
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.
- “PCF Directives” describes how PCF implements a general model of parallelism.
- “Synchronization Intrinsic” describes synchronization operations.

Overview

The MIPSpro Fortran 77 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 wall-clock run time of the job. This chapter discusses techniques for analyzing your program and converting it to multiprocessing operations. Chapter 6,

“ Parallel Programming on Origin2000,” describes the support provided for writing parallel programs on Origin2000. Chapter 7, “ Compiling and Debugging Parallel Fortran,” gives compilation and debugging instructions for parallel processing.

Directives

Directives enable, disable, or modify a feature of the compiler. Essentially, directives are command line options specified within the input file instead of on the command line. Unlike command line options, directives have no default setting. To invoke a directive, you must either toggle it on or set a desired value for its level.

Directives allow you to enable, disable, or modify a feature of the compiler in addition to, or instead of, command line options. Directives placed on the first line of the input file are called *global directives*. The compiler interprets them as if they appeared at the top of each program unit in the file. Use global directives to ensure that the program is compiled with the correct command line options. Directives appearing anywhere else in the file apply only until the end of the current program unit. The compiler resets the value of the directive to the global value at the start of the next program unit. (Set the global value using a command line option or a global directive.)

Some command line options act like global directives. Other command line options override directives. Many directives have corresponding command line options. If you specify conflicting settings in the command line and a directive, the compiler chooses the most restrictive setting. For Boolean options, if either the directive or the command line has the option turned off, it is considered off. For options that require a numeric value, the compiler uses the minimum of the command line setting and the directive setting.

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, suppose a **DO** loop consisting of 200 iterations will run on a machine with four processors using the **SIMPLE** scheduling method (described in “ **CHUNK, MP_SCHEDTYPE**” on page 70). 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. By default, the multiprocessing code does not use more than 8 processors. If you want to use more processors, set the environment variable `MP_SET_NUMTHREADS` (see “ Environment Variables: `MP_SET_NUMTHREADS`, `MP_BLOCKTIME`, `MP_SETUP`” on page 96 for more information). 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 be run on single-processor machines. So the above loop is divided into one block of 200 iterations. This allows code to be developed on a single-processor Silicon Graphics workstation, 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 to the number of processors on the machine or 8, whichever is smaller. If you want, you can override the default and explicitly control the number of threads of execution used by a parallel 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 when 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, many 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 adopted 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 the proposed Parallel Computing Forum (PCF) standard (ANSI-X3H5 91-0023-B Fortran language binding) is available (refer to “ PCF Directives” on page 102). Additionally, a number of special routines exist that permit more direct control over the parallel execution (refer to “ Advanced Features” on page 93 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” on page 98. This section describes the others.

C\$DOACROSS

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

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

where valid values for the optional *clause* are

```
[ IF (logical_expression) ]  
[ {LOCAL | PRIVATE} (item[ ,item ... ] ) ]  
[ {SHARE | SHARED} (item[ ,item ... ] ) ]  
[ {LASTLOCAL | LAST LOCAL} (item[ ,item ... ] ) ]  
[ REDUCTION (item[ ,item ... ] ) ]  
[ MP_SCHEDTYPE=mode ]  
[ CHUNK=integer_expression ]
```

The preferred form of the directive (as generated by WorkShop Pro MPF) uses the optional commas between clauses. This section discusses the meaning of each clause.

IF

The **IF** clause determines whether the loop is actually executed in parallel. If the logical 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 4000 CPU clocks of work, which normally translates to about 1000 floating point operations.

LOCAL, SHARE, LASTLOCAL

The **LOCAL**, **SHARE**, and **LASTLOCAL** clauses specify lists of variables used within parallel loops. A variable can 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 Specifies variables that are local to each process. If a variable is declared as **LOCAL**, each iteration of the loop is given its own uninitialized copy of the variable. You can declare a variable as **LOCAL** if its value does not depend on any other iteration of the loop 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. The name **LOCAL** is preferred over **PRIVATE**.

SHARE Specifies variables that are shared across all processes. If a variable is declared as **SHARE**, all iterations of the loop use the same copy of the variable. You can declare a variable as **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. The name **SHARE** is preferred over **SHARED**.

LASTLOCAL Specifies variables that are local to each process. Unlike with the **LOCAL** clause, the compiler saves only the value of the logically last iteration of the loop when it exits. The name **LASTLOCAL** is preferred over **LAST LOCAL**.

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” on page 93). 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.

REDUCTION

The **REDUCTION** clause specifies variables involved in a reduction operation. In a reduction operation, the compiler keeps local copies of the variables and combines them when it exits the loop. For an example and details see Example 5-17 in “Breaking Data Dependencies.” An element of the **REDUCTION** list must be an individual variable (also called a scalar variable) and cannot be an array. However, it can be an individual element of an array. In a **REDUCTION** clause, it would appear in the list with the proper subscripts.

One element of an array can 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, the entire array can also appear in the **SHARE** list.

The four types of reductions supported are **sum(+)**, **product(*)**, **min()**, and **max()**. Note that **min(max)** reductions must use the **min(max)** intrinsic functions to be recognized correctly.

The compiler confirms that the reduction expression is legal by making some simple checks. The compiler does not, however, check all statements in the **DO** loop for illegal reductions. You must ensure that the reduction variable is used correctly in a reduction operation.

CHUNK, MP_SCHEDTYPE

The **CHUNK** and **MP_SCHEDTYPE** clauses affect the way the compiler schedules work among the participating tasks in a loop. These clauses do not affect the correctness of the loop. They are useful for tuning the performance of critical loops. See “Load Balancing” on page 91 for more details.

