Chapter 3

CM Fortran Timing and Analysis Tools

To improve the performance of a program, it helps to know how efficient it already is. There are two basic mechanisms for analyzing the performance of a CM Fortran program:

- The **`CM_timer`** facility. You insert CM timer procedure calls in your program; these calls start and stop timers as the program runs, and report information about how much time was spent in specific portions of the code. The **`CM_timer`** procedure calls are described in the *CM Fortran User's Guide*.

- The performance analysis tools of the Prism programming environment. You compile your program with the **`-cmprofile`** option, and then use Prism's collection and analysis techniques to display information about the program's performance. Prism performance analysis is described in detail the *Prism User's Guide*.

These two methods use the same mechanism: the Prism timing routines make calls to **`CM_timer`** procedures to get timing data. The difference is the level of detail involved. You can use Prism's performance analysis tools as a "shotgun" method for tabulating the overall performance of every part of your program, and use direct calls to the **`CM_timer`** routines for fine-tuning of a specific program fragment.

In general, Prism is more convenient. It doesn't require changing your source code, and it provides an easy-to-read graphic interface to the results.

Note: For both Prism performance analysis and the **`CM_timer`** facility, compiling with the **`-g`** switch (full debugging) will distort the results, since it forces each statement to be in a separate code block.

Copyright © 1994 Thinking Machines Corporation

3.1 CM Timers, Timesharing, and You

Timesharing can significantly impact your timing results, especially when you run a program on a CM that has a large number of users (whether or not they are executing CM programs). There are two ways your program can be swapped out by CM timesharing:

- By the OS on the partition manager, so that it cannot execute on the PM. (PE code blocks may still be executing on the nodes, however.)
- By the CM timesharing daemon (the *ts-daemon*), so that it cannot execute on the CM processing nodes. (This implicitly means the PM part of the program cannot execute.)

There are three different timing metrics you can apply to your program, each of which is affected in different ways by the presence of other users on your partition manager:

- *Wallclock time*: The total execution time taken by your program or one of its subroutines. You can use UNIX timing tools (**`gettimeofday`**, etc.) to obtain this timing information. Wall clock time is affected by all users sharing your PM.
- *CM elapsed time*: The elapsed execution time while the program is not swapped out by the OS on the partition manager or by the timesharing daemon. You can use the **`CM_timer_read_elapsed`** procedure to get the elapsed time from a CM timer. Elapsed time is only affected by other users running purely scalar programs (that is, non-CM) on your PM.
- *CM busy time*: The time spent actually executing parallel computation on the PNs. When the CM busy time differs significantly from the elapsed time, it indicates that the program is also executing scalar operations that do not involve the PNs, while the PNs sit idle. You can use the **`CM_timer_read_cm_busy`** function to get the CM busy time, and the **`CM_timer_read_cm_idle`** function to get the CM idle time (the elapsed time minus the CM busy time).

For consistent and accurate timing results, you should try to time your code when few other users are running programs on the same PM at the same time. If you can't guarantee that you will have exclusive use of a CM-5 during your timing run, try to run the program several times; the lowest elapsed time reported will be the most accurate.

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

3.2 The CM_timer Facility

Using the **CM_timer** facility is simple: you bracket a section of code with calls to **CM_timer_start** and **CM_timer_stop**, and use calls to **CM_timer_read_elapsed** and similar functions to obtain the timing results recorded by these routines.

One important factor to note is that **CM_timer** routines are scalar procedure calls, which interrupt PE code blocks. If you're not careful where you place these timer calls, you may alter the performance information you receive.

As an example, consider the following two timing examples:

```
CALL CM_TIMER_CLEAR(0)
CALL CM_TIMER_START(0)
   Y=X*X+Z
   Z=Y*Y+X*X
CALL CM_TIMER_STOP(N)
T = CM_TIMER_READ_ELAPSED(N)

CALL CM_TIMER_CLEAR(0)
CALL CM_TIMER_START(0)
   Y=X*X+Z
CALL CM_TIMER_STOP(N)
TA = CM_TIMER_READ_ELAPSED(N)
CALL CM_TIMER_CLEAR(0)
CALL CM_TIMER_START(0)
   Z=Y*Y+X*X
CALL CM_TIMER_STOP(N)
TB = CM_TIMER_READ_ELAPSED(N)
```

Both examples involve the same set of computations on the arrays **x**, **y**, and **z**. The first example times the two computations together; the second times each one separately. The first example allows the two computations to be compiled as one PE code block, thus executing almost twice as fast as the second example (that is, the times **ta** and **tb** added together are almost twice the single time **t**).

