Chapter 9.
Multiprocessing #pragma Directives

Table 10, contains an alphabetical list of the #pragma directives discussed in this chapter, along with a brief description of each and the compiler versions in which the directive is supported.

Table 10. Multiprocessing #pragma Directives

#pragma	Short Description	Compiler Versions
#pragma copyin	Copies the value from the master thread's version of an -Xlocal–linked global variable into the slave thread's version.	6.0 and la
#pragma critical	Protects access to critical statements.	6.0 and la
#pragma enter gate(see Section 9.3)	Indicates the point that all threads must clear before any threads are allowed to pass the corresponding exit gate.	6.0 and la
#pragma exit gate(see Section 9.3)	Stops threads from passing this point until all threads have cleared the corresponding enter gate.	6.0 and la
#pragma independent	Tells the compiler to run independent code section in parallel with the rest of the code in the parallel region.	6.0 and la
#pragma local	Tells the compiler the names of all the variables that must be local to each thread.	6.0 and la
#pragma no side effects	Tells the compiler to assume that all of the named functions are safe to execute concurrently.	7.1 and la
#pragma one processor	Causes the next statement to be executed on only one processor.	6.0 and la
#pragma parallel(see also Section 9.9)	Marks the start of a parallel region.	6.0 and la
#pragma pfor (see also Section 9.11)	Marks a for loop to run in parallel.	6.0 and la
#pragma pure	Tells the compiler to use only values of named functions to computer return value with no side effects.	7.3 and la
#pragma set chunksize	Tells the compiler which values to use for chunksize.	6.0 and la
#pragma set numthreads	Tells the compiler which values to use for numthreads.	6.0 and la
#pragma set schedtype	Tells the compiler which values to use for schedtype.	6.0 and la
#pragma shared	Tells the compiler the names of all the variables that the threads must share.	6.0 and la
#pragma synchronize	Stops threads until all threads reach this point.	6.0 and la

9.1 #pragma copyin

The #pragma copyin directive allows you to copy values from the master thread's version of an -Xlocal–linked global variable into the slave thread's version.

#pragma copyin has the following syntax:

```
#pragma copyin item1[, item2 ...]
```

Each *item* must be a localized (that is, linked -Xlocal) global variable.

Do not place this directive inside a parallel region.

9.1.1 Example of #pragma copyin

The following line of code demonstrates the use of the #pragma copyin directive:

```
#pragma copyin x,y, A[i]
```

This propagates the master thread's values for *x*, *y*, and the *i*th element of array *A* into each slave thread's copy of the corresponding variable. All of these items must be linked -Xlocal. This directive is translated into executable code, so in this example *i* is evaluated at the time this statement is executed.

9.2 #pragma critical

Sometimes the bulk of the work done by a loop can be done in parallel, but the entire loop cannot run in parallel because of a single data–dependent statement. Often, you can move such a statement out of the parallel region. When that is not possible, you can use the #pragma critical directive to place a lock on the statement to preserve the integrity of the data.

The syntax of the #pragma critical directive is as follows:

```
#pragma critical [(lock_variable)]  
{ code }
```

The statement after the #pragma critical directive code is executed by all threads, one at a time.

In the multiprocessing C/C++ compiler, you can use the #pragma critical directive to put a lock on a critical statement (or compound statement using { }). When you put a lock on a statement, only one thread at a time can execute that statement. If one thread is already working on a #pragma critical protected statement, any other thread that needs to execute that statement must wait until the first thread has finished executing it.

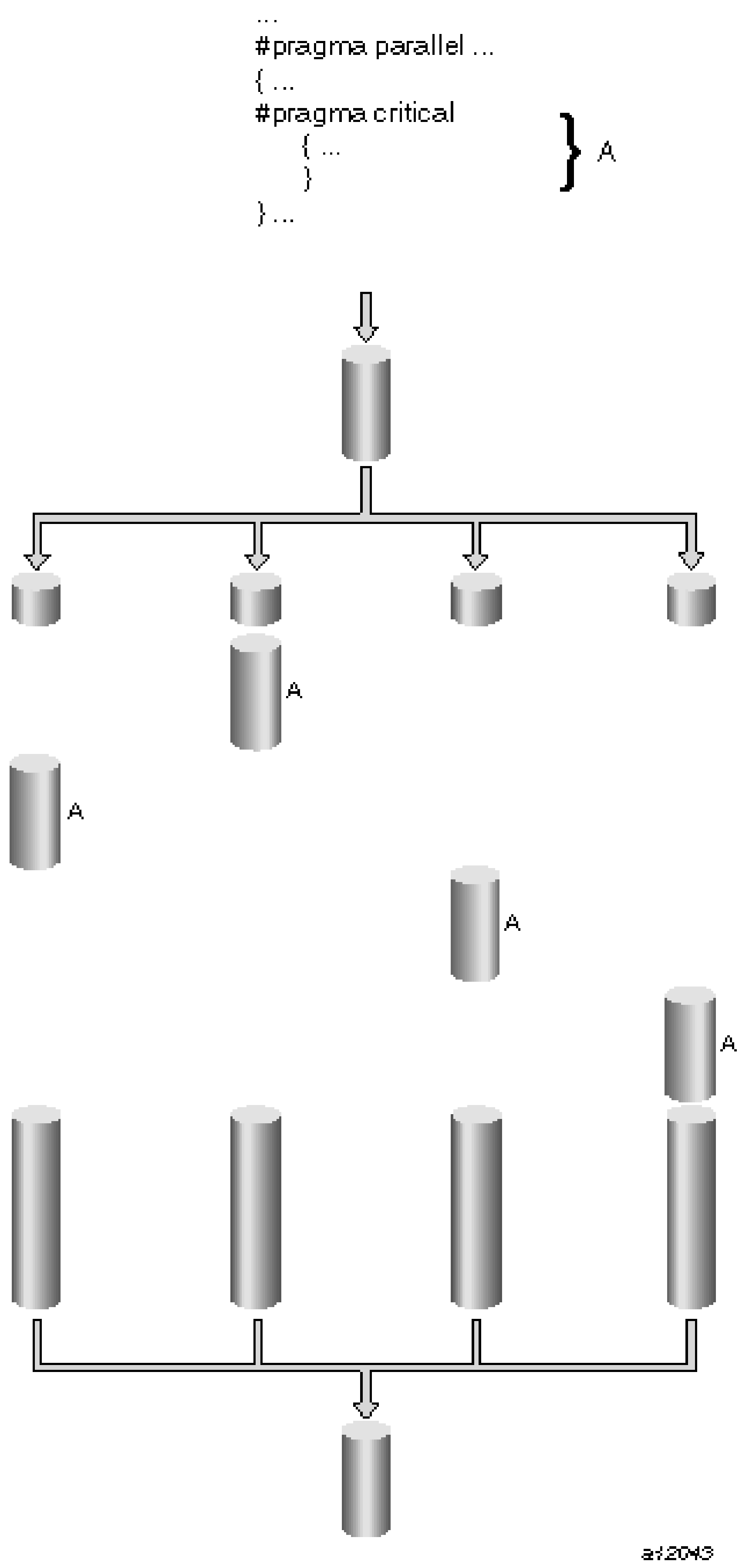
The lock variable is an optional integer variable that must be initialized to zero. The parentheses are required. If you do not specify a lock variable, the compiler automatically uses a global lock variable. Multiple critical constructs inside the same parallel region are considered to be dependent on each other unless they use distinct explicit lock variables.