For the **MP_SCHEDTYPE=mode** clause, *mode* can be one of the following:

```
[SIMPLE | STATIC]
[DYNAMIC]
[INTERLEAVE INTERLEAVED]
[GUIDED GSS]
[RUNTIME]
```

You can use any or all of these modes in a single program. The **CHUNK** clause is valid only with the **DYNAMIC** and **INTERLEAVE** modes. **SIMPLE**, **DYNAMIC**, **INTERLEAVE**, **GSS**, and **RUNTIME** are the preferred names for each mode.

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

In dynamic scheduling (**MP_SCHEDTYPE=DYNAMIC**) the iterations are broken into pieces the size of which is specified with the **CHUNK** clause. 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 interleave method (**MP_SCHEDTYPE=INTERLEAVE**) breaks the iterations 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.

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 system can achieve good load balancing while reducing the number of entries into the critical section.

In addition to these four methods, you can specify the scheduling method at run time (**MP_SCHEDTYPE=RUNTIME**). Here, the scheduling routine examines values in your run-time environment and uses that information to select one of the other four methods. See “Advanced Features” on page 93 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 5-1 Simple DOACROSS

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 5-2 DOACROSS LOCAL

Consider the following code fragment:

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

You can be fully explicit, as shown below:

```
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
```

You can also 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 5-8 in “Analyzing Data Dependencies for Multiprocessing” on page 76 for more information on this example.

Example 5-3 DOACROSS LAST LOCAL

Consider the following code fragment:

```
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 shown below:

```
C$DOACROSS LOCAL(Y,J), LASTLOCAL(I,X),  
C$& 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
```

You can also 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 assigned the default value of **SHARE**, which produces an incorrect answer.

C\$&

Occasionally, the clauses in the **C\$DOACROSS** directive are longer than one line. Use the **C\$&** directive to continue the directive onto multiple lines. For example:

```
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.

The following code demonstrates the use of the **C\$** directive:

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

C\$MP_SCHEDTYPE and C\$CHUNK

The **C\$MP_SCHEDTYPE=*mode*** directive acts as an implicit **MP_SCHEDTYPE** clause for all **C\$DOACROSS** directives in scope. *mode* is any of the modes listed in the section called “**CHUNK, MP_SCHEDTYPE**” on page 70. A **C\$DOACROSS** directive that does not have an explicit **MP_SCHEDTYPE** clause is given the value specified in the last directive prior to the loop, rather than the normal default. If the **C\$DOACROSS** does have an explicit clause, then the explicit value is used.

The **C\$CHUNK=*integer_expression*** directive affects the **CHUNK** clause of a **C\$DOACROSS** in the same way that the **C\$MP_SCHEDTYPE** directive affects the **MP_SCHEDTYPE** clause for all **C\$DOACROSS** directives in scope. 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.

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.

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 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 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 or if it is passed to a function or subroutine, 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**. “**REDUCTION**” on page 70 describes the use of **REDUCTION** variables.

It is often difficult to analyze loops for data dependence information. Each use of each variable must be examined to determine if it fulfills the criteria for **LOCAL**, **LASTLOCAL**, **SHARE**, or **REDUCTION**. If all of the variables’ uses 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**” on page 82 for information on rewriting code in parallel form.)

An alternative to analyzing variable usage by hand is to use Power Fortran. This optional software package is a Fortran preprocessor that analyzes loops for data dependence. If Power Fortran determines that a loop is data-independent, it automatically inserts the required compiler directives (see “**Writing Parallel Fortran**” on page 68). If Power Fortran cannot determine whether the loop is independent, it produces a listing file detailing where the problems lie. You can use Power Fortran in conjunction with WorkShop Pro MPF to visualize these dependencies and make it easier to understand the obstacles to parallelization.

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

Example 5-4 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 5-5 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 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 5-6 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 5-7 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 5-8 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, does not modify global data, and does not use static storage. We say that **SQRT** has no *side effects*.

All the Fortran intrinsic functions listed in Appendix A of the *MIPSpro 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 user's responsibility to ensure that the routines can be correctly multiprocessed.

Caution: Do not use the **-static** option when compiling routines called within a parallel loop.

Example 5-9 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 5-14 in “Breaking Data Dependencies” on page 82).

Example 5-10 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 5-11 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 5-12 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 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 read the value before it is written and some after it is written is not important because they 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 5-13 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” on page 68). You can use WorkShop Pro MPF with MIPSpro Power Fortran 77 to identify the problem areas. Once you have identified these areas, you can use various techniques 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 5-14 Loop Carried Value

```
INDX = 0
DO I = 1, N
    INDX = INDX + I
    A(I) = B(I) + C(INDX)
END DO
```

This code segment is the same as in “Rewritable Data Dependence” on page 80. **INDX** has its value carried from iteration to iteration. However, you can compute the appropriate value for **INDX** without making reference to any previous value.

For example, consider the following code:

```
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 5-15 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. This will be true if **IXOFFSET** or **IYOFFSET** are permutation vectors.

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 5-16 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 5-17 Sum Reduction

```
SUM = 0.0
DO I = 1,N
  SUM = SUM + A(I)
END DO
```

This operation is known as a *reduction*. Reductions occur when an array of values is 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
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*PIECE_SIZE -PIECE_SIZE +1, MIN(K*PIECE_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

```

The previous code 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 do not produce the same results as the original code. 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. Both answers are equally “correct.” Most of the time the difference is irrelevant, but sometimes it can be significant, so some caution is in order. If the difference is significant, neither answer is really trustworthy.

