Chapter 9

# Memory Management

Although it is usually a less important factor than parallel computation or communication in determining program speed, memory management is still one of the major considerations in evaluating the performance of a CM Fortran program. As the number and size of the arrays used in your program increases, you may find yourself running out of PE memory, even when the sizes of your arrays are such that they would seem to fit nicely into memory.

This chapter describes the things you can do to determine how memory is being used in your program, and how you can make the best use of memory in constructing and running your program.

## 9.1 CM Memory Management

### 9.1.1 Scalar and Parallel Memory

"Parallel memory" is a region of memory allocated in all four VU memory banks on each PN, across the entire CM. The region in each memory bank has the same size and begins at the same address, so parallel memory can be regarded as a "stripe" across the memory of the entire CM.

"Scalar memory" is any memory region not dedicated to storing parallel data. This includes the scalar memory bank of the PM, as well as the portions of the VU memory banks on each PN that are not used for parallel data structures.


*Copyright © 1994 Thinking Machines Corporation*

### 9.1.2 The Parallel Stack and Heap

The CM uses two different kinds of parallel memory:

- *Stack memory*: Temporary memory allocated in a fixed order; arrays allocated on the stack must be freed (deallocated) in reverse order
- *Heap memory*: Relatively permanent memory, allocated and deallocated arbitrarily

These memory regions work like the stack and heap on ordinary non-parallel computers, except that the heap never gets compacted, so it can become fragmented if arrays are not deallocated over time. Figure 26 shows the typical layout of the parallel heap and stack in CM memory.

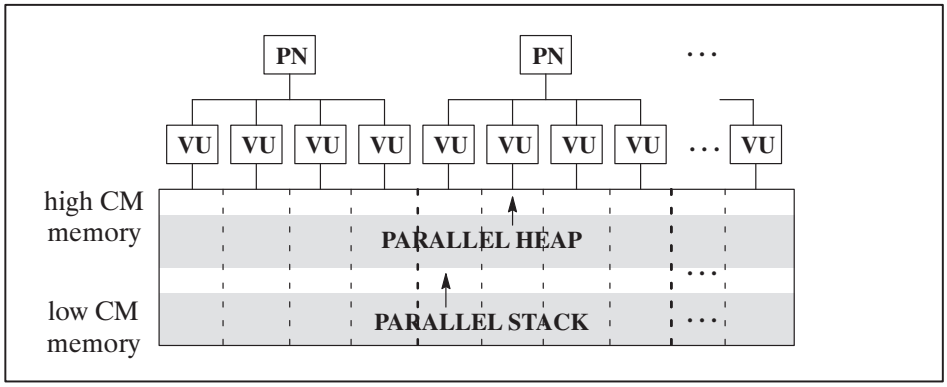


**Figure 26. Parallel stack and heap memory on CM-5 with VUs.**

The memory available per PN is either 32 or 128 Mbytes, depending on which memory option is installed. On CM-5s with VUs, this is divided among the VUs, giving either 8 or 32 Mbytes per VU. On non-VU CMs, all memory is under the direct control of the PN.

### 9.1.3 CM Memory Mapping

On each processing node, the basic mapping of scalar and parallel memory looks something like the diagram in Figure 27. On the left is the PN microprocessor's view of the available memory, and on the right is the actual distribution of parallel and scalar memory regions in the four physical



Copyright © 1994 Thinking Machines Corporation

memory banks of the VUs. Dark-shaded regions represent scalar memory, light-shaded regions represent parallel memory.