Caution: This #pragma directive works slightly differently in the IRIS POWER C Analyzer (PCA) for compiler versions 7.1 and older. See the*IRIS POWER C User's Guide* for more information.

9.2.1 Diagram of #pragma critical

Figure 1, illustrates critical segment execution.

Figure 1. Critical Segment Execution


...
#pragma parallel ...
{ ...
#pragma critical
{ ...
}
} ...
A
A
A
A
A
ai2043

9.3 `#pragma enter gate` and `#pragma exit gate`

The `#pragma enter gate` and `#pragma exit gate` directives provide an additional tool for coordinating the processing of code within a parallel region. These directives work as a matched set, by establishing a section of code bounded by gates at the beginning and end. These gates form a special barrier. No thread can exit a gated region until all threads have entered it. This construct gives more flexibility when managing dependences between the work−sharing constructs in a parallel region.

By using `#pragma enter gate` and `#pragma exit gate` pairs, you can make subtle distinctions about which construct is dependent on which other construct.

The syntax of the `#pragma enter gate` directive is as follows:

```
#pragma enter gate
```

Put this directive after the work−sharing construct that all threads must clear before any can pass `#pragma exit gate`.

The syntax of the `#pragma exit gate` directive is as follows:

```
#pragma exit gate
```

Put this directive before the work−sharing construct that is dependent on the preceding `#pragma enter gate`. No thread enters this work−sharing construct until all threads have cleared the work−sharing construct controlled by the corresponding `#pragma enter gate`.

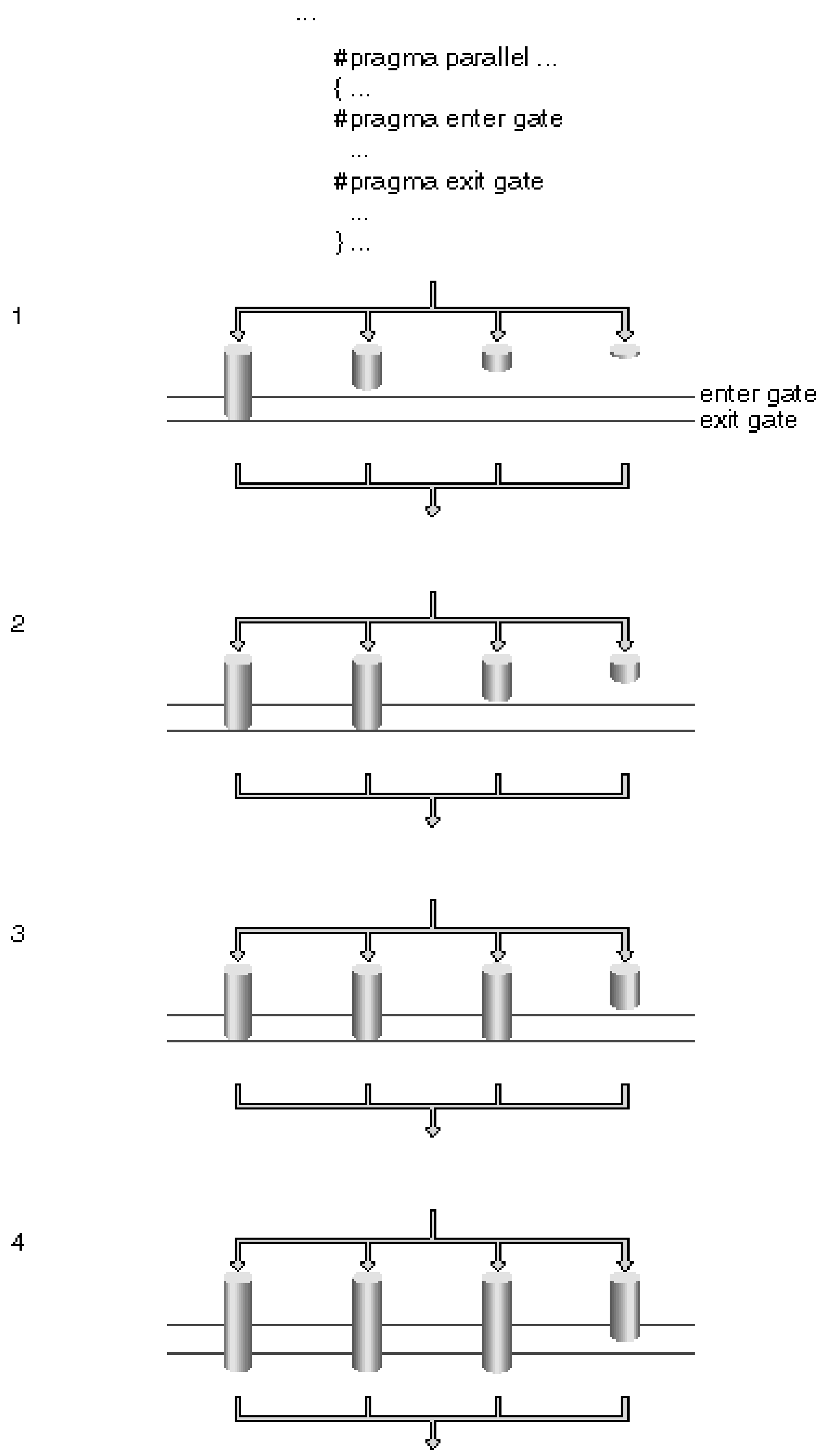
> **Note:** Nesting of the `#pragma enter gate` and `#pragma exit gate` directives is not supported.

