyright © 1994 Thinking Machines Corporation

5.5.2 Using Detailed Layout Axis Descriptors

The **LAYOUT** directive lets you describe an array's layout precisely in terms of its physical grid and subgrid size. Recall that the CMRTS layout algorithm divides an array into equal sized subgrids of elements. This division of the array into subgrids also implies an arrangement of PEs; this is the physical grid of the array.

The detailed layout descriptors allow you to specify the subgrid length (**:block**) and the physical grid length (**:procs** or **:pdesc**) for each axis of the array. Remember that these axis lengths must allow for garbage elements, if they are required to meet the standard layout restrictions (same subgrid size per PE, multiple-of-eight subgrid size on VU systems). If you specify an array layout that cannot legally be used by the CMRTS, an error message is signaled when the program is executed.

Each of the following **LAYOUT** directives describes the canonical layout of an 8 x 12 array for a 16-VU system:

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

CMF$ LAYOUT A( :BLOCK=4 :PDESC=8, :BLOCK=2 :PDESC=7 )
```

Notice how these **LAYOUT** directives correspond with the geometry object description in the **CMF_describe_array** output for the 8 x 12 array:

```
Array geometry id: 0x15c1b8
  ...
  Axis 0:
   Extent:    8 (2 physical x 4 subgrid)
   Off-chip:  1 bits, mask = 0x8
   Subgrid:   length = 4, axis-increment = 2
  Axis 1:
   Extent:    16 (8 physical x 2 subgrid)
   Off-chip:  3 bits, mask = 0x7
   Subgrid:   length = 2, axis-increment = 1
```

(Also compare these values with the picture of the array in Figure 30.)

The **:block** descriptor for each array axis corresponds to the **subgrid** part of the **Extent:** entry for that axis in the geometry object. The **:procs** descriptor corresponds to the physical part, and the alternate **:pdesc** descriptor corresponds to the **Off-chip: mask** entry. (Note that in the **LAYOUT** directive the **:pdesc** is expressed as a decimal value, while in the **CMF_describe_array** output it is represented as a hexadecimal bit-mask.)

Copyright © 1994 Thinking Machines Corporation

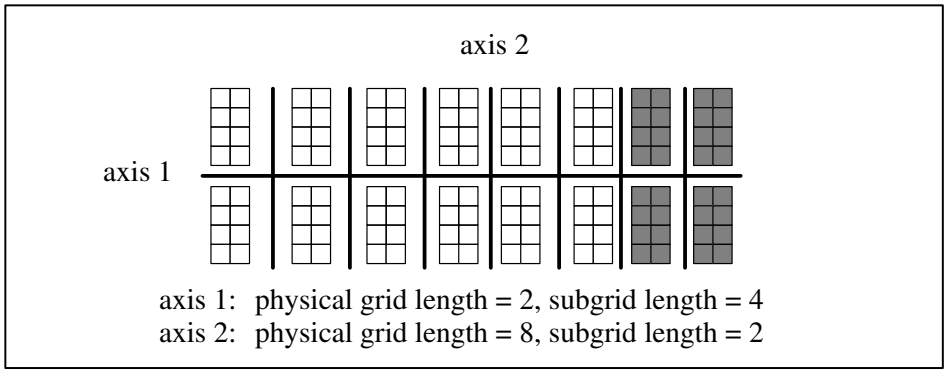

Figure 30. An 8 x 12 array laid out as an 8 x 16 array with garbage elements.

The **`:procs`** and **`:pdesc`** descriptors are equivalent ways of expressing the layout of the physical grid, with the **`:pdesc`** descriptor being slightly more precise. The **`:procs`** descriptor is pretty much self-explanatory, once you understand the concept of the physical grid of an array, but understanding the **`:pdesc`** descriptor requires a quick digression to look at *axis masks*.

5.5.3 Axis Masks (Processor Masks)

The physical grid of an array is the arrangement of PEs. When we number the PEs linearly (as in Figure 23), the number each PE receives is its *physical address*. The number of PEs is always a power of 2, so each physical grid axis must also be a power-of-2 PEs in length. When a node's physical address is represented as binary value, it can be broken down into binary values defining the node's position along each physical grid axis (see Figure 31).

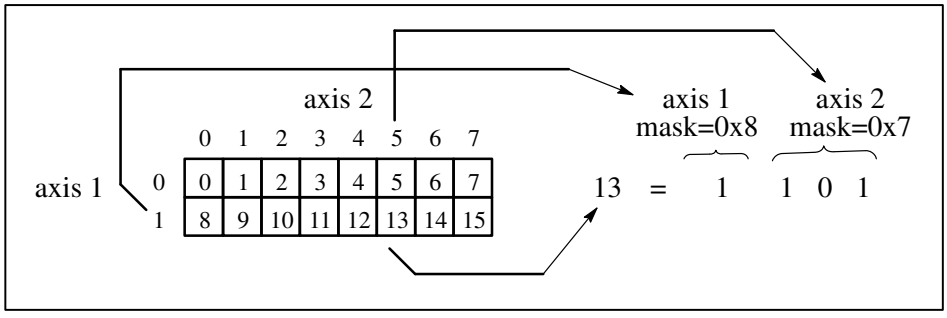

Figure 31. Correspondence between physical grid positions and axis masks.

Copyright © 1994 Thinking Machines Corporation

The bits assigned to each axis are determined by the *axis masks*: axis 1 has a mask of 0x8 hex = 1000 binary (so it is assigned the single high bit), while axis 2 has a mask of 0x7 hex = 0111 binary (so it is assigned the three low bits). (Note that the hexadecimal numbers above are represented in the C format used by CMRTS.) The axis mask is simply a hex value with 1 values for the $\log_2(d_i)$ bits corresponding to axis i. (CM Fortran programmers also refer to axis masks as *processor masks* — the two terms are synonymous.)

Note: In the current implementation, the bits assigned to any one axis must be contiguous. Thus, the axis mask value as a binary value must contain a sequence of consecutive 1's with no "gaps."

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. This means that, looking at the binary form of the physical address, there are $\log_2(d_i)$ bits representing an index along that axis. (Notice that the axis indexes we're using here are zero-based, not one-based as in Fortran, because once again we're dealing with internal CMRTS data structures.)

Thus, in the figure above, position 13 in the physical grid is located at (1,3) using zero-based axis indexing. The value 13 is equal to 1101 in binary. This contains a single bit for axis 1, having the value 1 (the index along axis 1) and three bits for axis 2 having the value 101 or 5 (the index along axis 2).

A **`:pdesc`** value in a **`LAYOUT`** directive is simply the decimal equivalent of the axis mask. For the physical grid in Figure 31, the **`LAYOUT`** directive would be

