Chapter 5

Fortran Enhancements for Multiprocessors

This chapter contains these sections:

- "Overview" provides an overview of this chapter.
- "Parallel Loops" discusses the concept of parallel **DO** loops.
- "Writing Parallel Fortran" explains how to use compiler directives to generate code that can be run in parallel.
- "Analyzing Data Dependencies for Multiprocessing" describes how to analyze **DO** loops to determine whether they can be parallelized.
- "Breaking Data Dependencies" explains how to rewrite **DO** loops that contain data dependencies so that some or all of the loop can be run in parallel.
- "Work Quantum" describes how to determine whether the work performed in a loop is greater than the overhead associated with multiprocessing the loop.
- "Cache Effects" explains how to write loops that account for the effect of the cache.
- "Advanced Features" describes features that override multiprocessing defaults and customize parallelism.
- "DOACROSS Implementation" discusses how multiprocessing is implemented in a **DOACROSS** routine.

Overview

The Silicon Graphics Fortran compiler allows you to apply the capabilities of a Silicon Graphics multiprocessor workstation to the execution of a single job. By coding a few simple directives, the compiler splits the job into concurrently executing pieces, thereby decreasing the run time of the job.

This chapter discusses techniques for analyzing your program and converting it to multiprocessing operations. Chapter 6, "Compiling and Debugging Parallel Fortran," gives compilation and debugging instructions for parallel processing.

Parallel Loops

The model of parallelism used focuses on the Fortran **DO** loop. The compiler executes different iterations of the **DO** loop in parallel on multiple processors. For example, using the **SIMPLE** scheduling method, a **DO** loop consisting of 200 iterations will run on a machine with four processors. The first 50 iterations run on one processor, the next 50 on another, and so on. The multiprocessing code adjusts itself at run time to the number of processors actually present on the machine. Thus, if the above 200–iteration loop was moved to a machine with only two processors, it would be divided into two blocks of 100 iterations each, without any need to recompile or relink. In fact, multiprocessing code can even be run on single–processor machines. The above loop would be divided into one block of 200 iterations. This allows code to be developed on a single–processor Silicon Graphics IRIS–4D Series workstation or Personal IRIS ™, and later run on an IRIS POWER Series multiprocessor.

The processes that participate in the parallel execution of a task are arranged in a master/slave organization. The original process is the master. It creates zero or more slaves to assist. When a parallel **DO** loop is encountered, the master asks the slaves for help. When the loop is complete, the slaves wait on the master, and the master resumes normal execution. The master process and each of the slave processes are called a *thread of execution* or simply a *thread*. By default, the number of threads is set equal to the number of processors on the particular machine. If you want, you can override the default and explicitly control the number of threads of execution used by a Fortran job.

For multiprocessing to work correctly, the iterations of the loop must not depend on each other; each iteration must stand alone and produce the same answer regardless of whether any other iteration of the loop is executed. Not all **DO** loops have this property, and loops without it cannot be correctly executed in parallel. However, any of the loops encountered in practice fit this model. Further, many loops that cannot be run in parallel in their original form can be rewritten to run wholly or partially in parallel.

To provide compatibility for existing parallel programs, Silicon Graphics has chosen to adopt the syntax for parallelism used by Sequent Computer Corporation. This syntax takes the form of compiler directives embedded in comments. These fairly high level directives provide a convenient method for you to describe a parallel loop, while leaving the details to the Fortran compiler. For advanced users, there are a number of special routines that permit more direct control over the parallel execution. (Refer to "Advanced Features" for more information.)

Writing Parallel Fortran

The Fortran compiler accepts directives that cause it to generate code that can be run in parallel. The compiler directives look like Fortran comments: they begin with a **C** in column one. If multiprocessing is not turned on, these statements are treated as comments. This allows the identical source to be compiled with a single–processing compiler or by Fortran without the multiprocessing option. The directives are distinguished by having a **$** as the second character. There are six directives that are supported: **C$DOACROSS**, **C$&**, **C$**, **C$MP_SCHEDTYPE**, **C$CHUNK**, and **C$COPYIN**. The **C$COPYIN** directive is described in "Local COMMON Blocks". This section describes the others.

C$DOACROSS

The essential compiler directive is **C$DOACROSS**. This directs the compiler to generate special code to run iterations of the **DO** loop in parallel. The **C$DOACROSS** statement applies only to the next statement (which must be a **DO** loop).

The **C$DOACROSS** directive has the form

`C$DOACROSS [`*clause*` [ , `*clause*`]... ]`

where a *clause* is one of the following:

`SHARE (`*variable list*`)`
`LOCAL (`*variable list*`)`
`LASTLOCAL (`*variable list*`)`

```
REDUCTION (scalar variable list)
IF (logical expression)
CHUNK=integer expression
MP_SCHEDTYPE=schedule type
```

The meaning of each clause is discussed below. All of these clauses are optional.

SHARE, LOCAL, LASTLOCAL

These are lists of variables as discussed in the "Analyzing Data Dependencies for Multiprocessing". A variable may appear in only one of these lists. To make the task of writing these lists easier, there are several defaults. The loop−iteration variable is **LASTLOCAL** by default. All other variables are **SHARE** by default.

LOCAL is a little faster than **LASTLOCAL**, so if you do not need the final value, it is good practice to put the **DO** loop index variable into the **LOCAL** list, although this is not required.

Only variables can appear in these lists. In particular, **COMMON** blocks cannot appear in a **LOCAL** list (but see the discussion of local **COMMON** blocks in "Advanced Features"). The **SHARE**, **LOCAL**, and **LASTLOCAL** lists give only the names of the variables. If any member of the list is an array, it is listed without any subscripts.

Note: There is a minor flaw in the way unlisted variables default to **SHARE**. There must be at least one reference to the variable in a nonparallel region or at least one appearance of that variable in the **SHARE** list of some loop. If not, the compiler will complain that the variable in the multiprocessed loop has not been previously referenced.

REDUCTION

The **REDUCTION** clause lists those variables involved in a reduction operation. The meaning and use of reductions are discussed in Example 4 of "Breaking Data Dependencies". An element of the **REDUCTION** list must be an individual variable (also called a scalar variable) and may not be an array. However, it may be an individual element of an array. In this case, it would appear in the list with the proper subscripts.

It is possible for one element of an array to be used in a reduction operation, while other elements of the array are used in other ways. To allow for this, if an element of an array appears in the **REDUCTION** list, it is legal for that array also to appear in the **SHARE** list.

There are four types of reduction supported: *sum*(+), *product*(*), *min*(), and *max*(). Note that *min*(*max*) reductions must use the *min*(*max*) functions in order to be recognized correctly.

The compiler makes some simple checks to confirm that the reduction expression is legal. The compiler does not, however, check all statements in the **DO** loop for illegal reductions. It is up to the programmer to assure legal use of the reduction variable.

IF

The **IF** clause gives a logical expression that is evaluated just before the loop is executed. If the expression is TRUE, the loop is executed in parallel. If the expression is FALSE, the loop is executed serially. Typically, the expression tests the number of times the loop will execute to be sure that there is enough work in the loop to amortize the overhead of parallel execution. Currently, the break−even point is about 400 CPU clocks of work, which normally translates to about 100 floating point operations.