Copyright © 1994 Thinking Machines Corporation

3.2.1 Tips on Timing Parallel Computation and Communication

Here are some tips to keep in mind when timing parallel computation:

- When timing small portions of code, time the code executing several times in a loop to get more accurate timing results. The time measured for a single execution might not accurately reflect the code's performance, particularly with small array subgrid sizes in which the execution time is less than 100 µs. In addition, there is a small amount of overhead involved in starting and stopping the timer, which might affect the results for very short times. Again, timing the code in a loop (and, if you wish to be exact, timing the overhead of a bare timer call and subtracting it) is a good method to use.

- Remember in writing timing loops that there is an instruction and data cache between the microprocessor and memory; hence, if your loop body is very short or has few array references, all references and instructions will be cached and subsequent loop iterations won't reflect the time taken by independent (non-cached) computations. In addition, keep the cache in mind when you select a small section of a large code region for more specific timing — if the small section is cacheable, you may see timing results significantly different than you obtained while timing it as part of a larger loop.

- When timing CMFS parallel I/O operations, the **`CM_timer`** facility does not report accurate results because its timings do not include the operating system's behind-the-scenes work in performing the I/O operations. You should use UNIX timers (such as **`gettimeofday`**) to obtain actual wallclock times for I/O operations. (See Section 3.5 for an example.)

You can measure the performance of parallel communication code in much the same way that you measure the performance of code that performs computation, by using the **`CM_timer`** calls. Because CM communication operations are not performed within PE code blocks, inserting calls to the **`CM_timer`** procedures among communication operations does not affect performance by breaking code blocks.

If your code contains a mixture of parallel computation and implicit communication operations in the same program statement, and you want to measure the two independently, separate the computation and communication, putting each on a separate line. (For example, you can assign intermediate results to temporary arrays.)

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

3.3 Using Prism to Analyze Performance

This section briefly describes aspects of using Prism to analyze the performance of a CM Fortran program. For more information, see the *Prism User's Guide*.

Prism performance analysis data is most useful for determining bottlenecks in a program. Where is a program spending its time? What resource is it depending on most?

To use Prism to analyze the performance of a program, you must

1. Compile your program with the **`-cmprofile`** switch.
2. Start up Prism and load the compiled executable program.
3. Turn on Prism's **Collection** option. Run the program to completion.
4. Select **Display Data** from the **Performance** menu. This will cause Prism to display a window of performance data graphs, as shown in Figure 7.

You can also compile and run your program outside of Prism, and then use Prism to view the captured performance data. To do this:

1. Set the following environment variables as shown:

   ```
   setenv CMPROFILING t
   setenv CMPROFILING_DATA_FILENAME myprog.perf
   setenv CMPROFILING_EXECUTABLE_FILENAME myprog
   ```

   (In this example, **`myprog`** is the name of your executable program.)

2. Compile and run your program as usual, either interactively or as a batch job. This will produce a data file, **`myprog.perf`**:

   ```
   cmf myprog.fcm –cmprofile –o myprog
   myprog
   ```

3. Access the performance data by loading it within Prism. (From the **Performance** menu, choose **Load Data**, and select the file.) You can then use **Display Data** to view the data file normally.

Copyright © 1994 Thinking Machines Corporation

3.3.1 Prism Performance Data Display

Prism provides three levels of performance data:

- *Resources*: The program's overall use of individual CM-5 resources: for example, general communication, grid communication, and PN computation (referred to as node CPU) time.

- *Functions*: The use of resources by each subroutine in the program. The data is available in both flat and call-graph mode. *Flat mode* displays each subroutine's total use of the resource, regardless of where the subroutine was called. *Call-graph mode* displays the dynamic call graph of the subroutines, and the use of the resource for each individual call.

- *Source lines*: Prism provides data on the use of a given resource for each source line within a procedure.