```
CMF$ LAYOUT A( :BLOCK=4 :PDESC=8, :BLOCK=2 :PDESC=7 )
```

Notice that in Figure 31 the PEs are numbered consecutively along axis 2. This is because the axis mask for axis 2 includes the low-order bits of the physical address. If the low-order bits were included in the axis mask for axis 1, the numbering would be consecutive along axis 1. For example,

```
CMF$ LAYOUT A( :BLOCK=2 :PDESC=7, :BLOCK=4 :PDESC=8 )
```

The **`:pdesc`** form of the **`LAYOUT`** directive gives you more control than the **`:procs`** form, because it lets you specify the axis masks explicitly. The **`:procs`** form of the **`LAYOUT`** directive lets you specify the number of processors along each axis (and thus implicitly the number of axis mask bits each axis gets), but it assigns the axis mask values according to the *highest-axis-varies-fastest* method: the highest-numbered axis gets the low-order bits, and the low-numbered axis gets the high-order bits. The **`:pdesc`** form of the **`LAYOUT`** directive allows you to specify a lowest-axis-varies-fastest ordering.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

5.5.4 Example

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 32. (Note that this assumes we are compiling with the **-nopadding** switch.)

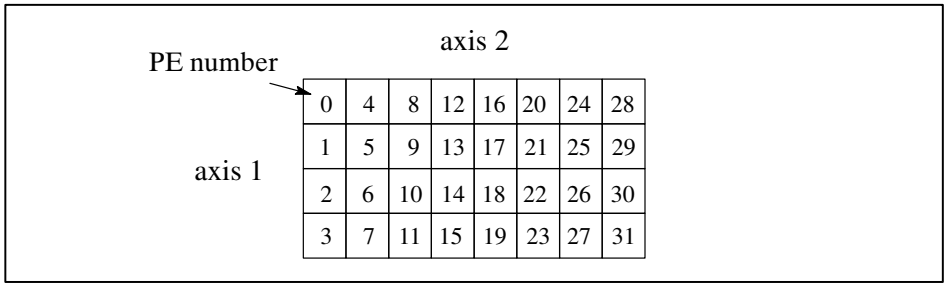

Figure 32. A physical grid with axis masks of 3 and 28.

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