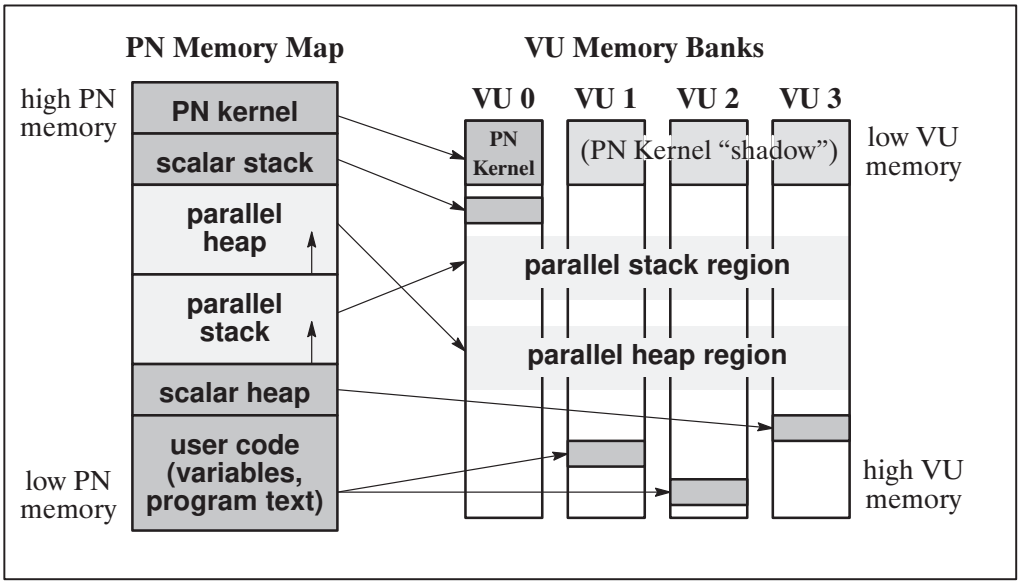


**Figure 27. Memory mapping of scalar and parallel memory regions.**

Each parallel memory region is always a stripe across all four VUs. Scalar memory regions, however, can be located at arbitrary locations in any of the four VU memory banks. The CM operating system is responsible for assigning pages of VU memory to scalar and parallel memory. For parallel memory, the OS must find four aligned regions of VU memory, one in each bank. For scalar memory, the OS can assign arbitrary memory pages.

The partition manager runs a complete UNIX operating system. Each processing node, however, runs a subset of OS code known as the *PN kernel*. The PN kernel occupies about 0.5 Mbytes of memory on each PN, and is located in the low memory of VU 0. Notice that because the OS requires aligned memory regions in all four VU banks to create parallel memory, the PN kernel effectively shadows the other VU memory banks, so that while it occupies only 0.5 Mbytes of memory, it prevents 2 Mbytes in total from being used for parallel memory. (This is usually not a problem, however, as the OS can still assign the "shadowed" memory regions as pages of scalar memory.)


Copyright © 1994 Thinking Machines Corporation

### 9.1.4 CM Fortran Array Types

CM arrays can be explicitly declared by a program, or can be implicitly generated as temporaries by the compiler. User-declared arrays are of four types:

- *Ordinary local arrays* (without SAVE or DATA attribute). These are allocated on the stack on entry to a program unit, deallocated on exit.
- *Permanent local arrays* with the SAVE or DATA attribute. These are allocated on the heap on first entry to a program unit, and are never deallocated.
- *Dynamically allocated arrays* (created by **CMF_ALLOCATE_..._ARRAY** or **ALLOCATE**). These are explicitly allocated and deallocated on the heap by function calls.
- *Common block arrays*. These are allocated on the heap when the array is first used (if ever), and persist indefinitely. If a common block array is initialized in a **BLOCK DATA** unit, the values are copied from the PM to the nodes on the first use of the array.

For example, here's a quick way to see how CM memory is allocated for arrays on your system:

```
      PROGRAM STACKHEAP
      CALL MAIN()
      END
      SUBROUTINE MAIN()
C-----ALLOCATED ON STACK
      INTEGER, DIMENSION(100) :: A,B,C
C-----ALLOCATED ON HEAP
      INTEGER, DIMENSION(100), SAVE :: D,E,F
C-----FORCE THEM ALL TO BE CM ARRAYS
      A=0
      B=0
      C=0
      D=0
      E=0
      F=0
C-----DISPLAY GEOMETRY INFO
C-----(ARRAYS ARE ALLOCATED IN REVERSE ORDER, SO:)
      CALL CMF_DESCRIBE_ARRAY(C)
      CALL CMF_DESCRIBE_ARRAY(B)
      CALL CMF_DESCRIBE_ARRAY(A)
```


Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

```
      CALL CMF_DESCRIBE_ARRAY(F)
      CALL CMF_DESCRIBE_ARRAY(E)
      CALL CMF_DESCRIBE_ARRAY(D)
      END
```