The data is available either as seconds (or μs) of elapsed time, or as a percentage of the total elapsed time. Figure 7 shows one view of the Prism performance data window for a sample program. At the top left of the window is the resource information. Each line refers to a different kind of CM operation. The resources timed are:

- **PM cpu (user)** — PM execution of the scalar part of your program.

  Note: The PM cpu (user) time includes time spent waiting for PN operations to complete, and so can be misleading if your program does not include much scalar code.

- **PM cpu (system)** — PM execution of OS operations, except I/O.
- **PM I/O** — PM execution of I/O operations.
- **PM Total** — Total PM execution time.
- **Node cpu (user)** — Parallel computation (PE code blocks).
- **Node cpu (system)** — PN kernel, dispatch loop execution.
- **Comm (Send/Get)** — Multiway send/get communication.
- **Comm (NEWS)** — Regular, patterned shifts of array data.
- **Comm (Reductions)** — Other types of parallel communication: parallel prefix, spreads, reductions.
- **Comm (PM<−−>Node)** — Data exchange between PM and nodes (global sum, etc.).

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

- **Node not profiled** — Routines not compiled with **`-cmprofile`**, etc.
- **Node I/O** — PN I/O operations (except the OS portion of writes).
- **Node+Comm Total** — Total PN resource use.

Note: You may have to scroll the resource window to see the last few items.

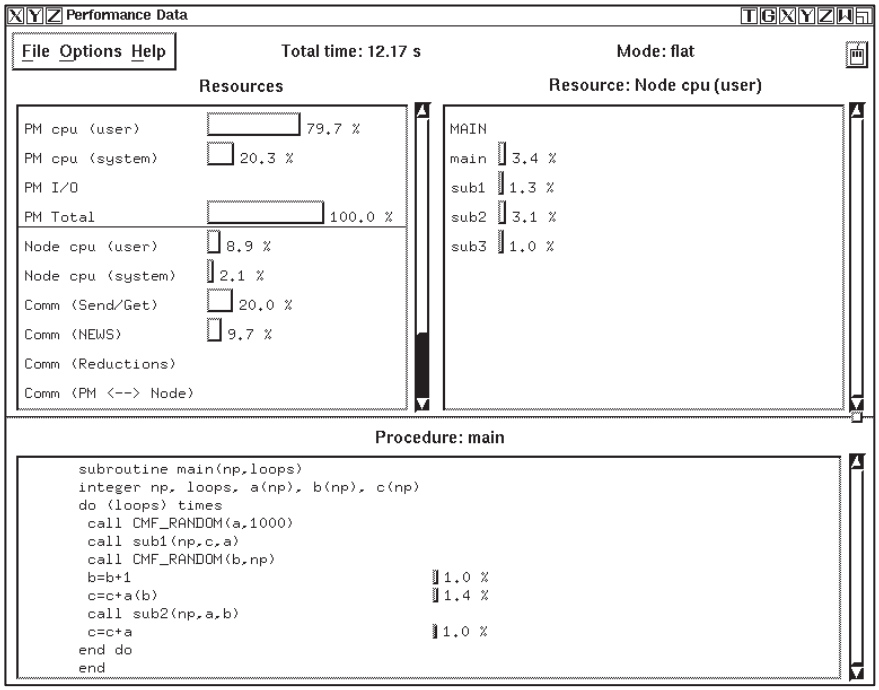

Figure 7. Performance data for a CM Fortran program.

At the top right of the data window is a flat-mode per-procedure display for the currently selected resource (you can choose any resource to view by clicking the appropriate bar in the resouce window). In the figure, the **Node cpu (user)** resource is selected. As the display shows, most of the parallel computation of the program was in the **`main`** and **`sub2`** subroutines.

Note: The reported PM and node resource times can overlap. The totals for the two types of resources can each range up to 100% of elapsed time.

At the bottom of the window is a source-line display for the currently selected subroutine of the program (again, you can select any subroutine for display by clicking the appropriate line in the subroutine resource display). In the figure, the **Node cpu (user)** times for the main subroutine are dis-

Copyright © 1994 Thinking Machines Corporation

played. Each source line that performs parallel computation has a usage time displayed beside it.

You can also instruct Prism to display a call-graph view of the resource data, by selecting **Mode−>Call Graph** from the **Options** menu. The call-graph display is shown in Figure 8. As you can see, the information in the resource window does not change, but the displayed data in the subroutine and source code windows is quite different.