```
  bbbbb
MSB     LSB
```

The physical addresses along axis 1 are consecutive, so axis 1 must be assigned the low-order bits of the physical address. Axis 1 has a length of 4, so it takes 2 bits to describe the zero-based index along this axis. Thus, the axis mask for axis 1 is 3, or, in binary,

```
  00011
MSB     LSB
```

Axis 2 gets the remaining (most-significant) bits, so its mask is 28 or, in binary,

```
  11100
MSB     LSB
```

The "subgrid" assigned to each VU is a single element, or a 1 x 1 array. So an appropriate **LAYOUT** directive would look like

```
CMF$  LAYOUT A(:BLOCK=1 :PDESC=3, :BLOCK=1 :PDESC=28)
```


Copyright © 1994 Thinking Machines Corporation

Note that the low-order bits assigned to axis 1 are 0's in the mask for axis 2. If we wanted the physical addressing to be consecutive along axis 2, the masks would need to be

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

or 24 and 7. The same number of bits are assigned to each axis; only the mask values change. The **LAYOUT** directive would then be

```
CMF$  LAYOUT A(:BLOCK=1 :PDESC=24, :BLOCK=1 :PDESC=7)
```

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
  ```

  This gives **:pdesc** numbers of 7 and 56, and a **LAYOUT** directive like:

  ```
  CMF$  LAYOUT A(:BLOCK=1 :PDESC=7, :BLOCK=1 :PDESC=56)
  ```

- To make vector units along axis 2 contiguous, reverse the masks: 56 for axis 1 and 7 for axis 2:

  ```
  CMF$  LAYOUT A(:BLOCK=1 :PDESC=56, :BLOCK=1 :PDESC=7)
  ```

5.5.5 Vector Unit Performance Note

On a CM-5 with vector units, the four VUs on each processing node always have sequential physical addresses. Thus, the two least significant bits of the physical address apply directly to the VUs.

Since exchange of data between PEs is fastest among the four VUs on a node, and especially between the two VUs on the same VU chip, the low-order "VU bits" should be assigned to axes of the array along which the most interprocessor communication is likely to take place.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

5.5.6 Serial Axes via Axis Masks

Notice that an axis of the physical grid may have *no* physical address bits assigned to it. In this case, it is a serial axis, as if you had declared it **`:serial`**. All elements along that axis are stored in PE memory (in other words, the length of the single PE subgrid along that axis is the total length of the array axis). For example, in the 32-PE case above, we could allocate all bits of the physical addressing to axis 2. This would create these masks:

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

or 0 and 31. Elements along axis 1 would always be stored in the memory of the same vector unit, and performance would be best along that axis.

5.5.7 Axis Mask Restrictions

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 have a 1 bit in the same position.
- The logical IOR of all the masks must use every bit from 0 through $n-1$, where n is an integer less than or equal to the total number of bits that represent the PEs on which the program will run.

  For example, if you are going to run on 32 PEs (on a 32-VU CM, say), 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 masks will collectively contain the total number of bits, except for very small arrays, which don't use all the PEs.

  If you use fewer than the total number of bits, the array will use fewer than the total number of PEs; this is usually efficient only for very small arrays that can't be evenly divided across all the PEs.
- A **`:serial`** axis has a processor mask of 0, by definition. (An axis with a zero processor mask is also treated as a **`:serial`** axis.)

Copyright © 1994 Thinking Machines Corporation

5.5.8 Using the CMF_ALLOCATE_DETAILED_ARRAY Procedure

The **CMF_allocate_detailed_array** procedure is similar in effect to the **:block :pdesc** version of the **LAYOUT** directive. The *SUBGRIDS* argument is an array of integers representing the subgrid lengths of the array to be defined (the same integers that would be the **:block** values in a **LAYOUT** directive). The *PMASKS* argument is an array of integers representing the axis masks for each axis (the same integers that would be the **:pdesc** values in a **LAYOUT** directive). For an example of the use of **CMF_allocate_detailed_array**, see the *CM Fortran Libraries Reference Manual*.

5.5.9 Example: Locating an Array Element

This section illustrates detailed layout information by showing how to locate a specified array element in a CM-5 under the VU execution model.

Locating an element means figuring out which VU it is assigned to, and then figuring out its offset from the base address of the subgrid within that VU. Detailed layout information includes subgrid axis lengths (abbreviated **sal** in the example below) for each axis, processor masks, or "**pdesc**s" for each axis, and axis ordering information. We use one-based axis numbering and indexing to describe Fortran arrays and zero-based numbering for the VUs and offsets within VU memory.

To calculate the **sal(n)** for each axis, divide the total physical axis length (including garbage positions, if any) by the number of processors along that axis (as determined by the number of 1 bits in the processor mask). Thus, if an axis is 1024 elements long, and there is no padding, and the processor mask is 12 (1100 = 2 bits set = 2^2 = 4), then **sal()** for that axis is 1024/4=256.

Example 1:

A(1024,1024) on 16 VUs (4 PNs), layout of **A** is (**:NEWS**, **:NEWS**)

```
sal(1) = 256
sal(2) = 256
pdesc(1) = 12 (binary 1100)
pdesc(2) = 3 (binary 0011)
```


Copyright © 1994 Thinking Machines Corporation

Problem: locate element **A(350,612)**:

```
350-1 = 349 =
  (349/sal(1))*sal(1) + MOD(349,sal(1)) = 1*sal(1) + 93
612-1 = 611 =
  (611/sal(2))*sal(2) + MOD(611,sal(2)) = 2*sal(2) + 99
```

So **A(350,612)** is on VU (1,2) in the physical grid and is element (93,99) within the subgrid on that VU. Now express the VU physical grid numbers in binary and shift them into the bitfields defined by the corresponding **pdesc**s:

(1,2) = (01,10) −−> 0110 = 6

To compute the offset from the base address within VU 6, we need to use the axis ordering information. With the default **-axisreorder**, we use right-to-left stride ordering. This means that element (93,99) is at offset 93*__sal__(2)+99=23907. With **-noaxisreorder**, we use left-to-right stride ordering. Thus element (93,99) is at offset 99*__sal__(1)+93=25437.

Example 2

A(1024,1024) on 32 VUs (8 PNs)
layout of **A** is (**:NEWS**,**:NEWS**)

```
sal(1) = 128
sal(2) = 256
pdesc(1) = 7 (binary 00111)
pdesc(2) = 24 (binary 11000)
```

Problem: locate **A(350,612)**

```
350-1 = 349 =
  (349/sal(1))*sal(1) + MOD(349,sal(1)) = 2*sal(1) + 93
612-1 = 611 =
  (611/sal(2))*sal(2) + MOD(611,sal(2)) = 2*sal(2) + 99
```

So **A(350,612)** is on VU (2,2) in the physical grid and is element (93,99) within the subgrid on that VU. Now express the VU physical grid numbers in binary and shift them into the bitfields defined by the corresponding **pdesc**s:

(2,2) = (010,10) −−> 10010 = 18

Copyright © 1994 Thinking Machines Corporation

To compute the offset from the base address within VU 18, we need to use the axis ordering information. With the default **-axisreorder**, we use right-to-left stride ordering. This means that element (93,99) is at offset 93*`sal`(2)+99=23907. With **-noaxisreorder**, we use left-to-right stride ordering. This means element (93,99) is at offset 99*`sal`(1)+93=12765.

5.6 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). Note that the discussion here refers to vector units, but applies also to programs running on CM-5s without vector units.

5.6.1 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 grid-communication operation of distance 1 along axis 2 — that is, each element sends a value to the element one coordinate higher along axis 2. Figure 33 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. 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-size*.

The number of moves within the vector unit is the total number of elements in the subgrid minus the subgrid-orthogonal-size (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

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

axis will, however, result in better communication performance along the axis.