MP_SCHEDTYPE, CHUNK

These options affect the way the work in the loop is scheduled among the participating tasks. They do not affect the correctness of the loop. They are useful for tuning the performance of critical loops. See"Load Balancing" for more details.

Four methods of scheduling the iterations are supported. A single program may use any or all of them as it finds appropriate.

The simple method (**MP_SCHEDTYPE=SIMPLE**) divides the iterations among the processes by dividing them into contiguous pieces and assigning one piece to each process.

The interleave scheduling method (**MP_SCHEDTYPE=INTERLEAVE**) breaks the iterations up into pieces of the size specified by the **CHUNK** option, and execution of those pieces is interleaved among the processes. For example, if there are four processes and **CHUNK=2**, then the first process will execute iterations 1-2, 9-10, 17-18, …; the second process will execute iterations 3-4, 11-12, 19-20,…; and so on. Although this is more complex than the simple method, it is still a fixed schedule with only a single scheduling decision.

In dynamic scheduling (**MP_SCHEDTYPE=DYNAMIC**) the iterations are broken into **CHUNK**−sized pieces. As each process finishes a piece, it enters a critical section to grab the next available piece. This gives good load balancing at the price of higher overhead.

The fourth method is a variation of the guided self−scheduling algorithm (**MP_SCHEDTYPE=GSS**). Here, the piece size is varied depending on the number of iterations remaining. By parceling out relatively large pieces to start with and relatively small pieces toward the end, the hope is to achieve good load balancing while reducing the number of entries into the critical section.

In addition to these four methods, the user may specify the scheduling method at run time (**MP_SCHEDTYPE=RUNTIME**). Here, the scheduling routine examines values in the user's run−time environment and uses that information to select one of the four methods. See "Advanced Features" for more details.

If both the **MP_SCHEDTYPE** and **CHUNK** clauses are omitted, **SIMPLE** scheduling is assumed. If **MP_SCHEDTYPE** is set to **INTERLEAVE** or **DYNAMIC** and the **CHUNK** clause are omitted, **CHUNK=1** is assumed. If **MP_SCHEDTYPE** is set to one of the other values, **CHUNK** is ignored. If the **MP_SCHEDTYPE** clause is omitted, but **CHUNK** is set, then **MP_SCHEDTYPE=DYNAMIC** is assumed.

Example 1

The code fragment

```
DO 10 I = 1, 100
```

```
      A(I) = B(I)
10 CONTINUE
```

could be multiprocessed with the directive

```
C$DOACROSS LOCAL(I), SHARE(A, B)
   DO 10 I = 1, 100
      A(I) = B(I)
10 CONTINUE
```

Here, the defaults are sufficient, provided **A** and **B** are mentioned in a nonparallel region or in another **SHARE** list. The following then works:

```
C$DOACROSS
   DO 10 I = 1, 100
      A(I) = B(I)
10 CONTINUE
```

Example 2

```
   DO 10 I = 1, N
      X = SQRT(A(I))
      B(I) = X*C(I) + X*D(I)
10 CONTINUE
```

You can be fully explicit:

```
C$DOACROSS LOCAL(I, X), share(A, B, C, D, N)
   DO 10 I = 1, N
      X = SQRT(A(I))
      B(I) = X*C(I) + X*D(I)
10 CONTINUE
```

or you can use the defaults

```
C$DOACROSS LOCAL(X)
   DO 10 I = 1, N
      X = SQRT(A(I))
      B(I) = X*C(I) + X*D(I)
10 CONTINUE
```

See Example 5 in "Analyzing Data Dependencies for Multiprocessing" for more information on this example.

Example 3

```
   DO 10 I = M, K, N
      X = D(I)**2
      Y = X + X
      DO 20 J = I, MAX
```

```
         A(I,J) = A(I,J) + B(I,J) * C(I,J) * X + Y
20 CONTINUE
10 CONTINUE
   PRINT*, I, X
```

Here, the final values of **I** and **X** are needed after the loop completes. A correct directive is

```
C$DOACROSS LOCAL(Y,J), LASTLOCAL(I,X), SHARE(M,K,N,ITOP,A,B,C,D)
   DO 10 I = M, K, N
       X = D(I)**2
       Y = X + X
       DO 20 J = I, ITOP
           A(I,J) = A(I,J) + B(I,J) * C(I,J) *X + Y
20 CONTINUE
10 CONTINUE
   PRINT*, I, X
```

or you could use the defaults

```
C$DOACROSS LOCAL(Y,J), LASTLOCAL(X)
   DO 10 I = M, K, N
       X = D(I)**2
       Y = X + X
       DO 20 J = I, MAX
           A(I,J) = A(I,J) + B(I,J) * C(I,J) *X + Y
20 CONTINUE
10 CONTINUE
   PRINT*, I, X
```

I is a loop index variable for the **C$DOACROSS** loop, so it is **LASTLOCAL** by default. However, even though **J** is a loop index variable, it is not the loop index of the loop being multiprocessed and has no special status. If it is not declared, it is given the normal default of **SHARE**, which would be wrong.

C$&

Occasionally, the clauses in the **C$DOACROSS** directive are longer than one line. The **C$&** directive is used to continue the directive onto multiple lines.

```
C$DOACROSS share(ALPHA, BETA, GAMMA, DELTA,
C$&  EPSILON, OMEGA), LASTLOCAL(I,J, K, L, M, N),
C$&  LOCAL(XXX1, XXX2, XXX3, XXX4, XXX5, XXX6, XXX7,
C$&  XXX8, XXX9)
```

C$

The **C$** directive is considered a comment line except when multiprocessing. A line beginning with **C$** is treated as a conditionally compiled Fortran statement. The rest of the line contains a standard Fortran statement. The statement is compiled only if multiprocessing is turned on. In this case, the **C** and **$** are treated as if they are blanks. They can be used to insert debugging statements, or an experienced user can use them to insert arbitrary code into the multiprocessed version.

```
C$    PRINT 10
C$ 10 FORMAT('BEGIN MULTIPROCESSED LOOP')
C$DOACROSS LOCAL(Ii), SHARE(A,B)
      DO I = 1, 100
         CALL COMPUTE(A, B, I)
      END DO
```

C$MP_SCHEDTYPE, C$CHUNK

The **C$MP_SCHEDTYPE=***schedule_type* directive acts as an implicit **MP_SCHEDTYPE** clause. A **DOACROSS** directive that does not have an explicit **MP_SCHEDTYPE** clause is given the value specified in the directive, rather than the normal default. If the **DOACROSS** does have an explicit clause, then the explicit value is used.

The **C$CHUNK=***integer_expression* directive affects the **CHUNK** clause of a **DOACROSS** in the same way that the **C$MP_SCHEDTYPE** directive affects the **MP_SCHEDTYPE** clause. Both directives are in effect from the place they occur in the source until another corresponding directive is encountered or the end of the procedure is reached.