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

1. sum: $p = p+a(i)$
2. product: $p = p*a(i)$
3. min: $m = \min(m,a(i))$
4. max: $m = \max(m,a(i))$

For example,

```
C$DOACROSS LOCAL(I), REDUCTION(BG_SUM, BG_PROD, BG_MIN, BG_MAX)
  DO I = 1, N
    BG_SUM = BG_SUM + A(I)
    BG_PROD = BG_PROD * A(I)
    BG_MIN = MIN(BG_MIN, A(I))
    BG_MAX = MAX(BG_MAX, A(I))
  END DO
```

One further example of a reduction transformation is noteworthy. Consider this code:

```
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 might look as if the inner loop should be parallelized with a **REDUCTION** clause. 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 5-18 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 shows 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 5-19 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 500.

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.500), 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 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.

Topics covered in this section include:

- “ Performing a Matrix Multiply” on page 89
- “ Understanding Trade-Offs” on page 90
- “ Load Balancing” on page 91
- “ Reorganizing Common Blocks To Improve Cache Behavior” on page 93

Performing a Matrix Multiply

Consider the following code segment:

```
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 5-18 in “ Work Quantum” on page 87 (after inter change). 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
```

Understanding 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
```

However, **M** needs to be large for the work quantum to show any improvement. In this example, **A(I)** is used to do a sum reduction, and it is possible to use the reduction techniques shown in Example 5-17 of “Breaking Data Dependencies” on page 82 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, the conversion can take more memory than you can spare. It can also take extra time to initialize the expanded array and increase the memory bandwidth requirements.

```

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 5-20 Load Balancing

```

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

```

The previous code segment 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, cures a load balancing problem.

You can use the **MP_SCHEDTYPE** clause to automatically perform this desirable transformation.

```
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
```

The previous code has the same meaning as the rewritten form above.

Note that interleaving can cause poor cache performance because the array is no longer stepped through at stride 1. You can improve performance somewhat by adding a **CHUNK=*integer_expression*** clause. Usually 4 or 8 is a good value for *integer_expression*. 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(1) or from pc sampling 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” on page 68 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 take a page fault, 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.

Reorganizing Common Blocks To Improve Cache Behavior

You can use the **-OPT:reorg_common** option, which reorganizes common blocks to improve the cache behavior of accesses to members of the common block. This option produces consistent results as long as the code follows the standard and array references are made within the bounds of the array. It produces unexpected results if you violate the standard, for example, if you access an array out of its declared bounds.

The option is enabled by default at **-O3** only if all files referencing the common block are compiled at that optimization level. It is disabled if any file with the common block is compiled at either **-O2** and below, **-OPT:reorg_common=OFF**, or **-Wl,-noivpad**.

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 puts the slave threads into a blocked state using the system call **blockproc**. The slave threads stay blocked until a call is made to **mp_unblock**. 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 if you neglect to do so.

mp_setup, mp_create, and mp_destroy

The **mp_setup**, **mp_create**, and **mp_destroy** 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 should 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 **MP_SET_NUMTHREADS** environment variable (described in “ Environment Variables: MP_SET_NUMTHREADS, MP_BLOCKTIME, MP_SETUP” on page 96), or by the number of CPUs on the current hardware platform. **mp_setup** is called automatically when the first parallel loop is entered 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)** reference 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** 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 one second. 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** 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 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_suggested_numthreads

The **mp_suggested_numthreads**(uint32) interface is available to any user program. You can call this routine at any time, and if possible, the suggested number of threads will be used at the start of the *next* parallel region. The implementation reserves the right to ignore the suggestion if it feels there is an over-riding reason to do so. You can call **mp_suggested_numthreads** directly. You can not vary the number of threads during execution of a parallel region. You can call **mp_suggested_numthreads** while in a parallel region, but the value will not take effect until the next one.

The automatic adjustment feature is enabled by setting the environment variable **MP_SUGNUMTHD** or **MPC_SUGNUMTHD**. The **mp_suggested_numthreads** interface is available whether or not the automatic feature is turned on.

mp_my_threadnum

mp_my_threadnum 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 cases.

Environment Variables: MP_SET_NUMTHREADS, MP_BLOCKTIME, MP_SETUP

The `MP_SET_NUMTHREADS`, `MP_BLOCKTIME`, and `MP_SETUP` environment variables act as an implicit call to the corresponding routine(s) of the same name at program start-up time.

For example, the `ssh` 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 multiple multiprocessors and multiple CPUs, the environment variable `MP_SET_NUMTHREADS` also accepts an expression involving integers `+`, `-`, `min`, `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_SUGNUMTHD, MP_SUGNUMTHD_MIN, MP_SUGNUMTHD_MAX, MP_SUGNUMTHD_VERBOSE

Prior to the current (6.02) compiler release, the number of threads utilized during execution of a multiprocessor job was generally constant, set for example using `MP_SET_NUMTHREADS`.

In an environment with long running jobs and varying workloads, it may be preferable to vary the number of threads during execution of some jobs.

Setting `MP_SUGNUMTHD` causes the run-time library to create an additional, asynchronous process that periodically wakes up and monitors the system load. When idle processors exist, this process increases the number of threads, up to a maximum of `MP_SET_NUMTHREADS`. When the system load increases, it decreases the number of threads, possibly to as few as 1. When `MP_SUGNUMTHD` has no value, this feature is disabled and multithreading works as before.

The environment variables `MP_SUGNUMTHD_MIN` and `MP_SUGNUMTHD_MAX` are used to limit this feature as desired. When `MP_SUGNUMTHD_MIN` is set to an integer value between 1 and `MP_SET_NUMTHREADS`, the process will not decrease the number of threads below that value.

When `MP_SUGNUMTHD_MAX` is set to an integer value between the minimum number of threads and `MP_SET_NUMTHREADS`, the process will not increase the number of threads above that value.

If you set any value in the environment variable `MP_SUGNUMTHD_VERBOSE`, informational messages are written to `stderr` whenever the process changes the number of threads in use.

Calls to `mp_numthreads` and `mp_set_numthreads` are taken as a sign that the application depends on the number of threads in use. The number in use is frozen upon either of these calls; and if `MP_SUGNUMTHD_VERBOSE` is set, a message to that effect is written to `stderr`.

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.

mp_setlock, mp_unsetlock, mp_barrier

mp_setlock, **mp_unsetlock**, and **mp_barrier** are zero-argument subroutines that provide convenient (although limited) access to the locking and barrier functions provided by **ussetlock**, **usunsetlock**, and **barrier**. These subroutines are convenient because you do not need to initialize them; calls such as **usconfig** and **usinit** are done automatically. The limitation is that there is only one lock and one barrier. For most programs, this amount is sufficient. If your program requires more complex or flexible locking facilities, use the **ussetlock** family of subroutines directly.

Local COMMON Blocks

A special *ld* option allows named **COMMON** blocks to be local to a process. 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 **-Wl,Xlocal** 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 -Wl,Xlocal,foo_
```