> **Caution:** These directives work slightly differently in the IRIS POWER C Analyzer (PCA) for compiler versions 7.1 and older. See the *IRIS POWER C User's Guide* for more information.

9.3.1 Diagram of `#pragma enter gate` and `#pragma exit gate`

Figure 2, is a "time−lapse" sequence showing execution using enter and exit gates.

Figure 2. Execution Using Gates


...
#pragma parallel ...
{ ...
#pragma enter gate
...
#pragma exit gate
...
} ...
1
enter gate
exit gate
2
3
4
a12044

9.3.2 Example of `#pragma enter gate` and `#pragma exit gate`

This example shows how to use these two directives to work with parallelized segments that have some dependences.

Suppose you have a parallel region consisting of the work−sharing constructs A, B, C, D, E, and so forth. A dependence may exist between B and E such that you cannot execute E until all the work on B has completed (see the following code).

```
#pragma parallel ...  
{  
..A..  
..B..  
..C..  
..D..  
..E.. (depends on B)  
}
```

One option is to put a `#pragma synchronize` before E. But this `#pragma` directive is wasteful if all the threads have cleared B and are already in C or D. All the faster threads pause before E until the slowest thread completes C and D.

```
#pragma parallel ...  
{  
..A..  
..B..  
..C..  
..D..  
#pragma synchronize  
..E..  
}
```

To reflect this dependence, put `#pragma enter gate` after B and `#pragma exit gate` before E. Putting `#pragma enter gate` after B tells the system to note which threads have completed the B work−sharing construct. Putting `#pragma exit gate` prior to the E work sharing construct tells the system to allow no thread into E until all threads have cleared B. See the following example:

```
#pragma parallel ...  
{  
..A..  
..B..  
#pragma enter gate  
..C..  
..D..  
#pragma exit gate  
..E..  
}
```

9.4 `#pragma independent`

Running a loop in parallel is a class of parallelism sometimes called "fine−grained parallelism" or "homogeneous parallelism." It is called homogeneous because all the threads execute the same code on different data. Another class of parallelism is called "coarse−grained parallelism" or "heterogeneous parallelism." As the name suggests, the code in each thread of execution is different.

Ensuring data independence for heterogeneous code executed in parallel is not always as easy as it is for homogeneous code executed in parallel. (Ensuring data independence for homogeneous code is not a trivial task, either.)

The syntax of the `#pragma independent` directive is as follows:

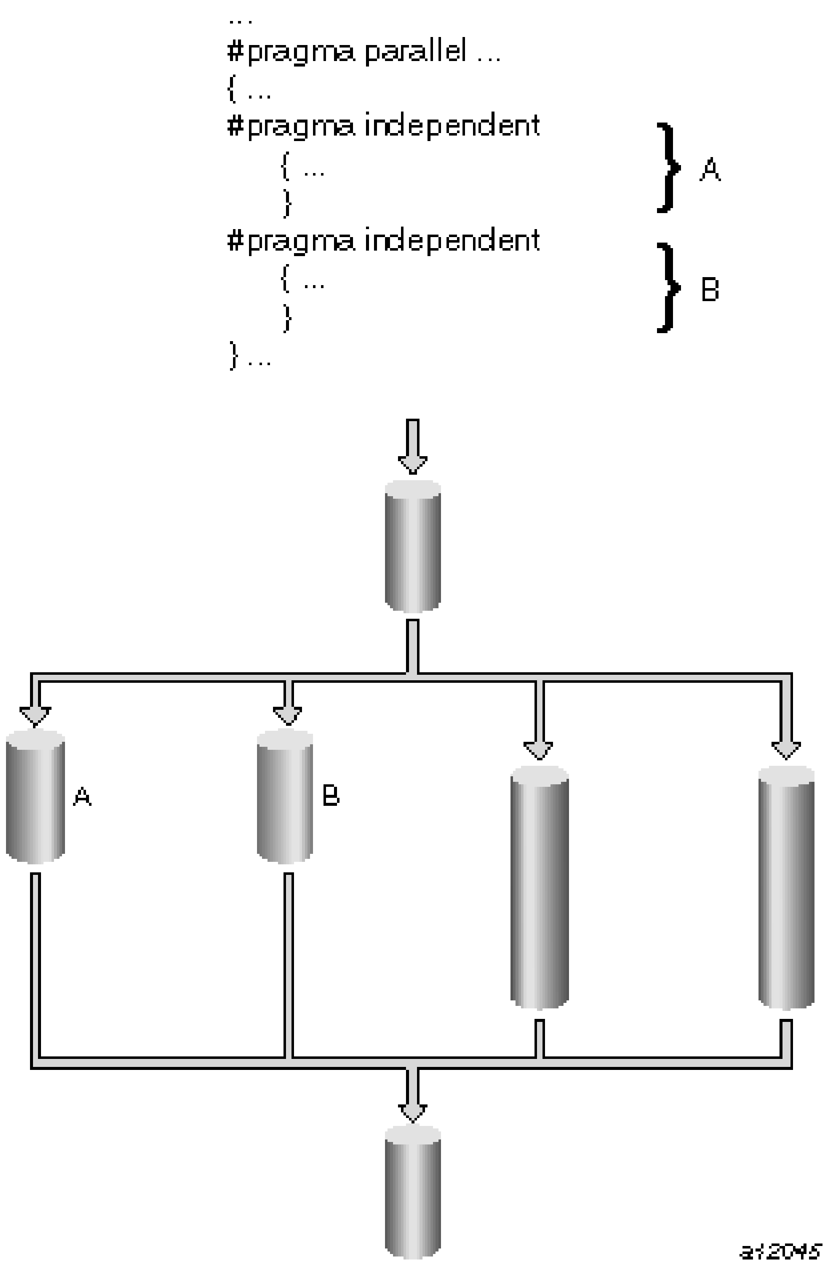
```
#pragma independent  
{ code }
```

The `#pragma independent` directive has no modifiers. Use this directive to tell the multiprocessing C/C++ compiler to run code in parallel with the rest of the code in the parallel region. Other threads can proceed past this code as soon as it starts execution.

9.4.1 Diagram of `#pragma independent`

Figure 3, shows an independent segment with execution by only one thread.

