Appendix A

CMRTS and CM Memory Allocation

This appendix describes the features of the CM Run-Time System (CMRTS) that you are likely to use in CM Fortran programming. It also discusses methods for allocating parallel CM memory through the CMRTS.

Note: To make direct use of the CMRTS functions and data structures described in this chapter, you must write a subroutine in the C programming language, and include the CMRTS header file in the subroutine source file:

```
#include <cm/rts.h>
```

A.1 The CM Run-Time System (CMRTS)

The CMRTS is a set of low-level CM code libraries that define and manipulate array data structures in CM parallel memory. CMRTS functions allocate blocks of CM memory and manipulate their contents "at run-time" (that is, during execution of CM programs), hence the name of the library. The CMRTS is divided into three main libraries of functions:

- CMRT — the CM Run-Time Library
- CMCOM — the CM Communications Library
- CMIP — the CM In-Processor Library

The CMRT layer is the topmost layer of the RTS software, and represents an "external," machine-independent interface for the RTS. CMRT functions and data types provide access to all CM operations defined in the RTS. The CMCOM and CMIP layers are "internal" support software for the CMRT

Copyright © 1994 Thinking Machines Corporation

layer. CMCOM functions perform CM communication operations (sends, scans, etc.). CMIP functions perform in-processor operations (arithmetic and logical computations, etc., that don't require communication between CM processors). If your program makes any direct calls to the CMRTS, it is typically through the CMRT functions. However, references to data structures defined at the CMCOM level (in particular, CMCOM *machine geometries*) are common.

A.1.1 Arrays in the CMRTS

A high-level CM array, as defined in a parallel programming language such as CM Fortran or C*, is represented in the RTS by an *array descriptor*, of type **`CMRT_desc_t`**. This array descriptor is the topmost level of a hierarchy of data structures (see Figure 37) that form a bridge between high-level arrays and the physical memory of the CM hardware.

A high-level parallel array can have any reasonable shape and size, as permitted by the syntax of the programming language in use, and by the memory space available on the CM. However, the number of processing nodes available on the CM and the amount of memory available within each node typically remains fixed, placing a physical constraint on the sizes and shapes of arrays that can be conveniently stored in parallel memory.

The RTS allocates array memory in bands across the individual memory banks of the nodes, so that the starting memory address and size of the array region is the same for each node. (If the CM has vector units, these bands are across the individual memory banks of the VUs, as shown in Figure 37 below.)

Implementation Note: The RTS memory allocation routines ensure that each allocated region of memory is doubleword aligned (that is, starts at an address that is a multiple of 8 bytes).

An array with a number of elements that is an exact multiple of the number of processing nodes (or VUs) can be stored very neatly in such a memory stripe. But if the array is not an exact multiple of the machine size (or if the program that uses the array is run on a CM of a different size), then the array cannot be stored in CM memory without wasting some space on one or more of the nodes. This *garbage space* must be kept track of, so that the invalid values it contains are not confused with the actual data of the array.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

