NASA Contractor Report 181950

CPU Timing Routines for a CONVEX C220 Computer System

Mary Ann Bynum

PRC Kentron, Inc.
Aerospace Technologies Division
Hampton, Virginia

CONTRACT NAS1-18000
DECEMBER 1989

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

CPU Timing Routines for a CONVEX C220 Computer System

Mary Ann Bynum

PRC Kentron, Inc.
Aerospace Technologies Division
Hampton, Virginia

Introduction

The successful solution of computationally large problems depends not only on the existence of supercomputers with very fast CPUs and large memories, but on the ability of programmers to write fast code optimized for specific computers. CPU timing routines are subroutines or functions which allow an application programmer to measure the time necessary to execute a program or section of a program. The time measured can be elapsed time or can be the actual time the CPU has dedicated to a particular program. The knowledge of the execution time can then be used by the programmer to pinpoint sections of code which consume large amounts of CPU time. An examination of these CPU-intensive sections can reveal inefficient coding techniques or program structure whose correction can result in improved, shortened execution times. Shortened execution times place lighter loads on computer resources and allow large problems to be solved in a shorter time. Timing results allow an algorithm developer to compare the efficiency of different algorithms on the same computer architecture and of the same algorithm on different computer architectures. Any comparison of execution times must be based on an understanding of the meaning of the data reported by the timing routines on a particular computer. This report collects in one document background information, descriptions, and examples of the timing routines on the CONVEX C220 computer system in the Structural Mechanics Division at NASA Langley Research Center.

The purpose of this report is to describe the timing routines available on the CONVEX C220 computer system[†] in the Structural Mechanics Division (SMD) at NASA Langley Research Center. The report describes the function of the timing routines, the use of the timing routines in sequential, parallel and vector code, and the interpretation of the results from the timing routines with respect to the CONVEX model of computing. The timing routines available on the SMD CONVEX fall into two groups. The first group includes standard timing routines generally available on computers with UNIX 4.3 BSD operating systems, while the second group includes routines unique to the SMD CONVEX. The standard timing routines described in this report are **/bin/csh time**, **/bin/time**, **etime**, and **ctime**. The routines unique to the SMD CONVEX are

---

[†] The following remarks apply to Version 7.0 of the operating system and Version 5.0 of the FORTRAN compiler. The SMD CONVEX C220 is a two-processor model in the CONVEX C200 series.

getinfo, second, cputime, toc, and a parallel profiling package made up of **palprof, palinit,** and **palsum.**

The CONVEX Model of Computing

The CONVEX model of computing affects the interpretation of results from timing routines [1]. In standard sequential UNIX operating systems, including the CONVEX C220, the execution of a program in a single logical address space is referred to as a process. Each process is assigned a unique process identification number (PID). The CONVEX model of computing defines an additional entity called a thread. In the CONVEX model of computing, "a thread is an execution stream through the instruction space of a process" [2]. Like a process, each thread is assigned a unique thread identification number (TID). The terms *thread* and *process* describe software entities. A thread is created by the hardware scheduling system, while a process is created by the system software. Because it is created by the hardware, a thread may be considered the basic unit of identification for any executing program.

Sequential and vector programs have a single process and a single thread when executed. The execution of parallel programs or parallel sections of code in a program on the CONVEX C220 consists of the creation and execution of multiple concurrent threads for a single process. When a process reaches a section of parallel code, the hardware creates a new thread, increments the thread counter, and posts a notice to the processors that there is additional work to be done. The process then continues. The next available processor begins executing the new thread. When the work of any thread is complete, the CPU is freed and the thread count decremented. When all of a section of parallel code has completed execution and the thread count equals one, the current thread continues sequential execution until another parallel section of code is encountered. CONVEX refers to this dynamic scheduling scheme as *Automatic Self-Allocating Processors* (ASAP) [1].

The ASAP scheduling system is designed to ensure that a thread is executed as soon as there is a processor available to do so. It also minimizes the time that all processors are idle. The disadvantage of this system to the designer of parallel algorithms is the fact that the execution of a thread and the selection of a processor for that execution is beyond user control. A thread may be executed entirely on one processor, or may be switched between processors during its execution. The ASAP mechanism and the operating system cooperate to schedule the execution of a thread. Another result of the ASAP mechanism is that the actual concurrent execution of parallel code on different processors occurs only under the control of the ASAP scheduling system. CONVEX provides a utility, **mpa** [3], which allows the user to modify the attributes of a process. Using this utility, the user can gurantee that all processors are available for the execution of the threads of a process. Even with the use of **mpa**, however, the concurrent execution of parallel threads is not guaranteed. Several of the timing routines discussed in this paper provide a *concurrency factor*, a number between one and the total number of processors available, which reflects the degree to which a program executes concurrently.

