FORTRAN Parallelization

Application Note

Document No. 720-001718-200

November 1988

CONVEX Computer Corporation

Richardson, Texas

Printed in the United States of America

Introduction

With the release of the CONVEX FORTRAN V5.0 compiler, you gain the power of parallel processing. This paper provides information you need to use that power successfully: what the compiler does for you, what you can tell it to do, and the possible effects on how your programs and your system perform.

Parallelization differs from vectorization in that it does not reduce CPU usage. Instead, it spreads processing of a single program across multiple CPUs, improving turnaround or "wall clock" time for that program.

The FORTRAN V5.0 compiler, invoked with the *-O3* option, generates code that allows your program to be executed by as many CPUs as are available when the program runs. It is important to note that the generation of parallel code alone does not guarantee parallel execution. You can prevent or suppress parallel processing by

- recompiling your program at a lower level of optimization
- using the C shell *limit* command
- executing your program under *mpa*, with the *-t* option

The CONVEX C200 Series machine is designed so that each CPU seeks out and executes the next available piece of work, as soon as possible. Known as ASAP (Automatic Self-Allocating Processors), this scheme optimizes system use and throughput in multiuser environments by preventing the wasted CPU cycles characteristic of other multiple-CPU systems. The hardware performs CPU allocation without incurring the overhead of a system call.

The pieces of work that a CPU performs are streams of execution, called threads. A program always incorporates one thread, and may incorporate many. True parallel processing occurs when different CPUs simultaneously execute the separate threads of a single program.

CONVEX UNIX V7.0 implements a thread scheduler to act as an interface between the hardware's creation of threads and the operating system's own allocation of CPUs to execute them. The operating system assigns each process a priority and places it in a run queue. Every second, the operating system recalculates the priority of each process, taking into account the accumulated CPU time of all the process' threads. Processes that consume great amounts of CPU time are assigned a lower priority, so that other processes can move up in the run queue.

The FORTRAN V5.0 compiler generates parallel code that is independent of the number of CPUs available. It identifies parallelizable loops by constructing a flow control graph, based on dependency analysis.

A loop is parallelizable if there are no dependencies between iterations. A cross-iteration dependency, or loop carried dependency (LCD), exists when a calculation in one iteration of a loop depends on the result of a previous iteration. You cannot perform two calculations in parallel if one calculation depends on the result of the other.

The CONVEX FORTRAN V5.0 compiler is designed to identify dependencies, protecting the integrity of your data. If the compiler cannot determine whether a loop can execute safely in parallel, it does not generate parallel code. Instead, it includes in the optimization report its reason for not parallelizing the loop.

To avoid errors associated with parallelization, you need to understand the nature of your code and data. Be sure that what you think is independent really is. Debug your program with little or no optimization and check your output carefully. Recompile at a higher level and check again. Continue this process of incremental optimization, checking at each level that your program produces valid output. Debugging your program at increasing levels of optimization lets you know at what level an error is introduced and may help prevent some errors along the way.

The CONVEX compiler includes a set of directives with which you can enhance its ability to identify dependencies, and even override its judgment regarding a given segment of code. The only danger in telling the compiler to ignore any detected or assumed dependencies is that the compiler may have been right.

If you alter the compiler's behavior in the mistaken belief that no data dependency exists, the resulting code is nondeterministic. That is, you may get good answers one time your program runs and bad answers the next. This unpredictability exists because the errors are order-dependent, and you cannot predict the execution order of your program's parallel threads.

For this reason, it is always safer to let the compiler choose which code segments to parallelize. That is its job.

Automatic parallelization

The CONVEX FORTRAN V5.0 compiler performs several transformations on your program, taking every opportunity to enhance parallel performance. The most important transformations are reordering transformations. Reordering transformations do not eliminate operations from a program or replace them with simpler operations, but rearrange them so they can be executed more efficiently.

Reordering transformations performed and orchestrated by the FORTRAN V5.0 compiler, when invoked at optimization level 3, include the following:

- vectorization
- partial vectorization
- parallelization
- loop interchange
- loop distribution

The compiler automatically parallelizes the outermost loop in a nest (if it can be parallelized), including the strip-mine loop created when a loop has been vectorized. The compiler also distributes and interchanges loops so it can generate parallel code for the outer loop. Most scalar reductions and assignments can be parallelized, at the cost of some additional synchronization code.