Figure 3. Independent Segment Execution



9.5 `#pragma local`

The `#pragma local` directive tells the multiprocessing C/C++ compiler the names of all the variables that must be local to each thread.

The syntax of the `#pragma local` directive is as follows:

```
#pragma local (variable1[, variable2...])
```

> **Note:** A variable in a local clause cannot have initializers and cannot be an array element or a field within a class, structure, or union.

9.6 `#pragma no side effects`

The `#pragma no side effects` directive tells the compiler that the only observable effect of a call to any of the named functions is its return value. In particular, the function does not modify an object or file that exists before it is called, and does not create a new object or file that persists after the completion of the call. This implies that if its return value is not used, the call may be skipped.

The syntax of the `#pragma no side effects` directive is as follows:

```
#pragma no side effects (function1[, function2...])
```

The functions named must be declared before the directive.

`#pragma no side effects` is not currently supported in C++, except for symbols marked `extern"C"`.

9.7 `#pragma one processor`

The `#pragma one processor` directive causes the statement that follows it to be executed by one thread.

The syntax of the `#pragma one processor` directive is as follows:

```
#pragma one processor  
{ code }
```

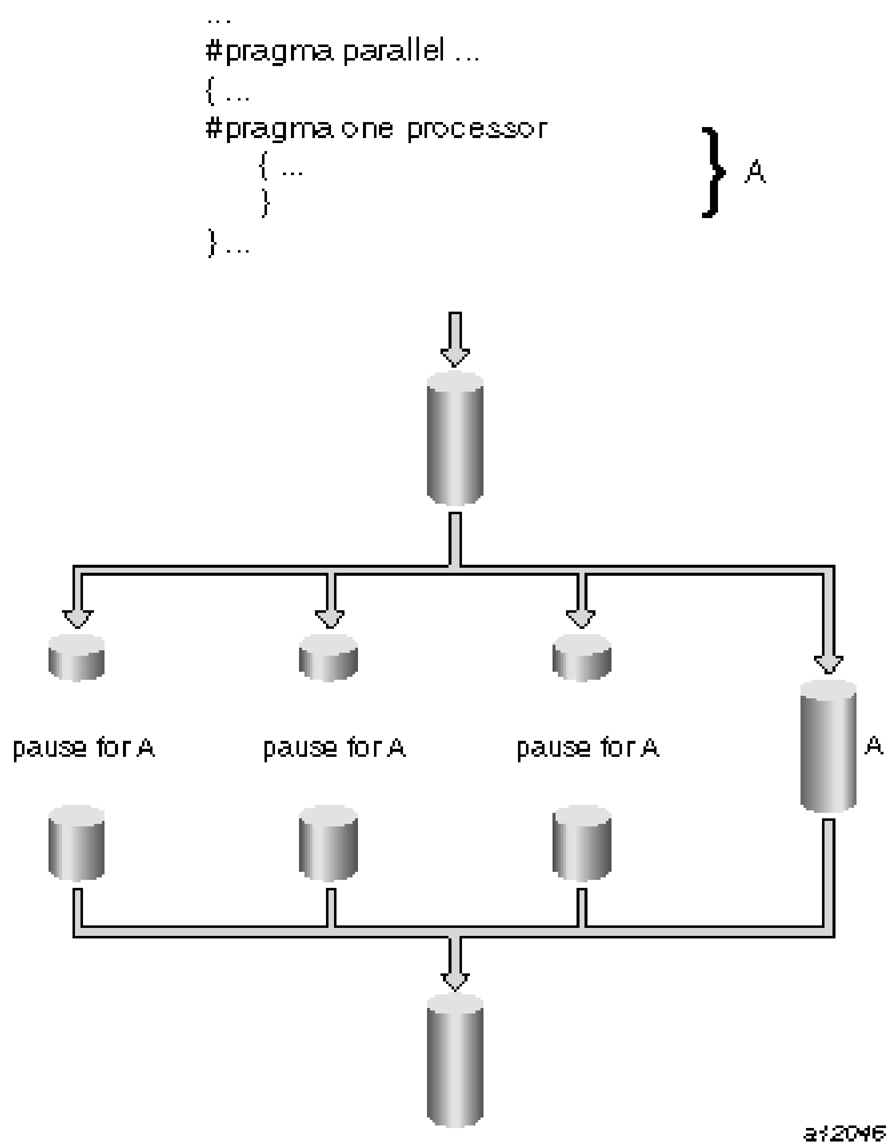
If a thread is executing the statement enclosed by this directive, other threads that encounter this statement must wait until the statement has been executed by the first thread, then skip the statement and continue.

If a thread has completed execution of the statement enclosed by this directive, then all threads encountering this statement skip the statement and continue without pause.

9.7.1 Diagram of `#pragma one processor`

Figure 4, shows code executed by only one thread. No thread can proceed past this code until it has been executed.

Figure 4. One Processor Segment



9.8 #pragma parallel

The #pragma parallel directive indicates that the subsequent statement (or compound statement) is to be run in parallel. #pragma parallel has four clauses, shared, local, if, and numthreads, that provide the compiler with more information on how to run the block of code (see Section 9.9). These clauses can either be listed on the same line as the #pragma parallel directive or broken out into separate #pragma directives(see Section 9.8.1).

The syntax of the #pragma parallel directive is as follows:

```
#pragma parallel [clause1[, clause2 ...]]
```

Use the #pragma parallel directive to start a parallel region. This directive has a number of clauses (see Section 9.9 for more details), but to run a single loop in parallel, the only clauses you usually need are shared and local. These options tell the multiprocessing C/C++ compiler which variables to share between all threads of execution and which variables to treat as local.

The code that makes up the parallel region is usually delimited by curly braces ({ }) and immediately follows the #pragma parallel directives and its modifiers.

Objects are shared by default unless declared within a parallel program region. If they are declared within a parallel program region, they are local by default. For example:

```
main() {  
int x, s, l;  
#pragma parallel shared (s) local (l)  
{  
int y;  
  
/* within this parallel region, by the default rules  
x and s are shared  whereas l and y are local */  
  
...  
}  
...  
}
```

Caution: This directive works slightly differently in the IRIS POWER C™ Analyzer (PCA) for compiler versions 7.1 and older. See the *IRIS POWER C User's Guide* for more information.

