Chapter 4

Program Development Tools

The Connection Machine system provides the integrated Prism environment to assist with program development. See the Prism documentation set for information about

- debugging
- performance analysis
- data visualization

In addition, various tools and facilities are available from the shell of the CM control processor. This chapter describes the following:

- timing
- profiling
- traceback
- C preprocessor
- CM Fortran simulator
- error and core files

Some additional tools are managed with compiler switches and are described in Chapter 2 of this manual:

- run-time safety checking (using the switch **`-safety`**)
- reporting array homes and the kinds of communication being generated (using the switch **`-list`**)
- locating symbols and line labels (using the switch **`-cross_reference`**)

Copyright © 1991–1994 Thinking Machines Corporation

4.1 The Timing Facility

The CM system provides a timing facility that lets you determine how much elapsed time and CM time any part of a program takes to execute. The timer consists of a set of procedures that you call at the appropriate places in the program. It records times with microsecond precision.

"CM busy time" refers to time spent by the parallel processors; "elapsed time" refers to total time consumed by the process, *not* wallclock time. Be aware, when timing I/O, that much of the work for I/O happens when your process is inactive and, therefore, cannot be measured by the normal timers. To time I/O, use the cmf77 function **TIME**, which returns wallclock time.

Within a single program, you can have multiple timers (up to 64), you can nest timers, and you can do consecutive or cumulative timings. Using the timer involves at least four procedures: **CM_timer_clear**, **CM_timer_start**, **CM_timer_stop**, and **CM_timer_print**. Your first call to the timer should be **CM_timer_clear**, as this call zeroes the timer and its number-of-starts.

For example, you might want to establish both the total time spent in a loop and the time spent on one key part of the loop. You could do that as follows:

```
CALL CM_TIMER_CLEAR(0)              !Initialize the timers.
CALL CM_TIMER_CLEAR(1)

CALL CM_TIMER_START(0)              !Start outer timer.
   ! Start the loop
    . . .

CALL CM_TIMER_START(1)              !Start and then stop
   ! Perform key portion of loop    !the second timer.
    . . .                           !Since it's not cleared
CALL CM_TIMER_STOP(1)               !between starts,
    .                               !timings will be
    .                               !cumulative.
    .
   ! End the loop
CALL CM_TIMER_STOP(0)      !Stop outer timer.
CALL CM_TIMER_PRINT(0)     !Print times for entire loop.
CALL CM_TIMER_PRINT(1)     !Print total time for
                           !key portion of loop.
```


Version 2.2, October 1994
Copyright © 1991–1994 Thinking Machines Corporation

Functions that return wallclock time (as well as date and other such information) from the control processor are provided by the operating system. The cmf77 library, described in the *CM Fortran Libraries Reference Manual*, provides a convenient CM Fortran interface to these functions.

The procedures of the CM timing facility are listed below. All are subroutines except the functions **CM_timer_read_***xxx*. If you use any of these functions in a CM Fortran program, you need to include the header file **timer-fort.h**.

```
INCLUDE 'timer-fort.h'
```

CM_timer_clear(*timer***)**

Sets the total elapsed time, total idle time, and number of starts of *timer* to zero. The value of *timer* is an unsigned integer between 0 (inclusive) and the integer variable **CM_*number_of_timers***. This variable represents the maximum number of timers that can be allocated at one time; it is currently equal to 64.

CM_timer_start(*timer***)**

Starts the clock running for the specified *timer*.

CM_timer_stop(*timer***)**

Stops the clock for the specified *timer* and increments its idle time and wallclock time.

CM_timer_print(*timer***)**

Prints out information about the specified *timer,* including (but not limited to) its number of starts, its total elapsed time, and the total time that the CM was busy while this timer was active.

INTEGER = CM_timer_read_starts(*timer***)**

Returns the number of times **CM_timer_start** has been called on this timer.

Copyright © 1991–1994 Thinking Machines Corporation

CM_timer_set_starts(*timer*, *integer***)**

Sets the number of starts for **CM_timer_start**, for the timer indicated by the *timer* argument, to the integer value specified in the second argument.