An example

Consider a common algorithm for the matrix multiplication C=A*B (for N by N arrays A, B, and C).

```
DO I=1,N
  DO J=1,N
    C(I,J)=0
    DO K=1,N
      C(I,J)=C(I,J)+A(I,K)*B(K,J)
    ENDDO
  ENDDO
ENDDO
```

The compiler processes this loop nest by first distributing the I and J loops.

```
DO I=1,N
  DO J=1,N
    C(I,J)=0
  ENDDO
ENDDO

DO I=1,N
  DO J=1,N
    DO K=1,N
      C(I,J)=C(I,J)+A(I,K)*B(K,J)
    ENDDO
  ENDDO
ENDDO
```

It then moves the I loop to the innermost position in each nest, so that contiguous storage elements can be vectorized.

```
DO J=1,N
  DO I=1,N
    C(I,J)=0
  ENDDO
ENDDO

DO J=1,N
  DO K=1,N
    DO I=1,N
      C(I,J)=C(I,J)+A(I,K)*B(K,J)
    ENDDO
  ENDDO
ENDDO
```

Both I loops are now strip mined to the optimal vector length (some function of N). MVSL represents that function. In the second nest, the induction variable IOUTER is recognized as loop-invariant with respect to the K loop, so the compiler interchanges those two loops.

```
DO J=1,N
  M=MVSL(N)
  DO IOUTER=1,N,M
    DO I=IOUTER,MIN(N,IOUTER+M-1)
      C(I,J)=0
    ENDDO
  ENDDO
ENDDO

DO J=1,N
  DO IOUTER=1,N,M
    DO K=1,N
      DO I=IOUTER,MIN(N,IOUTER+M-1)
        C(I,J)=C(I,J)+A(I,K)*B(K,J)
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Now the outer loops are parallelized, and vector code is generated for the inner nests. V0 and V1 represent vector registers that can contain up to 128 64-bit elements.

```
DOALL J=1,N
  M=MVSL(N)
  DO IOUTER=1,N,M
    C(IOUTER:MIN(N,IOUTER+M-1),J)=0
  ENDDO
ENDDO

DOALL J=1,N
  DO IOUTER=1,N,M
    DO K=1,N
      V0=C(IOUTER:MIN(N,IOUTER+M-1),J)
      V1=A(IOUTER:MIN(N,IOUTER+M-1),K)
      V0=V0+V1*B(K,J)
      C(IOUTER:MIN(N,IOUTER+M-1),J)=V0
    ENDDO
  ENDDO
ENDDO
```

Finally, global vector register allocation removes a vector load and a vector store from the K loop. The remaining reference to vector V1 chains with the vector addition and vector multiplication in the next statement, resulting in an even greater speedup (roughly 2.5 times).

```
DOALL J=1,N
  M=MVSL(N)
  DO IOUTER=1,N,M
    C(IOUTER:MIN(N,IOUTER+M-1),J)=0
  ENDDO
ENDDO

DOALL J=1,N
  DO IOUTER=1,N,M
    V0=C(IOUTER:MIN(N,IOUTER+M-1),J)
    DO K=1,N
      V1=A(IOUTER:MIN(N,IOUTER+M-1),K)
      V0=V0+V1*B(K,J)
    ENDDO
    C(IOUTER:MIN(N,IOUTER+M-1),J)=V0
  ENDDO
ENDDO
```

The following table summarizes the effect of these transformations:

Extent of Optimization	Time
global optimization	1.000
+ vectorization	0.107
+ vector register allocation	0.046
+ parallelization (4 CPUs)	0.012

Without programmer intervention, the compiler generates code that runs up to 83 times as fast (with all four CPUs) as the scalar version.

The optimization report

The compiler generates an optimization report for each program unit. This report consists of a loop table, an array table, or both.

For example, consider the matrix multiplication loop just described, with N=200:

```
 1      PROGRAM EXAMPLE1
 2      REAL A(200,200), B(200,200), C(200,200)
 3
 4      DO I=1,200
 5        DO J=1,200
 6          C(I,J)=0
 7          DO K=1,200
 8            C(I,J)=C(I,J)+A(I,K)*B(K,J)
 9          ENDDO
10        ENDDO
11      ENDDO
12
13      END
```

The following screen shows the optimization report output from compilation of the above program. No array table is generated for this program.