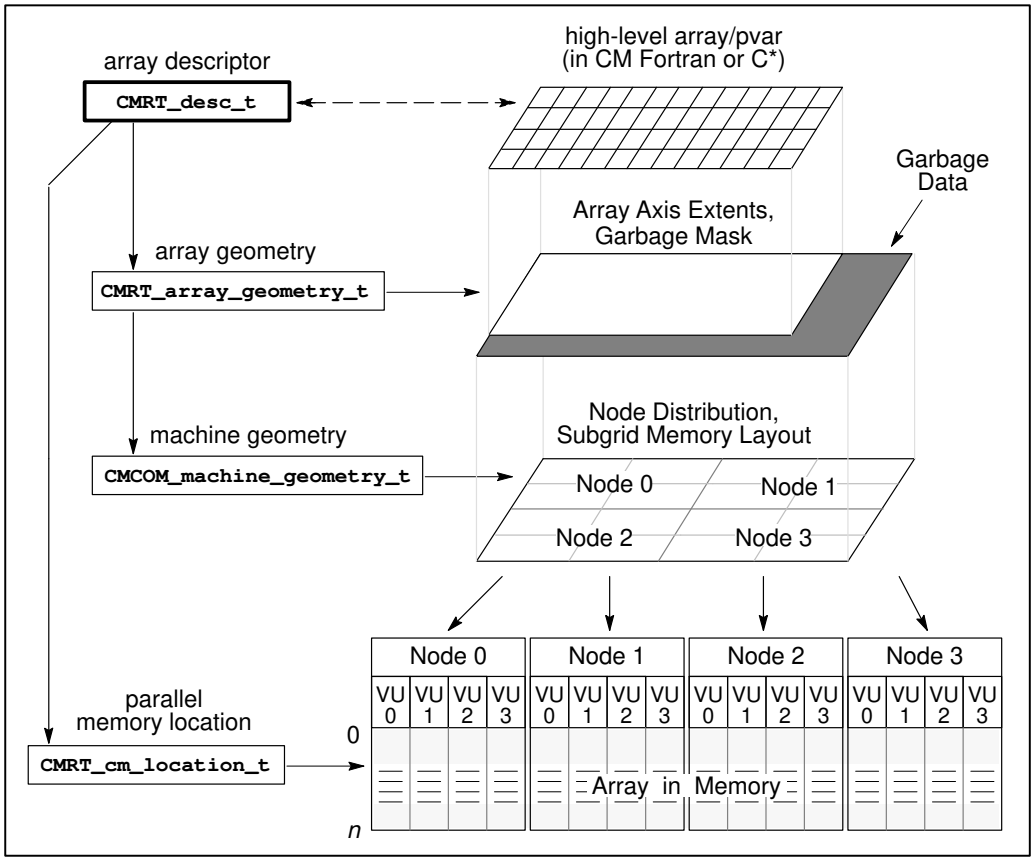


Figure 37. Internal structure of a CM array.

The many-layered structure of RTS arrays deals with these hardware constraints by using data structures known as *geometries* to define the arrangement of array data in CM memory. Two types of geometry are used to define the layout of a high-level array:

- A *machine geometry*, of type **`CMCOM_machine_geometry_t`**, which describes the structure of an arbitrary array that is sized and shaped to fit exactly into a stripe of CM memory.
- An *array geometry*, of type **`CMRT_array_geometry_t`**, which refers to a specific machine geometry and selects a region of it to represent the actual data of a high-level array. (The unused space defined by the machine geometry is considered *garbage data*, and is ignored.)

Copyright © 1994 Thinking Machines Corporation

A CM array descriptor (a **CMRT_desc_t** data structure) includes

- an array geometry, which defines the layout of the array
- a parallel memory location, of type **CMRT_cm_location_t**, which defines the start of the band of parallel memory that holds the array data

For the Curious: This multi-layer array data structure saves storage space, since a single machine geometry data structure can be shared among the descriptors of many array geometries. It also makes it possible to use a simple pointer comparison to determine whether two arrays have the same machine geometry (because arrays with the same machine geometry actually reference an identical **CMCOM_machine_geometry_t** data structure).

A.1.2 An Example of a CMRTS Array

Let's take a specific example. Suppose that we have a CM-5 with a very small number of processing nodes — four, to be exact (see Figure 38). This CM-5 has vector units, so that the number of *processing elements* in the machine is 16: four nodes with four VUs per node. (If this CM-5 did not have vector units, then there would be only four processing elements — the four processing nodes themselves.)

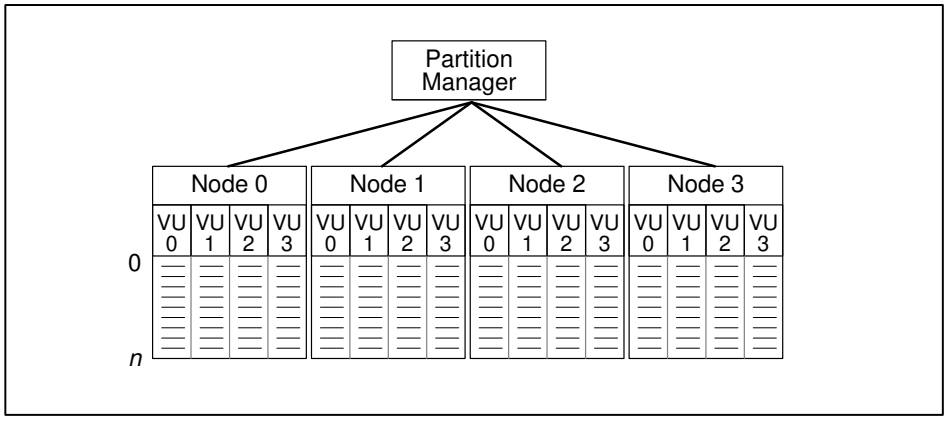


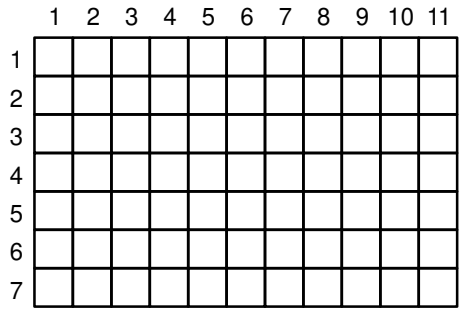
Figure 38. A hypothetical 4-node CM-5 system (with VUs).

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

Suppose further that we compile and run a CM Fortran program on this machine, defining a floating-point array as follows:

```
DOUBLE PRECISION LUCKY (7,11)
```

How is this array stored in the memory of the CM? First, look at the array itself:



It's a two-dimensional array of 77 elements, each of which is a double-precision floating-point number. This is not evenly divisible across 16 VUs, so a small number of garbage elements will need to be added. In general, the garbage space of an array is added by extending the axes of the array, adding garbage elements at the high ends of one or more axes.

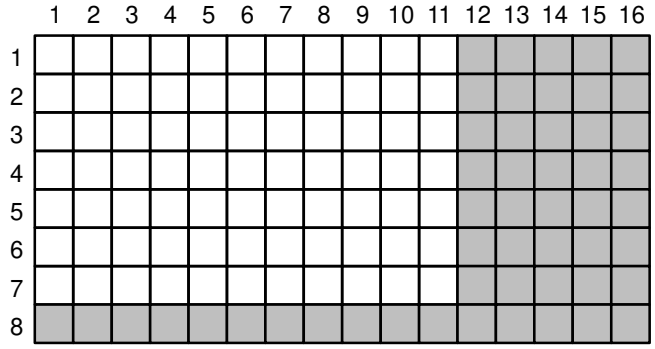
The amount of garbage space to add is determined as follows:

- Enough garbage space must be added so that the array can be divided into pieces of equal size and shape for each CM node.
- The part of the array assigned to each node must furthermore be divisible into 4 parts of equal size and shape, one for each of the node's VUs.
- Each node's portion of the array should be of the same rank as the entire array, and should have the same basic shape.
- The amount of added garbage space should be as small as possible.

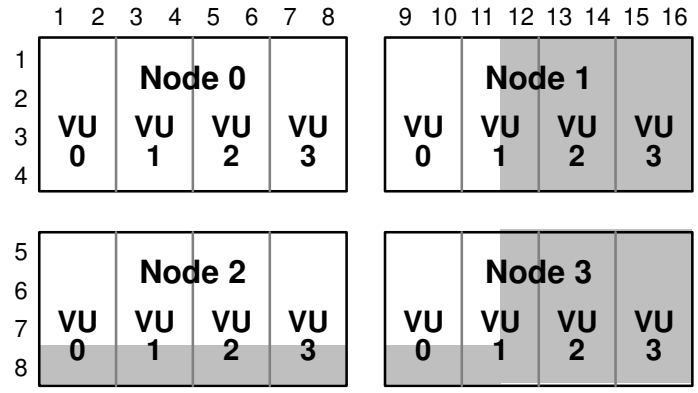
In addition, there is an implementation-dependent restriction: the number of array elements assigned to each VU must be a multiple of 8, unless you compile with the **`-nopadding`** switch.)

Copyright © 1994 Thinking Machines Corporation

Following these guidelines, a 7 x 11 array is padded out into an 8 x 16 array,



and is divided among the nodes and VUs as follows:



Note that this assigns a 4 x 2 portion of the array to each vector unit.

The portion of the array assigned to each vector unit is called the array's *subgrid*. This subgrid is the same size and shape for each VU, and basically represents the part of the array that is stored in the memory of each VU. The size of the subgrid determines the total amount of parallel memory allocated for the array. Exactly enough memory for one subgrid is allocated in the memory bank of each VU.

The subgrid of our sample array contains 8 double-precision floating-point values. A double-precision float in Fortran occupies two words of memory, so 16 words of parallel memory must be allocated on each CM node to contain the array. The *n*-dimensional subgrid stored on each VU is "unwrapped" and stored as a one-dimensional series of VU memory words.

Copyright © 1994 Thinking Machines Corporation

As you might expect, this is done systematically by defining a single memory storage order for all subgrids. In the case of the sample (7,11) array, the memory storage order is as shown below (remember that each double-precision subgrid element is stored as 2 words in memory):

subgrid axis 0 \ subgrid axis 1	**1**	**2**
1	1 2	3 4
2	5 6	7 8
3	9 10	11 12
4	13 14	15 16

The resulting actual memory distribution of the array is shown below:

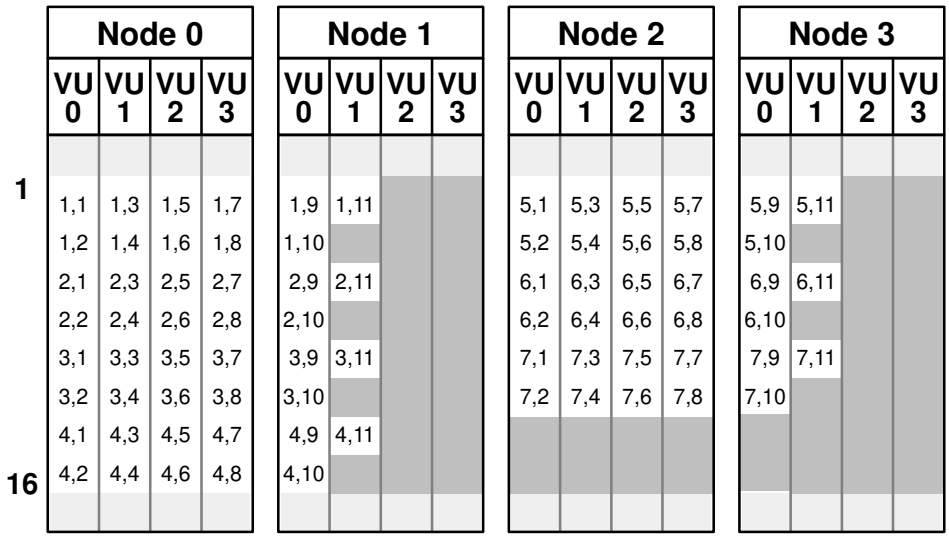


Figure 39. The actual memory distribution of a 7 x 11 CM Fortran array.

As you can see, this arrangement of array data values in CM memory bears almost no relation to the shape of the original high-level array. It is the array's geometry alone that determines how the array data in CM memory is interpreted. With a different array geometry, the values of this two-dimensional array could just as readily be accessed as a one-dimensional array or

Copyright © 1994 Thinking Machines Corporation

a three-dimensional array, by suitably adjusting the axis lengths. A complete discussion of the algorithms used to determine the layout of a CM array in memory is beyond the scope of this appendix, but this example should help you understand the information found in CMRTS array descriptors and geometry objects, as described in the following sections.

A.1.3 CMRTS Data Structures

CMRT_desc_t

This is the top-level array descriptor data structure. (See Figure 37.) It contains references to all component data structures that define a single high-level array.

The user-accessible structure slots are:

`CMRT_cm_location_t` **`cm_location`**

This is the parallel memory location (the same address on each VU) at which the array's allocated memory region begins. The amount of memory allocated is determined by the array size and shape specified in the **`array_geometry`** structure.

`CMRT_array_geometry_t` **`array_geometry`**

This is the array geometry, which specifies the size, shape, and garbage region of the array. See the description of the **`CMRT_array_geometry_t`** data structure below.

`int4` **`element_size`**

This is the size, in bytes of memory, of a single array element.

The CMRTS functions used to access these structure slots are:

```
CMRT_cm_location_t CMRT_desc_get_cm_location(descriptor)
    CMRT_desc_t descriptor;
CMRT_array_geometry_t CMRT_desc_get_geometry(descriptor)
    CMRT_desc_t descriptor;
int4 CMRT_desc_get_element_size(descriptor)
    CMRT_desc_t descriptor;
```


Copyright © 1994 Thinking Machines Corporation

CMRT_array_geometry_t

This is the array geometry, which specifies an array's extents, garbage space, and machine geometry (only the first two are directly specified by the array geometry; the machine geometry is specified by referring to a CMCOM data structure).

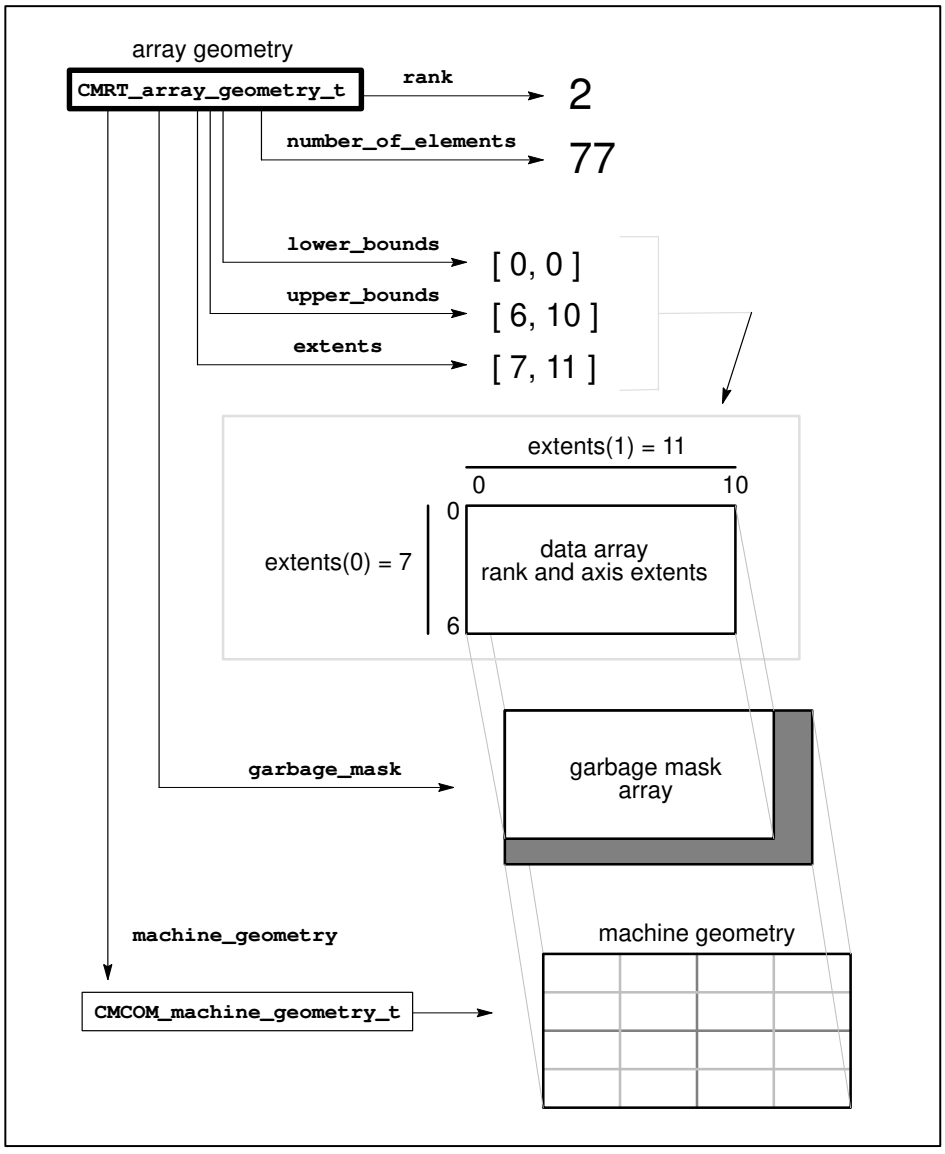


Figure 40. The CMRT_array_geometry data structure.

Copyright © 1994 Thinking Machines Corporation

The user-accessible structure slots are:

`int4` **`rank`**

This is the rank (number of dimensions) of the array. This is typically the same as the rank specified by the **`machine_geometry`**.

`int8` **`number_of_elements`**

This is the total number of actual (non-garbage) data elements in the array (simply the product of the array's axis extents, defined below).

`int8` **`*extents, *lower_bounds, *upper_bounds`**

These are integer arrays specifying the lengths and lower/upper axis indices for each array axis. Note that these values are completely independent of the array extents specified by the **`machine_geometry`** — the bounds arrays specify what part of the array is used for actual data.

Note: The lower and upper bound values in these arrays are zero-based, unlike Fortran array indices, which are one-based, so the values in these arrays are always one less than the actual values in the CM Fortran source program. The CMRTS is coded in C, and thus follows the C conventions for array indexing.

`CMCOM_machine_geometry_t` **`machine_geometry`**

This is the machine geometry, which specifies the parallel memory layout for the array. See the description of the **`CMCOM_machine_geometry_t`** data structure below.

`CMRT_cm_location_t` **`garbage_mask`**

This is the *garbage mask* array, a region of parallel memory that defines the garbage space of the array. The garbage mask contains boolean values, with a **`TRUE`** value representing garbage elements. Essentially, the garbage mask provides the same information as the extents and bounds arrays described above, but in an array format that is more convenient for some CMRTS operations. The garbage mask is typically stored in a compressed format to save space, so extracting the appropriate boolean value for a given array element is not as simple as it would be for an ordinary logical array.

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

To access these slots, you can use the following accessor functions (note that these functions are applied to the *array descriptor*, not to the geometry object):

```
int4 CMRT_desc_get_rank(descriptor)
    CMRT_desc_t descriptor;

int8 CMRT_desc_get_number_of_elements(descriptor)
    CMRT_desc_t descriptor;
```

(The following accessors take an extra **axis** argument, zero-based as in C, and the appropriate value for the specified array axis is returned:)

```
int8 CMRT_desc_get_lower_bound(descriptor, axis)
    CMRT_desc_t descriptor;
    int4 axis;

int8 CMRT_desc_get_upper_bound(descriptor, axis)
    CMRT_desc_t descriptor;
    int4 axis;
```

You can also access the structure slots directly (you must do this to access the **extents** and **machine_geometry** slots, for example):

```
( array_descriptor->array_geometry ) -> extents[axis]

( array_descriptor->array_geometry ) -> machine_geometry
```


Copyright © 1994 Thinking Machines Corporation

CMRT_machine_geometry_t

This is the machine geometry, which specifies the actual parallel memory layout of the array (the division of the array among the nodes and VUs, and the size and shape of the subgrid stored in each VU's memory).

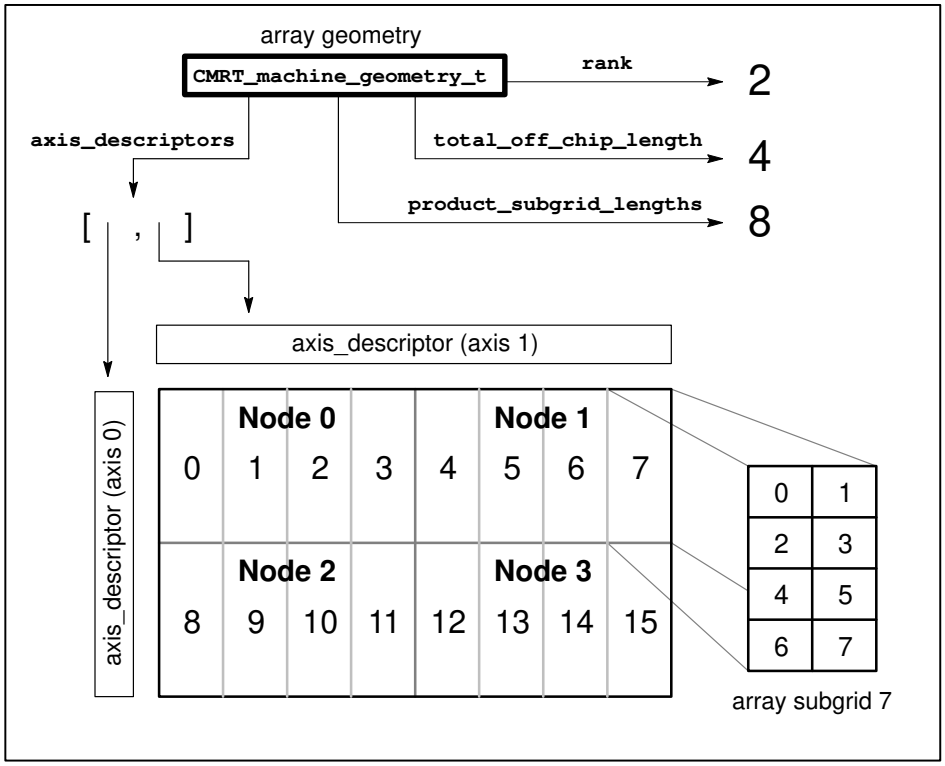


Figure 41. The `CMCOM_machine_geometry` data structure.

The user-accessible structure slots are:

`int4` **`rank`**

This is the rank (number of dimensions) of the array. This is typically the same as the rank specified by the array geometry.

`unsigned char` **`total_off_chip_length`**

This is the logarithm (base 2) of the total number of subgrids (specifically, this is the total number of physical, or "off-chip" bits required

Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

in send addresses of array elements). **Note:** While this value typically represents the number of VUs in your CM, it may sometimes be less. This is particularly the case for small arrays that do not use all the nodes of the CM to store array data.

`int4` **`product_subgrid_lengths`**

This is the total number of elements in each subgrid (that is, the product of all the subgrid axis lengths).

`CMCOM_axis_descriptor *`**`axis_descriptors`**

This is an array of axis descriptor data structures, one for each axis of the array. See the description of the **`CMCOM_axis_descriptor`** data structure below.

To access the **`product_subgrid_lengths`** slot, you can use the following accessor function:

```
int4 CMRT_desc_get_subgrid_size(descriptor)
    CMRT_desc_t descriptor;
```

You can also access the structure slots directly (you must do this to access the remaining slots):

```
CMCOM_machine_geometry_t m_geometry =
  array_descriptor->array_geometry -> machine_geometry

m_geometry -> rank

m_geometry -> total_off_chip_length

m_geometry -> axis_descriptors[axis]
```


Copyright © 1994 Thinking Machines Corporation

CMCOM_axis_descriptor

This is the axis descriptor data structure, which defines the geometry information for a single axis of a machine geometry.

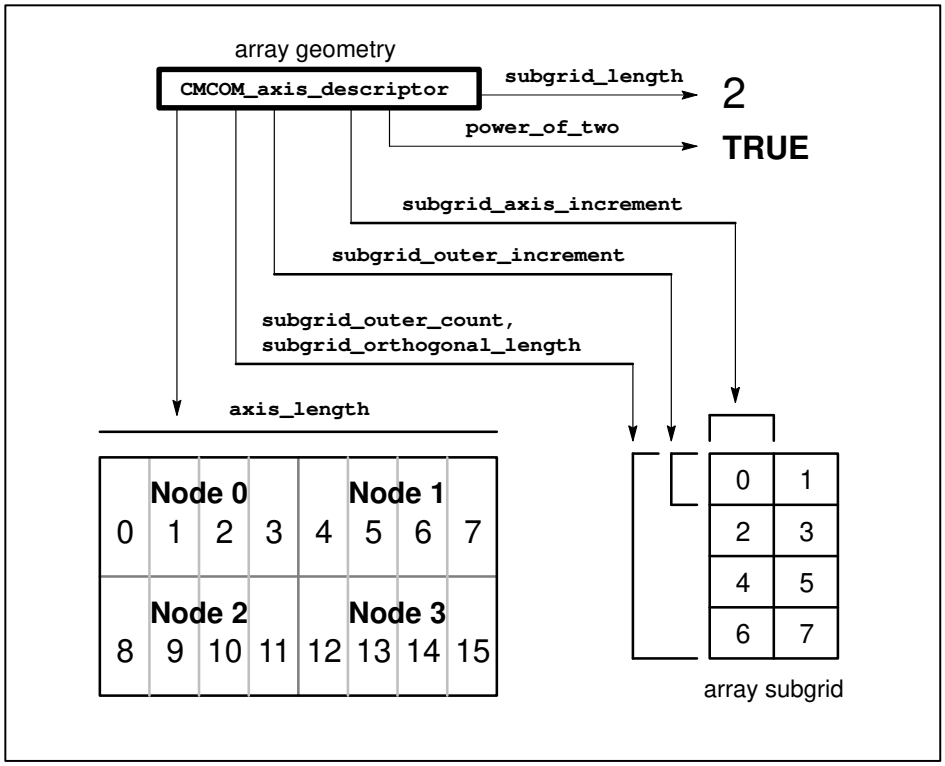


Figure 42. The `CMCOM_axis_descriptor` data structure.

The user-accessible structure slots are:

`int4` **`subgrid_length, power_of_two`**

The **`subgrid_length`** is the number of subgrid elements along the given axis. The **`power_of_two`** slot is a flag that is **`TRUE`** if and only if the subgrid length is an exact power of two.

Copyright © 1994 Thinking Machines Corporation

`int8` **`axis_length`**

This is the total length (number of array elements) of the array axis. (This is basically the subgrid length along the given axis times the number of subgrids.)

`int4` **`subgrid_axis_increment`**

This is the number of array elements in memory that must be skipped to move from each subgrid element along the given axis to the next element.

`int4` **`subgrid_outer_increment`**

This is the product of the **`subgrid_axis_increment`** and the **`subgrid_length`**; in other words the number of array elements in memory that must be skipped to move past *all* the subgrid elements along a single axis. (If **`subgrid_axis_increment`** is the distance between elements in a row, for example, then **`subgrid_outer_increment`** is the distance between the first elements of successive rows of the subgrid.)

`int4` **`subgrid_outer_count`**

This is the result of dividing the subgrid size (number of elements) by the value of **`subgrid_outer_increment`**. In other words, it is the number of iterations that would be needed to step through the entire subgrid using increments of **`subgrid_outer_increment`**. (This slot is used internally in the CMRTS to quickly calculate looping limits for operations that take place over the entire subgrid.)

`int4` **`subgrid_orthogonal_length`**

This is the product of the subgrid lengths of all other axes in the array. In other words, this is the number of subgrid elements in a single multidimensional “slice” through the array that is perpendicular to the given axis. (In a three-dimensional array, for example, this would be the number of rows and columns of elements in each horizontal “slice” of the vertical axis.)

Copyright © 1994 Thinking Machines Corporation

(The remaining slots are used to specify the send addresses of array elements. These slots are not shown in Figure 42.)

```
unsigned char off_chip_positions, off_chip_length
unsigned4 off_chip_mask
```

These slots specify the physical, or "off-chip" part of an array element's send address that corresponds to the given axis. Literally, these values are the starting position of the physical, or "off-chip" bits assigned to the axis, the number of bits assigned to the axis, and a binary mask that selects only those bits from a send address. Note that the bit positions corresponding to a single axis must all be contiguous.

```
unsigned char subgrid_bits_position, subgrid_bits_length;
int4 subgrid_bits_mask;
```

These slots specify the subgrid part of an array element's send address that corresponds to the given axis. (Literally, these values are the starting position of the subgrid bits assigned to the axis, the number of bits assigned to the axis, and a binary mask that selects only those bits from a send address.)

Note: The **subgrid_bits** slots are valid only if the **power_of_two** slot is **TRUE** — in other words, if the number of elements in the subgrid is an exact power of two.

To access the **subgrid_length** slot, you can use this accessor function:

```
int4 CMRT_desc_get_subgrid_dimension(descriptor, axis)
    CMRT_desc_t descriptor;
    int4 axis;
```

You can also access the structure slots directly (you must do this to access the remaining slots):

```
CMCOM_machine_geometry_t m_geometry =
  array_descriptor->array_geometry -> machine_geometry

CMCOM_axis_descriptor axis_d =
  m_geometry -> axis_descriptors[axis]

axis_d -> axis_length

axis_d -> subgrid_axis_increment

axis_d -> subgrid_outer_count
```


Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

A.2 CMRTS Parallel Memory Allocation

For some special-purpose applications, it is necessary to allocate parallel CM memory other than by using a high-level language to define an array. To do this, you can use the memory allocation functions of the CMRTS.

