Appendix B

Memory Layout on the CM-5

This appendix describes the memory layout used by CM-5 C*, and explains how to access physical memory characteristics of parallel variables. This information is not ordinarily necessary for C* programming. It is provided for advanced C* users who may need the information to mix their C* code with low-level code, such as C/DPEAC, or to perform special manipulation of data through shape aliasing, as described in Section B.4.

The representation of parallel variables is very much implementation-dependent and could change in future releases of CM-5 C*. In many cases, making use of the information provided in this appendix requires a detailed understanding of how the C* memory layout mechanisms work. For the sake of reliable and portable programming, we urge you to avoid depending on these mechanisms.

Note that we describe memory layout on vector units. The discussion also applies to the nodes if you use the **`-sparc`** option to compile for execution on the nodes.

B.1 Memory Layout of Parallel Variables

Appendix A describes how the shape layout mechanism works and describes how to access specific layout information for a shape. This section reviews some of the concepts from that appendix and extends the discussion to specific memory layout issues for parallel variables.

A C* parallel variable on the CM-5 has the same amount of memory and the same address in each vector unit. In general, a parallel variable may have

Copyright © 1990–1995 Thinking Machines Corporation

more than one value in each VU. The parallel variable's shape determines the exact layout of the variable's positions on the VUs.

On each VU, the data representing a parallel variable is arranged in a *subgrid*. The subgrids on each VU are always the same size. Elements in the subgrids on some VUs may not correspond to positions of the parallel variable; these are called *garbage positions* and are explained in Appendix A.

Each subgrid can be thought of as a multidimensional array of elements. For the purposes of computation, however, it is simpler to view the subgrid as a 1-dimensional array of elements, since we're not particularly concerned with the ordering of the elements in the subgrid. The total number of positions in each subgrid is called the *subgrid size*. The function **CMC_shape_subgrid_size** returns a shape's subgrid size; see Section A.5. In the current C* implementation, the subgrid size is always a multiple of eight on the vector units; this restriction does not apply on the nodes.

Suppose we have a parallel integer declared in a shape that has 16 subgrid elements in each VU:

```
shape [positionsof(physical) * 16]S;
int:S a;
```

The representation of **a** in VU memory is such that each VU has 16 integer elements, contiguous in memory. If **a** is stored at the memory location 0x50001448 on each VU, then the VU memory representing **a** would be

```
0x50001448:        a subgrid 0
0x5000144c:        a subgrid 1
0x50001450:        a subgrid 2
 . .
 . .
 . .
0x50001484:        a subgrid 15
```

Parallel structures and arrays are represenented such that the structure members and array elements are contiguous in memory. Suppose we declare

```
struct tri { int x, y, z; };
struct tri:S b;
```

If **b** is stored at location 0x50001488, it would be stored as

```
0x50001488:        b.x subgrid 0
0x5000148c:        b.y subgrid 0
0x50001490:        b.z subgrid 0
```


Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

```
0x50001494:        b.x subgrid 1
0x50001498:        b.y subgrid 1
0x5000149c:        b.z subgrid 1
0x500014a0:        b.x subgrid 2
 . .
 . .
 . .
0x50001544:        b.z subgrid 15
```

Similarly, if we declare

```
float:S c[4];
```

and **c** is stored at location 0x50001548, it would be stored as

```
0x50001548:        c[0] subgrid 0
0x5000154c:        c[1] subgrid 0
0x50001540:        c[2] subgrid 0
0x50001544:        c[3] subgrid 0
0x50001558:        c[0] subgrid 1
 . .
 . .
 . .
0x50001644:        c[3] subgrid 15
```

The distance in memory between successive subgrid elements is called the *memory stride*. For **a** the memory stride is 4 bytes, for **b** it is 12 bytes, and for **c** it is 16 bytes. (In general, the memory stride for a parallel variable is the number of bytes that **sizeof** would return when applied to the variable.)

B.2 Pointers to Parallel Variables

In C*, the representation of a pointer to a parallel variable is different from that of a pointer to a scalar variable, since it must contain more information than a simple memory address. In particular, a pointer to a parallel variable carries three pieces of information:

- the parallel variable's memory address
- its memory stride
- its shape

Copyright © 1990–1995 Thinking Machines Corporation

A pointer to a parallel variable is simply a scalar data type that contains all of this information. (The current CM-5 C* implementation uses 16 bytes to represent a pointer to a parallel variable.)