DOUBLE = CM_timer_read_elapsed(*timer***)**

Returns the total elapsed time, in seconds, accumulated while the specified *timer* was running. "Elapsed time" refers to process time, not wallclock time.

DOUBLE = CM_timer_read_cm_busy(*timer***)**

Returns the total CM busy time, in seconds, accumulated while the specified *timer* was running.

DOUBLE = CM_timer_read_cm_idle(*timer***)**

Returns the total CM idle time, in seconds, accumulated while the specified *timer* was running. CM idle time is equal to total elapsed time minus the CM busy time.

INTEGER = CM_timer_read_run_state(*timer***)**

Returns **.TRUE.** if and only if the specified timer is running. Otherwise, returns **.FALSE.**.

Version 2.2, October 1994
Copyright © 1991–1994 Thinking Machines Corporation

4.2 Profiling CM Fortran Programs

You can obtain an execution profile of a CM Fortran program by using Prism. To collect performance data, compile your program with the **-cmprofile** switch.

Prism Version 2.0 does not support node-level profiling. Do not use the CM Fortran switch **-cmprofile** together with **-local**.

However, profiling the global part of a global/local program is supported. Prism shows how much total time is spent in a local routine, but you cannot determine how that time is being spent. For this purpose, compile the global part of the code with **-cmprofile**, compile the local part without **-cmprofile**, and then link it all together.

For more information, refer to your Prism documentation.

4.3 Collecting Performance Data

The Prism development environment provides a thorough, easy-to-use tool for performance analysis. For more information, see your Prism documentation.

You can also collect performance data by setting environment variables, without entering Prism. This is convenient if you can't enter Prism for some reason (for example, because the CM is only accepting batch jobs).

To turn on collection of performance data, set the environment variable **CMPROFILING** to **t**:

```
% setenv CMPROFILING t
```

To turn collection off, set the environment variable to **f**.

To specify the program on which data is to be collected, set the environment variable **CMPROFILING_EXECUTABLE_FILENAME** to the name of the executable program. For example:

```
% setenv CMPROFILING_EXECUTABLE_FILENAME a.out
```


Copyright © 1991–1994 Thinking Machines Corporation

To specify the file to which the performance data is to be sent, set the environment variable **CMPROFILING_DATA_FILENAME** to the name of the file. For example:

```
% setenv CMPROFILING_DATA_FILENAME perf.data
```

You can load this file into Prism for examination at a later time. See the Prism documentation for further information.

Version 2.2, October 1994
Copyright © 1991–1994 Thinking Machines Corporation

4.4 The Traceback Facility

The CM Fortran run-time system provides a traceback facility that displays information about the nest of procedure calls that are active when an error is detected. The program below illustrates the facility.

```
 1    PROGRAM TRACEBACK
 2    CALL FOO( 13 )
 3    END

 5    SUBROUTINE FOO( I )
 6    CALL BAR( 42*I )
 7    END

 9    SUBROUTINE BAR( I )
10    CALL GRILL( I*0 )
11    END

13    SUBROUTINE GRILL( I )
14    I = 1/I     ! scalar division by zero
15    END
```

Execution of the program results in the following output:

```
*** RTS-FATAL-UNIX: Integer division by zero
Traceback follows:
   pc = 0x625        GRILL, line 15
   pc = 0x606        BAR, line 10
   pc = 0x5e6        FOO, line 6
   pc = 0x5c6        _main, line 2
```

All active procedures are listed, with the most recent reference first. For each active procedure, the program counter value (in hexadecimal), the procedure name, and possibly a line number are given. For the procedure in which the error was detected, the line number indicates the next statement to be executed; in other procedures, it indicates the line number of an active procedure reference. The line number is provided for a given procedure only if the file containing the procedure is compiled using a debugging switch (the **-g** or the **-cmdebug** switch).