The CONVEX model of computing defines the execution time of a process as the sum of the thread times for the execution of that process. This definition leads to apparently anomalous results when sequential, vector, and parallel versions of the same program are compared with the standard UNIX timing routine **etime**. The process time for the parallel, vectorized version of a program will be longer than the process time for the sequential version or the vectorized version alone. For sequential or vector programs in which there is one thread and one process, the thread time is identical to the process time. For parallel programs, however, there are several threads executing the same code. The elapsed time to execute each thread of the parallel program will be less than the time to perform the same work sequentially, but the sum of the thread times will be longer due to the overhead incurred in parallelization.

The expression of execution time as the sum of the thread times of a process measures the work which is performed by the entire computer in the execution of a single program and as such is a good measure of the load which that program places upon the operating system. It is not, however, a good measure of the benefits yielded by the parallelization of code, nor is it useful as a tool to dissect the behaviour of parallel algorithms. Elapsed time, the time from start to finish of the execution of a program, measures the benefits of the parallelization of code on a computer which is dedicated to that job, but does not give reproducible results on a computer run in the multi-user mode such as the SMD CONVEX C220. Local timing routines have been developed on the SMD CONVEX C220 which will measure and report the execution times of each thread of a parallel program and report a value for the CPU-time to execute the program which reflects the benefits due to the parallelization of that program.

Categories of Timing Routines

The UNIX timing routines on the SMD CONVEX C220 produce CPU times for sequential and vector code which are directly comparable to CPU times measured with the same routines on other sequential or vector machines. The UNIX timing routines on the SMD CONVEX C220 produce CPU times for parallel versions of code which are longer than the CPU times of sequential version of the same code run on the same CONVEX C220. The apparent slowness of parallel versions of code is due to the CONVEX model of computing in which CPU time is defined as the sum of the thread times for a program, whether it is executed sequentially, in vector mode or in parallel.

Local timing routines on the SMD CONVEX C220 measure thread times independent of the UNIX timing routines. A parallel profiling package has been developed which allows users to measure the thread times of each parallel loop in a subroutine and report the user time for the subroutine like the standard UNIX timing routines and in a way which measures the sum of the longest thread times for each loop in the subroutine. The former method is useful for measuring throughput, while the latter method is useful in measuring the efficiency of the parallelization of an algorithm.

Standard UNIX Timing Routines

1. /bin/time

The timing routine **/bin/time** is described in the *CONVEX UNIX Programmer's Manual* under **time(1)**† [4]. This routine reports the total time elapsed during the execution of a program (denoted by *real* in the results below), the time attributed to the program itself (denoted by *user* in the results below), and the time used by the system in the execution of that command (*sys* in the results below). With the **-e** option, times are reported with microsecond accuracy; otherwise, times are reported with tenth of a second accuracy.

Example:

```
/bin/time -e test4
```

Results:

```
0.024773 real        0.001160 user        0.016373 sys
```

2. /bin/csh time

The **time** command built into the **/bin/csh** command interpreter is described in the *CONVEX UNIX Programmer's Manual* under **csh(1)** [4]. This routine reports a timing summary for a specified command. The resource usage contained in the timing summary is based on data from the system call **getrusage(2)** [5]. The user may control the selective extraction of information by the command or use the default format which requires no specific specification of formatting variables. The format statement consists of two fields, separated by a space. The first field is a measure of CPU seconds and is an integer. Any command which takes more than this amount of CPU seconds causes a line to be printed that gives a time and resource usage summary. The second field is a *printf*-like formatting string, similar to the C programming language *printf* formatting string, which allows the user to customize the output of the **time** command. The following conversion specifications are recognized for the **time** command and have specific meanings:

%U Amount of time spent executing in user mode (seconds).

%S Amount of time spent in system mode executing on behalf of the process (seconds).

%E Real (elapsed) time (minutes, seconds).

---

† The number in parentheses refers to the section number of the *CONVEX UNIX Programmer's Manual*.

%P Percentage of CPU utilization (ratio of user plus system time to real time, scaled by the number of processors on the system).

%C CPU parallelization factor or concurrency level (1 < **%C** < number of processors on the system). This value is designed to be used when timing a particular process and will display *N/A* when **time** is issued as a command with no arguments.

%W Number of process swaps out of main memory.

%X Amount of memory shared among other processes (kilobytes).

%D Combined size of unshared data and stack segments (kilobytes).

%K Total size of shared memory, unshared data, and unshared stack sizes (kilobytes).

%M Maximum resident set size utilized (kilobytes).

%F Number of page faults serviced which required I/O activity.

%R Number of page faults serviced without I/O activity; here, I/O activity is avoided by reclaiming a page from the list of pages awaiting reallocation.

%I Number of times the file system had to perform block input.

%O Number of times the file system had to perform block output.

The default format string for **time** follows:

Example:

```
%Uu %Ss %E %P %X+%Dk %I+%Oio %Fpf+%Ww
```

Results:

```
1.01u 1.35s 0:03 40% 116+1249k 0+lio 104pf+0w
```

This cryptic output is a report of user, system and real times, a utilization percentage, the amount of shared memory, the combined size of unshared data and stack segments, the number of times the file system had to perform block input and output, the number of page faults, and the number of process swaps. The user may customize the output to produce a more readable format. The following example consists of lines a user may put in a shell script, and the results consist of the output from executing that shell script.

Example:

```
#!/bin/csh
set time =(0 "User time = %U System Time = %S Real time = %E \
Block input = %I Block output = %O \
Concurrency Factor = %C")
time test4-3
```

Results:

```
User time = 0.55 System time = 0.44 Real time = 0:01
Block input = 0 Block output = 1
Concurrency Factor = 1.16
```

3. etime

The command **etime** is described in the *CONVEX UNIX Programmer's Manual* under **etime(3F)** [6]. The function may be called in a FORTRAN program and returns the sum of the user and system times for a process. The user time is returned with microsecond accuracy, while the system time is returned with sixtieth of a second accuracy. User time and system time can be reported individually as well. In parallel sections of programs compiled with the **-O3** compiler option the user time reported by **etime** is the sum of the thread times for the process; therefore, a parallel program will appear to take as long or longer to execute than a sequential or vector version of the same program. The following example illustrates the use of **etime** in a FORTRAN program and is followed by the results.

Example:

```
      Program test1

      real tarray(2)
      real utime1, utime2, stime1, stime2
      real A(10000), B(10000), C(10000)
      t2 = etime (tarray)
      utime1 = tarray(1)
      stime1=tarray(2)
      do 100 i = 1,10000
            A(i) = B(i) + C(i)
100   continue

      t5 = etime (tarray)
      utime2 = tarray(1)
      stime2 = tarray(2)
      write (6,61)
      write(6,131) t5 - t2
      write (6,141) utime2 - utime1
```

```
      write (6,151 stime2 - stime1
61    format ('Main program call to doloop:  ')
131   format (      Time for call to doloop (etime) = ', f10.6)
141   format (      User time for call to doloop (etime) = ', f10.6)
151   format (      Stime for call to doloop (etime) = ', f10.6)
      end
```

Results:

```
Main program call to doloop:
    Time for call to doloop (etime) =          0.011167
    User time for call to doloop (etime) =     0.001167
    Stime for call to doloop (etime) =         0.010000
```

4. ctime

The command **ctime** is described in the *CONVEX UNIX Programmer's Manual* under **ctime(3)** [7]. Four related commands are also described under the manual entry **ctime(3)**: **localtime**, **gmtime**, **asctime**, and **timezone**. All of the routines described under **ctime(3)** are called from C programs. The comand **ctime** converts the current date and time as maintained by the operating system into ASCII format and returns a pointer to a 26-character string. The format of the string is that returned by the **date(1)** [4] command. The following example illustrates the output from a call to **ctime** in a C program.

Example:

```
Sun Sep 16 01:03:52 1973
```

The commands **localtime** and **gmtime** allow the user to access the system data structure which maintains information about the current time and date. The command **localtime** corrects for time zone and daylight savings time, while **gmtime** returns Greenwich Mean Time. The command **asctime** converts the times reported in **localtime** or **gmtime** to ASCII format and returns a pointer to a 26-character string. The **timezone** command returns the name of the time zone associated with its first argument. The data structure in which the information reported by the commands described under **ctime(3)** is stored by the system is listed in the following C example:

Example:

```
struct tm {
        int tm_sec;          /* 0-59      seconds */
        int tm_min;          /* 0-59      minutes */
        int tm_hour;         /* 0-23      hour */
        int tm_mday;         /* 1-31      day of month */
        int tm_mon;          /* 0-11      month */
        int tm_year;         /*0-         year - 1900 */
        int tm_wday;         /* 0-6       day of week (Sunday - 0) */
```

```
        int tm_yday;            /* 0-365    day of year */
        int tm_isdst;           /* flag:    daylight savings time in effect */
}
```

5. cputime

CONVEX VECLIB contains the timing routine **cputime** which returns the total amount of user-mode CPU time with microsecond accuracy [8].

The following example illustrates the use of **cputime** to time a section of FORTRAN code.

Example:

```
        Program test2

        real *4 time, tzero
        real *8 A(100)

        tzero = cputime (0.0)
        do 10 i = 1,100
                A(i) = 5.0 * i
10      continue
        time = cputime (tzero)
        write (6,100) time
100     format ('Time in seconds = ', f10.6)
        end
```

The output from the execution of the previous program *test2* follows.

Results:

```
Time in seconds =          0.000163
```

Local Timing Routines Available on the SMD CONVEX C220

In addition to the standard UNIX timing routines on the SMD CONVEX C220, a set of local routines which provide additional special purpose timing information is provided on the SMD CONVEX C220. All of these routines are contained in the library **/usr/local/lib/libtm.a**, which should be linked with the user's program. Manuals for these routines are not available at the present time.

The CONVEX C200 series of computers has two 64-bit hardware timers, TTR and TOC, which are implemented in microcode [1]. The timers TTR and TOC are accessible to the programmer without the overhead of a system call. The special purpose timing routines described in this section are based on TTR and TOC.

The timer TTR is a 64-bit register that accumulates microseconds of time elapsed since the creation of a thread. The TTR timer allows each thread of a program to be timed separately. The thread timer register is saved on the stack between context switches, allowing the timing of a thread as it migrates between CPUs during its execution. The thread times are reported with microsecond accuracy. The assembly instruction *mov TTR, Sk* atomically reads the TTR information [1]. The local timing routines **getinfo** and **palprof** use this assembly language instruction to measure individual thread times.

The timer TOC is a 64-bit register that accumulates microseconds of time elapsed since the *epoch*, a UNIX construct defining an arbitrary origin point for the beginning of time. It measures elapsed time rather than user time. The TOC timer is neither saved nor restored during context switches, although it is not altered by the context switch, and will keep time indefinitely. The assembly instruction *mov TOC, Sk* atomically reads the TOC information [1]. The local timing routines **getinfo, palprof,** and **toc** use this assembly instruction to measure elapsed time.

1. getinfo

The timing routine **getinfo** is a simple assembly language routine listed in the CONVEX documentation to the FORTRAN Version 5.0 compiler [3]. This routine allows the user to access the hardware timers and the thread and CPU identification registers. The subroutine returns four pieces of information: the thread identification number (TID), the CPU identification number (CPUID), the amount of time the current thread has been executing (the TTR register), and the elapsed time (the TOC register). Each of these values is returned as a 64-bit integer value. The TID specifies which thread is running. The number of threads which are created is an operating system configuration variable and is set equal to the number of CPUs; therefore, threads may be numbered 0 and 1 on the SMD CONVEX C220. The CPUID specifies the physical CPU processing the current thread. The routine **getinfo** is intended to be used within a section of parallel code. Two calls to **getinfo**, one at the beginning, and one at the end of a parallel section of code, allows the measurement of the time each thread uses to execute the intervening code. The arguments to the call to **getinfo** are global variables. The routine **getinfo** places the returned values in these addresses. Because these global arguments are not thread specific, the same argument should not be passed to parallel invocations of **getinfo**; array indices should be passed instead.

The following code section is the assembly source code for the routine:

```
;SUBROUTINE GETINFO (TID, CPUID, TTR, TOC)
;INTEGER*8 TID, CPUID, TTR, TOC

        .fpmode native
        .text
        .globl  _getinfo_
```

```
_getinfo_:
        psh.l   s0              ;       Save s0
        and     #0, s0          ;       Clear s0
        mov     TID, s0         ;       Get thread id
        st.l    s0,@0(ap)       ;       Return TID
        mov     CPUID, s0       ;       Get physical CPUID
        st.l    s0,@4(ap)       ;       Return CPUID
        mov     TTR, s0         ;       Get total thread time
        st.l    s0,@8(a0)       ;       Return TTR
        mov     TOC,s0          ;       Get elapsed time
        st.l    s0,@12(ap)      ;       Return TOC
        pop.l   s0              ;       Restore s0
        rtn
```

The following examples illustrate the use of **getinfo**. The first example is a short FORTRAN program which contains calls to **getinfo**.

Example:

```
        Program test3

C       tid     Thread ID Value
C       cid     CPU ID Value
C       toc     Elapsed Time
C       ttr     Thread Time

        integer *8 tid(5), cid(5,2), tod(5,2), ttr(5,2)
        integer *8 total_ttr, total_toc

C$DIR FORCE_PARALLEL
        do 10 i = 1,5
                call getinfo (tid(i), cid(i,1), ttr(i,1), toc(i,1))
                A = 5.0 + 5.0
                call getinfo (tid(i), cid(i,2), ttr(i,2), toc9(i,2)
10      continue
C       Write output
C       Note:  cid(i,1) is the CPU on which the thread begins execution
C              cid(i,2) is the CPU on  which the thread ends execution.
C              These values may differ
C       Thread time and elapsed time are reported in microseconds.
        do 20 i = 1,5
                total_ttr = ttr(i,2) - ttr(i,1)
                total_toc = toc(i,2), - toc(i,1)
                write (*, 200) i, tid(i), cid(i,1), cid(i,2),total_ttr, total_toc
20      continue
100     format ('Loop  Thread  CPU  Thread_time  Elapsed_time')
200     format (i3, i8, i4, i2, i10, i10)
end
```

The next example illustrates the command to compile a program, (*e.g.*, the program *test3.f*), containing calls to **getinfo** and to link it with the library of timing routines containing the executable code for **getinfo**, */usr/local/lib/libtm.a*. The *-o* option directs the compiler to name the excutable program *test3*, while the *-O3* option directs the compiler to attempt to optimize the code, using both vectorization and parallelization.

Example:

```
fc -o test3  -O3 test3.f /usr/local/lib/libtm.a
```

The following are results from the program *test3* with times reported in microseconds. The first column under *CPU* is the processor on which the thread begins execution, while the second column under *CPU* is the processor on which the thread ends execution.

Results:

Loop	Thread	CPU	Thread_time	Elapsed_time
1	0	1 1	7	6
2	1	0 0	5	4
3	0	1 1	4	4
4	0	1 1	5	5
5	0	1 1	4	4

The routine **getinfo** should not be called outside parallel code because of the ASAP mechanism by which the hardware handles the creation and deletion of multiple threads when parallel code is entered and exited. The thread which begins every program is numbered thread 0. When thread 0 executes a *spawn* instruction to begin a parallel section of code, it posts a fork indicating a need for more threads to enter the execution of the process. The first thread to finish its work executes a *join* instruction and posts a notice that no more threads are needed. As each thread finishes, it executes a *join* instruction. The last thread which executes a *join* clears the fork and continues as a sequential process. The thread number of this continuing thread remains the same as it was in the parallel portion of the code; thus the thread exiting the section of parallel code may not have the same thread number as the thread entering the section of parallel code. Since the thread times are maintained on a thread-specific basis, the thread time of the exiting thread is not related to the thread time of the entering thread.

Ambiguous results can occur if calls to **getinfo** occur outside parallel code. The following examples show the effect of measuring thread times before and after a section of parallel code with **getinfo** . The first example is the FORTRAN code for a sample program, *testgetinfo*.

Example:

```
      Program testgetinfo

C     tid     Thread ID Value
C     cid     CPU ID Value
C     toc     Elapsed Time
C     ttr     Thread Time

      integer *8 tid(1,2), cid(1,2), toc(1,2), ttr(1,2)
      integer *8 total_ttr, total_toc

      call getinfo (tid(1,2), cid(1,1), ttr(1,1), toc(1,1))

C$DIR FORCE_PARALLEL
      do 10 i = 1,5
            A = 5.0 + 5.0
10    continue
      call getinfo (tid(1), cid(1,2), ttr(1,2), toc9(1,2)

C     Write output
C     Note:  tid(1,1) is the thread number which enters the parallel section.
C            tid(1,2) is the thread number which exits the parallel section.
C            cid(1,1) is the CPU on which the thread begins. execution
C            cid(1,2) is the CPU on  which the thread ends execution.
C            These values may differ
C     Thread time and elapsed time are reported in microseconds.

      total_ttr = ttr(1,2) - ttr(1,1)
      total_toc = toc(1,2), - toc(1,1)
      write (*, 100)
      write (*, 200) 1, tid(1,1), tid(1,2),  cid(1,1), cid(1,2),total_ttr, total_toc
100   format ('Loop        Thread-Enter Thread-Exit   CPU   Thread_time
     1 Elapsed_time')
200   format (i3, i8, i14, i2, i10, i10)
end
```

The following results are the output from several executions of the program *testgetinfo*. Note that **getinfo** performs correctly when the thread entering and exiting the parallel code is the same, but that meaningless negative times are reported if the entering and exiting thread numbers are different. Meaningless times occur when the exit time in the TTR register for Thread 1 is subtracted from the entrance time in the TTR register for thread 0, for example. Times are reported in microseconds. The first colum under *CPU* is the processor on which the thread begins execution, while the second column under *CPU* is the processor on which the thread ends execution.

Results:

Loop	Thread-Enter	Thread-Exit	CPU		Thread_time	Elapsed_time
1	0	0	0	0	36	257
1	0	0	1	1	38	272
1	0	1	0	1	-2861	696

2. toc.

The routine **toc** is an assembly language routine which is a subset of **getinfo**†. It returns elapsed time,the contents of the TOC register, as a 64-bit integer value. Time is reported in microseconds. The following code section is the assembly language code for the routine.

```
;SUBROUTINE TOC (TOC)
;integer *8 TOC

        .fpmode native
        .text
        .globl  _toc_

_toc_:
        psh.l   s0              ;       Save s0
        and     #0, s0          ;       Clear s0
        mov     TOC, s0         ;       Get elapsed time
        st.l    s0,@0(ap)       ;       Return TOC
        pop.l   s0              ;       Restore s0
        rtn
```

The following example illustrates the call to **toc** in a FORTRAN program, *test4*.

Example:

```
        Program test4
        integer *8 toc1, toc2
        real *8 A(100)

        call toc(toc1)
        do 10 i = 1,100
                A(i) = 5.0 * i
10      continue
        call toc(toc2)
        write (6,100) toc2 - toc1
100     format ('Total elapsed time = ', i20)
        end
```

† **toc** was written by the technical advisers at CONVEX Computer Corporation.

The following result is the output from the execution of the program *test4*. Time is reported in microseconds.

Results

```
Total elapsed time =          1249
```

3. second

The function **second** is a C language function callable from FORTRAN†. This routine returns user time, the same value returned by **etime**, with microsecond accuracy. Like **etime**, in programs compiled with the **-O3** option, the user time reported by **second** is the sum of the thread times for the process. Time is reported in seconds.

The following code section is the C code for the routine.

```
float second_()
{
        long getrusage ();
        struct rusage rusage;
        float x, y;
        getrusage (RUSAGE_SELF, &rusage);
        x = ((float) rusage.ru_exutime.tv_usec) * 0.000001;
        y = ((float) rusage.ru_exutime.tv_sec) + x;
        return (y);
}
```

The following example illustrates the use of **second** in a FORTRAN program, *test5*.

Example:

```
        Program test5

        real *8 A(100)
        real *4 t1, t2

        t1 = second ()
        do 10 i = 1,100
                A(i) = 5.0 * i
10      continue
        t2 = second ()
        write (6,100) t2 - t1
100     format ('Total user time (seconds) = ', f10.6)
        end
```

† The function **second** was written by the technical advisers at CONVEX Computer Corporation. It has been incorporated into the Force programming language [9] as **ctime** by Greg Astfalk of CONVEX Computer Corporation.

The following results is the output from the execution of the FORTRAN program, *test5*. Time is reported in seconds.

Results:

Total user time (seconds) = 0.000161

4. palprof, palinit.o , palprof.o, palsum.o

The four routines **palprof**, **palinit.o**, **palprof.o**, and **palsum.o** together form a programming tool whose function is to return a user time for parallel code which reflects the maximum thread time for a parallel section of code rather than the sum of the thread times as reported by **etime**†. This new user time, also referred to as *corrected user time*, represents the theoretical best-case time which can be achieved by a particular parallel algorithm on the CONVEX.

The **/bin/csh** script called **palprof**, which is located in **/usr/local/bin** on the SMD CONVEX C220, performs much of the code modification needed to report *corrected user time*. This script compiles the user's FORTRAN program with the **-S** and **-O3** options. The **-O3** option produces code with parallelization and vectorization; the **-S** option produces assembly code from the FORTRAN code. The routine **palprof** then uses *awk* to insert assembly code timing routines around each *spawn* and *join* instruction which resulted from the parallelization of FORTRAN *DO* loops. The compilation is then continued on the expanded assembly language code.

The subroutine object modules, **palinit.o**, **palprof.o** and **palsum.o** , are contained in the library **/usr/lib/libtm.a.** The subroutine **palinit** sets up the environment for the timing of a subroutine and calls **etime** to establish the starting time for the subroutine. The subroutine **palprof** returns individual thread times, total CPU time (sum of the thread times), and elapsed time for each *DO* loop in the timed subroutine. The subroutine **palsum** reports the user time as reported by **etime** for the entire timed subroutine, the total user time for the parallel loops in the timed subroutine, the sum of the longest thread times for all parallel loops in the timed subroutine, and a *corrected user time* for the timed subroutine. This *corrected user time* is the user time for the entire timed subroutine reported by **etime** minus the total user time for all parallel loops in the timed subroutine plus the sum of the longest thread times for all parallel loops in the timed subroutine.

The user must take the following steps to use the parallel profiling routines.

A call to **palinit** must be added before a call to a subroutine which contains parallel *DO* loops and a call to **palsum** must be added after a call to a subroutine which contains parallel *DO* loops. In the following example, *doloop* is a subroutine which contains several parallel *DO* loops.

---

† The routines **palprof** and **palprof.o** were written by Brad Funkhouser of CONVEX Computer Corporation. The routines **palinit** and **palsum** were written by Mary Ann Bynum of PRC/Kentron.

Example:

```
call palinit
call doloop
call palsum
```

A call to **palprof** must be added after each parallel *DO* loop in the called subroutine which the user wishes to time. The following example is one of the *DO* loops in the subroutine *doloop*. In this example, *name* is the name of the subroutine in which the loop appears or any other identifier which the user wishes to use. It is a FORTRAN character string. Each loop timed with a call to **palprof** within a subroutine is automatically given an integer identification number beginning with 1.

Example:

```
      do 10 i = 1, 10000
            (computation)
10    continue
      call palprof ('name')
```

After including the proper calls to **palinit**, **palprof** and **palsum**, the user must compile the programs containing the parallel timing routines with the script **palprof**. The results of executing the following example is a file, *test.o*, which may be linked with other files to produce an executable object file.

Example:

```
palprof test6.f
```

As a final step, the user must build the final executable program by linking the results of the compilation step, *test.o*, with the library **/usr/local/lib/libtm.a**. The *-o* option directs the compiler to name the excutable program *test6*.

Example:

```
fc -o test6 test6.o /usr/local/lib/libtm.a
```

In order to ensure that both processors are available when the parallel code is run, the user's program should be executed using the **mpa** utility with the **-f** option. Since one processor will wait for the other to be free to execute parallel code, this utility often results in an increase in elapsed time. An example of executing the program *test6* with the **mpa** option follows.

Example:

```
mpa -f test6
```

The results of executing program *test6* with the **mpa** utility follow. In the comments that follow the output for the individual loops, the *CPU Time for Parallel Loops in Subroutine xxxxx* is equal to the user time reported by **etime**, and the *Max Thread Time for Loops in Subroutine xxxxx* is the sum of the longest threads in all parallel loops within the subroutine. As stated earlier, the *Corrected CPU Time for Subroutine xxxxx using Max Time* is the user time for the entire timed subroutine reported by **etime** minus the total user time for all parallel loops in the timed subroutine plus the sum of the longest thread times for all parallel loops in the timed subroutine. Times are reported in seconds with microsecond accuracy.

Results:

```
---------------
      Doloop:          1
Thread 0:      0.8482180
Thread 1:      0.8397850
Thread 2:      0.0000000
Thread 3:      0.0000000
Total CPU:     1.6880030
Wallclock:     1.0057740
---------------
      Doloop:          2
Thread 0:      0.6571360
Thread 1:      0.6503410
Thread 2:      0.0000000
Thread 3:      0.0000000
Total CPU:     1.3074770
Wallclock:     0.7705900
---------------
        Work:          1
Thread 0:      0.0002490
Thread 1:      0.0052300
Thread 2:      0.0000000
Thread 3:      0.0000000
Total CPU:     0.0054790
Wallclock:     0.0265080

CPU Time for Parallel Loops in Subroutine Doloop  :          2.9954801
Time for Subroutine Doloop via etime (usr time):             2.9994712
Max Thread Time for Loops in Subroutine Doloop  :            1.5053540
Corrected CPU Time for Subroutine Doloop  using Max Time:    1.5093452

CPU Time for Parallel Loops in Subroutine Work :             0.0054790
Time for Subroutine Work via etime (usr time):               0.0070379
Max Thread Time for Loops in Subroutine Work  :              0.0052300
Corrected CPU Time for Subroutine Work  using Max Time:      0.0067889
```

The parallel profiling package can only time parallel *DO* loops in subroutines called from the immediately higher level. It cannot be used to time nested calls to subroutines in which the timed *DO* loops are more than one nesting level away.

System Level Foundations for Timing Routines

Both the standard UNIX timing routines and the local timing routines access software data structures which are maintained by the operating system for each particular process and any processes which that particular process has created (child processes). The programmer who wishes to incorporate information about process resource utilization may access these data structures through the system routines **getrusage** and **cvxprusage**.

1. getrusage

The system function **getrusage** is described in the *CONVEX UNIX Programmer's Manual* under **getrusage(2)** [5]. The function may be called in a C program and, if successful, returns a pointer to a buffer containing process resource utilization information. The process resource utilization information consists of user time used, system time used, shared memory size, unshared data size, unshared stack size, page reclaims, page faults, swaps, block input operations, block output operations, messages sent, messages received, signals received, voluntary context switches, involuntary context switches, and user time used reported with microsecond accuracy. Standard UNIX routines such as **/bin/csh time** and **etime**, and local timing routines such as **second** are built on the system routine **getprusage**.

2. cvxprusage

The system function **cvxprusage** is described in the *CONVEX UNIX Programmer's Manual* under **cvxprusage(2)** [5]. The function may be called in a C program and, if successful, returns a pointer to a buffer containing information about parallel resource utilization by the current process. This information consists of user time in microseconds, system time in seconds, cumulative number of threads in user and system space, and the frequency of sampling user and system space for the number of threads. The system samples the number of threads in a process at a fixed rate, currently 100 times a second, and continually updates the parallel process information buffer. A estimate of user level parallelization can be obtained by dividing the cumulative number of threads in user space by the number of times the system sampled user space. This calculation underlies the *Concurrency Factor* reported by **/bin/csh time** †

Summary

The SMD CONVEX C220 has a group of timing routines which allow users to dissect the efficiency of their sequential, vector or parallel computer programs. Standard UNIX

† See **Standard UNIX Timing Routines 2. /bin/csh/time** in this paper.

routines allow the user to compare the results of executing his program on the CONVEX C220 with other UNIX machines without making changes to his code. They also allow him to measure the total work done by the CPUs in the execution of programs containing parallel sections. Local timing routines on the CONVEX C220 allow the user to monitor the execution of a parallel program and measure the execution time of parallel sections, both as the sum of the thread times and as the sum of the times of the longest threads for each parallel section. In this way, the time to execute sequential, vector and parallel versions of the same code can be compared and the benefits of parallelization measured.

References

1. Anon.: *CONVEX Architecture Reference.* Third Edition. CONVEX Computer Corporation, Richardson, Texas, October, 1988.

2. Funkhauser, B.: *Understanding the CONVEX Parallel System: Application Note.* CONVEX Computer Corporation, Richardson, Texas, September, 1988, p. 1-1.

3. Anon.: *FORTRAN Parallelization.* Application Note. CONVEX Computer Corporation, Richardson, Texas, November, 1988.

4. Anon.: *CONVEX UNIX Programmer's Manual Part I, Section 1: Commands and Application Programs.* Eighth Edition. CONVEX Computer Corporation, Richardson, Texas, October, 1988.

5. Anon.: *CONVEX UNIX Programmer's Manual Part II, Section 2: System Calls.* Eighth Edition. CONVEX Computer Corporation, Richardson, Texas, October, 1988.

6. Anon.: *CONVEX UNIX Programmer's Manual Part II, Section 3F: FORTRAN Library Subroutines.* Eighth Edition. CONVEX Computer Corporation, Richardson, Texas, October, 1988.

7. Anon.: *CONVEX UNIX Programmer's Manual Part II, Section 3C: Compatibility Library Subroutines.* Eighth Edition. CONVEX Computer Corporation, Richardson, Texas, October, 1988.

8. Anon.: *CONVEX VECLIB User's Guide.* Third Edition, Revision 1. CONVEX Computer Corporation, Richardson, Texas, March, 1988.

9. Jordan, H. F.: The Force. L.H. Jamieson, D. Gannon, and R. J. Douglas (editors), *Characteristics of Parallel Algorithms,* Chapter 16, MIT Press, Cambridge, Massachusetts, 1987.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181950		

4. Title and Subtitle	5. Report Date
CPU Timing Routines for a CONVEX C220 Computer System	December 1989
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Mary Ann Bynum	
9. Performing Organization Name and Address	10. Work Unit No.
PRC Kentron, Inc. Aerospace Technologies Division 303 Butler Farm Road Hampton, VA 23605	505-63-01-10
	11. Contract or Grant No.
	NAS1-18000
12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	Contractor Report
	14. Sponsoring Agency Code

15. Supplementary Notes
NASA Langley Research Center Technical Monitor: Ronnie E. Gillian

16. Abstract

The purpose of this report is to describe the timing routines available on the CONVEX C220 computer system in the Structural Mechanics Division (SMD) at NASA Langley Research Center. The report describes the function of the timing routines, the use of the timing routines in sequential, parallel, and vector code, and the interpretation of the results from the timing routines with respect to the CONVEX model of computing. The timing routines available on the SMD CONVEX fall into two groups. The first group includes standard timing routines generally available on computers with UNIX 4.3 BSD operating systems, while the second group includes routines unique to the SMD CONVEX. The standard timing routines described in this report are **/bin/csh time**, **/bin/time**, **etime**, and **ctime**. The routines unique to the SMD CONVEX are **getinfo**, **second**, **cputime**, **toc**, and a parallel profiling package made up of **palprof**, **palinit**, and **palsum**.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
CPU Timing Routines CONVEX C200 CPU Timing Routines Parallel CPU Timing Routines	Unclassified—Unlimited Subject Category 61

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	21	A03

NASA FORM 1626 OCT 86
For sale by the National Technical Information Service, Springfield, Virginia 22161-2171