makes the **COMMON** block **/foo/** a local **COMMON** block in the resulting **a.out** file. You can specify multiple **-Wl,Xlocal** options if necessary.

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.

Note: The **C\$COPYIN** directive cannot be executed from inside a parallel region.

For example,

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

propagates 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** do not apply to the threads created by the Fortran routines. However, the Fortran threads are ordinary subprocesses; using the routine **kill** with the arguments **0** and **sig** (for example, **kill(0,sig)**) to signal all members of the process group might kill threads used to execute **C\$DOACROSS**. If you choose to intercept the **SIGCLD** 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** 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 a debugger or interpret the results of an execution profile.

Loop Transformation

When the Fortran compiler encounters a **C\$DOACROSS** directive, it spools the body of the corresponding **DO** loop into a separate subroutine and replaces the loop with a call to a special library routine **__mp_parallel_do**.

The newly created routine is named by appending **.pregion** to the name of the original routine, followed by the number of the parallel loop in the routine (where 0 is the first loop). For example, the first parallel loop in a routine named **foo** is named **foo.pregion0**, the second parallel loop is **foo.pregion1**, 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 routine uses subroutine arguments to specify which part of the loop a particular process is to execute. The spooled routine has three arguments: the starting value for the index, the number of times to execute the loop, 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.pregion
X ( _LOCAL_START, _LOCAL_NTRIP, _THREADINFO)
INTEGER*4 _LOCAL_START
INTEGER*4 _LOCAL_NTRIP
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

```

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**.

The **__mp_parallel_do** routine divides the work and signals the slaves. The master process then calls the spooled routine to do its share of the 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 its portion of the spooled routine, it waits in the special routine **mp_wait_for_loop_completion** until all the slaves have completed processing. The master then returns to the main routine and continues execution.

PCF Directives

In addition to the simple loop-level parallelism offered by the **C\$DOACROSS** directive (described in “ Parallel Loops” on page 66), the compiler supports a more general model of parallelism. This model is based on the work done by the Parallel Computing Forum (PCF), which itself formed the basis for the proposed ANSI-X3H5 standard. The compiler supports this model through compiler directives, rather than extensions to the source language.

The main concept in this model is the *parallel region*, which can be any arbitrary section of code (not just a **DO** loop). Within the parallel region, there are special *work-sharing constructs* that can be used to divide the work among separate processes or threads. The parallel region can also contain a *critical section* construct, where exactly one process executes at a time.

The master thread executes the user program until it reaches a parallel region. It then spawns one or more slave threads that begin executing code at the beginning of a parallel region. Each thread executes all the code in the region until a work sharing construct is encountered. Each thread then executes some portion of the work sharing construct, and then resumes executing the parallel region code. At the end of the parallel region, all the threads synchronize, and the master thread continues execution of the user program.

The PCF directives, summarized in Table 5-1, implement the general model of parallelism. They look like Fortran comments, with a **C** in column one. The compiler recognizes these directives when multiprocessing is enabled with either the **-mp** option. (Multiprocessing is also enabled with the **-pfa** option if you have purchased MIPSpro Power Fortran 77.) If multiprocessing is not enabled, the compiler treats these statements as comments. Therefore, you can compile identical source with a single-processing compiler or by Fortran without the multiprocessing option. The PCF directives start with the characters **C\$PAR**.

Table 5-1 Summary of PCF Directives

Directive	Description
C\$PAR BARRIER	Ensures that each process waits until all processes reach the barrier before proceeding.
C\$PAR [END] CRITICAL SECTION	Ensures that the enclosed block of code is executed by only one process at a time by using a lock variable.
C\$PAR [END] PARALLEL	Encloses a parallel region, which includes work-sharing constructs and critical sections.
C\$PAR PARALLEL DO	Precedes a single DO loop for which separate iterations are executed by different processes. This directive is equivalent to the C\$DOACROSS directive.
C\$PAR [END] PDO	Separate iterations of the enclosed loop are executed by different processes. This directive must be inside a parallel region.
C\$PAR [END] PSECTION[S]	Parcels out each block of code in turn to a process.
C\$PAR SECTION	Signifies a starting line for an individual section within a parallel section.
C\$PAR [END] SINGLE PROCESS	Ensures that the enclosed block of code is executed by exactly one process.
C\$PAR &	Continues a PCF directive onto multiple lines.

Parallel Region

A parallel region encloses any number of PCF constructs (described in “PCF Constructs” on page 104). It signifies the boundary within which slave threads execute. A user program can contain any number of parallel regions. The syntax of the parallel region is