9.8.1 Example of `#pragma parallel`

For example, suppose you want to start a parallel region in which to run the following code in parallel:

```
for (idx=n; idx; idx--) {  
a[idx] = b[idx] + c[idx];  
}
```

Enter the following code before the statement or compound statement (code in curly braces, { }) that makes up the parallel region:

```
#pragma parallel shared( a, b, c ) shared(n) local( idx )  
#pragma pfor
```

Or you can enter the following code:

```
#pragma parallel  
#pragma shared( a, b, c )  
#pragma shared(n)  
#pragma local(idx)  
#pragma pfor
```

Any code within a parallel region, but not within any of the explicit parallel constructs (pfor, independent, one processor, and critical), is local code. Local code typically modifies only local data and is run by all threads.

9.9 `#pragma parallel` Clauses

The `#pragma parallel` directive has four possible clauses; each clause may also be written as a separate directive, following the `#pragma parallel` directive:

· `shared`

· `local`

· `if`

· `numthreads`

9.9.1 `shared`: Specifying Shared Variables

The `shared` clause tells the multiprocessing C/C++ compiler the names of all the variables that the threads must share.

The syntax of `#pragma parallel` with the `shared` clause is as follows:

```
#pragma parallel shared (var1[, var2 ...])
```

Note: A variable in a shared clause cannot be an array element or a field within a class, structure, or union.

9.9.2 `local`: Specifying Local Variables

The `local` clause tells the multiprocessing C/C++ compiler the names of all the variables that must be local to each thread.

The syntax of `#pragma parallel` with the `local` clause is as follows:

```
#pragma parallel local (var1[, var2 ...])
```

A variable in a local clause cannot have initializers and cannot be any of the following:

- An array element
- A field within a class, structure, or union
- An instance of a C++ class

9.9.3 `if`: Specifying Conditional Parallelization

The `if` clause lets you set up a condition that is evaluated at run time to determine whether to run the statements serially or in parallel. At compile time, it is not always possible to judge how much work a parallel region does (for example, loop indices are often calculated from data supplied at run time). The `if` clause lets you avoid running trivial amounts of code in parallel when the possible speedup does not compensate for the overhead associated with running code in parallel.

The syntax of `#pragma parallel` with the `if` clause is as follows:

```
#pragma parallel if (expr)
```

The `if` condition, *expr*, must evaluate to an integer. If *expr* is false (evaluates to zero), then the subsequent statements run serially. Otherwise, the statements run in parallel.

9.9.4 `numthreads`: Specifying the Number of Threads

The `numthreads` clause tells the multiprocessing C/C++ compiler how many of the available threads to use when running this region in parallel. (The default is all the available threads.)

In general, you should avoid having more threads of execution than you have processors, and you should specify `numthreads` with the `MP_SET_NUMTHREADS` environment variable at run time. If you want to run a loop in parallel while you run other code, you can use this option to tell the compiler to use only some of the available threads.

The syntax of `#pragma parallel` with the `numthreads` clause is as follows:

```
#pragma parallel numthreads(expr)
```

The variable *expr* should evaluate to a positive integer.

9.10 `#pragma pfor`

The `#pragma pfor`directive marks a for loop to run in parallel. This directive must follow a `#pragma parallel` directive and be contained within a parallel region. `#pragma pfor` takes several clauses (see Section 9.9, for more details), which control the following aspects:

- How the work load is partitioned over the available processors
- Which variables are local to each process

- Which variables are involved in a reduction operation
- Which iterations are assigned to which threads
- How the iterations are shared by the available processors
- How many iterations make up the "chunks" assigned to the threads

Use `#pragma pfor` to run a for loop in parallel only if the loop meets all of the following conditions:

- The `#pragma pfor` is contained within a parallel region.
- All the values of the index variable can be computed independently of the iterations.
- All iterations are independent of each other; that is, data used in one iteration does not depend on data created by another iteration. If the loop can be run backwards, the iterations are probably independent.
- The number of iterations is known (no infinite or data−dependent loops) at execution time. The number of times the loop must be executed must be determined once, upon entry to the loop, and based on the loop initialization, loop test, and loop increment statements.

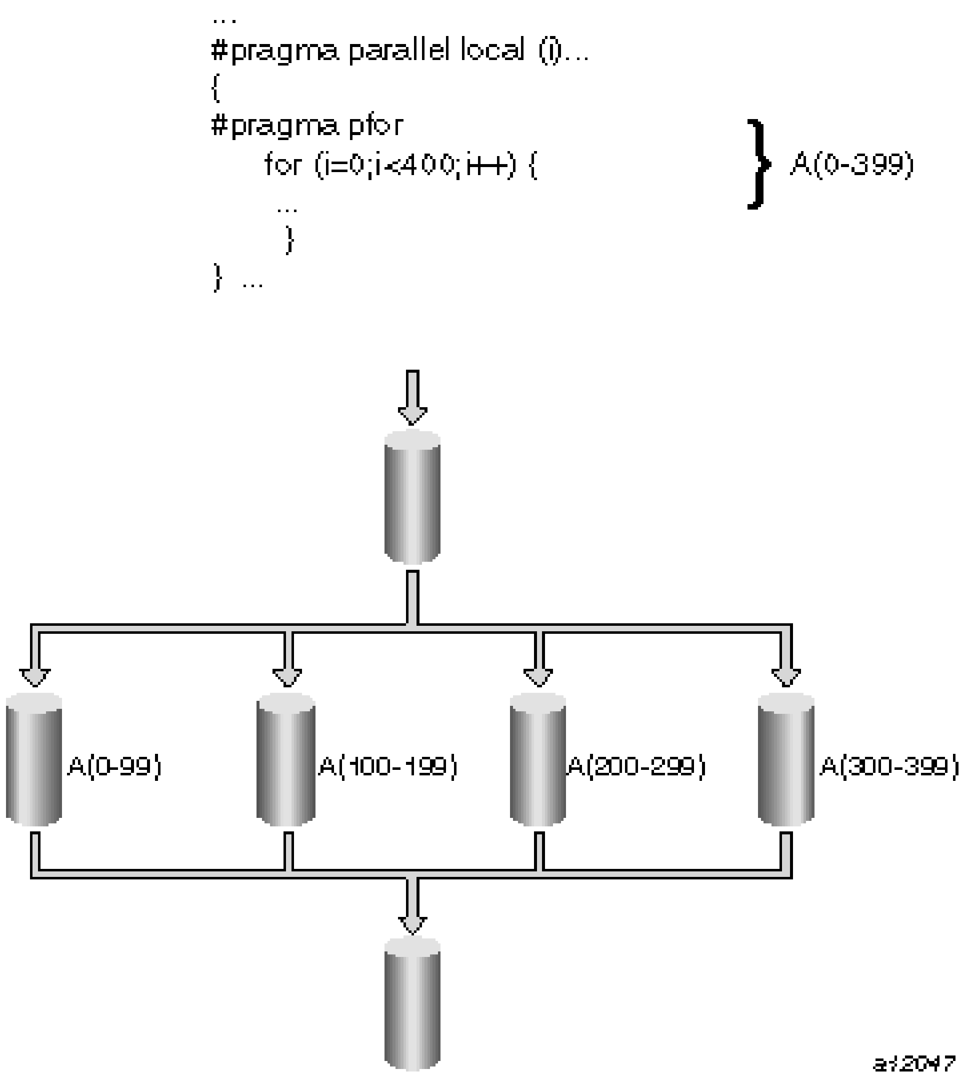
  Note: If the number of times the loop is actually executed is different from what is computed above, the results are undefined. This can happen if the loop test and increment change during the execution of the loop, or if there is an early exit from within the for loop. An early exit or a change to the loop test and increment during execution may have serious performance implications.
- The chunksize, if specified, is computed before the loop is executed, and the behavior is undefined if its value changes within the loop.
- The loop control variable cannot be an array element, or a field within a class, structure, or union.
- The test or the increment should not contain expressions with side effects.