```
% fc -O3 example1.f

            Optimization by Loop for Routine EXAMPLE1

Line     Iter.   Reordering            Optimizing / Special      Exec.
Num.     Var.    Transformation         Transformation            Mode
-------------------------------------------------------------------------
   5-1   I       FULL VECTOR Dist Inter
   5-2   I       FULL VECTOR Inter
   6-2   J       PARALLEL
   6-2   J       PARALLEL
   8-2   K       Outer loop

Line     Iter.   Analysis
Num.     Var.
-------------------------------------------------------------------------
   5-1   I       Interchanged to innermost
   5-2   I       Interchanged to innermost

%
```

The loop table lists the optimizations performed on each loop and consists of two parts. Note that the line numbers in this example consist of two numerals, separated by a hyphen, indicating that the loop was distributed. The first numeral is the actual line number; the second indicates which of the resulting loops (distributends) contains the line being described.

The first part of the loop table shows that the following transformations were performed:

- the I loop at line 5 was distributed, then interchanged and fully vectorized in both distributends
- the J loop in the second distributend was parallelized
- the K loop in the second distributend became an outer loop

The second part of the loop table provides specific information about the transformations performed on the I loop.

As an example of other transformations performed by the compiler, the following screen shows a simple subroutine and the optimization report generated by the compiler.

```
% cat -n example2.f

 1        PROGRAM EXAMPLE2
 2        INTEGER*8 A(512),SUM
 3
 4        DO I = 1,512
 5           SUM = SUM + A(I)
 6        ENDDO
 7
 8        END

% fc -O3 -re -c example2.f


            Optimization by Loop for Routine SUB

Line      Iter.    Reordering            Optimizing / Special      Exec.
Num.      Var.     Transformation         Transformation            Mode
-------------------------------------------------------------------------
   4      I        PARA/VECTOR            Reduction                 S

%
```

The "S" in the "Exec. Mode" column indicates that the compiler creates a scalar version of the loop, as well as a parallel/vector version. "Reduction" (in the adjacent column) indicates that the compiler recognizes the reduction in the I loop, and parallelizes it. On this four-CPU system, the compiler strip mines the loop into four parallel strips of 128 iterations each, which it then vectorizes. In execution, information from the processing of each vector strip accumulates in a communication register, from which SUM takes its final value.

Analyze the optimization report to find the parallelized segments of your program. Based on your knowledge of the algorithms and data, add directives to tune that parallelization or to override the compiler's decisions.

User-assisted parallelization

One group of parallelization directives allows you to provide the compiler with additional information that may supplement or enhance its ability to generate parallel code. You can use the *-ep* option and the *VSTRIP* and *PSTRIP* directives to tune parallelization.

Expected processors

At compile time, the compiler detects the number of CPUs on your system. Known as ep, for "expected processors," this value is used to calculate the optimal vector and parallel strip lengths. You can alter the compiler's selection by specifying the expected number of CPUs with the *-ep* option. This allows you to compile a program on one machine (a two-CPU system, for example) that you intend to run on another (a four-CPU system, for example).

The purpose of parallelization is to maximize program performance. By increasing the number of CPUs that can work on one program, however, you increase the likelihood that those CPUs will be unavailable for other programs. So, you might use the *-ep* option to limit the number of CPUs that can participate in the execution of your program.

For instance, a side effect of using *-ep1* is that the compiler does not perform variable vector strip mining or parallel strip mining. The program does not run at maximum performance, but total system throughput improves.

Vector strip mining

The compiler calculates the maximum vector strip length (mvsl), according to the following formula:

for ep=1, mvsl=128
for ep>1, mvsl=max(min((n+ep-1)/ep,128),8)

where n is the actual length of the vector (i.e., the loop trip count) and 128 is the size of the vector register. The following table shows the maximum vector strip lengths calculated and the strip lengths actually processed on a four-CPU system for vectors of various lengths:

Trip Count	MVSL	Actual Strip Length(s) Processed
3	8	3 (one iteration)
100	25	25, 25, 25, 25 (four iterations)
514	128	128, 128, 128, 128, 2 (five iterations)

Notice that the actual strip length for a given iteration is the optimal value (the mvsl) or the number of vector elements left to process.

If you believe that you can improve the efficiency this formula provides, you may override the compiler's calculation for a specific loop by applying the *VSTRIP(n)* directive. This directive tells the compiler to vectorize the immediately following loop with vectors of length n.

For example, on a two-CPU system, the compiler normally chooses a vector strip length of 64 for the following loop:

```
DO I=1,128
   A(I)=0
ENDDO
```

Preceding this loop with the line

```
C$DIR VSTRIP(32)
```

tells the compiler to use a vector length of 32, thereby creating a four-iteration parallel strip-mine loop, which in turn allows up to four CPUs to participate in executing the loop. In practice, if you expected to run the program on a four-CPU system, you would probably compile the program with the *-ep4* option.

The *VSTRIP* directive does not force vectorization. If you apply it to a non-vector loop, the compiler generates a warning message and ignores the directive.

Parallel strip mining

A parallelizable loop consists of multiple iterations that can execute independently. Each thread involved in executing the loop performs a certain number of iterations. The compiler calculates the size of those iteration blocks with the intent of spreading the work evenly across the available CPUs and of minimizing the time spent binding threads and CPUs.

The compiler calculates the maximum parallel strip length (mpsl) according to the following formula:

for ep=1, mpsl=1
for ep>1, mpsl=max(n/(2*ep),1)

where n is the actual length of the vector (trip count).

The *PSTRIP(n)* directive tells the compiler to generate parallel code, in strips of length n, for the immediately following loop. You would apply the *PSTRIP* directive to a large-bodied loop if the trip count of the loop typically falls in a known range. For instance, the compiler may be unable to determine the trip count of the following loop:

```
DO I=1,N
   .
   .
   .

ENDDO
```

If you know that N is usually around 1000 — perhaps 90% of the time — you would set the parallel strip length to 250 (assuming a four-CPU system). You would do so by preceding the loop with the line

```
C$DIR PSTRIP(250)
```

Unnecessary overhead is incurred when N is far from 1000; however, if the body of code in the loop is large enough, that overhead is offset by the improved load-balancing and reduced overhead more frequently achieved.

The *PSTRIP* directive does not force parallelization. If you apply it to a non-parallel loop, the compiler generates a warning message and ignores the directive.

User-specified parallelization

Another group of directives allows you to override the compiler's judgment about whether to parallelize a particular code segment. You can use the *FORCE_PARALLEL* directive, the tasking directives, and the *SYNCH_PARALLEL* directive to control parallelization.

The compiler does not automatically parallelize loops that contain subroutine calls or function invocations. Arguments of subroutine calls, other than induction variables, are passed as global data addresses. Parallel invocations of a given subroutine are not independent if they manipulate the same global data. Because the compiler cannot determine whether the called subroutine manipulates global data, it does not parallelize such loops.

Forcing parallelization

If the compiler is unable to determine whether a particular loop can be parallelized safely, it assumes that it cannot. If you know that the subroutine call does not jeopardize data independence, you can direct the compiler to behave against its better judgment. To do so, you must recompile the called subroutine for reentrancy, using the *-re* option. Compiled in this way, each invocation of the subroutine maintains a thread-private copy of its local data and a thread-private stack to store compiler-generated temporary variables.

You must also tell the compiler to generate parallel code for the loop in which the call occurs, by preceding the loop with the *FORCE_PARALLEL* directive. This directive forces the compiler to parallelize the immediately following loop, regardless of dependencies it may have detected.

Note that, if you force the compiler to parallelize a loop that carries dependencies, the generated code may work every time you test it and fail on its first use. No amount of testing can guarantee the correctness of code generated by forcing the compiler to ignore presumed dependencies. You must analyze your data and algorithms to ensure that the code can be run safely in parallel.

For example, the loop in the following example would not be parallelized automatically because of the subroutine call. The *FORCE_PARALLEL* directive overrides the compiler's decision, and causes it to generate parallel code for the loop.

```
C$DIR FORCE_PARALLEL
      DO I=1,N
         CALL SUB(A,B,I,N)
      ENDDO
          .
          .
          .