This program produces a large amount of **CMF_DESCRIBE_ARRAY** output, but the part we're interested in is the **cm_location** of each array descriptor. Here's a sample run of this program, with the important lines selected:

```
% a.out | grep "cm_location"
  cm_location          :  1342178408
  cm_location          :  1342178440
  cm_location          :  1342178472
  cm_location          :  1879048600
  cm_location          :  1879048632
  cm_location          :  1879048664
```

As you can see from the increasing values of the addresses, the parallel stack and heap grow upwards in CM memory. (The **CMF_DESCRIBE_ARRAY** calls are in inverted order because CM Fortran arrays are allocated in reverse order within a single declaration; for example, try compiling the above program with the **-S** switch, and examine the **.s** file that is produced.)

## 9.2 Compiler Temporary Arrays

The CM Fortran compiler generates two kinds of internal temporary arrays:

- *Communication-computation (comm-comp) temporaries*: These arrays hold temporary results generated in expressions, and are released when the calculation of the expression is completed.
- *Common subexpression (CSE) temporaries*: These arrays hold values of common subexpressions between the first time they are required and the last time they are needed.

### 9.2.1 Communication-Computation Temporary Arrays

In separating communication and computation implied by array expressions, the compiler generates comm-comp arrays to hold intermediate array results. These temporary arrays are effectively identical to those you might


Copyright © 1994 Thinking Machines Corporation

use yourself in dividing an expression into separate communication and computation statements. For example:

```
A=B+CSHIFT(C*D,DIM=1,SHIFT=1) ==>
                          T1 = C*D
                          T2 = CSHIFT(T1,DIM=1,SHIFT=1)
                          A = B + T2
```

Comm-comp temporaries are also used for element numbering (grid coordinate generation, etc.) used by array constructors, triplet subscripts, and **FORALL** assignments:

```
F=[1:100] ==> CALL MY_GRID_COORD(T3,1)
              F=T3+1

FORALL(I=2:99) G(I)=I ==>     CALL MY_GRID_COORD(T4,1)
                              WHERE(1<=T4<=98) G=T4+1

H(::K)=X   ==>CALL MY_GRID_COORD(T5,1)
                    WHERE(MOD(T5-1,K).EQ.0) H=X
```

In evaluating a given expression, many comm-comp temporaries may be needed, but each is needed only for a short time to hold a result. As a result, comm-comp temporaries have very short life spans. Comm-comp temporaries are allocated at first use and deallocated right after their last use.

For example, here's the **CSHIFT** example above expanded to show allocation and deallocation of temporaries:

```
A=B+CSHIFT(C*D,DIM=1,SHIFT=1) ==>
                          ALLOCATE(T1)
                          T1 = C*D
                          ALLOCATE(T2)
                          T2 = CSHIFT(T1,DIM=1,SHIFT=1)
                          DEALLOCATE(T1)
                          A = B + T2
                          DEALLOCATE(T2)
```

The compiler may *collapse* two or more temporaries into one; the collapsed temporary is reused as many times as needed before being deallocated. This occurs only when:

- the temporaries involved have the same shape
- the temporaries are used in non-overlapping regions of code
- the temporaries are used within the same *basic block* of code



Copyright © 1994 Thinking Machines Corporation

A *basic block* is a segment of CM Fortran statements bounded by control-flow statements. Specifically, a basic block begins at the entry point of a routine or at the first non-control-flow statement after another block. It ends at or before the first line that involves a transfer of control. Single flow-control statements (such as the **DO** or **END DO** of a loop) also form basic blocks. For example, here's how a typical region of code is divided into basic blocks:

```
C-----BASIC BLOCK 1-----------------
      A=B+C
      WHERE(M) X=Y
      D=CSHIFT(E*F, DIM=1, SHIFT=-1)
      COUNT=COUNT+1
C-----BASIC BLOCK 2-----------------
      DO I=1,10
C-----BASIC BLOCK 3-----------------
        A(I,:)=B(I,:)
C-----BASIC BLOCK 4-----------------
      END DO
C-----BASIC BLOCK 5-----------------
      IF (COUNT>10) GOTO 99
C-----BASIC BLOCK 6-----------------
 99   CONTINUE
      ...
```

**Note:** Be sure not to confuse *basic blocks* with the PE code blocks described previously. A basic block is a lexical unit of code used in determining whether temporaries can be collapsed together. A PE code block, by contrast, is a region of pure parallel computation involving no control-flow statements at all.