  Caution: This directive works differently in the IRIS POWER C™ Analyzer (PCA) for compiler versions 7.1 and older. See the *IRIS POWER C User's Guide* for more information.

9.10.1 Diagram of `#pragma pfor`

Figure 5, shows parallel code segments using `#pragma pfor` running on four threads with simple scheduling.

Figure 5. Parallel Code Segments Using #pragma pfor



9.10.2 C++ Multiprocessing Considerations With `#pragma pfor`

If you are writing a `#pragma pfor` loop for the multiprocessing C++ compiler, the index variable *i* can be declared within the `for` statement using the following:

```
int i = 0;
```

The ANSI C++ Standard states that the scope of the index variable declared in a `for` statement extends to the end of the `for` statement, as in the following example:

```
#pragma pfor  
for (int i = 0, ...) { ... }
```

The MIPSpro 7.2 C++ compiler does not enforce this rule. By default, the scope extends to the end of the enclosing block. The default behavior can be changed by using the command line option `-LANG:ansi-for-init-scope=on` which enforces the ANSI C++ standard.

To avoid future problems, write `for` loops in accordance with the ANSI standard, so a subsequent change in the compiler implementation of the default scope rules does not break your code.

9.11 `#pragma pfor` Clauses

The `#pragma pfor` directive accepts the following clauses:

`iterate`	Tells the multiprocessing C compiler the information it needs to partition the work load over the available processors.
`local`	Specifies the variables that are local to each process.
`lastlocal`	Specifies the variables that are local to each process, saving only the value of the variables from the logically last iteration of the loop.
`reduction`	Specifies variables involved in a reduction operation.
`affinity`	Assigns certain iterations to specific threads (for Origin200™ and Origin2000™ only).
`nest`	Exploits nested concurrency.
`schedtype`	Specifies how the loop iterations are to be shared among the processors.
`chunksize`	Specifies how many iterations make up a chunk.

9.11.1 `iterate`: Specifying the `for` Loop

The syntax of `#pragma pfor` with the `iterate` clause is as follows:

`#pragma pfor iterate (`*index* `=` *expr1*`;` *expr2*`;` *expr3*`)`

The iterate clause tells the multiprocessing C compiler the information it needs to identify the unique iterations of the loop and partition them to particular threads of execution. This clause is optional. The compiler automatically infers the appropriate values from the subsequent `for` loop.

Table 11, describes the components of the `iterate` clause.

Table 11. Components of the `iterate` Clause

Component	Description
index	The index variable of the `for` loop you want to run in parallel.
expr1	The starting value for the index variable.
expr2	The number of iterations for the loop you want to run in parallel.
expr3	The increment of the `for` loop you want to run in parallel.

The following is an example using the `iterate` clause:

Consider this `for` loop:

```
for (idx=n; idx; idx--) {  
a[idx] = b[idx] + c[idx];  
}
```

The `iterate` clause to `pfor` should be as follows:

```
iterate(idx=n;n;-1)
```

This loop counts down from the value of *n*, so the starting value is the current value of *n*. The number of trips through the loop is *n*, and the increment is −1.

9.11.2 `local` and `lastlocal`: Specifying Local Variables

The syntax of `#pragma pfor` with the `local` clause is as follows:

```
#pragma pfor local (var1[, var2,...])
```

The `local` clause specifies the 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 `pfor local` clause has the same restrictions as the `parallel local` clause (see Section 9.9.2).

The syntax of `#pragma pfor` with the `lastlocal` clause is as follows:

```
#pragma pfor lastlocal (var1[, var2,...])
```

The `lastlocal` clause specifies the variables that are local to each process. Unlike with the `local` clause, the compiler saves the value from only the logically last iteration of the loop when it completes.

9.11.3 `reduction`: Specifying Variables for Reduction

The syntax of `#pragma pfor` with the `reduction` clause is as follows:

```
#pragma pfor reduction (var1[, var2,...])
```

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. An element of the reduction list must be an individual variable (also called a scalar variable) and cannot be an array or structure. However, it can be an individual element of an array. When the `reduction` clause is used, it appears in the list with the correct 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 two types of reductions supported are `sum`(+) and `product`(*). For more information, see the *C Language Reference Manual* .

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

9.11.4 `affinity`: Thread Affinity

Thread affinity assigns particular iterations to a particular thread.

The syntax of `#pragma pfor` with the `affinity` clause for thread affinity is as follows:

```
#pragma pfor affinity variable = thread (expr)
```

The effect of thread affinity is to execute iteration *i* on the thread number given by the user−supplied expression (modulo the number of threads). Because the threads may need to evaluate this expression in each iteration of the loop, the variables used in the expression (other than the loop induction variable) must be declared shared and must not be modified during the execution of the loop. Violating these rules may lead to incorrect results.

If the expression does not depend on the loop induction variable, then all iterations will execute on the same thread and will not benefit from parallel execution.

Thread affinity is often used in conjunction with the `#pragma page-place` directive (Section 5.4).

Data affinity for loops with non−unit stride can sometimes result in non−linear affinity expressions. In such situations the compiler issues a warning, ignores the affinity clause, and defaults to simple scheduling.