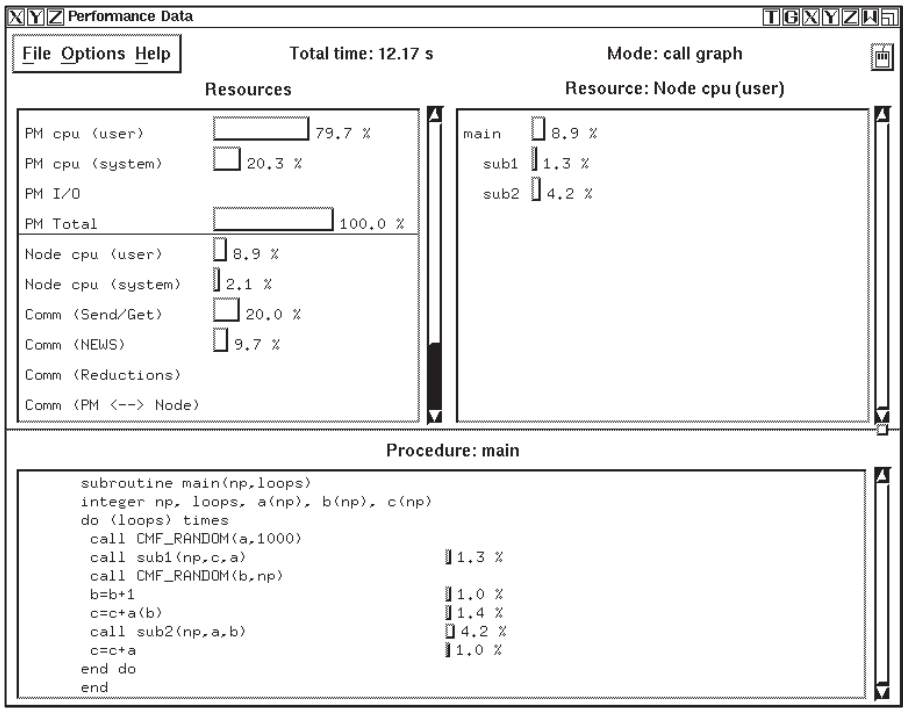

Figure 8. Call-graph display of performance data.

The subroutine times now show the total time for a selected subroutine (the main subroutine in this example, plus the components of that time spent in each subroutine call. (You can move up or down the call tree, from subroutine to subroutine, by clicking on the data bars in this window.) In the figure, the data displayed shows that at least half the **Node cpu (user)** time spent in the **`main`** subroutine comes from the call to **`sub2`**.

The source-line data at the bottom of the window also shows the time allocated to each subroutine call from within **`main`**. It shows the large percentage of time spent in the **`sub2`** routine.

Copyright © 1994 Thinking Machines Corporation

3.3.2 Timing of PE Code Blocks

Prism calculates source-line usage per PN code block, not per source line. If a single code block contains multiple source lines, Prism divides the usage equally among these source lines for display purposes. Thus, if a number of adjacent source lines have equal times, it may be an indication that they are part of the same code block.

If you require specific times for each line, you may have to compile with the `-g` compiler switch (to force single-line code blocks), or insert code that causes the source lines to be divided into separate code blocks. Remember that this produces less than optimal times overall.

3.3.3 Timing of Parallel I/O Operations

Prism reports only some of the time spent performing parallel I/O write operations (via the CMFS interface), because the nodes can "finish" the write and continue execution before all the data has been written to disk; the OS asynchronously handles the leftover I/O work.

The time reported for parallel I/O write operations includes the time spent re-ordering the parallel data (converting it to and from New Mexico Order) for the I/O operation, but does not include asynchronous OS activity used to complete the operation. The I/O times reported for parallel read operations are complete, however, because for reads the nodes must wait until all data is received.

For more information on CM Fortran I/O, see the *CM Fortran Language Reference Manual* and the *CM Fortran User's Guide*. For an example of the use of UNIX timing routines to time I/O operations, see Section 3.5.

3.3.4 Prism Restrictions

When using Prism to analyze the performance of a CM Fortran program, you should be aware of the restrictions listed below. These restrictions may be removed in future compiler and Prism releases.