```
C$PAR PARALLEL [clause [, clause]...]
           code
C$PAR END PARALLEL
```

where valid clauses are

```
[IF ( logical_expression )]
[ {LOCAL | PRIVATE} (item [, item ...]) ]
[ {SHARE | SHARED} (item [, item ...]) ]
```

The **IF**, **LOCAL**, and **SHARED** clauses have the same meaning as in the **C\$DOACROSS** directive (refer to “Writing Parallel Fortran” on page 68).

The preferred form of the directive has no commas between the clauses. The **SHARED** clause is preferred over **SHARE** and **LOCAL** is preferred over **PRIVATE**.

In the following code, all threads enter the parallel region and call the routine **foo**:

```
subroutine ex1(index)
integer i
C$PAR PARALLEL LOCAL(i)
i = mp_my_threadnum()
call foo(i)
C$PAR END PARALLEL
end
```

PCF Constructs

The three types of PCF constructs are work-sharing constructs, critical sections, and barriers. All master and slave threads synchronize at the bottom of a work-sharing construct. None of the threads continue past the end of the construct until they all have completed execution within that construct.

The four work-sharing constructs are

- parallel **DO**
- PDO
- parallel sections
- single process

If specified, the PDO, parallel section, and single process constructs must appear inside of a parallel region; the parallel **DO** construct cannot. Specifying a parallel **DO** construct inside of a parallel region produces a syntax error.

The critical section construct protects a block of code with a lock so that it is executed by only one thread at a time. Threads do not synchronize at the bottom of a critical section.

The barrier construct ensures that each process that is executing waits until all others reach the barrier before proceeding.

Parallel DO

The parallel **DO** construct is the same as the **C\$DOACROSS** directive (described in “**C\$DOACROSS**” on page 68) and conceptually the same as a parallel region containing exactly one PDO construct and no other code. Each thread inside the enclosing parallel region executes separate iterations of the loop within the parallel **DO** construct. The syntax of the parallel **DO** construct is

```
C$PAR PARALLEL DO [clause [[,] clause]. . .]
```

“**C\$DOACROSS**” on page 68 describes valid values for *clause* with the exception of the **MP_SCHEDTYPE=*mode*** clause. For the **C\$PAR PARALLEL DO** directive, **MP_SCHEDTYPE=** is optional; you can just specify *mode*.

PDO

Each thread inside the enclosing parallel region executes a separate iteration of the loop within the PDO construct. The syntax of the PDO construct, which can only be specified within a parallel region, is

```
C$PAR PDO [clause [[,] clause]. . .]  
      code  
[C$PAR END PDO [NOWAIT]]
```

where valid values for *clause* are

```
[{LOCAL | PRIVATE} (item[,item ...])]
[ {LASTLOCAL | LAST LOCAL} (item[,item ...])]
[ (ORDERED) ]
[ sched ]
[ chunk ]
```

LOCAL, **LASTLOCAL**, *sched*, and *chunk* have the same meaning as in the **C\$DOACROSS** directive (refer to “ Writing Parallel Fortran” on page 68). Note in particular that it is legal to declare a data item as **LOCAL** in a PDO even if it was declared as **SHARED** in the enclosing parallel region. The **(ORDERED)** clause is equivalent to a *sched* clause of **DYNAMIC** and a *chunk* clause of **1**. The parenthesis are required.

LASTLOCAL is preferred over **LAST LOCAL** and **LOCAL** is preferred over **PRIVATE**.

The **END PDO** directive is optional. If specified, this directive must appear immediately after the end of the **DO** loop. The optional **NOWAIT** clause specifies that each process should proceed directly to the code immediately following the directive. If you do not specify **NOWAIT**, the processes will wait until all have reached the directive before proceeding.

As an example of the PDO construct, consider the following code:

```
subroutine ex2(a,n)
  real a(n)
C$PAR PARALLEL local(i) shared(a)
C$PAR PDO
  do i = 1, n
    a(i) = a(i) + 1.0
  enddo
C$PAR END PARALLEL
end
```

This sample code is the same as a **C\$DOACROSS** loop. In fact, the compiler recognizes this as a special case and generates the same (more efficient) code as for a **C\$DOACROSS** directive.

Parallel Sections

The parallel sections construct is a parallel version of the Fortran 90 **SELECT** statement. Each block of code is parcelled out in turn to a separate thread. The syntax of the parallel sections construct is

```
C$PAR PSECTION[S] [clause [, clause] ...
    code
[C$PAR SECTION
    code] ...
C$PAR END PSECTION[S] [NOWAIT]
```

where the only valid value for *clause* is

```
[{LOCAL | PRIVATE} (item [, item]) ]
```

LOCAL is preferred over **PRIVATE** and has the same meaning as for the **C\$DOACROSS** directive (refer to “ **C\$DOACROSS**” on page 68). Note in particular that it is legal to declare a data item as **LOCAL** in a parallel sections construct even if it was declared as **SHARED** in the enclosing parallel region.

The optional **NOWAIT** clause specifies that each process should proceed directly to the code immediately following the directive. If you do not specify **NOWAIT**, the processes will wait until all have reached the **END PSECTION** directive before proceeding.

Parallel sections must appear within a parallel region. They can contain critical section constructs (described in “ **Critical Section**” on page 112) but cannot contain any of the following types of constructs:

- PDO
- parallel **DO** or **C\$DOACROSS**
- single process

Each code block is executed in parallel (depending on the number of processes available). The code blocks are assigned to threads one at a time, in the order specified. Each code block is executed by only one thread.

For example, consider the following code:

```
      subroutine ex3(a,n1,b,n2,c,n3)
      real a(n1), b(n2), c(n3)
C$PAR PARALLEL local(i) shared(a,b,c)
C$PAR PSECTIONS
C$PAR SECTION
      do i = 1, n1
        a(i) = 0.0
      enddo
C$PAR SECTION
      do i = 1, n2
        b(i) = 0.5
      enddo
C$PAR SECTION
      call normalize(c,n3)
      do i = 1, n3
        c(i) = c(i) + 1.0
      enddo
C$PAR END PSECTION
C$PAR END PARALLEL
      end
```

The first thread to enter the parallel sections construct executes the first block, the second thread executes the second block, and so on. This example has only three sections, so if more than three threads are in the parallel region, the fourth and higher threads wait at the **C\$PAR END PSECTION** directive until all threads are finished. If the parallel region is being executed by only two threads, whichever thread finishes its block first continues and executes the remaining block.

This example uses **DO** loops, but a parallel section can be any arbitrary block of code. Be aware of the significant overhead of a parallel construct. Make sure the amount of work performed is enough to outweigh the extra overhead.

The sections within a parallel sections construct are assigned to threads one at a time, from the top down. There is no other implied ordering to the operations within the sections. In particular, a later section cannot depend on the results of an earlier section, unless some form of explicit synchronization is used. If there is such explicit synchronization, you must be sure that the lexical ordering of the blocks is a legal order of execution.

Single Process

The single process construct, which can only be specified within a parallel region, ensures that a block of code is executed by exactly one process. The syntax of the single process construct is

```
C$PAR SINGLE PROCESS [clause [[,] clause]...]
    code
C$PAR END SINGLE PROCESS [NOWAIT]
```

where the only valid value for *clause* is

```
[{LOCAL | PRIVATE} (item [,item]) ]
```

LOCAL is preferred over **PRIVATE** and has the same meaning as for the **C\$DOACROSS** directive (refer to “ C\$DOACROSS” on page 68). Note in particular that it is legal to declare a data item as **LOCAL** in a single process construct even if it was declared as **SHARED** in the enclosing parallel region.

The optional **NOWAIT** clause specifies that each process should proceed directly to the code immediately following the directive. If you do not specify **NOWAIT**, the processes will wait until all have reached the directive before proceeding.

This construct is semantically equivalent to a parallel sections construct with only one section. The single process construct provides a more descriptive syntax. For example, consider the following code:

```
    real function ex4(a,n, big_max, bmax_x, bmax_y)
    real a(n,n), big_max
    integer bmax_x, bmax_y
C$ volatile big_max, bmax_x, bmax_y
C$ volatile cur_max, index_x, index_y
    index_x = 0
    index_y = 0
    cur_max = 0.0
C$PAR PARALLEL local(i,j)
C$PAR& shared(a,n,index_x,index_y,cur_max,
C$PAR& big_max,bmax_x,bmax_y)
C$PAR PDO
    do j = 1, n
        do i = 1, n
            if (a(i,j) .gt. cur_max) then
```

```
C$PAR CRITICAL SECTION
      if (a(i,j) .gt. cur_max) then
          index_x = i
          index_y = j
          cur_max = a(i,j)
      endif
C$PAR END CRITICAL SECTION
      endif
      enddo
      enddo

C$PAR SINGLE PROCESS
      if (cur_max .gt. big_max) then
          big_max = (big_max + cur_max) / 2.0
          bmax_x = index_x
          bmax_y = index_y
      endif
C$PAR END SINGLE PROCESS

C$PAR PDO
      do j = 1, n
          do i = 1, n
              a(i,j) = a(i,j)/big_max
          enddo
      enddo

C$PAR END PARALLEL

      ex4 = cur_max

      end
```

The first thread to reach the single process section executes the code in that block. All other threads wait at the end of the block until the code has been executed.

This example contains a number of interesting points to be examined. First, note the use of the **VOLATILE** declaration. Any data item that might be written by one thread and then read by a different thread must be marked as **VOLATILE**. Making a variable **VOLATILE** can reduce opportunities for optimization, so the declarations are prefixed by **C\$** to prevent the single-processor version of the code from being penalized. Refer to the *MIPSpro Fortran 77 Language Reference Manual* for more information about the **VOLATILE** statement.

Second, note the use of the odd looking repetition of the **IF** test in the first parallel loop:

```

        if (a(i,j) .gt. cur_max) then
C$PAR CRITICAL SECTION
        if (a(i,j) .gt. cur_max) then

```

This practice is usually called *test&test&set*. It is a multi-processing optimization. Note that the following straight forward code segment is incorrect:

```

        do i = 1, n
        if (a(i,j) .gt. cur_max) then
C$PAR CRITICAL SECTION
        index_x = i
        index_y = j
        cur_max = a(i,j)
C$PAR END CRITICAL SECTION
        endif
        enddo

```

Because many threads execute the loop in parallel, there is no guarantee that once inside the critical section, **cur_max** still has the same value it did in the **IF** test outside the critical section (some other thread may have updated it). In particular, **cur_max** may now have a value that is larger than **a(i,j)**. Therefore, the critical section must be locked before testing the value of **cur_max**. Changing the previous code into the equally straightforward

```

        do i = 1, n
C$PAR CRITICAL SECTION
        if (a(i,j) .gt. cur_max) then
        index_x = i
        index_y = j
        cur_max = a(i,j)
        endif
C$PAR END CRITICAL SECTION
        enddo

```

works correctly, but suffers from a serious performance penalty: the critical section lock must be acquired and released (an expensive operation) for each element of the array. Because the values are rarely updated, this process involves a lot of wasted effort. It is almost certainly slower than just executing the loop serially.

Combining the two methods, as in the original example, produces code that is both fast and correct. If the **IF** test outside of the critical section fails, you can be certain that the values will not be updated, and can proceed. You can expect that the outside **IF** test will

account for the majority of cases. If the outer **IF** test passes, then the values *might* be updated, but you cannot always be certain. To ensure correctness, you must perform the test again after acquiring the critical section lock.

You can prefix one of the two identical **IF** tests with **C\$** to reduce overhead in the non-multiprocessed case.

Lastly, note the difference between the single process and critical section constructs. If several processes arrive at a critical section construct, they execute the code one at a time. However, they will *all* execute the code. If several processes arrive at a single process construct, only one process executes the code. The other processes bypass the code and wait at the end of the construct for the chosen process to finish.

Critical Section

The critical section construct restricts execution of a block of code so that only one process can execute it at a time. Another process attempting to gain entry to the critical section must wait until the previous process has exited.

The critical section construct can appear anywhere in a program, including inside and outside a parallel region and within a **C\$DOACROSS** loop. The syntax of the critical section construct is

```
C$PAR CRITICAL SECTION [ ( lock_variable ) ]
    code
C$PAR END CRITICAL SECTION
```

The *lock_variable* is an optional integer variable that must be initialized to zero. The parenthesis are required. If you do not specify *lock_variable*, the compiler automatically supplies one. Multiple critical section constructs inside the same parallel region are considered to be independent of each other unless they use the same explicit *lock_variable*.

Consider the following code:

```
integer function num_exceptions(a,n,biggest_allowed)
double precision a(n,n,n), biggest_allowed

integer count
integer lock_var

volatile count

count = 0
lock_var = 0
```

```
C$PAR PARALLEL local(i,j,k) shared(count,lock_var)
C$PAR PDO
  do 10 k = 1,n
    do 10 j = 1,n
      do 10 i = 1,n
        if (a(i,j,k) .gt. biggest_allowed) then

C$PAR CRITICAL SECTION (lock_var)
          count = count + 1
C$PAR END CRITICAL SECTION (lock_var)

          else
            call transform(a(i,j,k))
            if (a(i,j,k) .gt. biggest_allowed) then

C$PAR CRITICAL SECTION (lock_var)
              count = count + 1
C$PAR END CRITICAL SECTION (lock_var)

            endif
          endif
        10 continue
C$PAR END PARALLEL

  num_exceptions = count

  return
end
```

This example demonstrates the use of the lock variable (**lock_var**). A **C\$PAR CRITICAL SECTION** directive ensures that no more than one process executes the enclosed block of code at a time. However, if there are multiple critical sections, different processes can be in different critical sections at the same time. This example does not allow different processes to be in different critical sections at the same time because both critical sections control access to the same variable (**count**). Specifying the same lock variable for both critical sections ensures that no more than one process is executing either of the critical sections that use that lock variable. Note that the **lock_var** must be **SHARED** (so that all processes use the same lock), and that **count** must be **volatile** (because other processes might change its value).

Barrier Constructs

A barrier construct ensures that each process waits until all processes reach the barrier before proceeding. The syntax of the barrier construct is

```
C$PAR BARRIER
```

C\$PAR &

Occasionally, the clauses in PCF directives are longer than one line. You can use the **C\$PAR &** directive to continue a directive onto multiple lines.

For example,

```
C$PAR PARALLEL local(i,j)
C$PAR& shared(a,n,index_x,index_y,cur_max,
C$PAR& big_max,bmax_x,bmax_y)
```

Restrictions

The three work-sharing constructs, **PDO**, **PSECTION**, and **SINGLE PROCESS**, must be executed by all the threads executing in the parallel region (or none of the threads). The following is illegal:

```
.
.
.
C$PAR PARALLEL
    if (mp_my_threadnum() .gt. 5) then
C$PAR SINGLE PROCESS
        many_processes = .true.
C$PAR END SINGLE PROCESS
    endif
.
.
.
```

This code will hang forever when run with enough processes. One or more process will be stuck at the **C\$PAR END SINGLE PROCESS** directive waiting for all the threads to arrive. Because some of the threads never took the appropriate branch, they will never encounter the construct. However, the following kind of simple looping is supported:

```
code
C$PAR PARALLEL local(i,j) shared(a)
    do i= 1,n
C$PAR PDO
    do j = 2,n
code
```

The distinction here is that all of the threads encounter the work-sharing construct, they all complete it, and they all loop around and encounter it again.

Note that this restriction does not apply to the critical section construct, which operates on one thread at a time without regard to any other threads.

Parallel regions cannot be lexically nested inside of other parallel regions, nor can work-sharing constructs be nested. However, as an aid to writing library code, you can call an external routine that contains a parallel region even from within a parallel region. In this case, only the first region is actually run in parallel. Therefore, you can create a parallelized routine without accounting for whether it will be called from within an already parallelized routine.

A Few Words About Efficiency

The more general PCF constructs are typically slower than the special case parallelism offered by the **C\$DOACROSS** directive. They are slower because of the extra synchronization required. When a **C\$DOACROSS** loop executes, there is a synchronization point at entry and another at exit. When a parallel region executes, there is a synchronization point at entry to the region, another at each entry to a work-sharing construct, another at each exit from a work-sharing construct, and one at exit from the region. Thus, several separate **C\$DOACROSS** loops typically execute faster than a single parallel region with several PDO constructs. Limit your use of the parallel region construct to those few cases that actually need it.

Synchronization Intrinsic

The intrinsic described in this section provide a variety of primitive synchronization operations. Besides performing the particular synchronization operation, each of these intrinsic has two key properties:

- The function performed is guaranteed to be atomic (typically achieved by implementing the operation using a sequence of load-linked and/or store-conditional instructions in a loop).
- Associated with each intrinsic are certain *memory barrier* properties that restrict the movement of memory references to *visible data* across the intrinsic operation (by either the compiler or the processor).

A *visible memory reference* is a reference to a data object potentially accessible by another thread executing in the same shared address space. A visible data object can be one of the following types:

- Fortran COMMON data
- data declared extern
- volatile data
- static data (either file-scope or function-scope)
- data accessible via function parameters
- automatic data (local-scope) that has had its address taken and assigned to some object that is visible (recursively)

The memory barrier semantics of an intrinsic can be one of the following types:

- *acquire barrier*, which disallows the movement of memory references to visible data from after the intrinsic (in program order) to before the intrinsic (this behavior is desirable at lock-acquire operations)
- *release barrier*, which disallows the movement of memory references to visible data from before the intrinsic (in program order) to after the intrinsic (this behavior is desirable at lock-release operations)
- *full barrier*, which disallows the movement of memory references to visible data past the intrinsic (in either direction), and is thus both an acquire and a release barrier. A barrier only restricts the movement of memory references to visible data across the intrinsic operation: between synchronization operations (or in their absence), memory references to visible data may be freely reordered subject to the usual data-dependence constraints.

Caution: Conditional execution of a synchronization intrinsic (such as within an **if** or a **while** statement) does not prevent the movement of memory references to visible data past the overall **if** or **while** construct.

Atomic fetch-and-op Operations

```
<type> __fetch_and_add (<type>* ptr, <type> value)
<type> __fetch_and_sub (<type>* ptr, <type> value)
<type> __fetch_and_or (<type>* ptr, <type> value)
<type> __fetch_and_and (<type>* ptr, <type> value)
<type> __fetch_and_xor (<type>* ptr, <type> value)
<type> __fetch_and_nand(<type>* ptr, <type> value)
```

Where <type> can be one of:

```
int
long
long long
unsigned int
unsigned long
unsigned long long
```

Behavior:

1. Atomically performs the specified operation with the given value on *ptr, and returns the old value of *ptr.

```
{ tmp = *ptr; *ptr <op>= value; return tmp; }
```

2. Full barrier.

Atomic op-and-fetch Operations

```
<type> __add_and_fetch (<type>* ptr, <type> value)
<type> __sub_and_fetch (<type>* ptr, <type> value)
<type> __or_and_fetch (<type>* ptr, <type> value)
<type> __and_and_fetch (<type>* ptr, <type> value)
<type> __xor_and_fetch (<type>* ptr, <type> value)
<type> __nand_and_fetch(<type>* ptr, <type> value)
```

Where <type> can be one of the following:

```
int
long
long long
unsigned int
unsigned long
unsigned long long
```

Behavior:

1. Atomically performs the specified operation with the given value on *ptr, and returns the new value of *ptr.

```
{ *ptr <op>= value; return *ptr; }
```

2. Full barrier.

Atomic BOOL Operation

```
BOOL __compare_and_swap (<type>* ptr, <type> oldvalue, <type> newvalue)
```

Where <type> can be one of the following:

```
int
long
long long
unsigned int
unsigned long
unsigned long long
```

Behavior:

1. Atomically do the following: compare *ptr to old value. If equal, store the new value and return 1, otherwise return 0.

```
if (*ptr != oldvalue) return 0;
else {
    *ptr = newvalue
    return 1;
}
```

2. Full barrier.

Atomic synchronize Operation

```
__synchronize ()
```

Behavior:

1. Full barrier.

Atomic lock and unlock Operations

```
<type> __lock_test_and_set (<type>* ptr, <type> value)
```

Where <type> can be one of the following:

```
int
long
long long
unsigned int
unsigned long
unsigned long long
```

Behavior:

1. Atomically store the supplied value in *ptr and return the old value of *ptr.
{ tmp = *ptr; *ptr = value; return tmp; }
2. Acquire barrier.

```
void __lock_release (<type>* ptr)
```

Where <type> can be one of the following:

```
int
long
long long
unsigned int
unsigned long
unsigned long long
```

Behavior:

1. Set *ptr to 0.
{ *ptr = 0 }
2. Release barrier.

Example of Implementing a Pure Spin-Wait Lock

The following example shows implementation of a spin-wait lock.

```
int lockvar = 0;
while (__lock_test_and_set (&lockvar, 1) != 0); /* acquire the lock */
    ... read and update shared variables ...
__lock_release (&lockvar); /* release the lock */
```

The memory barrier semantics of the intrinsics guarantee that no memory reference to visible data is moved out of the above critical section, either before of the lock-acquire or after the lock-release.

Note: Pure spin-wait locks can perform poorly under heavy contention.

If the data structures protected by the lock are known precisely (for example, *x*, *y*, and *z* in the example below), then those data structures can be precisely identified as follows:

```
while (__lock_test_and_set (&lockvar, 1, x, y, z) != 0);
    ... read/modify the variables x, y, and z ...
__lock_release (&lockvar, x, y, z);
```