A.2.1 Standard CMRTS Memory Allocation Functions

Parallel VU memory is mapped into two general regions of memory, the parallel stack and the parallel heap. Both the stack and the heap regions grow upward, toward higher memory addresses. When you allocate new space in these regions, it is allocated as a stripe across the physical memory of the VUs. **Note:** Stack and heap memory regions are always allocated as 8-byte aligned regions by rounding each memory allocation request up to the next 8-byte boundary.

There are two ways to allocate either stack or heap memory space: by a *physical* block of memory, or by a *geometry* block of memory. The difference is essentially one of convenience. If you know exactly how many bytes of memory you want to allocate on each VU, use the physical memory allocation functions:

```
CMRT_cm_location_t
  CMRT_allocate_physical_stack_field(num_bytes)
    int4 num_bytes;
CMRT_cm_location_t
  CMRT_allocate_physical_heap_field(num_bytes)
    int4 num_bytes;
```

Both functions take a number of bytes as an argument, and return a **CMRT_cm_location_t** pointer to the allocated stack or heap memory region. To deallocate a memory region allocated in this way, use the following functions:

```
CMRT_deallocate_physical_stack_through(field, num_bytes)
     CMRT_cm_location_t field;
     int4 num_bytes;
CMRT_deallocate_physical_heap_field(field, num_bytes)
     CMRT_cm_location_t field;
     int4 num_bytes;
```


Copyright © 1994 Thinking Machines Corporation

These functions take a starting address and a number of bytes, and return the indicated space to free storage. (Note that deallocating a stack field implicitly deallocates all stack fields with higher stack addresses.)

If, on the other hand, you have a specific **CMRT_array_geometry_t** and want to allocate enough memory to give each element of the geometry a specific number of bytes, you should use these allocation functions:

```
CMRT_cm_location_t
  CMRT_allocate_stack_field(geometry, num_bytes)
    CMRT_array_geometry_t geometry;
    int4 num_bytes;
CMRT_cm_location_t
  CMRT_allocate_heap_field(geometry, num_bytes)
    CMRT_array_geometry_t geometry;
    int4 num_bytes;
```

Both functions take a geometry object and a number of bytes, and return a **CMRT_cm_location_t** pointer to the allocated stack or heap region. The difference between these functions and the physical ones shown above is that these functions allocated the specified number of bytes for *each element* of the array's subgrid — the **num_bytes** argument is multiplied by the array's subgrid size. Thus, the following equivalences hold:

```
CMRT_allocate_stack_field(geometry, num_bytes) ==
CMRT_allocate_physical_stack_field (num_bytes *
    geometry->machine_geometry->product_subgrid_lengths)

CMRT_allocate_heap_field(geometry, num_bytes) ==
CMRT_allocate_physical_heap_field (num_bytes *
    geometry->machine_geometry->product_subgrid_lengths)
```

To deallocate these fields, you can use the following deallocation functions:

```
CMRT_deallocate_stack_field_through
  (field, geometry, num_bytes)
    CMRT_cm_location_t field;
    CMRT_array_geometry_t geometry;
    int4 num_bytes;
CMRT_deallocate_heap_field
  (field, geometry, num_bytes)
    CMRT_cm_location_t field;
    CMRT_array_geometry_t geometry;
    int4 num_bytes;
```


Version 2.1, January 1994
Copyright © 1994 Thinking Machines Corporation

A.2.2 Node-Level Stack Operations

When you are writing a node-level routine that is to be called as part of a global program, you can also use CMRTS functions to allocate temporary stack space independently on each node. The only condition to this is that you must ensure that the allocated space is freed before your node-level routine returns.

The following routines can be used at this level:

```
CMCOM_cm_address_t CMCOM_pe_get_stack_pointer ()

CMCOM_cm_address_t
  CMCOM_pe_allocate_stack_space (nbytes)
    int4 nbytes;

CMCOM_pe_set_stack_pointer (new_sp)
    CMCOM_cm_address_t new_sp;
```

The proper (and recommended) method to do this is:

- At the start of a node-level subroutine, get the current value of the stack pointer and store it in a temporary variable:

  ```
  CMCOM_cm_address_t temp;
  temp = CMCOM_pe_get_stack_pointer();
  ```

- When you need to allocate stack space, call the allocation function:

  ```
  space = CMCOM_pe_allocate_stack_space (nbytes);
  ```

- At the end of the routine (or before any return point), free all allocated stack space by resetting the stack pointer to its original value.

  ```
  CMCOM_pe_set_stack_pointer (temp);
  ```

Important: This method is only applicable for node routines that are called directly as part of a global (PM and nodes) program. If you are running code under the global/local version of the RTS, in which each node is treated as a parallel machine in and of itself, you can make calls to the standard RTS memory allocation routines as described in Section A.2.1 above. These will work in either the global or the local parts of a global/local program.

Copyright © 1994 Thinking Machines Corporation