Here is an example of the merging of temporaries. The two statements

```
A=B+CSHIFT(C*D,DIM=1,SHIFT=1)
E=F+CSHIFT(G*H,DIM=1,SHIFT=1)
```


Copyright © 1994 Thinking Machines Corporation

are compiled as shown below, without and with collapsing of temporaries:

**Without Collapsing:**

```
ALLOCATE(T1)
T1 = C*D
ALLOCATE(T2)
T2 = CSHIFT(T1,DIM=1,SHIFT=1)
DEALLOCATE(T1)
A = B + T2
DEALLOCATE(T2)
ALLOCATE(T3)
T3 = G*H
ALLOCATE(T4)
T4 = CSHIFT(T3,DIM=1,SHIFT=1)
DEALLOCATE(T3)
E = F + T4
DEALLOCATE(T4)
```

**With Collapsing:**

```
ALLOCATE(T1)
T1 = C*D
ALLOCATE(T2)
T2=CSHIFT(T1,DIM=1,SHIFT=1)

A = B + T2


T1 = G*H

T2=CSHIFT(T1,DIM=1,SHIFT=1)
DEALLOCATE(T1)
E = F + T2
DEALLOCATE(T2)
```

In this example, collapsing of temporaries saves the allocation and deallocation of two temporary arrays (**`T3`** and **`T4`**). This does not reduce the amount of memory used, but makes more effective use of already-allocated space.

### 9.2.2 Common Subexpression (CSE) Temporary Arrays

The compiler identifies common subexpressions (CSEs) in array computations and assigns them to temporary arrays to save computation time. For example, the two occurrences of the expression (**`B+C`**) in the following code fragment can be computed once and used twice:

```
A = B + C
D = B + C + 2
```

Detection of CSEs depends on the level of opimization that is enabled.

In the local optimization step of the compiler (which is always enabled), CSEs are recognized only within basic blocks. In addition, CSEs that occur within the same loop, the same PE code block, and the same basic block are assigned to registers, not memory, which increases speed and efficiency.

CSEs in different code blocks, but still in the same basic block, are also recognized and evaluated once per basic block. For example, here is an example of CSEs across code blocks:


Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

```
INTEGER X,A(100),B(100),C(100),D(100)
A = B + C     (parallel computation)
X = X + 2     (scalar computation, breaks code blocks)
D = B + C + X (parallel computation)
```

In the global optimization step (enabled by the **-O** compiler switch), CSEs can be recognized in different basic blocks. In this case, a heap temporary is allocated when the expression is first used, and deallocated at the end of the program unit in which the CSEs appear. For example, the above four-line code example compiles into a single PE code block with the **-O** switch.

Other examples of CSE temporary use are

- Calling an array-valued function:

  ```
  B = MATID(n)
  C = B * MATID(n)
  ```

- Passing a CM array expression to a subprogram:

  ```
  CALL FOO(3*A-1)
  ```

  This generates a temporary array of the same shape as A, using a canonical layout, as in

  ```
  T1=3*A-1
  CALL FOO(T1)
  ```

**WHERE** masks are a special case of CSE temporary. The compiler generates a CSE temporary to store *every* **WHERE** mask, whether it is an array expression or simply a logical array. The rationale for this is that a **WHERE** block might alter the values of data used in the arrays specified as the mask; the mask values on entry to the **WHERE** block are the valid ones, so these must be saved as a temporary. The CSE temporary for a **WHERE** block is heap allocated on entry to the block, and deallocated on exit from the program unit containing the block. **Note:** Since the **ELSEWHERE** mask can be computed by inverting the **WHERE** mask, this is not stored as a temporary.