- Currently, only the parallel resource items (**Node**, **Comm**, etc.) can be viewed by source line. If you select any of the PM items, the source code window displays a warning message to this effect.

Copyright © 1994 Thinking Machines Corporation

- Prism does not account for the time spent executing procedures that were not compiled using **-cmprofile**. *This includes all CM Fortran library procedures, in particular the procedures in the utilities library.*
- The time attributed to scalar computation includes time spent waiting for PNs to complete operations. Because of this, the time reported as **PM cpu time** can be misleadingly large.

3.3.5 Using Prism and the CM Timer Utility Together

It is possible to use Prism and the **CM_timer** facility at the same time. However, you should note that Prism times the programmer's **CM_timer** routines, and the programmer's **CM_timer** routines time Prism's timing activity. This usually has a greater effect on the programmer's **CM_timer** results, since Prism usually inserts more timing calls than the programmer.

In addition, Prism reserves some CM timers for itself (by default, all timers with numbers higher than 4). If you need more timers for **CM_timer** calls, you must instruct Prism to reserve fewer timers for itself. This is done by setting a UNIX environment variable, **CMPROF_N_USER_TIMERS**, to the number of timers you want to use, prior to compiling your program. For example, to reserve timers 0 through 7,

```
setenv CMFPROF_N_USER_TIMERS 8
```

3.4 Examining Assembler Source Code

Another source of information about the compilation of a CM Fortran routine is the PE assembly code of the program, obtained by compiling with the **-veccode** switch. From the assembly code, you can

- count the number of code blocks used in the program
- determine the number of SPARC ticks (cycles) used
- see where the program is using floating-point operations

This information is available in the *program*.**pe.dp** file produced by the compiler (or *program*.**pe.s**, produced by the **-S** switch, for non-VU systems).

Version 2.2, October 1994
Copyright © 1994 Thinking Machines Corporation

3.4.1 Counting Code Blocks

To determine the number of code blocks, look for line labels of the form **_cmpe_foo_*n*_:**, where *n* ranges from 0 up to some arbitrary integer. Each of these labels marks the beginning of a code block. (Be sure not to confuse these with the **L*x*$_cmpe_foo_*n*_:** branch labels used within code blocks.)

Note: PE code blocks are not always labeled with consecutively increasing values, i.e., looking at the end of the **.dp** file for the last label is not sufficient, especially when the routine is compiled with the **-O** switch.

3.4.2 Estimating SPARC Cycles

Each line of the assembly code output includes a comment line estimating the number of SPARC clock ticks (or cycles) required to execute the instruction. For example,

```
                                              ![ cyc , flop]
sub %g0, 96, %g2                              ![    1,    0]
load 0, %g6                                   ![    2,    0]
imulv V2,V2,V2; uloadv [%i2+%g6]:4,V2;        ![   16,    0]
                                              !-------------
                                              ![   19,    0]
```

In this routine, the SPARC **sub** and **load** instructions take 1 and 2 ticks respectively. The DPEAC **imulv**/**uloadv** instruction takes 16 SPARC ticks (8 VU ticks, one for each vector element).

By totalling the number of SPARC ticks required for a given PE code block, then multiplying by the number of times the code block is executed (the subgrid size of your arrays divided by 8, the vector length), you can get an estimate of the amount of time the PE code block should take.

A 40MHz SPARC chip executes 40,000 ticks per millisecond, so you can compare the value you get via the above arithmetic with the actual performance of the microprocessor. For example, if you're executing the routine above on a CM with 40MHz SPARC microprocessors, you should at best be able to execute it 40,000/19 or 1684 times per millisecond. Assuming a vector length of 8 elements, in one millisecond you can expect to handle arrays with subgrid sizes of 1684*8=13472 elements or less.

Copyright © 1994 Thinking Machines Corporation

3.4.3 Counting Floating-Point Operations

The comment line for each assembly code source line also includes a count of the number of flops executed by the instruction. For example:

```
                                                    ![ cyc , flop]
fnopv; dustorev S4:0, [%i3 + %o0]:8; nopad;         ![   16,    0]
dfmulv V2, V2, V2; duloadv [%i2 + %o0]:8, V2;       ![   16,    8]
dfmadiv V2, V4, V4; duloadv [%i3 + %o0]:8, V4;      ![   16,   16]
                                                    !-------------
                                                    ![   48,   24]
```

The arithmetic operations here are a no-op (0 flops), a floating-point multiplication (8 flops), and a floating-point mult-add (16 flops). In general, the flop column lists only computations involving floating-point operators, so you should not be surprised to see long columns of 0's in routines that do no floating-point calculations.