      OPTIONS -re
      SUBROUTINE SUB(A,B,I,N)
      REAL A(N),B(N)
      A(I) = B(I)*3.14
      RETURN
      END
```

Because SUB does not contain any operations that violate data independence, it can safely execute in parallel.

If the subroutine is called only from within a parallelized loop, compile it at optimization level 2 rather than 3. Only one loop at a time can be run in parallel, and the loop containing the call will already be doing so. Thus, any parallelizable code within the subroutine will never execute in parallel. The additional code generated by compiling for parallelization is useless overhead.

Parallelizing non-loop code

The compiler does not recognize parallelizable code that occurs outside of a loop. You may use the tasking directives to tell the compiler that certain non-loop sections of code can execute safely in parallel.

The *BEGIN_TASKS* directive tells the compiler to generate parallel code for the immediately following series of tasks. The *NEXT_TASK* directive denotes to the compiler the end of the preceding task and the start of another. The *END_TASKS* directive denotes to the compiler the end of the preceding series of tasks.

For example, the following sequence causes the compiler to construct a parallelizable loop around the three tasks, with control directed to a different task for each iteration.

```
C$DIR BEGIN_TASKS
      <stmt1>
C$DIR NEXT_TASK
      <stmt2>
C$DIR NEXT_TASK
      <stmt3>
C$DIR END_TASKS
```

The code generated is functionally equivalent to the code generated for the following sequence:

```
C$DIR FORCE_PARALLEL
      DO 100, I=1,3
         GOTO(10,20,30),I
  10     <stmt1>
         GOTO 100
  20     <stmt2>
         GOTO 100
  30     <stmt3>
 100  CONTINUE
```

Effective uses of the tasking directives include the initialization of large, independent arrays. The tasked blocks should be of sufficient size to offset the additional overhead incurred by parallelization. It is best if the blocks are also roughly equal in size.

Synchronized parallelization

The turnaround time of certain loops can be decreased by inserting synchronization points to satisfy the cross-iteration dependencies. The *SYNCH_PARALLEL* directive tells the compiler to perform the immediately following loop in synchronized parallel, even though doing so appears to achieve less than full efficiency. Typically, the compiler can generate more efficient code by partially vectorizing such loops.

One reason to use the *SYNCH_PARALLEL* directive is to speed up a benchmark, where turnaround time is the only consideration, and it is known that synchronized parallelization is faster than the partial vectorization chosen by the compiler. For synchronized code to execute efficiently in parallel, the independent (parallel) portion of the loop must be large relative to the dependent (sequential) portion.

The *SYNCH_PARALLEL* directive is currently supported only on the innermost loop in a nest.

Debugging parallelized programs

Debugging should be an integral part of a general development strategy, incremental optimization. In this strategy, you eliminate errors that pertain to a given level of optimization before moving to a higher level. At each level, you check for those errors that arise as a result of greater optimization and for those that were overlooked at the previous level. When you compile your program for parallelization, you have already eliminated most errors at a lower level where they are much easier to find.

Once you are satisfied that your program works correctly in parallel without added directives, use the optimization report to determine what loops have and have not been parallelized. Use your knowledge of the algorithms and data to determine what enhancements you can make by

applying directives. Keep checking your program output as you add directives, making sure that your assumptions about data independence are valid.

The remainder of this section discusses the steps you might follow in debugging your parallelized program and some tools you can use in the process.

A debugging strategy

If your program produces correct answers when compiled at optimization level 2, but not at 3, run the parallelized code serially. To do so, use the C shell command *limit concurrency 1*, or run your program under the *mpa* utility with the *-t1* option (see "The *mpa* utility" below).

If you get correct answers from the serial run of your parallelized code, some dependency has been overlooked. This may suggest a hidden alias (such as an item in COMMON being passed as an argument) or a floating-point round-off discrepancy being propagated into significance.

Any errors at this stage probably result from a misused directive. You are forcing parallelization at an inappropriate point. These errors most often involve a subroutine, compiled for reentrancy and executed in parallel, in which a loop carried dependency exists. Determining reentrancy is difficult, but important when you direct the compiler to act against its judgment.

Another common mistake is applying the *FORCE_PARALLEL* directive to a loop that calls a subroutine, but failing to compile that subroutine (or some subroutine called by it) for reentrancy. A related error is assuming that the local variables of a subroutine compiled for reentrancy are initialized to 0 — the local variables must be initialized explicitly in the subroutine code.

The source-level debugger

Part of the CONVEX CONSULTANT V8.0 package, *csd* provides thread-specific commands to let you monitor the processing of parallel code. You can set process breakpoints, monitor individual threads, examine thread-specific machine registers, and display thread stack backtraces. Use *csd* to help you track down elusive dependencies. For more information on *csd*, see the *CONVEX CONSULTANT User's Guide*, 8th edition.

The *mpa* utility

The *mpa* (modify process attributes) utility sets or modifies the attributes of a process or command. Using the *-f* option, you can cause a command to run as a fixed-schedule job. A fixed-schedule job is always scheduled to run on all CPUs; the job waits until all CPUs are available before executing. Running your program under *mpa* as a fixed-schedule job increases the time your program takes to complete, and you are not advised to use *mpa* as a standard mode of operation.

Fixed-schedule execution under *mpa* is, however, a useful debugging tool. Other than running on a dedicated machine (i.e., standalone), fixed-schedule execution is the only way to guarantee that the parallelized portions of your program run in parallel. This helps you in the development stages to check for errors related to parallel execution.

The *-t* option allows you to limit concurrency for individual executions of your parallelized program by specifying the maximum number of threads your program may entail at any given time. That is, you get the same effect with the command