Copyright © 1994 Thinking Machines Corporation

## 9.3 Running Out of Memory

The basic parallel memory-management routines of the CM are located at the CMRTS level of the software. Thus, when you run out of PE memory on the CM, you'll get a CMRTS-level error like:

```
*** RTS-FATAL-CMRTS:
*** Fatal RTS Error ***:
CMCOM_allocate_stack_space: couldn't allocate
10000000 bytes on VU stack (SP=0x50000468)
```

Out-of-memory errors arise in both visible and invisible ways:

- You're allocating arrays with subgrids that are too large to fit in the available memory on the PEs.
- You're implicitly creating temporary arrays that take up more total memory space than is available.

As an example of the visible kind of memory use, consider the following short program:

```
      PROGRAM MEMOUT
      INTEGER S,NP,CMF_NUMBER_OF_PROCESSORS
      NP=CMF_NUMBER_OF_PROCESSORS()
      PRINT *,'NUMBER OF PES: ',NP
      PRINT *,'ARRAY SUBGRID SIZE (BYTES)?'
      READ *,S
C---- CONVERT SIZE FROM BYTES TO 4-BYTE INTEGERS
      S=S/4
      CALL MAIN(S,NP)
      END

      SUBROUTINE MAIN(X,Y)
      INTEGER X,Y,A(X,Y)
CMF$  LAYOUT A(:SERIAL,:NEWS)
      A=0
      END

% cmf -nopad memout.fcm
% a.out
 number of PEs:           16
 array subgrid size (bytes)?
100000000

*** RTS-FATAL-CMRTS:
```



Copyright © 1994 Thinking Machines Corporation

```
*** Fatal RTS Error ***:
CMCOM_allocate_stack_space: couldn't allocate
 100000000 bytes on VU stack (SP=0x50000468)
Traceback follows:
   pc = 0x46c74      _cmrt_allocate_stack_memory
   pc = 0x44870      _CMRT_allocate_physical_stack_field
   pc = 0x4491c      _CMRT_allocate_stack_field
   pc = 0x23ec       _MAIN_
   pc = 0x1be9c      _main
Abort
```

But memory use can be invisible, as when the compiler needs to create temporary arrays used in computation or communication. These temporary arrays can take up just as much memory as your declared arrays, and if enough of them are created, they can exhaust the available memory even if your declared arrays are well within the available memory size. For example, here's a program that generates a number of temporary arrays:

```
      PROGRAM MEMOUT2
      INTEGER S,NP,CMF_NUMBER_OF_PROCESSORS
      NP=CMF_NUMBER_OF_PROCESSORS()
      PRINT *,'NUMBER OF PES: ',NP
      PRINT *,'ARRAY SUBGRID SIZE (BYTES)?'
      READ *,S
C---- CONVERT SIZE FROM BYTES TO 4-BYTE INTEGERS
      S=S/4
      CALL MAIN(S,NP)
      END

      SUBROUTINE MAIN(X,Y)
      INTEGER X,Y,A(X,Y)
CMF$  LAYOUT A(:SERIAL,:NEWS)
      CALL CMF_RANDOM(A,10)
      PRINT *,'ARRAY IS ALLOCATED'
      A = A + CSHIFT(A+A,DIM=1,SHIFT=1)
     $      + CSHIFT(-A,DIM=2,SHIFT=1)
     $      + CSHIFT(A*A,DIM=1,SHIFT=-1)
     $      + CSHIFT(100/A,DIM=2,SHIFT=-1)
      PRINT *,'DONE'
      END
```

Only one declared array is used, but each of the **CSHIFT** calls in the parallel computation also generates a temporary array of equal size. When run with


Copyright © 1994 Thinking Machines Corporation

a sufficiently large subgrid size, the declared array can be allocated, but the temporary arrays cause the program to run out of memory:

```
% cmf -nopad memout2.fcm
% a.out
 NUMBER OF PES:           16
 ARRAY SUBGRID SIZE (BYTES)?
1000000
 ARRAY IS ALLOCATED

*** RTS-FATAL-CMRTS:
*** Fatal RTS Error ***:
CMCOM_allocate_heap_memory:
  unable to allocate 1000000 bytes on VU heap
. . .
```