3.4.4 WARNING: Don't Misinterpret the Assembler Numbers

Some parts of the DPEAC code blocks files are looped over many times, so just summing the cycles and flops won't give you the exact counts.

Also note that you should take care in using the flop counts provided in the assembly code files to estimate your program's total flop rate — other factors such as memory page-faulting will influence the flop rate you actually see. (See Chapter 7 for a discussion of these factors and their effects on PE code block performance.)

3.4.5 Sample PE Assembler Output File

For the short CM Fortran program

```
PROGRAM FOO
REAL*8 A(100),B(100)
A = 3.14
B = 2.0
A = A * A + B * B
END
```

the PE assembly code file (produced by **`cmf -c -veccode foo.fcm`**) is as shown below.

Copyright © 1994 Thinking Machines Corporation

```
!! Connection Machine Fortran Version: 2.2  Source Rev: f2100
!! Compiler options: -dp foo.fcm
!! At the end of each instruction CYC stands for SPARC cycles.
!! FLOP stands for floating point operations on one VU.
!! When a cycle is marked with a *, then it is a SPARC operation
!! that is overlapped by the issue of a previous VU operation.

                                                    ![ cyc , flop]
    .seg "text"                                     ![    0,    0]
    .global _cmpe_foo_0_                            ![    0,    0]
                                                    !-------------
                                                    ![    0,    0]
_cmpe_foo_0_:
                                                    ![ cyc , flop]
    sub %g0, 96, %g2                                ![    1,    0]
    save %sp, %g2, %sp                              ![    1,    0]
    dpregs %g1=, %g4=, %g2                          ![    3,    0]
    uors=8 S2=2, S2, S2; memnop; vmmode:=always;    ![    8,    0]
    mov 8, %l7                                      ![    1,    0]
    fnops; memnop []:=0;                            ![    8,    0]
                                                    !-------------
                                                    ![   22,    0]
L1$_cmpe_foo_0_:
                                                    ![ cyc , flop]
    loadd 0d3.14000000000000012 , %g6               ![    4,    0]
    dpwrtd *, %g6, S2                               ![   10,    0]
    loadd 0d2  , %o0                                ![    4,    0]
    dpwrtd *, %o0, S4                               ![   10,    0]
                                                    !-------------
                                                    ![   28,    0]
L2$_cmpe_foo_0_:
                                                    ![ cyc , flop]
    load 0, %o0                                     ![    2,    0]
    srl %i4, 3, %o1                                 ![    1,    0]
    and %i4, 7, %o2                                 ![    1,    0]
    fnops=8; memnop;                                ![    8,    0]
    load 8, %l7                                     ![    2,    0]
    subcc %i4, %g0, %i4                             ![    1,    0]
    bneg L7$_cmpe_foo_0_                            ![    1,    0]
    nop                                             ![    1,    0]
                                                    !-------------
                                                    ![   17,    0]
L3$_cmpe_foo_0_:
                                                    ![ cyc , flop]
    subcc %o1, %g0, %o1                             ![    1,    0]
    bz L5$_cmpe_foo_0_                              ![    1,    0]
    nop                                             ![    1,    0]
                                                    !-------------
                                                    ![    3,    0]
```