```
% mpa -t1 a.out
```

as with the series

```
% limit concurrency 1
% a.out
      .
      .
      .
% unlimit concurrency
```

Monitoring parallel programs

The following example shows how you can use simple assembly-language routines to gain access to hardware timers and to thread and CPU identification registers. This information can help you trace some of the effects of parallelization on your program.

```
                                ; 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 CPU id
        st.l    s0,@4(ap)       ; Return CPUID
        mov     TTR,s0          ; Get total thread time
        st.l    s0,@8(ap)       ; Return TTR
        mov     TOC,s0          ; Get wall clock time
        st.l    s0,@12(ap)      ; Return TOC
        pop.l   s0              ; Restore s0
        rtn
```

Routine *getinfo* returns four pieces of information:

TID	Specifies which thread is running. Threads are numbered 0 and 1 on a C220; 0, 1, and 2 on a C230; 0, 1, 2, and 3 on a C240.
CPUID	Specifies the physical CPU processing the current thread. On a C220, CPUs are numbered 0 and 1; on a C230, 0, 1, and 2; on a C240, 0, 1, 2, and 3.
TTR	Indicates the amount of time the current thread has been executing. TTR is a 64-bit register that accumulates microseconds of CPU time used by a given thread.
TOC	Contains the current time. TOC is a 64-bit register that accumulates microseconds of "wall clock" time elapsed since the epoch.

Because *getinfo* contains no local data, you can consider it reentrant. You can call *getinfo* from parallelized loops in your FORTRAN program. The information returned by *getinfo* is placed in the addresses passed to it in the call. Typically, these addresses are shared (not thread-specific), so do not pass the same global argument to parallel invocations of *getinfo*. Pass array indices, instead, as in the following example.

```
      PROGRAM INFO
      INTEGER*8 TID(10)         ! Thread Id Value
      INTEGER*8 CID(10,2)       ! CPU Id Value
      INTEGER*8 TOC(10,2)       ! Wallclock Time
      INTEGER*8 TTR(10,2)       ! Thread Time
      INTEGER*8 TOTAL_TTR, TOTAL_TOC
      REAL A(100000,10), B(100000,10)
      COMMON /DATA/ A,B

C$DIR FORCE_PARALLEL
      DO I=1,10
         CALL GETINFO(TID(I),CID(I,1),TTR(I,1),TOC(I,1))
         DO J=1,100000
            B(J,I)=100*J
            A(J,I)=B(J,I)*3.14
         ENDDO
         CALL GETINFO(TID(I),CID(I,2),TTR(I,2),TOC(I,2))
      ENDDO

      WRITE(*,100)
      DO I=1,10
         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
      ENDDO

  100 FORMAT('Loop Thread CPU Thread_time Wall_time')
  200 FORMAT(i3,i6,i4,i2,i10,i10)

      END
```

The following screen shows how to assemble the *getinfo* routine and compile *info* to link with it. Notice that the compiler warns you of the two potential dependencies it detects in *info*.

```
% as getinfo.s
% fc -O3 info.f getinfo.o
fc: Warning on line 16.6 of info.f: WARNING: IGNORING DEPENDENCE FROM 'A'
 TO 'GETINFO' CAN CAUSE GENERATION OF INCORRECT OBJECT CODE !!
fc: Warning on line 15.6 of info.f: WARNING: IGNORING DEPENDENCE FROM 'B'
 TO 'GETINFO' CAN CAUSE GENERATION OF INCORRECT OBJECT CODE !!


            Optimization by Loop for Routine INFO

Line      Iter.    Reordering            Optimizing / Special       Exec.
Num.      Var.     Transformation         Transformation             Mode
-------------------------------------------------------------------------
  12      I        PARALLEL
  14      J        FULL VECTOR

  22      I        Scalar

Line      Iter.    Analysis
Num.      Var.
-------------------------------------------------------------------------
  22      I        Insufficient vector code

%
```

The following screen shows the output from the execution of the program *info*:

```
% a.out
Loop Thread CPU Thread_time Wall_time
  1      0   0 0      13998     85379
  2      0   0 1      13978     31034
  3      1   1 0      13999     41389
  4      1   0 0      13995     63471
  5      0   1 0      13876     71497
  6      0   0 0      13812     56877
  7      1   0 1      13850     51868
  8      1   1 1      13735     47107
  9      1   1 1      13631     34769
 10      1   1 1      13872     34621

%
```

The threads are numbered 0 and 1, indicating that the maximum number of concurrent threads (for this two-CPU system) was created. The columns under the CPU heading illustrate that a thread may be started by one CPU and finished by another. Processing of a thread by more than one CPU may be the result of a page fault, a system call, or a clock interrupt (which occurs every ten milliseconds), all of which may cause the scheduler to bind threads with CPUs.

This program provides a convenient opportunity to see the effect of limited concurrency on a parallelized program. The following output results from three consecutive executions of the same program under *mpa* with the *-tl* option:

```
% mpa -tl a.out
Loop Thread CPU Thread_time Wall_time
   1      0   0 0      14024      33375
   2      0   0 1      13971      32725
   3      0   1 1      13934      29609
   4      0   1 1      13963      29831
   5      0   1 1      13981      29920
   6      0   1 1      13962      30168
   7      0   1 1      13932      29731
   8      0   1 1      13999      30059
   9      0   1 1      13462      29174
  10      0   1 1      14052      36038

% mpa -tl a.out
Loop Thread CPU Thread_time Wall_time
   1      0   1 1      13990      33414
   2      0   1 1      13892      29832
   3      0   1 1      13940      31053
   4      0   1 1      13895      29900
   5      0   1 1      13928      29894
   6      0   1 1      13889      29840
   7      0   1 1      13887      30166
   8      0   1 1      14000      35362
   9      0   1 1      14058      33081
  10      0   1 1      14919      40033

% mpa -tl a.out
Loop Thread CPU Thread_time Wall_time
   1      0   0 0      14203      34401
   2      0   0 0      14147      30670
   3      0   0 0      14138      30937
   4      0   0 0      14092      30609
   5      0   0 0      14101      30388
   6      0   0 0      14130      30727
   7      0   0 0      14209      34353
   8      0   0 0      14139      30918
   9      0   0 0      14171      31786
  10      0   0 0      14805      38639