These directives are mostly intended for users of Silicon Graphics POWER Fortran Accelerator™ (PFA). The **DOACROSS** directives supplied by PFA do not have **MP_SCHEDTYPE** or **CHUNK** clauses. These directives provide a method of specifying what kind of scheduling option is desired and allowing PFA to supply the **DOACROSS** directive. These directives are not PFA–specific, however, and can be used by any multiprocessing Fortran programmer.

It is also possible to invoke this functionality from the command line during a compile. The **-mp_schedtype=***schedule_type* and **-chunk=** *integer* command line options have the effect of implicitly putting the corresponding directive(s) as the first lines in the file.

Nesting C$DOACROSS

The Fortran compiler does not support direct nesting of **C$DOACROSS** loops. For example, the following is illegal and generates a compilation error:

```
C$DOACROSS LOCAL(I)
      DO I = 1, N
C$DOACROSS LOCAL(J)
         DO J = 1, N
            A(I,J) = B(I,J)
         END DO
      END DO
```

However, to simplify separate compilation, a different form of nesting is allowed. A routine that uses **C$DOACROSS** can be called from within a multiprocessed region. This can be useful if a single routine is called from several different places: sometimes from within a multiprocessed region, sometimes not.

Nesting does not increase the parallelism. When the first **C$DOACROSS** loop is encountered, that loop is run in parallel. If while in the parallel loop a call is made to a routine that itself has a **C$DOACROSS**, this subsequent loop is executed serially.

Parallel Blocks

The Silicon Graphics Fortran compiler supports parallel execution of **DO** loops only. However, another kind of parallelism frequently occurs: different blocks of code independent of one another can be executed simultaneously. As a simple example,

```
CALL MAKE1(A, B, C, D)

CALL MAKE2(E, F, G, H)
```

If you know that these two routines do not interfere with each other, you can call them simultaneously. The following example shows how to use **DO** loops to execute parallel blocks of code.

```
C$DOACROSS LOCAL(I), MP_SCHEDTYPE=SIMPLE
      DO I = 1, 2
         IF (I .EQ. 1) THEN
             CALL MAKE1(A, B, C, D)
         ELSEIF (I .EQ. 2) THEN
             CALL MAKE2(E, F, G, H)
         END IF
      END DO
```

Analyzing Data Dependencies for Multiprocessing

The essential condition required to parallelize a loop correctly is that each iteration of the loop must be independent of all other iterations. If a loop meets this condition, then the order in which the iterations of the loop execute is not important. They can be executed backward or even at the same time, and the answer is still the same. This property is captured by the notion of *data independence*. For a loop to be data–independent, no iterations of the loop can write a value into a memory location that is read or written by any other iteration of that loop. It is also all right if the same iteration reads and/or writes a memory location repeatedly as long as no others do; it is all right if many iterations read the same location, as long as none of them write to it. In a Fortran program, memory locations are represented by variable names. So, to determine if a particular loop can be run in parallel, examine the way variables are used in the loop. Because data dependence occurs only when memory locations are modified, pay particular attention to variables that appear on the left–hand side of assignment statements. If a variable is not modified, there is no data dependence associated with it.

The Fortran compiler supports four kinds of variable usage within a parallel loop: **SHARE**, **LOCAL**, **LASTLOCAL**, and **REDUCTION**. If a variable is declared as **SHARE**, all iterations of the loop use the same copy. If a variable is declared as **LOCAL**, each iteration is given its own uninitialized copy. A variable is declared **SHARE** if it is only read (not written) within the loop or if it is an array where each iteration of the loop uses a different element of the array. A variable can be **LOCAL** if its value does not depend on any other iteration and if its value is used only within a single iteration. In effect the **LOCAL** variable is just temporary; a new copy can be created in each loop iteration without changing the final answer. As a special case, if only the very last value of a variable computed on the very last iteration is used outside the loop (but would otherwise qualify as a **LOCAL** variable), the loop can be multiprocessed by declaring the variable to be **LASTLOCAL**. The use of **REDUCTION** variables is discussed later.

It is often difficult to analyze loops for data dependence information. Each use of each variable must be examined to see if it fulfills the criteria for **LOCAL**, **LASTLOCAL**, **SHARE**, or **REDUCTION**. If all the variables conform, the loop can be parallelized. If not, the loop cannot be parallelized as it stands, but possibly can be rewritten into an equivalent parallel form. (See "Breaking Data Dependencies" for information on rewriting code in parallel form.)

An alternative to analyzing variable usage by hand is to use PFA. This optional software package is a Fortran preprocessor that analyzes loops for data dependence. If it can determine that a loop is data−independent, it automatically inserts the required compiler directives (see"Writing Parallel Fortran"). If PFA cannot determine the loop to be independent, it produces a listing file detailing where the problems lie.

The rest of this section is devoted to analyzing sample loops, some parallel and some not parallel.

Example 1: Simple Independence

```
   DO 10 I = 1,N
10     A(I) = X + B(I)*C(I)
```

In this example, each iteration writes to a different location in **A**, and none of the variables appearing on the right−hand side is ever written to, only read from. This loop can be correctly run in parallel. All the variables are **SHARE** except for **I**, which is either **LOCAL** or **LASTLOCAL**, depending on whether the last value of **I** is used later in the code.

Example 2: Data Dependence

```
   DO 20 I = 2,N
20     A(I) = B(I) − A(I−1)
```

This fragment contains **A(I)** on the left−hand side and**A(I−1)**on the right. This means that one iteration of the loop writes to a location in **A** and that the next iteration reads from that same location. Because different iterations of the loop read and write the same memory location, this loop cannot be run in parallel.

Example 3: Stride Not 1

```
   DO 20 I = 2,N,2
20     A(I) = B(I) − A(I−1)
```

This example looks like the previous example. The difference is that the stride of the **DO** loop is now two rather than one. Now **A(I)** references every other element of **A**, and **A(I−1)**references exactly those elements of **A** that are not referenced by **A(I)**. None of the data locations on the right−hand side is ever the same as any of the data locations written to on the left−hand side. The data are disjoint, so there is no dependence. The loop can be run in parallel. Arrays **A** and **B** can be declared **SHARE**, while variable **I** should be declared **LOCAL** or **LASTLOCAL**.

Example 4: Local Variable

```
DO I = 1, N
   X = A(I)*A(I) + B(I)
   B(I) = X + B(I)*X
END DO
```

In this loop, each iteration of the loop reads and writes the variable **X**. However, no loop iteration ever needs the value of **X** from any other iteration. **X** is used as a temporary variable; its value does not survive from one iteration to the next. This loop can be parallelized by declaring **X** to be a **LOCAL** variable within the loop. Note that **B(I)** is both read and written by the loop. This is not a problem because each iteration has a different value for **I**, so each iteration uses a different **B(I)**. The same **B(I)** is allowed to be read and written as long as it is done by the same iteration of the loop. The loop can be run in parallel. Arrays **A** and **B** can be declared **SHARE**, while variable **I** should be declared **LOCAL** or **LASTLOCAL**.

Example 5: Function Call

```
   DO 10 I = 1, N
      X = SQRT(A(I))
      B(I) = X*C(I) + X*D(I)
10 CONTINUE
```

The value of **X** in any iteration of the loop is independent of the value of **X** in any other iteration, so **X** can be made a **LOCAL** variable. The loop can be run in parallel. Arrays **A**, **B**, **C**, and **D** can be declared **SHARE**, while variable **I** should be declared **LOCAL** or **LASTLOCAL**.

The interesting feature of this loop is that it invokes an external routine, **sqrt**. It is possible to use functions and/or subroutines (intrinsic or user defined) within a parallel loop. However, make sure that the various parallel invocations of the routine do not interfere with one another. In particular, **sqrt** returns a value that depends only on its input argument, that does not modify global data, and that does not use static storage. We say that **sqrt** has no *side effects*.

All the Fortran intrinsic functions listed in Appendix A of the *Fortran 77 Language Reference Manual* have no side effects and can safely be part of a parallel loop. For the most part, the Fortran library functions and VMS intrinsic subroutine extensions (listed in Chapter 4, "System Functions and Subroutines,") cannot safely be included in a parallel loop. In particular, **rand** is not safe for multiprocessing. For user−written routines, it is the responsibility of the user to ensure that the routines can be correctly multiprocessed.

Caution: Routines called within a parallel loop cannot be compiled with the **-static** flag.

Example 6: Rewritable Data Dependence

```
INDX = 0
```

```
DO I = 1, N
   INDX = INDX + I
   A(I) = B(I) + C(INDX)
END DO
```

Here, the value of **INDX** survives the loop iteration and is carried into the next iteration. This loop cannot be parallelized as it is written. Making **INDX** a **LOCAL** variable does not work; you need the value of **INDX** computed in the previous iteration. It is possible to rewrite this loop to make it parallel (see Example 1 in "Breaking Data Dependencies").

Example 7: Exit Branch

```
     DO I = 1, N
        IF (A(I) .LT. EPSILON) GOTO 320
        A(I) = A(I) * B(I)
     END DO
 320 CONTINUE
```

This loop contains an exit branch; that is, under certain conditions the flow of control suddenly exits the loop. The Fortran compiler cannot parallelize loops containing exit branches.

Example 8: Complicated Independence

```
DO I = K+1, 2*K
   W(I) = W(I) + B(I,K) * W(I-K)
END DO
```

At first glance, this loop looks like it cannot be run in parallel because it uses both **W(I)** and **W(I−K)** Closer inspection reveals that because the value of **I** varies between **K**+1 and *2**K**, then **I−K**goes from 1 to **K**. This means that the **W(I−K)**term varies from **W(1)** up to **W(K)**, while the **W(I)** term varies from **W(K+1)** up to **W(2*K)**. So **W(I−K)**in any iteration of the loop is never the same memory location as **W(I)** in any other iterations. Because there is no data overlap, there are no data dependencies. This loop can be run in parallel. Elements **W**, **B**, and **K** can be declared **SHARE**, while variable **I** should be declared **LOCAL** or **LASTLOCAL**.

This example points out a general rule: the more complex the expression used to index an array, the harder it is to analyze. If the arrays in a loop are indexed only by the loop index variable, the analysis is usually straightforward though tedious. Fortunately, in practice most array indexing expressions are simple.

Example 9: Inconsequential Data Dependence

```
INDEX = SELECT(N)
DO I = 1, N
   A(I) = A(INDEX)
END DO
```

There is a data dependence in this loop because it is possible that at some point **I** will be the same as

INDEX, so there will be a data location that is being read and written by different iterations of the loop. In this particular special case, you can simply ignore it. You know that when **I** and **INDEX** are equal, the value written into **A(I)** is exactly the same as the value that is already there. The fact that some iterations of the loop will read the value before it is written and some after it is written is not important because they will all get the same value. Therefore, this loop can be parallelized. Array **A** can be declared **SHARE**, while variable **I** should be declared **LOCAL** or **LASTLOCAL**.

Example 10: Local Array

```
DO I = 1, N
   D(1) = A(I,1) - A(J,1)
   D(2) = A(I,2) - A(J,2)
   D(3) = A(I,3) - A(J,3)
   TOTAL_DISTANCE(I,J) = SQRT(D(1)**2 + D(2)**2 + D(3)**2)
END DO
```

In this fragment, each iteration of the loop uses the same locations in the **D** array. However, closer inspection reveals that the entire **D** array is being used as a temporary. This can be multiprocessed by declaring **D** to be **LOCAL**. The Fortran compiler allows arrays (even multidimensional arrays) to be **LOCAL** variables with one restriction: the size of the array must be known at compile time. The dimension bounds must be constants; the **LOCAL** array cannot have been declared using a variable or the asterisk syntax.

Therefore, this loop can be parallelized. Arrays **TOTAL_DISTANCE** and **A** can be declared **SHARE**, while array **D** and variable **I** should be declared **LOCAL** or **LASTLOCAL**.

Breaking Data Dependencies

Many loops that have data dependencies can be rewritten so that some or all of the loop can be run in parallel. The essential idea is to locate the statement(s) in the loop that cannot be made parallel and try to find another way to express it that does not depend on any other iteration of the loop. If this fails, try to pull the statements out of the loop and into a separate loop, allowing the remainder of the original loop to be run in parallel.

The first step is to analyze the loop to discover the data dependencies (see "Writing Parallel Fortran"). Once the problem areas are identified, various techniques can be used to rewrite the code to break the dependence. Sometimes the dependencies in a loop cannot be broken, and you must either accept the serial execution rate or try to discover a new parallel method of solving the problem. The rest of this section is devoted to a series of "cookbook" examples on how to deal with commonly occurring situations. These are by no means exhaustive but cover many situations that happen in practice.

Example 1: Loop Carried Value

```
INDX = 0
DO I = 1, N
INDX = INDX + I
```

```
A(I) = B(I) + C(INDX)
END DO
```

This is the same as Example 6 in "Writing Parallel Fortran". Here, **INDX** has its value carried from iteration to iteration. However, it is possible to compute the appropriate value for **INDX** without making reference to any previous value:

```
C$DOACROSS LOCAL (I, INDX)
  DO I  = 1, N
     INDX = (I*(I+1))/2
     A(I) = B(I) + C(INDX)
  END DO
```

In this loop, the value of **INDX** is computed without using any values computed on any other iteration. **INDX** can correctly be made a **LOCAL** variable, and the loop can now be multiprocessed.

Example 2: Indirect Indexing

```
   DO 100 I = 1, N
      IX = INDEXX(I)
      IY = INDEXY(I)
      XFORCE(I) = XFORCE(I) + NEWXFORCE(IX)
      YFORCE(I) = YFORCE(I) + NEWYFORCE(IY)
      IXX = IXOFFSET(IX)
      IYY = IYOFFSET(IY)
      TOTAL(IXX, IYY) = TOTAL(IXX, IYY) + EPSILON
100 CONTINUE
```

It is the final statement that causes problems. The indexes **IXX** and **IYY** are computed in a complex way and depend on the values from the **IXOFFSET** and **IYOFFSET** arrays. We do not know if **TOTAL (IXX,IYY)** in one iteration of the loop will always be different from **TOTAL (IXX,IYY)** in every other iteration of the loop.

We can pull the statement out into its own separate loop by expanding **IXX** and **IYY** into arrays to hold intermediate values:

```
C$DOACROSS LOCAL(IX, IY, I)
    DO I  = 1, N
       IX = INDEXX(I)
       IY = INDEXY(I)
       XFORCE(I) = XFORCE(I) + NEWXFORCE(IX)
       YFORCE(I) = YFORCE(I) + NEWYFORCE(IY)
       IXX(I) = IXOFFSET(IX)
       IYY(I) = IYOFFSET(IY)
    END DO
    DO 100 I = 1, N
       TOTAL(IXX(I),IYY(I)) = TOTAL(IXX(I), IYY(I)) + EPSILON
100 CONTINUE
```

Here, **IXX** and **IYY** have been turned into arrays to hold all the values computed by the first loop. The first loop (containing most of the work) can now be run in parallel. Only the second loop must still be run serially.

Before we leave this example, note that, if we were certain that the value for **IXX** was always different in every iteration of the loop, then the original loop could be run in parallel. It could also be run in parallel if **IYY** was always different. If **IXX** (or **IYY**) is always different in every iteration, then **TOTAL(IXX,IYY)** is never the same location in any iteration of the loop, and so there is no data conflict.

This sort of knowledge is, of course, program–specific and should always be used with great care. It may be true for a particular data set, but to run the original code in parallel as it stands, you need to be sure it will always be true for all possible input data sets.

Example 3: Recurrence

```
DO I = 1,N
   X(I) = X(I-1) + Y(I)
END DO
```

This is an example of *recurrence*, which exists when a value computed in one iteration is immediately used by another iteration. There is no good way of running this loop in parallel. If this type of construct appears in a critical loop, try pulling the statement(s) out of the loop as in the previous example. Sometimes another loop encloses the recurrence; in that case, try to parallelize the outer loop.

Example 4: Sum Reduction

```
        sum  = 0.0
        amax = a(1)
        amin = a(1)
c$doacross local(1), REDUCTION(asum, AMAX, AMIN)
        do i = 1,N
           asum = asum + a(i)
           if (a(i) .gt. amax) then
               imin = a(i)
           else if (a(i) .lt. amin) then
                    imin = a(i)
           end if
      end do
```

This operation is known as a *reduction*. Reductions occur when an array of values are combined and reduced into a single value. This example is a sum reduction because the combining operation is addition. Here, the value of **sum** is carried from one loop iteration to the next, so this loop cannot be multiprocessed. However, because this loop simply sums the elements of **a(i)**, we can rewrite the loop to accumulate multiple, independent subtotals.

Then we can do much of the work in parallel:

```
   NUM_THREADS = MP_NUMTHREADS()
C
C  IPIECE_SIZE = N/NUM_THREADS ROUNDED UP
C
   IPIECE_SIZE = (N + (NUM_THREADS -1)) / NUM_THREADS
   DO K = 1, NUM_THREADS
      PARTIAL_SUM(K) = 0.0
C
C  THE FIRST THREAD DOES 1 THROUGH IPIECE_SIZE, THE
C  SECOND DOES IPIECE_SIZE + 1 THROUGH 2*IPIECE_SIZE,
C  ETC. IF N IS NOT EVENLY DIVISIBLE BY NUM_THREADS,
C  THE LAST PIECE NEEDS TO TAKE THIS INTO ACCOUNT,
C  HENCE THE "MIN" EXPRESSION.
C
   DO I =K*IPIECE_SIZE -IPIECE_SIZE +1, MIN(K*IPIECE_SIZE,N)
      PARTIAL_SUM(K) = PARTIAL_SUM(K) + A(I)
   END DO
   END DO
C
C  NOW ADD UP THE PARTIAL SUMS
   SUM = 0.0
   DO I = 1, NUM_THREADS
      SUM = SUM + PARTIAL_SUM(I)
   END DO
```

The outer **K** loop can be run in parallel. In this method, the array pieces for the partial sums are contiguous, resulting in good cache utilization and performance.

This is an important and common transformation, and so automatic support is provided by the **REDUCTION** clause:

```
   SUM = 0.0
C$DOACROSS LOCAL (I), REDUCTION (SUM)
   DO 10 I = 1, N
      SUM = SUM + A(I)
10 CONTINUE
```

This has essentially the same meaning as the much longer and more confusing code above. It is an important example to study because the idea of adding an extra dimension to an array to permit parallel computation, and then combining the partial results, is an important technique for trying to break data dependencies. This idea occurs over and over in various contexts and disguises.

Note that reduction transformations such as this are not strictly correct. Because computer arithmetic has limited precision, when you sum the values together in a different order, as was done here, the round–off errors accumulate slightly differently. It is likely that the final answer will be slightly different from the original loop. Most of the time the difference is irrelevant, but it can be significant, so some caution is in order.

This example is a *sum* reduction because the operator is plus (+). The Fortran compiler supports three other types of reduction operations:

1. product: p = p*a(*i*)
2. mm: m = mm(m,a(i))
3. max: m = max(m,a(i))

For example,

```
c$doacross local(1), REDUCTION(asum, AMAX, AMIN)
        do i = 1,N
           big_sum  = big_sum + a(i)
           big_prod = big_prod * a(i)
           big_min  = min(big_min, a(i))
           big_max  = max(big_max, a(i)
        end do
```

One further reduction is noteworthy.

```
      DO I = 1, N
         TOTAL = 0.0
         DO J = 1, M
            TOTAL = TOTAL + A(J)
         END DO
         B(I) = C(I) * TOTAL
      END DO
```

Initially, it may look as if the reduction in the inner loop needs to be rewritten in a parallel form. However, look at the outer **I** loop. Although **TOTAL** cannot be made a **LOCAL** variable in the inner loop, it fulfills the criteria for a **LOCAL** variable in the outer loop: the value of **TOTAL** in each iteration of the outer loop does not depend on the value of **TOTAL** in any other iteration of the outer loop. Thus, you do not have to rewrite the loop; you can parallelize this reduction on the outer **I** loop, making **TOTAL** and **J** local variables.

Work Quantum

A certain amount of overhead is associated with multiprocessing a loop. If the work occurring in the loop is small, the loop can actually run slower by multiprocessing than by single processing. To avoid this, make the amount of work inside the multiprocessed region as large as possible.

Example 1: Loop Interchange

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

```
      END DO
   END DO
END DO
```

Here you have several choices: parallelize the **J** loop or the **I** loop. You cannot parallelize the **K** loop because different iterations of the **K** loop will all try to read and write the same values of **A(I,J)**. Try to parallelize the outermost **DO** loop possible, because it encloses the most work. In this example, that is the **I** loop. For this example, use the technique called *loop interchange*. Although the parallelizable loops are not the outermost ones, you can reorder the loops to make one of them outermost.

Thus, loop interchange would produce

```
C$DOACROSS LOCAL(I, J, K)
      DO I = 1, N
         DO K = 1, N
            DO J = 1, N
               A(I,J) = A(I,J) + B(I,K) * C(K,J)
            END DO
         END DO
      END DO
```

Now the parallelizable loop encloses more work and will show better performance. In practice, relatively few loops can be reordered in this way. However, it does occasionally happen that several loops in a nest of loops are candidates for parallelization. In such a case, it is usually best to parallelize the outermost one.

Occasionally, the only loop available to be parallelized has a fairly small amount of work. It may be worthwhile to force certain loops to run without parallelism or to select between a parallel version and a serial version, on the basis of the length of the loop.

Example 2: Conditional Parallelism

```
   J = (N/4) * 4
   DO I = J+1, N
      A(I) = A(I) + X*B(I)
   END DO
   DO I = 1, J, 4
      A(I) = A(I) + X*B(I)
      A(I+1) = A(I+1) + X*B(I+1)
      A(I+2) = A(I+2) + X*B(I+2)
      A(I+3) = A(I+3) + X*B(I+3)
   END DO
```

Here you are using loop unrolling of order four to improve speed. For the first loop, the number of iterations is always fewer than four, so this loop does not do enough work to justify running it in parallel. The second loop is worthwhile to parallelize if **N** is big enough. To overcome the parallel loop overhead, **N** needs to be around 50.

An optimized version would use the **IF** clause on the **DOACROSS** directive:

```
   J = (N/4) * 4
   DO I = J+1, N
      A(I) = A(I) + X*B(I)
   END DO
C$DOACROSS IF (J.GE.50), LOCAL(I)
      DO I = 1, J, 4
         A(I) = A(I) + X*B(I)
         A(I+1) = A(I+1) + X*B(I+1)
         A(I+2) = A(I+2) + X*B(I+2)
         A(I+3) = A(I+3) + X*B(I+3)
      END DO
   ENDIF
```

Cache Effects

It is good policy to write loops that take the effect of the cache into account, with or without parallelism. The technique for the best cache performance is also quite simple: make the loop step through the array in the same way that the array is laid out in memory. For Fortran, this means stepping through the array without any gaps and with the leftmost subscript varying the fastest.

Note that this optimization does not depend on multiprocessing, nor is it required in order for multiprocessing to work correctly. However, multiprocessing can affect how the cache is used, so it is worthwhile to understand.

Example 1: Matrix Multiply

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

This is the same as Example 1 in "Work Quantum". To get the best cache performance, the **I** loop should be innermost. At the same time, to get the best multiprocessing performance, the outermost loop should be parallelized. For this example, you can interchange the **I** and **J** loops, and get the best of both optimizations:

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

```
            END DO
         END DO
      END DO
```

Example 2: Trade–Offs

Sometimes you must choose between the possible optimizations and their costs. Look at the following code segment:

```
DO J = 1, N
   DO I = 1, M
      A(I) = A(I) + B(J)*C(I,J)
   END DO
END DO
```

This loop can be parallelized on **I** but not on **J**. You could interchange the loops to put **I** on the outside, thus getting a bigger work quantum.

```
C$DOACROSS LOCAL(I,J)
   DO I = 1, M
      DO J = 1, N
         A(I) = A(I) + B(J)*C(I,J)
      END DO
   END DO
```

However, putting **J** on the inside means that you will step through the **C** array in the wrong direction; the leftmost subscript should be the one that varies the fastest. It is possible to parallelize the **I** loop where it stands:

```
   DO J = 1, N
C$DOACROSS LOCAL(I)
      DO I = 1, M
         A(I) = A(I) + B(J)*C(I,J)
      END DO
   END DO
```

but **M** needs to be large for the work quantum to show any improvement. In this particular example, **A(I)** is used to do a sum reduction, and it is possible to use the reduction techniques shown in Example 4 of "Breaking Data Dependencies" to rewrite this in a parallel form. (Recall that there is no support for an entire array as a member of the **REDUCTION** clause on a **DOACROSS**.) However, that involves converting array **A** from a one–dimensional array to a two–dimensional array to hold the partial sums; this is analogous to the way we converted the scalar summation variable into an array of partial sums.

If **A** is large, however, that may take more memory than you can spare.

```
   NUM = MP_NUMTHREADS()
   IPIECE = (N + (NUM–1)) / NUM
C$DOACROSS LOCAL(K,J,I)
   DO K = 1, NUM
```

```
      DO J = K*IPIECE - IPIECE + 1, MIN(N, K*IPIECE)
         DO I = 1, M
            PARTIAL_A(I,K) = PARTIAL_A(I,K) + B(J)*C(I,J)
         END DO
      END DO
   END DO
C$DOACROSS LOCAL (I,K)
   DO I = 1, M
      DO K = 1, NUM
         A(I) = A(I) + PARTIAL_A(I,K)
      END DO
   END DO
```

You must trade off the various possible optimizations to find the combination that is right for the particular job.

Load Balancing

When the Fortran compiler divides a loop into pieces, by default it uses the simple method of separating the iterations into contiguous blocks of equal size for each process. It can happen that some iterations take significantly longer to complete than other iterations. At the end of a parallel region, the program waits for all processes to complete their tasks. If the work is not divided evenly, time is wasted waiting for the slowest process to finish.

Example:

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

This can be parallelized on the **I** loop. Because the inner loop goes from 1 to **I**, the first block of iterations of the outer loop will end long before the last block of iterations of the outer loop.

In this example, this is easy to see and predictable, so you can change the program:

```
   NUM_THREADS = MP_NUMTHREADS()
C$DOACROSS LOCAL(I, J, K)
   DO K = 1, NUM_THREADS
      DO I = K, N, NUM_THREADS
         DO J = 1, I
            A(J, I) = A(J, I) + B(J)*C(I)
         END DO
      END DO
   END DO
```

In this rewritten version, instead of breaking up the **I** loop into contiguous blocks, break it into interleaved blocks. Thus, each execution thread receives some small values of **I** and some large values of **I**, giving a better balance of work between the threads. Interleaving usually, but not always, helps cure a load balancing problem.

This desirable transformation is provided to do this automatically by using the **MP_SCHEDTYPE** clause.

```
C$DOACROSS LOCAL (I,J), MP_SCHEDTYPE=INTERLEAVE
   DO 20 I = 1, N
      DO 10 J = 1, I
         A (J,I) = A(J,I) + B(J)*C(J)
10 CONTINUE
20 CONTINUE
```

This has the same meaning as the rewritten form above.

Note that this can cause poor cache performance because you are no longer stepping through the array at stride 1. This can be somewhat improved by adding a **CHUNK** clause. **CHUNK=** 4 or 8 is often a good choice of value. Each small chunk will have stride 1 to improve cache performance, while the chunks are interleaved to improve load balancing.

The way that iterations are assigned to processes is known as *scheduling*. Interleaving is one possible schedule. Both interleaving and the "simple" scheduling methods are examples of *fixed* schedules; the iterations are assigned to processes by a single decision made when the loop is entered. For more complex loops, it may be desirable to use **DYNAMIC** or **GSS** schedules.

Comparing the output from *pixie* or from pc–sample profiling allows you to see how well the load is being balanced so you can compare the different methods of dividing the load. Refer to the discussion of the **MP_SCHEDTYPE** clause in "C$DOACROSS" for more information.

Even when the load is perfectly balanced, iterations may still take varying amounts of time to finish because of random factors. One process may have to read the disk, another may be interrupted to let a different program run, and so on. Because of these unpredictable events, the time spent waiting for all processes to complete can be several hundred cycles, even with near perfect balance.

Advanced Features

A number of features are provided so that sophisticated users can override the multiprocessing defaults and customize the parallelism to their particular applications. This section provides a brief explanation of these features.

mp_block and mp_unblock

mp_block(3f) puts the slave threads into a blocked state using the system call **blockproc**(2). The slave threads stay blocked until a call is made to **mp_unblock**(3f). These routines are useful if the job has bursts of parallelism separated by long stretches of single processing, as with an interactive program. You can block the slave processes so they consume CPU cycles only as needed, thus freeing the machine for other users. The Fortran system automatically unblocks the slaves on entering a parallel region should you neglect to do so.

mp_setup, mp_create, and mp_destroy

The **mp_setup**(3f), **mp_create**(3f), and **mp_destroy**(3f) subroutine calls create and destroy threads of execution. This can be useful if the job has only one parallel portion or if the parallel parts are widely scattered. When you destroy the extra execution threads, they cannot consume system resources; they must be re−created when needed. Use of these routines is discouraged because they degrade performance; the **mp_block** and **mp_unblock** routines can be used in almost all cases.

mp_setup takes no arguments. It creates the default number of processes as defined by previous calls to **mp_set_numthreads**, by the environment variable **MP_SET_NUMTHREADS**, or by the number of CPUs on the current hardware platform. **mp_setup** is called automatically when the first parallel loop is entered in order to initialize the slave threads.

mp_create takes a single integer argument, the total number of execution threads desired. Note that the total number of threads includes the master thread. Thus, **mp_create(*n*)** creates one thread less than the value of its argument. **mp_destroy** takes no arguments; it destroys all the slave execution threads, leaving the master untouched.

When the slave threads die, they generate a **SIGCLD** signal. If your program has changed the signal handler to catch **SIGCLD**, it must be prepared to deal with this signal when **mp_destroy** is executed. This signal also occurs when the program exits; **mp_destroy** is called as part of normal cleanup when a parallel Fortran job terminates.

mp_blocktime

The Fortran slave threads spin wait until there is work to do. This makes them immediately available when a parallel region is reached. However, this consumes CPU resources. After enough wait time has passed, the slaves block themselves through **blockproc**. Once the slaves are blocked, it requires a system call to **unblockproc** to activate the slaves again (refer to the *unblockproc*(2) man page for details). This makes the response time much longer when starting up a parallel region.

This trade−off between response time and CPU usage can be adjusted with the **mp_blocktime**(3f) call. **mp_blocktime** takes a single integer argument that specifies the number of times to spin before blocking. By default, it is set to 10,000,000; this takes roughly 3 seconds. If called with an argument of 0, the slave threads will not block themselves no matter how much time has passed. Explicit calls to **mp_block**, however, will still block the threads.

This automatic blocking is transparent to the user’s program; blocked threads are automatically unblocked when a parallel region is reached.

mp_numthreads, mp_set_numthreads

Occasionally, you may want to know how many execution threads are available. **mp_numthreads**(3f) is a zero−argument integer function that returns the total number of execution threads for this job. The count includes the master thread.

mp_set_numthreads(3f) takes a single−integer argument. It changes the default number of threads to the specified value. A subsequent call to **mp_setup** will use the specified value rather than the original defaults. If the slave threads have already been created, this call will not change their number. It only has an effect when **mp_setup** is called.

mp_my_threadnum

mp_my_threadnum(3f) is a zero–argument function that allows a thread to differentiate itself while in a parallel region. If there are *n* execution threads, the function call returns a value between zero and *n* - 1. The master thread is always thread zero. This function can be useful when parallelizing certain kinds of loops. Most of the time the loop index variable can be used for the same purpose. Occasionally, the loop index may not be accessible, as, for example, when an external routine is called from within the parallel loop. This routine provides a mechanism for those rare cases.

Environment Variables: MP_SET_NUMTHREADS, MP_BLOCKTIME, MP_SETUP

These environment variables act as an implicit call to the corresponding routine(s) of the same name at program start–up time.

For example, the *csh* command

```
% setenv MP_SET_NUMTHREADS 2
```

causes the program to create two threads regardless of the number of CPUs actually on the machine, just like the source statement

```
CALL MP_SET_NUMTHREADS (2)
```

Similarly, the *sh* commands

```
% set MP_BLOCKTIME 0
% export MP_BLOCKTIME
```

prevent the slave threads from autoblocking, just like the source statement

```
call mp_blocktime (0)
```

For compatibility with older releases, the environment variable **NUM_THREADS** is supported as a synonym for **MP_SET_NUMTHREADS**.

To help support networks with several multiprocessors and several CPUs, the environment variable **MP_SET_NUMTHREADS** also accepts an expression involving integers **+**, **-**, **mm**, **max**, and the special symbol **all**, which stands for "the number of CPUs on the current machine."

For example, the following command selects the number of threads to be two fewer than the total number of CPUs (but always at least one):

```
% setenv MP_SET_NUMTHREADS max(1,all-2)
```

Environment Variables: MP_SCHEDTYPE, CHUNK

These environment variables specify the type of scheduling to use on **DOACROSS** loops that have their scheduling type set to **RUNTIME**. For example, the following *csh* commands cause loops with the **RUNTIME** scheduling type to be executed as interleaved loops with a chunk size of 4:

```
% setenv MP_SCHEDTYPE INTERLEAVE
% setenv CHUNK 4
```

The defaults are the same as on the **DOACROSS** directive; if neither variable is set, **SIMPLE** scheduling is assumed. If **MP_SCHEDTYPE** is set, but **CHUNK** is not set, a **CHUNK** of 1 is assumed. If **CHUNK** is set, but **MP_SCHEDTYPE** is not, **DYNAMIC** scheduling is assumed.

Environment Variable: MP_PROFILE

By default, the multiprocessing routines use the fastest possible method of doing their job. This can make it difficult to determine where the time is being spent if the multiprocessing routines themselves seem to be a bottleneck. By setting the environment variable **MP_PROFILE**, the multiprocessing routines use a slightly slower method of synchronization, where each step in the process is done in a separate subroutine with a long descriptive name. Thus *pixie* or pc−sample profiling can get more complete information regarding how much time is spent inside the multiprocessing routines.

Note: Only set/unset is important. The value the variable is set to is irrelevant (and typically is null).

mp_setlock, mp_unsetlock, mp_barrier

These zero−argument functions provide convenient (although limited) access to the locking and barrier functions provided by **ussetlock**(3p), **usunsetlock**(3p), and **barrier**(3p). The convenience is that no user initialization need be done because calls such as **usconfig**(3p) and **usinit**(3p) are done automatically. The limitation is that there is only one lock and one barrier. For a great many programs, this is sufficient. Users needing more complex or flexible locking facilities should use the **ussetlock** family of routines directly.

Local COMMON Blocks

A special *ld*(1) option allows named **COMMON** blocks to be local to a process. This means that each process in the parallel job gets its own private copy of the common block. This can be helpful in converting certain types of Fortran programs into a parallel form.

The common block must be a named **COMMON** (blank **COMMON** may not be made local), and it must not be initialized by **DATA** statements.

To create a local **COMMON** block, give the special loader directive
-Xlocaldata followed by a list of **COMMON** block names. Note that the external name of a **COMMON** block known to the loader has a trailing underscore and is not surrounded by slashes. For example, the command

```
% f77 −mp a.o −Xlocaldata foo_
```

would make the **COMMON** block **/foo/** be a local **COMMON** block in the resulting **a.out** file.

It is occasionally desirable to be able to copy values from the master thread’s version of the**COMMON** block into the slave thread’s version. The special directive **C$COPYIN** allows this. It has the form

```
C$COPYIN item [, item …]
```

Each *item* must be a member of a local **COMMON** block. It can be a variable, an array, an individual element of an array, or the entire **COMMON** block. For example,

```
C$COPYIN x,y, /foo/, a(i)
```

will propagate the values for **x** and **y**, all the values in the **COMMON** block **foo**, and the **i**th element of array **a**. All these items must be members of local **COMMON** blocks. Note that this directive is translated into executable code, so in this example **i** is evaluated at the time this statement is executed.

Compatibility With sproc

The parallelism used in Fortran is implemented using the standard system call **sproc**. It is recommended that programs not attempt to use both **C$DOACROSS** loops and **sproc** calls. It is possible, but there are several restrictions:

- Any threads you create may not execute **$DOACROSS** loops; only the original thread is allowed to do this.
- The calls to routines like **mp_block** and **mp_destroy** apply only to the threads created by **mp_create** or to those automatically created when the Fortran job starts; they have no effect on any user–defined threads.
- Calls to routines such as **m_get_numprocs**(3p) do not apply to the threads created by the Fortran routines. However, the Fortran threads are ordinary subprocesses; using the routine **kill**(2) with the arguments **0** and **sig** (**kill**(0,**sig**)) to signal all members of the process group might possibly result in the death of the threads used to execute **C$DOACROSS**.
- If you choose to intercept the **IGCLD** signal, you must be prepared to receive this signal when the threads used for the **C$DOACROSS** loops exit; this occurs when **mp_destroy** is called or at program termination.
- Note in particular that **m_fork**(3p) is implemented using **sproc**, so it is not legal to **m_fork** a family of processes that each subsequently executes **C$DOACROSS** loops. Only the original thread can execute **C$DOACROSS** loops.

DOACROSS Implementation

This section discusses how multiprocessing is implemented in a **DOACROSS** routine. This information is useful when you use the debugger and interpret the results of an execution profile.

Loop Transformation

When the Fortran compiler encounters a **C$DOACROSS** statement, it spools the corresponding **DO** loop into a separate subroutine and replaces the loop statement with a call to a special library routine. Exactly which routine is called depends on the value of **MP_SCHEDTYPE**. For discussion purposes, assume **SIMPLE** scheduling, so the library routine is **mp_simple_sched**.

The newly created subroutine is named using the following conventions. First, underscores are prepended and appended to the original routine name. For example, for a routine named **foo**, the first part of the name is **_foo_**. The next part of the name is the line number where the loop begins. This is the line number in the file, not the line number in the procedure. The last part of the name is a unique, four–character, alphabetic identifier. The first loop in a procedure uses **aaaa**, the second uses **aaab**, and so on. This "counter" is restarted to **aaaa** at the beginning of each procedure (not each file). So if the first parallel loop is at line 1234 in the routine named **foo**, the loop is named **_foo_1234_aaaa**. The second parallel loop, at line 1299, is named **_foo_1299_aaab**, and so on**.**

If a loop occurs in the **main** routine and if that routine has not been given a name by the **PROGRAM** statement, its name is assumed to be **main**. Any variables declared to be **LOCAL** in the original **C$DOACROSS** statement are declared as local variables in the spooled routine. References to **SHARE** variables are resolved by referring back to the original routine.

Because the spooled routine is now just a **DO** loop, the **mp_simple_sched** routine specifies, through subroutine arguments, which part of the loop a particular process is to execute. The spooled routine has four arguments: the starting value for the index, the number of times to execute the loop, the amount to increment the index, and a special flag word.

As an example, the following routine that appears on line 1000

```
   SUBROUTINE EXAMPLE(A, B, C, N)
   REAL A(*), B(*), C(*)
C$DOACROSS LOCAL(I,X)
   DO I = 1, N
      X = A(I)*B(I)
      C(I) = X + X**2
   END DO
   C(N) = A(1) + B(2)
   RETURN
   END
```

produces this spooled routine to represent the loop:

```
   SUBROUTINE _EXAMPLE_1000_aaaa
X ( _LOCAL_START, _LOCAL_NTRIP, _INCR, _THREADINFO)
   INTEGER*4 _LOCAL_START
   INTEGER*4 _LOCAL_NTRIP
   INTEGER*4 _INCR
   INTEGER*4 _THREADINFO
   INTEGER*4 I
   REAL X
   INTEGER*4 _DUMMY
   I = _LOCAL_START
   DO _DUMMY = 1,_LOCAL_NTRIP
       X = A(I)*B(I)
       C(I) = X + X**2
```

```
I = I + 1
END DO
END
```

Note: The compiler does not accept user code with an underscore (_) as the first letter of a variable name.

Executing Spooled Routines

The set of processes that cooperate to execute the parallel Fortran job are members of a process share group created by the system call **sproc**. The process share group is created by special Fortran start–up routines that are used only when the executable is linked with the **-mp** option, which enables multiprocessing.

The first process is the master process. It executes all the nonparallel portions of the code. The other processes are slave processes; they are controlled by the routine **mp_slave_control**. When they are inactive, they wait in the special routine **__mp_slave_wait_for_work**.

When the master process calls **mp_simple_sched**, the master passes the name of the spooled routine, the starting value of the **DO** loop index, the number of times the loop is to be executed, and the loop index increment.

The **mp_simple_sched** routine divides the work and signals the slaves. The master process then calls the spooled routine to do its work. When a slave is signaled, it wakes up from the wait loop, calculates which iterations of the spooled **DO** loop it is to execute, and then calls the spooled routine with the appropriate arguments. When a slave completes its execution of the spooled routine, it reports that it has finished and returns to **__mp_slave_wait_for_work**.

When the master completes its execution of the spooled routine, it returns to **mp_simple_sched**, then waits until all the slaves have completed processing. The master then returns to the main routine and continues execution.

Refer to Chapter 6 for an example of debugger output for the stack trace command *where*, which shows the calling sequence.