By saving the assembly-code version of the program, you can search for the references to allocation and deallocation of arrays:

```
% cmf -nopad -S -c memout2.fcm
% egrep '^\!|allocate_' memout2.s
!      PROGRAM MEMOUT2
!      NP=CMF_NUMBER_OF_PROCESSORS()
!      PRINT *,'NUMBER OF PES: ',NP
!      PRINT *,'ARRAY SUBGRID SIZE (BYTES)?'
!      READ *,S
!      S=S/4
!      CALL MAIN(S,NP)
!      END
!
!      SUBROUTINE MAIN(X,Y)
    call    _CMRT_allocate_stack_field, 2
!      CALL CMF_RANDOM(A,10)
!      PRINT *,'ARRAY IS ALLOCATED'
    call    _CMRT_allocate_heap_field, 2
!      A = A + CSHIFT(A+A,DIM=1,SHIFT=1)
      $      + CSHIFT(-A,DIM=2,SHIFT=1)
    call    _CMRT_allocate_heap_field, 2
    call    _CMRT_allocate_heap_field, 2
    call    _CMRT_allocate_heap_field, 2
    call    _CMRT_allocate_heap_field, 2
    call    _CMFRT_deallocate_heap_field, 3
    call    _CMFRT_deallocate_heap_field, 3
    call    _CMFRT_deallocate_heap_field, 3
    call    _CMFRT_deallocate_heap_field, 3
```


Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

```
      call    _CMFRT_deallocate_heap_field, 3
!       PRINT *,'DONE'
!       END
      call    _CMRT_deallocate_stack_field_through, 3
```

Here we can see five calls to **CMRT_allocate_heap_field** and its companion function **CMRT_deallocate_heap_field**, indicating the creation of five temporary arrays. (One is for the call to the utility function **CMF_RANDOM**, and the other four are for the evaluation of the **CSHIFT** expression.) Temporary arrays are generally allocated on the parallel heap of the CM. Declared arrays, such as A in the above program, are allocated on the parallel stack.

Notice that, depending on the way a program's statements are laid out, the **CMRT_allocate_heap_field** and **CMRT_deallocate_heap_field** calls may be interleaved, so that temporary heap space is allocated and then freed before new temporary arrays are created. In this case, however, all five temporaries must be created at the same time, for the four-**CSHIFT** source line.

## 9.4 Dealing with Memory Errors

When you get an out-of-memory error in a program:

### 9.4.1 Find Out Where You Ran Out

- Check the traceback to see what function or routine the program failed in.
- Single-step through that routine with Prism. Use the **Where** command to display the program's routine call stack.
- For a detailed analysis of memory usage, include the following subroutine in your program, and make calls to it within the offending subroutine:

```
SUBROUTINE MEMORY()
INCLUDE 'CMF_defs.h'
PRINT *,CMF_AVAILABLE_MEMORY()
END
```


Copyright © 1994 Thinking Machines Corporation

### 9.4.2 Check the Size of Your Arrays

For example, consider the following array declaration:

```
      DOUBLE COMPLEX A(512, 512, 128)
CMF$  LAYOUT A(:SERIAL, :SERIAL, :NEWS)
```

On a 32-node (128-VU) CM partition, this array is allocated with a subgrid of 512 x 512 x 8 = $2^{21}$ elements. (The extra factor of 8 in the subgrid elements is the result of vector length padding.) Since each element is 16 bytes in length, this array would occupy $2^{32}$ bytes (32 Mbytes) of memory per PE. Even if your CM has the 128 Mbytes memory-per-node option, this declaration would try to use all of the memory on the CM for a single array!

When laying out your arrays, then,

- Consider the effects of garbage element and vector-length padding; these can invisibly increase the storage size of your array.
- Use the function **`CMF_describe_array`** to check the actual layout of an array, to see whether it is really occupying the memory you expect.
- Consider the size of your partition, and whether you may have to run your program on fewer nodes (and thus less memory).

### 9.4.3 Try to Reduce Explicit Use of Memory

- Use fewer arrays and more complex array expressions.
- Revise your program to use a less memory-intensive algorithm.
- Try to reuse arrays (or parts of arrays) for multiple purposes.
- Consider using the aliasing functions to declare the same memory space for arrays of different layouts and purposes.
- Split up programs units so fewer arrays are allocated at once.
- Use scratch space on an parallel I/O device (a Scalable Disk Array or DataVault) to temporarily store data.
- If all else fails, try to run your program on a larger partition of processing nodes, which will give you more memory to work in.


Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

### 9.4.4 Reduce Use of Temporary Arrays

Depending on how your code is written, the compiler may generate hundreds of temporary arrays, or none at all. When you are working with very large arrays, it's obviously to your benefit to keep the number of temporaries under control. Here are some tips for avoiding the creation of temporaries where you don't want them:

- Avoid writing complicated array expressions that involve many array functions or changes of array layout; these generate comm-comp temporaries. For example, instead of this:

```
 P = CSHIFT(X,DIM=1,SHIFT=1)
$  + CSHIFT(X,DIM=2,SHIFT=1)
$  + CSHIFT(X,DIM=1,SHIFT=-1)
$  + CSHIFT(X,DIM=2,SHIFT=-1)
```

  which allocates and deallocates four temps, write this:

```
 P = CSHIFT(X,DIM=1,SHIFT=1)
$  + CSHIFT(X,DIM=2,SHIFT=1)
 P = P + CSHIFT(X,DIM=1,SHIFT=-1)
$  + CSHIFT(X,DIM=2,SHIFT=-1)
```

  which allocates and reuses two temporaries.

- Most communication functions require their source and destination to be distinct arrays. If you use the same array as both an argument and the result of a communication operation, a comm-comp temporary will be generated to hold the communication result until it can be copied into the destination array.

  For example,

```
X=CSHIFT(X,DIM=1,SHIFT=1) ==>
                    allocate(t1)
                    t1 = CSHIFT(X,DIM=1,SHIFT=1)
                    X=t1
                    deallocate(t1)
```

  Instead, use an explicitly defined array to hold the result:

```
Y=CSHIFT(X,DIM=1,SHIFT=1)
X=Y
```


Copyright © 1994 Thinking Machines Corporation

While this doesn't save you memory, it does make the use of the temporary array explicit, and thus makes it easier for you to track down memory usage problems.

You may also be able to use a single temporary array in multiple cases that the compiler on its own would not recognize.



Copyright © 1994 Thinking Machines Corporation

## 9.5 CMOST and Memory Allocation

Under the CMOST operating system, both processor time and memory are shared.

On the first stack and heap memory allocation calls in a program, CMOST allocates the memory as a chunk of fixed size, usually more than you ask for. Subsequent memory allocation requests can simply make use of the excess memory allocated, rather than having to ask the OS for more. Only when not enough space in the chunk is left to satisfy the request does the compiler resort to an OS call to allocate another chunk.

If deallocation of array space frees up an entire chunk of stack memory or exceeds a certain count of deallocations, the OS reclaims any chunk(s) made available thereby. While CMOST could reclaim heap memory in this fashion, it does not do so at present.

The consequences of this are

- That heap memory allocated by your program can only increase, and arbitrary deallocations of it can cause the heap region to become fragmented. Remember that this also holds true for other users on your system.
- The amount of CM memory available to your program varies over time, based on other users' needs.
- An excessive amount of execution time may be spent in memory management if the stack/heap quantum (the size of the chunks allocated/deallocated by the OS) is inappropriate for your program.


Copyright © 1994 Thinking Machines Corporation

Depending on your system administrator, you may be able to use the following environment variables to alter the memory management parameters for the execution of your program:

- **RTS_MIN_HEAP/STACK_MEMORY** — Minimum memory (in Kbytes per PE) with which your program runs. The default for this is 0. Note that abuse of this setting can cause severe memory consumption, preventing other CM jobs from executing.

- **RTS_ENABLE_STACK_DEALLOCATION** — This flag controls whether stack space is deallocated. The default value is 1, indicating that stack deallocation is to be used. Setting this variable to 0 will disable stack deallocation.

- **RTS_PM_STACK_GROWTH_QUANTUM** — Chunk size (in Kbytes per PE) by which the OS increases your memory allocation. Default is 64.

  There is no corresponding heap variable — heap space always has a quantum size of 4 Kbytes (1 page) per PE.

- **RTS_PM_STACK_SHRINK_QUANTUM** — Chunk size (in Kbytes per PE) by which the OS decreases your memory allocation. Default is 64.


Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation