# Appendix B

# Using allocate_detailed_shape for the CM-5

The CM-5's run-time system distributes the data for parallel variables among the processors (either nodes or vector units) of the partition on which the program is running. Parallel variables that have the same shape have their data distributed so that their elements are organized identically among and within the processors; this ensures that elemental operations don't require communication. The "layout" defines this organization; it is a property of the shape. Two shapes can have the same rank and dimensions but different layouts (this is why exchanging data between parallel variables of different shapes may require communication).

C* hides from the user the details of how shapes are actually laid out on a CM-5.[1] You may be able to improve your program's performance, however, by using the **`allocate_detailed_shape`** function to change a shape's physical layout from the default layout provided by the run-time system. You may also want to use **`allocate_detailed_shape`** if you are calling CM Fortran routines, and you need to have a shape's layout conform with a CM Fortran array layout.

Sophisticated use of **`allocate_detailed_shape`** requires an understanding of how the CM-5 run-time system maps shapes onto the nodes or vector units, and the implications of changing the default mapping. Sections B.1 through B.4 provide the necessary background information. If you already understand the issues involved, you can go directly to Section B.5, where we describe functions that you can call to determine a shape's layout, or to Section B.6, where we explain how to call **`allocate_detailed_shape.`**

1. Although we talk about "laying out a shape" in this appendix, note that what the run-time system really does is allocate memory for parallel variables of a given shape on the nodes or vector units.

# B.1 The Default Layout

This section describes how the current implementation of the run-time system lays out shapes on a CM-5 with vector units. Section B.2 discusses how shapes are laid out on a CM-5 without vector units. These procedures may change in future implementations.

Let's say you have a 2-dimensional shape with 8 positions along axis 0 and 12 positions along axis 1, and you are going to run your program on a 16-vector-unit (four-node) partition of a CM-5. How will the run-time system determine how to lay out the positions of the shape on the four nodes? Understanding how it does this requires understanding five concepts:

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

## B.1.1 Physical Grids

When laying out a shape, the run-time system arranges the physical vector units into a grid whose rank is the same as the rank of the shape. The total number of vector units in a partition is always a power of two; therefore, the number of vector units along each axis of the physical grid must be a power of two. Thus, for our example, the run-time system has these choices for arranging the 16 vector units into a 2-dimensional physical grid: grids of [4][4], [8][2], [2][8], [16][1], and [1][16].

## B.1.2 Garbage Positions

The run-time system tries to divide up the shape's positions equally among the vector units. In doing so, it follows these rules:

- Each vector unit must receive the same number of positions.
- The number of positions per vector unit must be a multiple of 8.

These rules also apply when you use **`allocate_detailed_shape`** to lay out the shape yourself.

Note, however, that it isn't possible to follow these rules in laying out an [8][12] shape on 16 vector units; each vector unit could receive the same number of positions, but the number of positions would not be a multiple of 8.

In such a case, the run-time system internally uses a layout with larger dimensions along one or more axes, so that the rules can be followed. It then lays out this new shape on the vector units. The actual shape can then be stored within this larger layout, leaving unused positions along the extended axes. These unused positions are referred to as *garbage positions*.

The run-time system always adds garbage positions to the high end of an axis, and adds as few garbage positions as possible. For our shape of [8][12] positions, the run-time system would pad axis 1 to make an underlying shape of [8][16]. Its layout on a physical grid of [2][8] is shown in Figure 94.

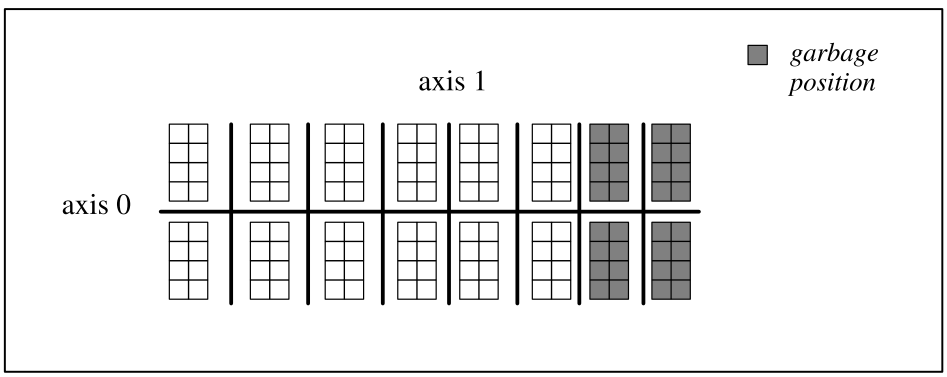


**Figure 94. An [8][12] shape laid out on a [2][8] physical grid.**

Typically you don't need to be aware that the garbage positions exist. You do need to take these positions into account, however, when determining how you want to lay out your shape using **`allocate_detailed_shape`**.

## B.1.3 Subgrids

As we mentioned above, the run-time system divides the shape into a number of sections, each section corresponding to a vector unit. These sections are called *subgrids*.

Note the layout requirements we have discussed so far:

- The physical grid must have a power-of-2 number of vector units along each dimension.
- Each vector unit must contain a multiple of 8 positions.
- Each vector must contain the same number of positions — in other words, the shape's subgrid must be the same size on each vector unit.

In the case of our sample [8][12]shape, padded to [8][16], we will have 8 positions per vector unit, but will they be laid out as a [4][2] subgrid, as shown in Figure 94, or, for example, as a [1][8] subgrid? The [1][8] layout implies a physical grid of [8][2], as shown in Figure 95.

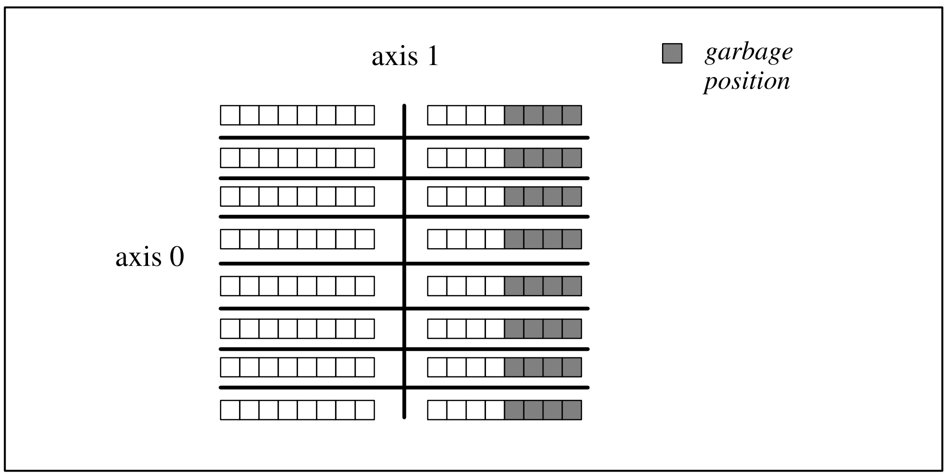


**Figure 95. An [8][12] shape laid out on an [8][2] physical grid.**

The basic process that the run-time system follows is to minimize the size of the subgrid. In other words, it uses as few garbage positions as possible; see Section B.1.2. This doesn't help us choose between the [2][4] and [1][8] subgrids, which both have the same number of garbage positions. It does, however, eliminate subgrids that use fewer than all 16 vector units. In practice, the run-time system chooses a layout that uses fewer than all of the vector units only when the number of positions in the shape is small relative to the number of vector units on which the program is to run.

To determine which layout the run-time system actually uses, you include in your program the appropriate functions discussed in Section B.5.

If you prefer a different subgrid size from the one that the run-time system uses, you can use **`allocate_detailed_shape`** to specify different lengths for the subgrid axes, as long as the resulting subgrid meets the requirements listed at the beginning of this section.

Note that, given the dimensions of a shape, the physical grid determines the subgrid, and vice versa — if you choose to minimize the number of garbage positions. You can specify either the physical grid or the subgrid when you use **`allocate_detailed_shape`**.

### B.1.4 Axis Sequence

One piece of information left out of the layouts shown in Figure 94 and Figure 95 is the numbering of the vector units within the physical grid. In the [2][8] physical grid layout, for example, Figure 96 shows two ways in which the vector units could be numbered.

*axis 1*

| *axis 0* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 |
| | 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 |

or

*axis 1*

| *axis 0* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

**Figure 96. 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 96, the vector-unit numbers increase fastest (that is, by the smallest interval) along axis 0; we call this an *axis sequence* of (0, 1) — the first axis in the sequence is the one that varies fastest. In the example on the bottom in Figure 96, they increase fastest along axis 1; this corresponds to an axis sequence of (1, 0).

By default for C*, the current implementation of the run-time system lays out multidimensional shapes so that the vector-unit numbers vary fastest along the highest-numbered axis — that is, it would choose the axis sequence of (1, 0) in our example. You can choose a different axis sequence via **allocated_detailed_shape**.

The axis sequence used is the only feature of the vector-unit numbering that you can currently control.

## B.1.5 Subgrid Sequence

The final issue with regard to the default layout for our sample shape is how the positions 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][2], there are two possible layouts of the positions in vector-unit memory:

```
[0][0]                [0][0]
[0][1]                [1][0]
[1][0]                [2][0]
[1][1]       or       [3][0]
[2][0]                [0][1]
[2][1]                [1][1]
[3][0]                [2][1]
[3][1]                [3][1]
```

By default for C*, the current implementation of the run-time system chooses the layout on the left; the highest-numbered axis varies fastest (that is, the adjacent subgrid positions along the highest-numbered axis are contiguous in memory).

You can use **allocate_detailed_shape** to choose the other subgrid sequence, in which positions along the lowest-numbered axis are contiguous in memory.

## B.1.6 Putting It All Together

It turns out that the run-time system chooses a [2][8] physical grid for our [8][12] shape. Given the information covered so far, we can now show exactly how the run-time system would lay out the shape using this physical grid. See Figure 97.

| VU 0 | | | | | | | 7 |
|---|---|---|---|---|---|---|---|
| [0][0] | [0][2] | [0][4] | [0][6] | [0][8] | [0][10] | [x][x] | [x][x] |
| [0][1] | [0][3] | [0][5] | [0][7] | [0][9] | [0][11] | [x][x] | [x][x] |
| [1][0] | [1][2] | [1][4] | [1][6] | [1][8] | [1][10] | [x][x] | [x][x] |
| [1][1] | [1][3] | [1][5] | [1][7] | [1][9] | [1][11] | [x][x] | [x][x] |
| [2][0] | [2][2] | [2][4] | [2][6] | [2][8] | [2][10] | [x][x] | [x][x] |
| [2][1] | [2][3] | [2][5] | [2][7] | [2][9] | [2][11] | [x][x] | [x][x] |
| [3][0] | [3][2] | [3][4] | [3][6] | [3][8] | [3][10] | [x][x] | [x][x] |
| [3][1] | [3][3] | [3][5] | [3][7] | [3][9] | [3][11] | [x][x] | [x][x] |
| | | | | | | | |
| [4][0] | [4][2] | [4][4] | [4][6] | [4][8] | [4][10] | [x][x] | [x][x] |
| [4][1] | [4][3] | [4][5] | [4][7] | [4][9] | [4][11] | [x][x] | [x][x] |
| [5][0] | [5][2] | [5][4] | [5][6] | [5][8] | [5][10] | [x][x] | [x][x] |
| [5][1] | [5][3] | [5][5] | [5][7] | [5][9] | [5][11] | [x][x] | [x][x] |
| [6][0] | [6][2] | [6][4] | [6][6] | [6][8] | [6][10] | [x][x] | [x][x] |
| [6][1] | [6][3] | [6][5] | [6][7] | [6][9] | [6][11] | [x][x] | [x][x] |
| [7][0] | [7][2] | [7][4] | [7][6] | [7][8] | [7][10] | [x][x] | [x][x] |
| [7][1] | [7][3] | [7][5] | [7][7] | [7][9] | [7][11] | [x][x] | [x][x] |
| 8 | | | | | | | 15 |

[x][x] = garbage data

**Figure 97. Default layout of an [8][12] shape 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 1.
- Data movement within the vector units of a node is faster than data movement across nodes.

## B.2 Layout without Vector Units

If your program is not going to use the CM-5's vector units, the layout is done in terms of nodes instead of vector units. The layout rules are basically the same, except that the rule that the subgrid must be a multiple of 8 positions does not apply.

If the run-time system were to lay out our sample [8][12] shape on 4 nodes, it could therefore choose among the physical grids and subgrids shown in Figure 98.