For example, if we declare

```
int:void *p;
```

and assign (using **b** declared in the previous section)

```
p = &b.y;
```

then **p** will have

- a memory address that is the address of the first subgrid element of **b.y** (four bytes offset from the address of **b**)
- a memory stride that is 12 bytes (the distance between successive subgrid elements of **b.y**)
- a shape, which is **S**

Because a pointer to a parallel variable may point to a member of a parallel structure or an element of a parallel array, the memory stride is not necessarily the size of the element pointed to. In the example above, although **p** points to a 4-byte integer, its memory stride is 12 bytes.

B.3 Manipulating Pointers to Parallel Variables

You can use the **shapeof** intrinsic to access the shape associated with a pointer to a parallel variable. For example,

```
shapeof(*p)
```

evaluates to **S** when **p** points to **b.y**.

Two functions declared in **<csshape.h>** allow access to a pointer's memory address and stride. These functions are

```
void *CMC_pointer_mem_addr(void:void *p);
size_t CMC_pointer_mem_stride(void:void *p);
```


Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

The memory address returned by **CMC_pointer_mem_addr** is a VU instruction-space address; it is not a valid address on the partition manager. The stride returned by **CMC_pointer_mem_stride** is in bytes.

It is also possible to construct a pointer to a parallel variable, given address, stride, and shape information. The function **CMC_make_pointer** declared in **<csshape.h>** accomplishes this:

```
void:void *CMC_make_pointer(void *addr, size_t stride,
                            shape s);
```

Finally, the function **CMC_change_pointer_shape**, also declared in **<csshape.h>**, changes the shape associated with a pointer:

```
void:void *CMC_change_pointer_shape(void:void *p,
                                    shape s);
```

Note that this function could be written in terms of **CMC_make_pointer**, **CMC_pointer_mem_addr**, and **CMC_pointer_mem_stride**. It is discussed in more detail in the next section.

B.4 Shape Aliasing

Advanced C* users may want to take advantage of the primitives discussed in the previous section to allow parallel variables to be *aliased* so that the data is used as if it were in another shape. This shape aliasing can allow data to be manipulated in different shapes without actually performing any communication operations, thus increasing performance.

Accomplishing shape aliasing requires a thorough understanding of how C* lays out parallel variables in memory. Appendix A and Section B.1 should both be consulted before attempting it. In particular, shape aliasing where garbage positions are used in a shape's layout can be tricky. Careful use of **allocate_detailed_shape** may be required in any case.

B.4.1 Examples

Two examples are provided to demonstrate shape aliasing.

Copyright © 1990–1995 Thinking Machines Corporation

The first example, shown below, demonstrates how data in a 2-dimensional shape can be viewed as data in a 1-dimensional shape. This technique is used to accomplish a **rank** operation on a 2-dimensional data set, treating it as if it were a 1-dimensional data set. It uses **CMC_change_pointer_shape** to accomplish shape aliasing.

```
#include <cscomm.h>
#include <assert.h>
#include <csshape.h>
#include <stdlib.h>
#include <stdio.h>

void check_shape_padding(shape s);

main()
{
     shape [128][positionsof(physical)]S1;
     shape [128 * positionsof(physical)]S2;

     int:S1 a, b;
     int:S2 *pa, *pb;

     int i, j;

     /*
      * Our shape aliasing in this example depends upon S1 and S2 having
      * identical subgrid sizes and upon there being no padding used in
      * the layout of each shape. The C* layout mechanism should guarantee
      * that for the shapes declared above, but the code below verifies
      * it.
      */
     assert(CMC_shape_subgrid_size(S1) == CMC_shape_subgrid_size(S2));
     check_shape_padding(S1);
     check_shape_padding(S2);

     /*
      * Initialize "a" with random values.
      */
     with(S1)
     {
          a = prand() % 100;
     }

     /*
      * Alias "pa" and "pb" so that they point to "a" and "b" but have
      * S2 as their shapes.
```


Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

```
     */
    pa = CMC_change_pointer_shape(&a, S2);
    pb = CMC_change_pointer_shape(&b, S2);

    /*
     * Rank the random values in "a" as if they were in a one-dimensional
     * shape.
     */
    with(S2)
    {
          *pb = rank(*pa, 0, CMC_upward, CMC_none, CMC_no_field);
    }

    /*
     * Print some of the results of the rank operation.
     */
    for(i = 0; i < 4; ++i)
    {
          for(j = 0; j < 4; ++j)
          {
                printf("[%d][%d] value:  %3d rank: %d\n",
                      i, j, [i][j]a, [i][j]b);
          }
    }
}

void check_shape_padding(shape s)
{
    int axis, physical_dim, axis_mask;

    /*
     * For each axis, ensure that the product of the physical grid
     * dimension and the subgrid dimension is equal to the shape's
     * dimension.  If this is not so, then the shape has positions that
     * are used for padding, and our shape aliasing example will not
     * work.
     */
    for(axis = 0; axis < rankof(s); ++axis)
    {
          /*
           * Calculate the dimension of the physical grid by counting the
           * number of bits in the physical axis mask.
           */
          axis_mask = CMC_shape_physical_axis_mask(s, axis);
          physical_dim = 1;
          while(axis_mask)
          {
                if(axis_mask & 0x1) physical_dim <<= 1;
                axis_mask >>= 1;
```


Copyright © 1990–1995 Thinking Machines Corporation

```
          }

          /*
           * Check that there is no padding for the axis.
          */
          assert(dimof(s, axis) ==
                CMC_shape_subgrid_axis_length(s, axis) * physical_dim);
     }
}
```

The second example demonstrates shape aliasing that allows a parallel array of integers to be viewed as a single 2-dimensional parallel integer. This requires both a sophisticated use of **`allocate_detailed_shape`** and manipulation of pointer strides.

```
#include <assert.h>
#include <csshape.h>
#include <stdio.h>

shape allocate_alias_shape(shape orig_shape, int new_axis_len);

main()
{
     shape [128 * positionsof(physical)]S1;
     int:S1 a[10], b;

     shape S2;
     int:void *pa;

     int i, j;

     with(S1)
     {
          for(i = 0; i < 10; ++i)
          {
               a[i] = i * pcoord(0);
          }
     }

     S2 = allocate_alias_shape(S1, 10);
     pa = CMC_make_pointer(CMC_pointer_mem_addr(a), sizeof(int), S2);

     with(S2)
     {
```


Copyright © 1990–1995 Thinking Machines Corporation

```
          /*
           * The data in the array "a" can now be treated as a single
           * two-dimensional parallel variable by dereferencing "pa".
           *
           * We check that the values pointed to by "pa" are correct, and
           * then add 1 to all of them.
           */
          assert(&= (*pa == pcoord(0) * pcoord(1)));

          *pa += 1;
    }

    with(S1)
    {
          /*
           * Similarly, we can still view the data as a 1-d parallel
           * array:
           */
          for(i = 0; i < 10; ++i)
          {
                printf("a[%d]:", i);
                for(j = 0; j < 4; ++j)
                {
                      printf(" %3d", [j]a[i]);
                }
                printf(" ...\n");
          }
    }
}


/*
 * allocate_alias_shape() creates a 2-d shape given a 1-d shape.
 * The second dimension is used to alias a parallel array to be
 * a simple type in this shape.
 */
shape allocate_alias_shape(shape orig_shape, int new_axis_len)
{
     shape new_shape;
     unsigned long extents[2];
     int subgrid_lengths[2];
     int subgrid_sequences[2];

     /*
      * The scheme would be somewhat more complicated for multidimensional
      * shapes.
      */
     assert(rankof(orig_shape) == 1);
```


Copyright © 1990–1995 Thinking Machines Corporation

```
    /*
     * The new shape will be 2-d, with axis 0 having the original
     * dimension and axis 1 having a dimension that is equal to the
     * dimension of the array we are aliasing.
     */
    extents[0] = dimof(orig_shape, 0);
    extents[1] = new_axis_len;

    /*
     * We use the original subgrid length for axis 0, and the array
     * dimension for the subgrid length of axis 1.  This ensures that
     * axis 1 has no off-VU component.
     */
    subgrid_lengths[0] = CMC_shape_subgrid_axis_length(orig_shape, 0);
    subgrid_lengths[1] = new_axis_len;

    /*
     * These subgrid sequences ensure that axis 1 has the lowest stride,
     * necessary to properly alias the array.
     */
    subgrid_sequences[0] = 1;
    subgrid_sequences[1] = 0;

    /*
     * The weights, axis orderings, and physical masks are not specified
     * because the other information sufficiently constrains our new
     * shape.
     */
    new_shape = allocate_detailed_shape(&new_shape,
                                        2,       /* rank */
                                        extents,
                                        NULL,    /* weights */
                                        NULL,   /* axis orderings */
                                        NULL,   /* physical masks */
                                        subgrid_lengths,
                                        subgrid_sequences);

    /*
     * Verify that the subgrid increments are correct.
     */
    assert(CMC_shape_subgrid_axis_increment(new_shape, 0) ==
                new_axis_len);
    assert(CMC_shape_subgrid_axis_increment(new_shape, 1) == 1);

    return new_shape;
}
```


Copyright © 1990–1995 Thinking Machines Corporation