Copyright © 1994 Thinking Machines Corporation

```
L4$_cmpe_foo_0_:
                                                        ![ cyc , flop]
    fnopv; dustorev S2:0, [%i2 + %o0]:8; pad:8;         ![   16,    0]
    sub %o1, 1, %o1                                     ![*   0,    0]
    mov %o1, %o7                                        ![*   0,    0]
    fnopv; dustorev S4:0, [%i3 + %o0]:8; nopad;         ![   16,    0]
    dfmulv V2, V2, V2; duloadv [%i2 + %o0]:8, V2;       ![   16,    8]
    dfmadiv V2, V4, V4; duloadv [%i3 + %o0]:8, V4;      ![   16,   16]
    fnopv; dustorev V4, [%i2 + %o0]:8; pad:8;           ![   16,    0]
    add %o0, 64, %o0                                    ![*   0,    0]
    subcc %o7, %g0, %o7                                 ![*   0,    0]
    bnz L4$_cmpe_foo_0_                                 ![*   0,    0]
    nop                                                 ![*   0,    0]
                                                        !-------------
                                                        ![   80,   24]
L5$_cmpe_foo_0_:
                                                        ![ cyc , flop]
    subcc %o2, %g0, %o2                                 ![    1,    0]
    bz L7$_cmpe_foo_0_                                  ![    1,    0]
    nop                                                 ![    1,    0]
                                                        !-------------
                                                        ![    3,    0]
L6$_cmpe_foo_0_:
                                                        ![ cyc , flop]
    fnops=%o2; memnop;                                  ![    8,    0]
    mov %o2, %l7                                        ![    1,    0]
    load 0, %o2                                         ![    2,    0]
    load 1, %o1                                         ![    2,    0]
    ba L4$_cmpe_foo_0_                                  ![    1,    0]
    nop                                                 ![    1,    0]
                                                        !-------------
                                                        ![   15,    0]
L7$_cmpe_foo_0_:
                                                        ![ cyc , flop]
    ret                                                 ![    1,    0]
    restore                                             ![    1,    0]
                                                        !-------------
                                                        ![    2,    0]
```


Copyright © 1994 Thinking Machines Corporation

3.5 Timing Parallel I/O — Example

Here is a sample program that demonstrates the use of UNIX timing routines to time I/O operations. This program prompts the user for a number of bytes. That many bytes are written to the SDA and then read back in. Results are checked for accuracy.

```
      PROGRAM IO_TEST
      DO
        WRITE(6,'(" NBYTES? (ENTER 0 TO EXIT) ",$)')
        READ(5,*)NBYTES
        IF (NBYTES.LE.0) EXIT
        CALL SUB(NBYTES)
      END DO
      STOP
      END

      SUBROUTINE SUB(NBYTES)
      CHARACTER*(*),PARAMETER :: FILENAME='/TEST1/TEST'
      DOUBLE PRECISION TIMEOFDAY, START_TIME
      COMPLEX*16, DIMENSION(NBYTES/16) :: X, Y
CMF$  LAYOUT X(), Y()

C Make up write array
      X=SQRT(DCMPLX(DBLE([1:UBOUND(X,1)]),
      &   0.5d0*UBOUND(X,1)))
C Initialize read array
      Y=DCMPLX(0.d0,0.d0)

C Time write
      OPEN(20,FILE=FILENAME,STATUS='UNKNOWN',
      &   FORM='PHYSICAL')
      START_TIME=TIMEOFDAY()
      WRITE(20)X
      WRITE_TIME=TIMEOFDAY()-START_TIME
      CLOSE(20)

C Time read
      OPEN(20,FILE=FILENAME,STATUS='OLD',
      &   FORM='PHYSICAL')
      START_TIME=TIMEOFDAY()
      READ(20)Y
      READ_TIME=TIMEOFDAY()-START_TIME
      CLOSE(20)
```


Copyright © 1994 Thinking Machines Corporation

```
C Check results
      III=COUNT(X.NE.Y)
      IF (III.NE.0) WRITE(6,*)III,' ERRORS!!!!!!!!'

C Report times
      AMOUNT=NBYTES/2.**20
      WRITE_RATE=NBYTES/WRITE_TIME/2.**20
      READ_RATE=NBYTES/READ_TIME/2.**20
      WRITE(6,'("                                WRITE

     &          READ")')
      WRITE(6,'("     MB          SEC        MB/S

     &   SEC      MB/S")')

      WRITE(6,'(5F10.3)')
     &   AMOUNT,WRITE_TIME,WRITE_RATE,READ_TIME,READ_RATE
      WRITE(6,*)

      RETURN
      END
```

The following C code is in a file with the suffix **.c**:

```
long float timeofday_()
{
    long int t1[2],t2[2];
    int gtod;

    gtod=gettimeofday(t1,t2);
    return ((double)t1[0]+0.000001 * (double)t1[1]);
}
```


Copyright © 1994 Thinking Machines Corporation