Appendix B

Non-RTS Parallel Memory Allocation

For some special-purpose applications, it is necessary to allocate parallel CM memory on a CM-5 with VUs without using the standard memory allocation and deallocation routines provided by the CM Run-Time System (CMRTS). An example of this is CMMD applications that include DPEAC subroutines.

Methods for allocating parallel memory without use of the CMRTS are described in the sections below.

B.1 The Parallel Stack and Parallel Heap

Parallel VU memory can be 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. Initially, no pages are allocated. When you allocate new space in these regions, it is as a stripe of memory across the physical memory of the four VUs. Thus, allocating a page of stack or heap actually allocates 4 pages of physical memory.

B.2 VU Regions

The stack and heap are each actually mapped as seven regions in memory in order to make the memory accessible in parallel or by individual VUs for the purpose of data or instruction access. (See the *DPEAC Reference Manual* for a description of VU space virtual addressing.)

By convention, pages of parallel memory are referenced by their *All-DP, Instruction Space* address. This is the address in the region of VU memory that causes all four VUs to execute a DPEAC instruction simultaneously. CMMD routines that take parallel memory addresses take such addresses (just as RTS routines do).

Whether coding in assembly (DPEAC) or C, you need to include this header file:

```
#include <cmsys/dp.h>
```

This header file defines a number of constants that are essential in navigating the VU memory regions. For example, the base of the parallel stack (in All-DP Instruction Space) is given by the symbol **`DPV_STACK_INST_PORT_ALL`** (**`0x50000000`**), and the base of the parallel heap region is given by the symbol **`DPV_HEAP_INST_PORT_ALL`** (**`0x70000000`**).

You can construct an address within these regions by adding a byte offset to these base addresses.

Important: Before you can access a stack or heap word, the memory region must have been expanded to include the address (that is, you must first allocate the memory before you can legally access it).

B.3 Expanding the Stack or Heap

When you want to expand the stack or heap, you make a CMOST system call to manipulate the pointer of the appropriate memory region. You can do this either from the partition manager or from a processing node. If you do this from a node, only one processing node must (and should) make the allocation call.

To access the appropriate CMOST routines, include the header file:

```
#include <cmsys/cm_memory.h>
```

The memory pointer system calls from the partition manager are:

```
CM_memaddr_t
  CM_set_dp_stack_ptr (CM_memaddr_t new_limit)

CM_memaddr_t
  CM_set_dp_heap_ptr (CM_memaddr_t new_limit)

CM_memaddr_t  CM_get_dp_stack_ptr ()

CM_memaddr_t  CM_get_dp_heap_ptr ()
```

The equivalent calls from the node are:

```
CM_memaddr_t
  CMPE_set_dp_stack_ptr (CM_memaddr_t new_limit)

CM_memaddr_t
  CMPE_set_dp_heap_ptr (CM_memaddr_t new_limit)

CM_memaddr_t CMPE_get_dp_stack_ptr ()

CM_memaddr_t CMPE_get_dp_heap_ptr ()
```

All of these routines return a **CM_memaddr_t** value, which is an *All-VU, Instruction Space* address, representing the current position of the memory pointer (in the case of the **set** routines, this is the value of the pointer after you have modified it). The value of the pointer is always one more than the highest allocated address in the memory region.

You cannot access allocated memory using the **CM_memaddr_t** values returned from these system calls, because they are in All-VU instruction space. You must translate this value into a *Single-VU Data Space* pointer, as described in Section B.4 below.

To use the **set** system calls, you pass in the highest address that you want to have allocated. The pointer value the call returns will always be greater than this value (unless there is insufficient memory remaining, in which case zero is returned), but it may not be exactly one more than the address you passed in.

Important: Don't make a "copy" of the stack or heap pointer and expect the copy to remain valid. Stack and heap memory can be allocated for other reasons than explicit system calls from your program. Thus, the stack and heap pointers can change without warning. You should always use the current value returned by the system calls mentioned above when determining the current size of the stack or heap.

If you want to deallocate parallel memory (in other words, shrink the stack or heap regions), call the appropriate **set** function with the new lower limit.

Note: CMOST currently does not allow the regions to shrink, and thus the above call will have no effect, and the current limit will be returned. Nevertheless, it is sensible to include deallocation calls, for compatibility with later software versions.

B.4 Translating Stack and Heap Addresses

You can change **CM_memaddr_t** values into valid data space addresses using the following C macro, which is defined in **cmsys/dp.h**:

```
data_address = TOGGLE_DPV_SPACE(instruction_address);
```

Note that the returned data space address is still an *All-VU* address. It cannot be used to read from memory, and if used to store to memory, the stored value will be written to all four VUs (broadcast).

You can change the data space address to point to a single VU by using one of the following macros:

```
VU_0_address = CHANGE_DP(data_address, DP_0);
VU_1_address = CHANGE_DP(data_address, DP_1);
VU_2_address = CHANGE_DP(data_address, DP_2);
VU_3_address = CHANGE_DP(data_address, DP_3);
```

The resulting addresses are pointers to single word/doubleword in stack or heap memory and can be used, for example, as a C pointer value to read or write memory values.

Note: Parallel memory, accessed by the node processor is always mapped with caching *disabled*. Thus, access to words/doublewords in the above fashion will be 2 to 3 times slower than normal cached accesses.

Also, all attempts to read/write parallel memory using pointers that are not word aligned will result in memory faults.

B.5 Using pmalloc to Allocate Memory

There is a sample C routine, **`pmalloc`**. An example of its definition and use can be found in the file:

```
/usr/examples/cmmd/hostless/c/mp_parallel_perf/parray.c
```