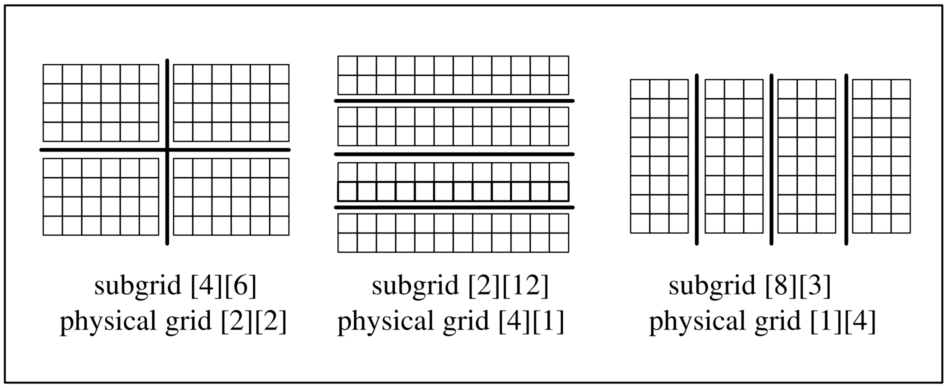


**Figure 98. Subgrids and physical grids for an [8][12] shape on four nodes.**

Note that no padding is required, because the 96 positions of the shape divide up evenly into 24 per subgrid.

The run-time system in this case would choose the [4][6] subgrid. The actual default layout is shown in Figure 99.

## B.3 Controlling Subgrid Layout: Using Serial Axes and Weighting Axes

As we mentioned earlier, **`allocate_detailed_shape`** lets you control the parameters that the run-time system uses in laying out a shape (subgrid lengths, axis sequence, etc.). The **`allocate_detailed_shape`** function provides two additional mechanisms for controlling subgrid layout:

- serial axes
- weighting of axes

We discuss serial axes below, and weighting of axes in Section B.3.2.

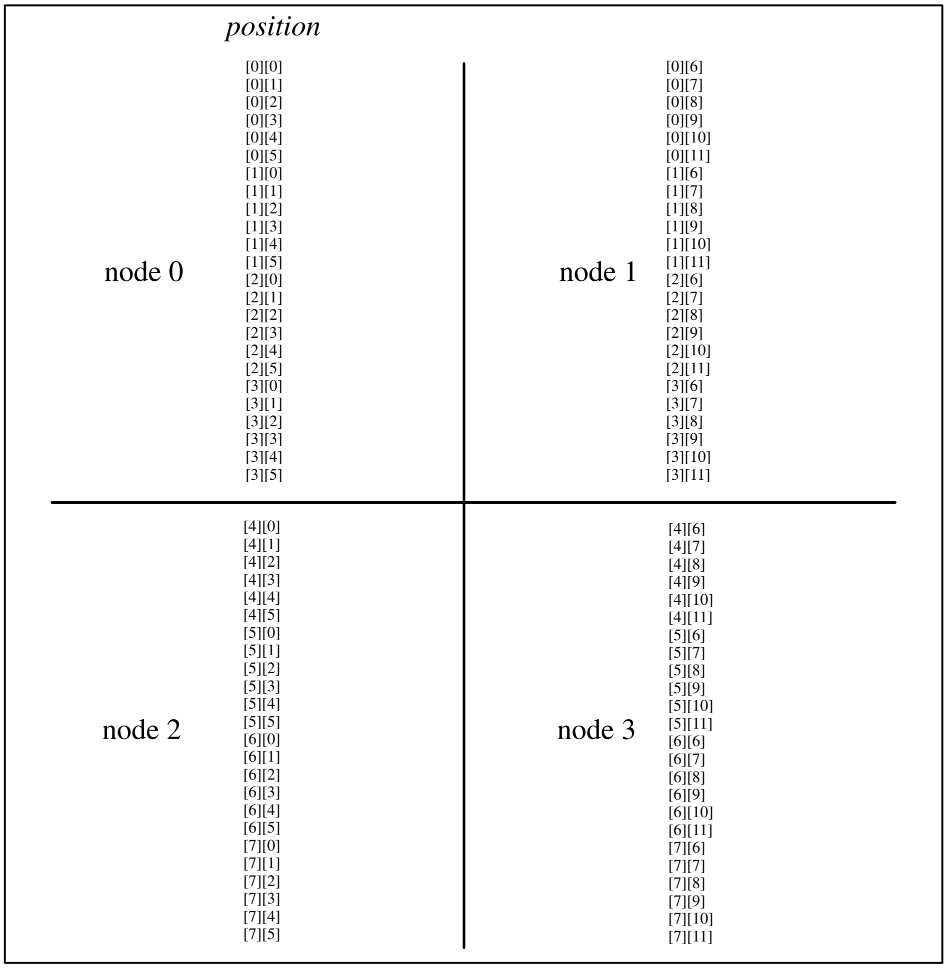


**Figure 99. Default layout of an [8][12] shape on four nodes.**

## B.3.1 Serial Axes

The **`allocate_detailed_shape`** function allows you to specify an axis as *serial*. Typically you would do this to create parallel variables that conform to CM Fortran arrays that contain a serial axis.