NOTE: If your program halts while executing a parallel code block, the traceback information could be misleading. Recall that the nodes run

Copyright © 1991–1994 Thinking Machines Corporation

asynchronously with respect to the partition manager, and that the partition manager is frequently ahead of the nodes in the program. Moreover, if a node has an error, the partition manager may continue working for a while before the error status is propagated to it and your program halts. Therefore, the routine that is executing on the partition manager when the nodes die may have nothing to do with the error.

Version 2.2, October 1994
Copyright © 1991–1994 Thinking Machines Corporation

4.5 The C Language Preprocessor

The **cmf** command driver accepts files with (uppercase) extensions of **.FCM** and **.F**, and invokes the C language preprocessor **cpp** on each file before passing it on to the appropriate compiler. (These extensions correspond to their lowercase counterparts used for CM Fortran and Sun FORTRAN 77, respectively). (**NOTE: cpp** is also invoked by the **cc** or **cs** compilers if the filename suffix is **.c** or **.cs**.)

The on-line man page for **cpp** describes the program switches and preprocessor command lines in detail, including a facility for defining macros with arguments.

4.5.1 Used with −D

The C preprocessor can provide a useful conditional compilation facility for CM Fortran source code when used with the **cmf** command line switch **-D**, described in Section 2.3. For example, the following program contains preprocessor control lines that conditionally define a parameter **N**, which is used in the declaration of a matrix **A**.

```
      PROGRAM CPP
#if ASIZE > 0 && ASIZE < 10
      PARAMETER (N = ASIZE)
#else
      PARAMETER (N = 9)
#endif
      CHARACTER*10 FMT
      INTEGER A(N,N)
      A = 0
      FORALL (I = 1:N, J=1:N) A(I,J) = I*10 + J
      WRITE (FMT, 10 ) N
10    FORMAT( "(1X,", I2.2, "I3)" )
      PRINT FMT, TRANSPOSE(A)
      END
```

The preprocessor control lines (those beginning with the character #) test whether the value of the symbol **ASIZE** is in the range **1** to **9** and, if so, select the first **PARAMETER** statement for compilation, otherwise the second. (The control lines themselves are filtered from the file actually passed to the

Copyright © 1991–1994 Thinking Machines Corporation

compiler, along with the unselected **PARAMETER** statement.) The value for **ASIZE** is substituted for all occurrences of the symbol **ASIZE** in the program; the value of symbol **ASIZE** can be defined in the source code, on the command line, or it can be left undefined (in which case it assumes the value zero). If the program is in the file **cpp.FCM**, then the command line

```
% cmf -DASIZE=7 cpp.FCM
```

causes the matrix **A** to be declared as a 7 x 7 array, while the command line

```
% cmf -DASIZE cpp.FCM
```

declares the matrix as a 1 x 1 array (the value of **ASIZE** is not specified, so it is assumed to be **1**). If no definition of **ASIZE** is supplied on the command line or in the source file, its value is taken as zero, and so the second **PARAMETER** statement is compiled, making it equal to the default value of **9**.

4.5.2 Predefined Symbols

The **cmf** compiler passes to **cpp** certain predefined symbols that identify the target machine, execution model, and several switches. This feature allows use of conditional **#ifdef** code that depends on compiler switches.

Symbols are defined that:

- Identify the target machine:

  -cm5 -DCM5
  -cmsim -DCMSIM

- Identify the target execution model:

  -sparc -DCMF_SPARC
  -vu -DCMF_VU
  -node -DCMF_NODE
  -local -DCMF_LOCAL (for **-local** code only)

Version 2.2, October 1994
Copyright © 1991–1994 Thinking Machines Corporation

- Identify use of certain switches that may affect performance or introduce incompatibilities between program units if not uniform.

-O	**-DCMF_OPTIMIZE**
-g	**-DCMF_DEBUG**
-cmdebug	**-DCMF_CMDEBUG**
-cmprofile	**-DCMF_CMPROFILE**
-padding	**-DCMF_PADDING**
-axisreorder	**-DCMF_AXISREORDER**
-f90syntax	**-DCMF_F90**

Copyright © 1991–1994 Thinking Machines Corporation

4.6 The CM Fortran Simulator

The CM Fortran simulator is an execution model whereby a CM Fortran program runs serially on a Sun-4 computer, without involving the CM parallel processors. The simulator can be useful in developing and debugging programs meant for any CM platform.