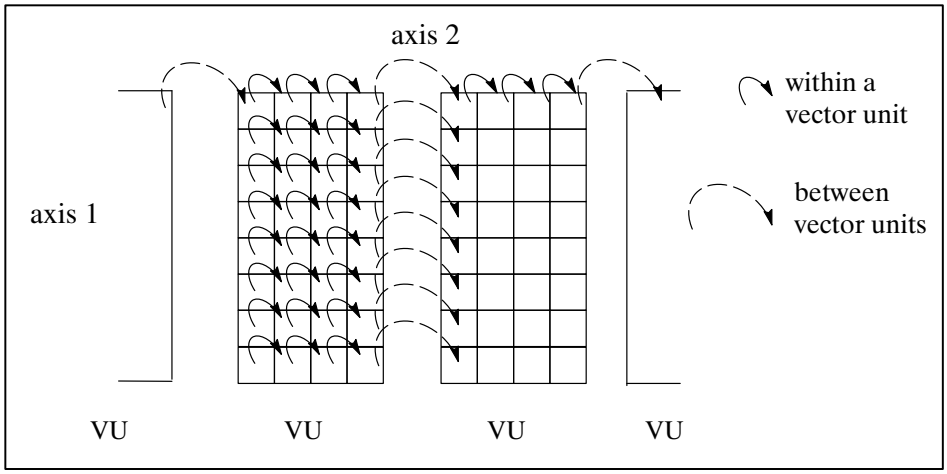

Figure 33. On-VU and off-VU data movement for a NEWS operation along axis 2.

5.6.2 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 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 with this default — whereby successive elements are placed "down memory" rather than across processors — it cannot lay out an array with 1 element of a parallel axis per processor. Instead, 8 elements (times the product of the serial axis lengths) are laid out in the first few processors, leaving the remaining processors with garbage elements only.

Copyright © 1994 Thinking Machines Corporation

5.6.3 Effect of Garbage Elements

If an array has garbage elements, they are not part of the array's data, but must be taken into account by communication procedures, requiring extra work to move data "around" the garbage locations. 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. For elemental operations, the only overhead that garbage elements add is that computations are carried out for those elements, even though the results are not used. Typically, this is not a problem because some other PN having no garbage elements will be doing useful work at the same time.

5.6.4 Geometries of Array Arguments

The geometry of a CM array argument to a procedure is determined from the descriptor of the actual argument (at the caller level), at the time of the procedure call. For example, the following program fails because the geometry used for **B** is the geometry stored in **A**'s descriptor, thus violating the CM Fortran restriction against changing array shape across program boundaries:

```
      REAL A(10,100)
CMF$  LAYOUT A(:SERIAL,:NEWS)
      CALL FOO(A)
      ...
      SUBROUTINE FOO(B)
      REAL B(2,5,100)     ! Error: shape mismatch
CMF$  LAYOUT B(:SERIAL,:SERIAL,:NEWS)
      ...
```

Similarly, the following program fails because of the attempt to change array **C**'s type and shape in the subroutine:

```
      COMPLEX C(8,128)
CMF$  LAYOUT C(:BLOCK=8 :PROCS=1,:BLOCK=1 :PROCS=128)
      CALL BAR(C)
      ...
      SUBROUTINE BAR(D)
      REAL D(16,128)      ! Error: type/shape mismatch
CMF$  LAYOUT D(:BLOCK=16:PROCS=1,:BLOCK=1 :PROCS=128)
      ...
```


Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

5.7 Querying an Array's Layout

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

Besides **CMF_DESCRIBE_ARRAY**, mentioned earlier, the library also provides these informational subroutines:

CMF_GET_AXIS_ORDERS
CMF_GET_GLOBAL_LOCATION
CMF_GET_PHYSICAL_LOCATION
CMF_GET_LOCAL_EXTENTS
CMF_GET_LOCAL_STRIDES
CMF_GET_PROCESSOR_MASKS

The last item, **CMF_GET_PROCESSOR_MASKS**, refers to the integer axis masks that describe the layout of axes across processors. See Section 5.5.3.

Note that **CMF_GET_AXIS_ORDERS** is not useful with the compiler switch **-noaxisreorder**. If **-noaxisreorder** is used, **CMF_GET_AXIS_ORDERS** returns **CMF_NEWS_ORDER** for all axes.

Copyright © 1994 Thinking Machines Corporation