Specifying an axis as serial has two effects:

- All positions along the axis will be located on the same vector unit. (This layout is useful if communication tends to occur along just one axis, since communication is faster within a vector unit than between vector units.)

For example, if your shape is [3][8][12] and you specify that axis 0 is to be serial, all positions along axis 0 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 0 didn't exist — see Figure 97. One result of this is that the run-time system never adds garbage positions to a serial axis; it always satisfies the requirements for garbage positions in the non-serial axes.

- The positions along a serial axis in the current implementation vary more slowly than positions along any non-serial axis — that is, they must be furthest away from each other in memory. For example, if you have a sub-grid that is [2][3][4], by default the order of positions in memory is:

```
[0][0][0]
[0][0][1]
[0][0][2]
[0][0][3]
[0][1][0]
[0][1][1]
[0][1][2]
[0][1][3]
[0][2][0]
[0][2][1] etc.
```

If you were to specify that axis 1 is serial, the sequence would change to:

```
[0][0][0]
[0][0][1]
[0][0][2]
[0][0][3]
[1][0][0]
[1][0][1]
[1][0][2]
[1][0][3]
[0][1][0]
[0][1][1] etc.
```

Axis 1 now varies most slowly. If you have more than one serial axis, by default the highest-numbered axis varies fastest, just as it does for non-serial axes. In effect, serial and non-serial axes are ordered independently.

### B.3.2 Weighting Axes

The **`allocate_detailed_shape`** function also lets you control layout by differentially weighting the shape axes.

Weighting one axis more heavily than another axis tells the run-time system that more communication is to take place along the more heavily weighted axis. The run-time system therefore tries to localize more of the positions along that axis within a node or vector unit, thereby reducing the cost of the communication along that axis. In our example of laying out a shape that is [8][12] on 16 vector units, weighting axis 1 would tend to result in a [1][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 run-time system to treat them similarly.

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

By default, the run-time system weights all axes evenly.

## B.4 Performance Issues

The main reason to use **`allocate_detailed_shape`** is to improve performance over what you can obtain using the default layout chosen for a shape by the run-time system. 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.

### B.4.1 Effect of Subgrid Length and Physical Grid

As we discussed above, the run-time system by default chooses the smallest subgrid size. This is the most efficient size for operations that are internal to a vector unit — that is, operations that don't involve communication between vector units.

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.

Let's look at the communication issue in more detail. Assume that we're using a subgrid size of [8][4] and we want to do a grid-communication operation of distance 1 along axis 1 — that is, each element of a parallel variable sends a value to an element that is one coordinate higher along axis 1. For example:

```
to_torus_dim(&dest, source, 1, 1);
```

Figure 100 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.

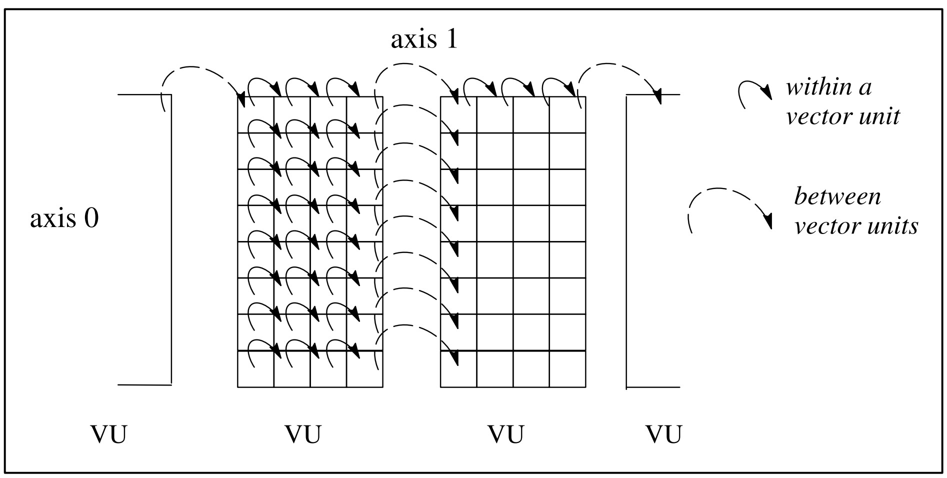


**Figure 100. On-VU and off-VU data movement for a NEWS operation along axis 1.**

In general, the number of off-VU moves in an operation like this is equal to the total number of positions 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 positions 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 will, however, result in better communication performance along the axis.

### B.4.2 Effect of Serial Axes

Making an axis serial guarantees that all positions 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 of the other constraints on shape layout must be satisfied by the non-serial axes, a shape with only a small number of positions along the non-serial axes can end up with an inefficient layout, because it will require a large number of garbage positions.

### B.4.3 Effect of Garbage Positions

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

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

For elemental operations, the only overhead that garbage positions add is that computations are carried out for those positions, even though the results are not used.

## B.5 Determining a Shape's Layout

C* provides several functions you can call to find out how the run-time system actually laid out a shape. You can use these functions to determine if you should use **allocate_detailed_shape** to specify a different layout.

Include the header file **<csshape.h>** when you call any of these functions.

The functions are defined as follows:

```
CMC_axis_order_t CMC_shape_axis_ordering(shape s, int axis);
int CMC_shape_physical_axis_mask(shape s, int axis);
int CMC_shape_subgrid_axis_length(shape s, int axis);
int CMC_shape_subgrid_axis_sequence(shape s, int axis);
int CMC_shape_subgrid_size(shape s);
int CMC_shape_subgrid_axis_increment(shape s, int axis);
int CMC_shape_subgrid_axis_orthogonal_length(shape s,
                        int axis);
int CMC_shape_subgrid_axis_outer_increment(shape s, int axis);
int CMC_shape_subgrid_axis_outer_count(shape s, int axis);
```

**CMC_shape_axis_ordering** returns **CMC_news_order** if the specified axis is in NEWS order (the standard order), **CMC_serial_order** if it's in serial order.

**CMC_shape_physical_axis_mask** returns an integer that represents the physical mask for the specified axis. See Section B.6 for a discussion of physical masks.

**CMC_shape_subgrid_axis_length** returns the subgrid length of the specified axis.

**CMC_shape_subgrid_axis_sequence** returns an integer that represents the specified axis's place in the sequence of axes within a subgrid.

**CMC_shape_subgrid_size** returns the total number of positions in the subgrid for the specified shape.

**CMC_shape_subgrid_axis_increment** returns an integer representing how many positions in memory separate consecutive subgrid positions along the specified axis. This is calculated by multiplying the subgrid lengths of all axes that have smaller subgrid axis increments (that is, the axes with lower subgrid sequences). If the positions along the subgrid axis are contiguous in memory, the function returns 1.

**CMC_shape_subgrid_orthogonal_length** returns the subgrid-orthogonal-length for the specified axis. This is the total number of positions in the subgrid divided by the subgrid length of the axis.

**CMC_shape_subgrid_axis_outer_increment** returns the product of the subgrid axis increment and the subgrid axis length for the specified axis.

**CMC_shape_subgrid_axis_outer_count** returns the product of the subgrid lengths of all axes that have larger subgrid axis increments (that is, axes with higher subgrid sequences) than the axis you specify.

## B.6 Using allocate_detailed_shape

This section describes how to specify a layout using **allocate_detailed_shape**.

Include the header file **<csshape.h>** when you call **allocate_detailed_shape**.

The format is as follows:

```
shape allocate_detailed_shape (
    shape *s,
    int rank,
    unsigned long extents[],
    unsigned long weights[],
    CMC_axis_order_t axis_orderings[],
    int physical_masks[],
    int subgrid_lengths[],
    int subgrid_sequence[]);
```

where:

**s** is a pointer to a shape. The remaining arguments specify this shape, and the function returns it. You must provide a value for this argument.

**rank** specifies the number of dimensions in the shape. You must provide a value for this argument. For the remaining arguments, you supply pointers to **rank** elements, one for each dimension, starting with axis 0.

**extents** specifies the number of positions along each axis of the shape. You must provide values for this argument.

**weights** specifies the relative frequency of communication along each axis; see Section B.3.2. For example, weights of 1 for axis 0 and 2 for axis 1 specify that communication occurs about half as often along axis 0. Only the relative values of the weights matter; for example, weights of 5 for axis 0 and 10 for axis 1 specify the same communication as weights of 1 and 2. Specifying the same values for different axes indicates that they have the same level of communication.

The **weights** values are used only if you specify neither the **physical_masks** nor the **subgrid_lengths** argument.

Pass **NULL** instead of the **weights** array to use the default weights, which are 1 for each axis.

**axis_orderings**

specifies the ordering of each axis, either **CMC_news_order** or **CMC_serial_order**. Specify **CMC_news_order** to get the standard ordering; specify **CMC_serial_order** to make the axis serial. Pass **NULL** instead of this array to specify the default ordering, which is **CMC_news_order** ordering for each dimension.

**physical_masks**

specifies the mapping of positions to physical nodes or vector units — that is, it specifies the physical grid and axis sequence; see Sections B.1.1 and B.1.4. See Section B.6.1, below, for an explanation of how to specify the physical masks.

You can pass **NULL** instead of an array, in which case the values for this argument are automatically determined from the extents and subgrid lengths; see Section B.6.1. If **subgrid_lengths** is also **NULL**, the **weights** argument is used to define the shape.

**subgrid_lengths**

specifies the subgrid length for each axis; see Section B.1.3.

You can pass **`NULL`** instead of an array, in which case the subgrid lengths will be determined from the extents and the physical masks (as described in Section B.6.1), or from the weights if **`physical_masks`** is also **`NULL`**.

**`subgrid_sequence`**
specifies the sequence of axes within a subgrid; see Section B.1.5. The default is for the highest-numbered axis to vary fastest (that is, the sequence is the reverse of the axis numbers). You can also specify that the lowest-numbered axis is to vary fastest (that is, the sequence is 0, 1, 2,...). In either case, serial axes must be last in the sequence. Other sequences are not allowed. For example, specify **`[0, 2, 3, 1]`** if axis 1 is serial and the lowest-numbered dimension is to vary fastest.

## B.6.1 In More Detail

The **`allocate_detailed_shape`** function provides you with several different options for specifying a shape's layout. The easiest option is to specify weights for the different axes, and let the run-time system figure out the appropriate layout. The most complete and flexible method is to use the **`physical_masks`** and **`subgrid_lengths`** arguments to specify the exact layout you want.

Note that the discussion of the **`physical_masks`** argument in this section assumes that you are compiling for a CM-5 with vector units. If you are not compiling for the vector units, the use of the argument is the same, except that it applies to nodes instead of vector units.

To understand the **`physical_masks`** argument, it is useful to review the concept of a *physical grid*. The physical grid of a shape 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 shape. The dimensions of the physical grid are each powers of two, since there must be a power-of-two 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 physical 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][8], as shown in Figure 101.

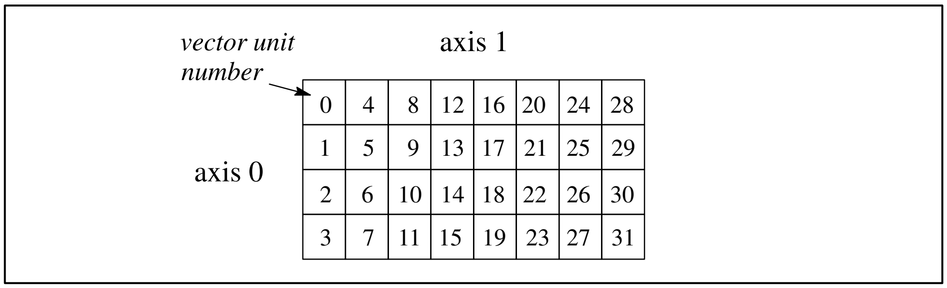


**Figure 101. A physical grid with a `physical_masks` argument of [3, 28].**

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

```
 bbbbb
MSB   LSB
```

To specify a [4][8] physical grid:

- Axis 0 (4 vector units) requires the lowest 2 bits of the mask because the vector units along this axis are contiguous; its **`physical_masks`** value is therefore 3:

  ```
   00011
  MSB   LSB
  ```

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

  ```
   11100
  MSB   LSB
  ```

If you want the vector units along axis 1 to be contiguous, the masks would be:

```
axis 0 = 11000     axis 1 = 00111
```

or [24, 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.

What if we want to maximize the speed of communication along axis 0? To do that, we could allocate all the bits of the physical mask to axis 1. This would create these masks:

```
axis 0 = 00000     axis 1 = 11111
```

or [0, 31]. Positions along axis 0 would be on the same vector unit, and performance would be best along that axis.

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

- To make vector units along axis 0 contiguous, specify axis 0 in the low-order bits:

  ```
  000111
  ```

  and axis 1 in the high-order bits:

  ```
  111000
  ```

  or [7, 56].

- To make vector units along axis 1 contiguous, reverse the masks: [56, 7].

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

- Each physical 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 shapes, which don't use all of the vector units.

If you use less than the total number of bits, the shape will use less than the total number of vector units; this is generally not a good idea.