9.11.5 `affinity`: Data Affinity

Data affinity applies only to distributed arrays and is supported only on Origin systems. See Chapter 5 for more information about distributed arrays.

The syntax of `#pragma pfor` with the `affinity` clause for data affinity is as follows:

```
#pragma pfor affinity(idx) = data(array(expr))
```

idx is the loop−index variable

array is the distributed array

expr indicates an element owned by the processor on which you want this iteration to execute

The following code shows an example of data affinity:

```
#pragma distribute A[block]  
#pragma parallel shared (A, a, b) local (i)  
#pragma pfor affinity(i) = data(A[a*i + b])  
for (i = 0; i < n; i++)  
     A[a*i + b] = 0;
```

The multiplier for the loop index variable (`a`) and the constant term (`b`) must both be literal constants, with `a` greater than zero.

The effect of this clause is to distribute the iterations of the parallel loop to match the data distribution specified for the array `A`, such that iteration `i` is executed on the processor that owns element `A[a*i + b]`, based on the distribution for `A`. The iterations are scheduled based on the specified distribution, and are not affected by the actual underlying data−distribution (which may differ at page boundaries, for example).

In the case of a multi−dimensional array, affinity is provided for the dimension that contains the loop−index variable. The loop−index variable cannot appear in more than one dimension in an affinity directive.

In the following example, the loop is scheduled based on the block distribution of the first dimension. See Chapter 5, for more information about distribution directives.

```
#pragma distribute A[block][cyclic(1)]  
#pragma parallel shared (A, n) local (i, j)  
#pragma pfor  
#pragma affinity (i) = data(A[i + 3, j])  
for (i = 0; i < n; i++)  
for (j = 0; j < n; j++)  
A[i + 3, j] = A[i + 3, j-1];
```

9.11.6 Data Affinity for Redistributed Arrays

By default, the compiler assumes that a distributed array is not dynamically redistributed, and directly schedules a parallel loop for the specified data affinity. In contrast, a redistributed array can have multiple possible distributions, and data affinity for a redistributed array must be implemented in the run−time system based on the particular distribution.

However, the compiler does not know whether or not an array is redistributed, because the array may be redistributed in another function (possibly even in another file). Therefore, you must explicitly specify the `#pragma dynamic` declaration for redistributed arrays. This directive is required only in those functions that contain a `pfor` loop with data affinity for that array (see Section 5.3, for additional information). This informs the compiler that the array can be dynamically redistributed. Data affinity for such arrays is implemented through a run−time lookup.

9.11.7 Data Affinity for a Formal Parameter

You can supply a `distribute` directive on a formal parameter, thereby specifying the distribution on the incoming actual parameter. If different calls to the subroutine have parameters with different distributions, then you can omit the `distribute` directive on the formal parameter; data affinity loops in that subroutine are automatically implemented through a run−time lookup of the distribution. (This is permissible only for regular data distribution. For reshaped array parameters, the distribution must be fully specified on the formal parameter.)

9.11.8 Data Affinity and the `#pragma pfor nest` Clause

The `nest` clause for `#pragma pfor` is described in Section 9.11.9. This section discusses how the `nest` clause interacts with the `affinity` clause when the program has reshaped arrays.

When you combine a `nest` clause and an `affinity` clause, the default scheduling is `simple`, except when the program has reshaped arrays and is compiled `-O3`. In that case, the default is to use data affinity scheduling for the most frequently accessed reshaped array in the loop (chosen heuristically by the compiler). To obtain `simple` scheduling even at `-O3`, you can explicitly specify the schedtype on the parallel loop.

The following example illustrates a nested `pfor` with an `affinity` clause:

```
#pfor nest(i, j) affinity(i, j) = data(A[i][j])  
for (i = 2; i < n; i++)  
for (j = 2; j < m; j++)  
A[i][j] = A[i][j] + i * j;
```

9.11.9 `nest`: Exploiting Nested Concurrency

The `nest` clause allows you to exploit nested concurrency in a limited manner. Although true nested parallelism is not supported, you can exploit parallelism across iterations of a perfectly nested loop–nest.

The syntax of `#pragma pfor` with the `nest` clause is as follows:

```
#pragma pfor nest(i, j[, ...])
```

This clause specifies that the entire set of iterations across the `(i, j[, ...])` loops can be executed concurrently. The restriction is that the loops must be perfectly nested; that is, no code is allowed between either the `for` statements or the ends of the respective loops, as illustrated in the following example:

```
#pragma pfor nest(i, j)  
for (i = 0; i < n; i++)  
for (j = 0; j < m; j++)  
A[i][j] = 0;
```

The existing clauses, such as `local` and `shared`, behave as before. You can combine a nested `pfor` with a schedtype of `simple` or `interleaved` (`dynamic` and `gss` are not currently supported). The default is `simple` scheduling.

> **Note:** The `nest` clause requires support from the MP run–time library (`libmp`). IRIX operating system versions 6.3 (and above) are automatically shipped with this new library. If you want to access these features on a system running IRIX 6.2, then contact your local Silicon Graphics service provider or Silicon Graphics Customer Support (1–800–800–4744) for `libmp`.

9.11.10 `schedtype`: Sharing Loop Iterations Among Processors

The syntax of `#pragma pfor` with the `schedtype` clause is as follows:

```
#pragma pfor schedtype (type)
```

The `schedtype` clause tells the multiprocessing C compiler how to share the loop iterations among the processors. The `schedtype` chosen depends on the type of system you are using and the number of programs executing (see Table 13).