%
```

Notice that the thread number is always 0. That is, only one thread is ever created. Either of the two CPUs may execute it, in part or whole. When two threads are created, the second is designated as thread number 1.

Conclusion

The CONVEX FORTRAN V5.0 compiler is well-equipped to optimize your programs for parallel execution with a minimum of programmer intervention. You are least likely to incur errors related to parallelization if you let the compiler follow its built-in evaluation process.

You may enhance that process through the careful application of the available directives. You have several tools available to help you track such errors as they arise, but effective use of the directives is based on your thorough understanding of the data and algorithms of your program.

Glossary

ASAP	Automatic Self-Allocating Processors, a unique architecture designed by CONVEX. A cornerstone of ASAP is the communication register, which allows CPUs to seek out and execute the next piece of work as soon as possible.
critical region	A segment of code that modifies data or computer resources shared by more than one thread.
communication register	A high-speed register used for communication among the threads of a process. Threads communicate by sending and receiving data through the communication registers. A hardware-maintained lock bit is associated with each communication register; the lock bit guarantees mutually exclusive access to the register.
concurrent	In parallel processing, threads that may execute at the same time are concurrent.
CPU	A physical processing unit. Each CPU in the configuration operates independently as a 64-bit CONVEX supercomputer.
dependency	A relationship between two statements, such that one statement must precede the other to produce the intended result.
distributend	A loop generated by the compiler in the process of loop distribution.
execution stream	A series of instructions executed by a CPU.
granularity	The amount of work executed by a separate thread, between the time it is created and the time it terminates. Granularity ranges from half the entire program (coarse), to the single iterations of a loop (fine), to individual source statements (very fine). The overhead, or system time required to create and manage multiple threads, determines the granularity of parallelization that is profitable.
loop distribution	The restructuring of a loop nest to create additional innermost loops and to enhance opportunities for loop interchange.

loop carried dependency (LCD)	A dependency between two operations executed on different iterations of a given loop and on the same iteration of all enclosing loops. The outermost loop for which the iterations are different is the loop that carries the dependency. A loop carries a dependency from an indexed assignment to an indexed use if, for some iteration of the loop, the assignment stores a value that is referred to on a later iteration of the loop. For example, there is an LCD from A(I+1) to A(I) in the following loop:
	
	`DO I=1,100`
	`  A(I+1)=A(I)+B(I)`
	`ENDDO`

loop interchange	The reordering of nested loops to achieve greater locality of reference for vectorization of greater granularity for parallelization.
multiprocessor	On a CONVEX supercomputer, the memory system, the I/O system, peripherals, and two or more CPUs.
mutual exclusion	A protocol that prevents access to a given resource by more than one thread at a time.
parallelization	The act of creating code that enables iterations of loops to run simultaneously and without synchronization on multiple CPUs. At optimization level 3, the CONVEX FORTRAN V5.0 compiler automatically parallelizes your program and recognizes compiler directives with which you can specify parallelization.
process	A collection of one or more execution streams within a single logical address space; an executable program. A process is made up of one or more threads.
reentrancy	As used in this paper, the capability of a code segment to have multiple versions in existence that may execute in parallel. Each version maintains a thread-private copy of its local data and a thread-private stack to store compiler-generated temporary variables.
strip length, parallel	The amount by which the induction variable of the inner loop is advanced on each iteration of the outer loop.
strip length, vector	The number of array elements processed in a given vector operation.

strip mining

The transformation of a single loop into two nested loops. CONVEX compilers perform the following parallelization and related strip-mine optimizations:

parallel	the outer loop (the parallel strip-mine loop) advances the initial value of the inner loop's induction variable by the parallel strip length.
variable	selects the parallel strip length based on the trip count of the loop and the amount of code contained in the loop body.
vector	the inner loop is vectorized, and the outer loop iterates over blocks of arrays in steps equal to the vector length of the target machine.

synchronization

A way to keep two threads from accessing the same critical region simultaneously. You can synchronize your programs using compiler directives or assembly-language instructions. You do so, however, at the cost of additional overhead: synchronization causes at least one CPU to wait for another.

thread

An independent execution stream that is fetched and executed by a CPU. One or more threads, each of which can execute on a different CPU, make up each process. Memory, files, signals, and other process attributes are generally shared among all the threads in a given process, enabling the threads to cooperate in solving the common problem. Threads are created and terminated by instructions that can be automatically generated by CONVEX compilers, inserted by adding compiler directives to source code, or coded explicitly in assembly-language programs.

thread-specific or thread-private

Data that is accessible by a single thread only (not shared among the threads constituting a process). Thread-specific data allows the same virtual address to refer to different physical memory locations.

vector chaining

The overlapping vector operations in the CPU. For instance, in the case of a vector load followed by a vector add, the add may be started as soon as the first operands are available.