- A serial axis must have a physical mask of 0.

Now let's consider how the physical mask works in conjunction with the subgrid length.

If you specify values for the **`physical_masks`** argument but omit the **`sub-grid_lengths`** argument (by passing a **`NULL`** value), **`allocate_detailed_shape`** will cause the appropriate subgrid lengths to be used. It calculates the subgrid length for an axis by dividing the total length of the shape axis by the length of the corresponding physical axis specified by the mask. If necessary, it rounds up to accommodate the whole axis.

For example, if you have a shape that is [64][50] and you specify a physical grid that is [4][8], the default subgrid lengths will be [16][7]; any other subgrid lengths would either be illegal (because the values were too small, and therefore didn't evenly divide the axis), or cause memory to be unused unnecessarily (because the values were too large).

Similarly, if you specify the subgrid lengths and omit the **`physical_masks`** argument, **`allocate_detailed_shape`** calculates the appropriate physical grid by dividing each shape axis by the subgrid length for that axis. It rounds up to the next power of 2 if necessary.

For example, if the shape is [32][128] and the subgrid lengths are [16][16], the physical grid will be [2][8]. If the shape is [34][128] and the subgrid lengths are [16][16], the physical grid will be [4][8].

If you specify subgrid lengths but pass **`NULL`** for the **`physical_masks`** argument, the *lowest-numbered* axis varies fastest; note that this is different from the default run-time system behavior.

If your shape requires garbage positions, **`allocate_detailed_shape`** provides the appropriate number required per axis.

### B.6.2 Example

The program below can be used to show the difference in speed of communication between the default layout and one specified via **allocated_detailed_shape**. The call to **allocate_detailed_shape** allocates all the bits of the physical mask to axis 1, and specifies a physical mask of 0 for axis 0; this maximizes communication performance along axis 0, at the expense of performance along axis 1.

```
#include <cscomm.h>
#include <cm/timers.h>
#include <stdio.h>

void time_grid(void)
{
    double:current a, b;
    int axis, i;

    for(axis = 0; axis < rankof(current); ++axis)
    {
         CM_timer_clear(0);
         CM_timer_start(0);

    for(i = 0; i < 20; ++i)
    {
        from_torus_dim(&b, &a,
              sizeof(double:current), axis, 1);
    }

    CM_timer_stop(0);
    printf("Timings for axis %d:\n", axis);
    CM_timer_print(0);
    }
}

main()
{
    shape ordinary, detailed;

    unsigned long extents[2];
    int physical_masks[2];

    extents[0] = 2048;
    extents[1] = 2048;

    physical_masks[0] = 0;
    physical_masks[1] = positionsof(physical) - 1;

    ordinary = allocate_shape(&ordinary, 2, extents);
```

```
    detailed = allocate_detailed_shape(&detailed, 2, extents,
                        NULL, /* weights */
                        NULL, /* axis orderings */
                        physical_masks, /* physical masks */
                        NULL, /* subgrid lengths */
                        NULL); /* subgrid sequences */

    with(ordinary)
    {
         printf("Ordinary layout:\n");
         time_grid();
    }

    {
         printf("Detailed layout:\n");
         time_grid();
    }
}
```

Note the use of **positionsof(physical) - 1** to specify the physical mask for axis 1. This makes the program portable among different-size partitions, and between CM-5s with and without vector units.

Here is the output from a sample run of this program on a 32-node partition with vector units:

```
Ordinary layout:
Timings for axis 0:
Starts: 1
CM Elapsed time: 0.146 seconds.
CM busy Time: 0.141 seconds.
Timings for axis 1:
Starts: 1
CM Elapsed time: 0.160 seconds.
CM busy Time: 0.155 seconds.
Detailed layout:
Timings for axis 0:
Starts: 1
CM Elapsed time: 0.127 seconds.
CM busy Time: 0.121 seconds.
Timings for axis 1:
Starts: 1
CM Elapsed time: 0.333 seconds.
CM busy Time: 0.325 seconds.
```

(Your results, of course, may vary.)

Note that the times for the **from_torus_dim** calls along axis 0 are somewhat faster for shape **detailed** than for shape **ordinary**. Timings along axis 1, however, are much slower along axis 1 for shape **detailed** than for shape **ordinary**. This shows the effect of using **allocate_detailed_shape**. The default layout provides superior performance in the general case, where there is communication along both axes. The detailed layout provides better performance for one case (communication along axis 0), but its performance in the general case is much worse than that provided by the default layout.