The CM simulator supports the CM Fortran language and utility library, with the following exceptions:

- parallel I/O utility procedures
- array aliasing
- controllable axis sequence (**-[no]axisreorder**)
- controllable local padding (**-[no]padding**)
- node-level programming with message passing
- parallel **READ**/**WRITE** to SDA and DataVault
- **READ**/**WRITE** character data to DataVault
- **PHYSICAL** keyword for **OPEN**
- 64-bit integers

The CM simulator does not support other CM libraries. It is compatible with the Prism development environment.

4.6.1 Timer Accuracy

The CM timing procedures can be used in simulator programs in the same way as in parallel programs. The Sun-4's UNIX timer is used in executing these procedures. Naturally, a program's execution time on the simulator bears no relation to its execution time on a CM system.

The results of UNIX timing tend to be accurate to about 10 milliseconds, whereas the CM hardware produces timing results accurate to about 1 microsecond. Timing results are therefore less precise when executing on the simulator. Also note that the cumulative error in successive timing calls increases far more rapidly on the simulator than it does on CM hardware.

Version 2.2, October 1994
Copyright © 1991–1994 Thinking Machines Corporation

4.6.2 Memory Limitations

The very large array sizes often used in CM Fortran programs may exhaust the memory available on a serial computer, or they may overrun the data structures used to simulate CM memory. If you encounter memory problems when executing a program on the simulator, you might consider the following remedies:

Reducing Array Sizes

You can conserve memory by reducing the sizes of large arrays while the program is under development. Declaring array dimensions with parameters rather than literal constants makes it convenient to adjust array sizes later when you run the program on the CM system.

Increasing Allocated Memory

Alternatively, you can direct the simulator to allocate more memory for your program (up to the limit of virtual memory your process is allowed).

The simulator simulates parallel memory with dynamically allocated memory. It creates two 4-megabyte arrays in which it places items that would, on CM hardware, be stored in the CM *heap area* or in the CM *stack area*. CM arrays in **COMMON**, dynamically allocated arrays, and various compiler temporaries are stored on the CM heap (the static area) or in the array that simulates it; user temporaries, automatic arrays, and various temporaries created by the run-time system are stored on the CM stack or in the array that simulates it. The error message that accompanies program demise on the simulator indicates whether it has run out of "stack memory" or "heap memory."

You can increase the amount of memory available for the simulated heap or stack by setting a user environment variable:

```
CMSIM_HEAP_SIZE
CMSIM_STACK_SIZE
```

The integer value supplied specifies the (decimal) number of bytes in the array that the simulator creates. For example, the following command line

Copyright © 1991–1994 Thinking Machines Corporation

increases the size of the simulator's "heap" from 4 megabytes to 8 megabytes:

```
% setenv CMSIM_HEAP_SIZE 8388600
```

The following increases the size of the simulator's "stack" from 4 megabytes to 5 megabytes:

```
% setenv CMSIM_STACK_SIZE 5000000
```

Making Expensive Operations Conditional

Finally, you can sometimes prevent an out-of-memory condition by calling the utility library function **CMF_AVAILABLE_MEMORY** from within the program. If you know that a particular operation is memory-intensive, you can make its execution conditional on the amount of memory available at the time. When used with the simulator, **CMF_AVAILABLE_MEMORY** returns the (decimal integer) number of bytes left in the arrays that simulate CM memory.

4.6.3 How the Simulator Works

The CM Fortran simulator works by linking with a run-time library that assumes that the machine has only one node. Because it is not modeling an *n*-processor system, the simulator is unable to estimate performance on such a system.

4.7 CM-5 Error Output and Debugging

For information on debugging CM Fortran programs, see the Prism documentation and the *CM-5 User's Guide*. Note that debugging nodal programs and the local part of global/local programs requires Prism Version 2.0 or later.

Version 2.2, October 1994
Copyright © 1991–1994 Thinking Machines Corporation