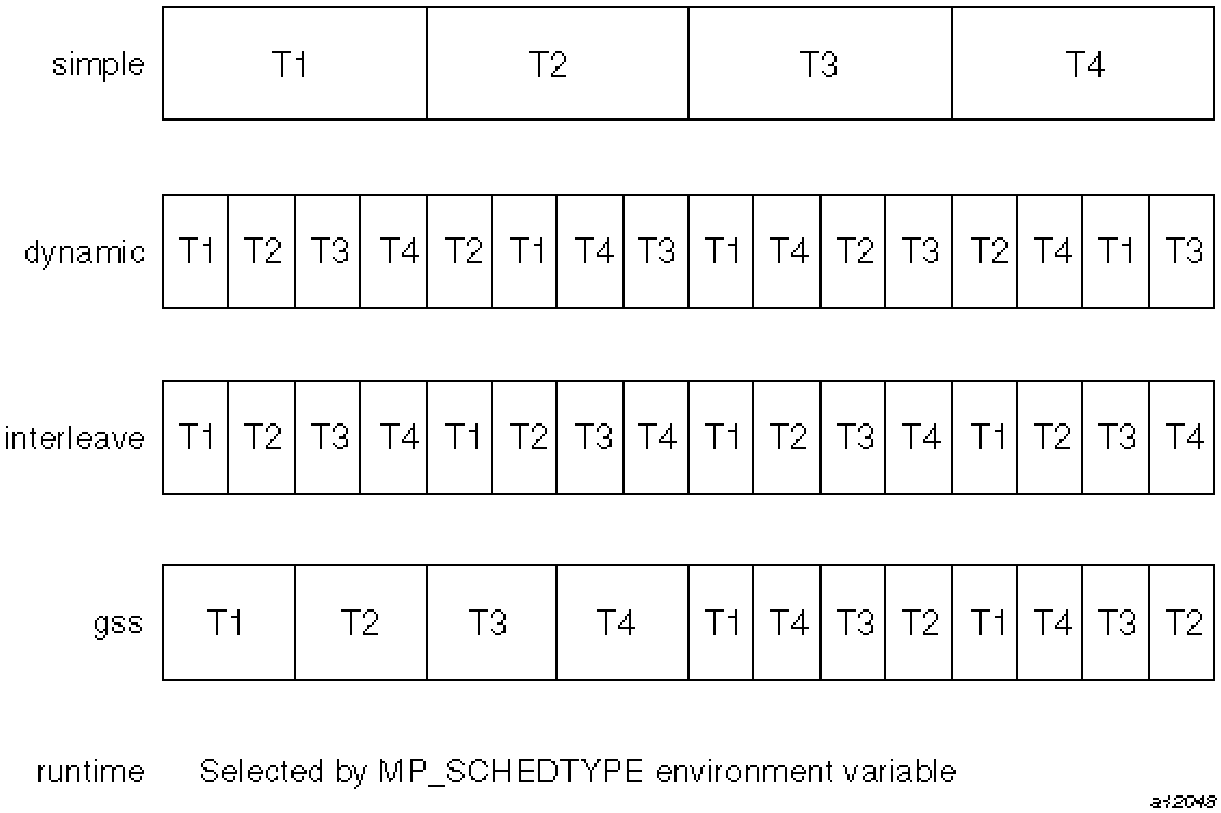
You can use the types in Table 12, to modify `schedtype`.

Table 12. Schedtype Types

Type	Function
`simple` (the default)	Tells the run–time scheduler to partition the iterations evenly among all the available threads.
`dynamic`	Tells the run–time scheduler to give each thread *chunksize* iterations of the loop. *chu* should be smaller than the number of total iterations divided by the number of thread advantage of dynamic over simple is that dynamic helps distribute the work more eve than simple.
`interleave`	Tells the run–time scheduler to give each thread chunksize iterations of the loop, whi then assigned to the threads in an interleaved way.
`gss`(guided self–scheduling)	Tells the run–time scheduler to give each processor a varied number of iterations of t loop. This is like `dynamic`, but instead of a fixed *chunksize*, the *chunksize* iterations l with big pieces and end with small pieces. If I iterations remain and P threads are working on them, the piece size is roughly I/(2P) + 1 Programs with triangular matrices should use `gss`.
`runtime`	Tells the compiler that the real schedule type will be specified at run time, based on environment variables (see the *C Language Reference Manual* for more information

Figure 6, shows how the iteration chunks are apportioned over the various processors by the different types of loop scheduling.

Figure 6. Loop Scheduling Types



The best `schedtype` to use for any given program depends on your system, program, and data. For instance, with certain types of data, some iterations of a loop can take longer to compute than others, so some threads may finish long before the others. In this situation, if the iterations are distributed by `simple`, then the thread waits for the others. But if the iterations are distributed by `dynamic`, the thread does not wait, but goes back to get another *chunksize* iteration until the threads of execution have run all the iterations of the loop.

The Table 13, describes how to choose a `schedtype`.

Table 13. Choosing a schedtype

For a...	**Where...**	**Use...**
Single−User System	iterations take same amount of time	`simple`
	data−sensitive iterations vary slightly	`gss`
	data−sensitive iterations vary greatly	`dynamic`
Multiuser System	data−sensitive iterations vary slightly	`gss`
	data−sensitive iterations vary greatly	`dynamic`

If you are on a single−user system but are executing multiple programs, select the scheduling from the multiuser rows.

If you are on a multiuser system, you should also consider using the environment variable, `MP_SUGNUMTHD`. Setting `MP_SUGNUMTHD` causes the run−time library to automatically adjust the number of active threads based on the overall 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. For more details about `MP_SUGNUMTHD`, see the "Run−time Environment Variables" section in the "Multiprocessing Advanced Features" chapter of the *C Language Reference Manual.*

9.11.11 `chunksize`: Specifying the Number of Iterations in a Chunk

The `chunksize` clause tells the multiprocessing C compiler how many iterations to define as a chunk when using the `dynamic` or `interleave` clause (see Section 9.11.10).

The syntax of `#pragma pfor` with the `chunksize` clause is as follows:

```
#pragma pfor chunksize (expr)
```

expr should be a positive integer. Silicon Graphics recommends using the following formula:

```
(number of iterations)/X
```

X should be between twice and ten times the number of threads. Select twice the number of threads when iterations vary slightly. Reduce the chunk size to reflect the increasing variance in the iterations. Performance gains may diminish after increasing X to ten times the number of threads.

9.12 `#pragma pure`

The `#pragma pure` directive tells the compiler that a call to any of the named functions has no side effects (see `#pragma no side effects`), and that its return value depends only on the values of its arguments. In particular, it does not access an existing object or file after its arguments have been evaluated. If the arguments of such a call are loop−invariant, then the compiler may move the call out of the loop.

The syntax of the `#pragma pure` directive is as follows:

```
#pragma pure (function1[, function2...])
```

The functions named must be declared before the directive.

`#pragma pure` is not currently supported in C++, except for symbols marked `extern"C"`.

9.13 `#pragma set chunksize`

The `#pragma set chunksize` directive sets the value of `chunksize`, which tells the multiprocessing C compiler how many iterations to define as a chunk when using the `dynamic` or `interleave` clause (see Section 9.15, and Section 9.11, for more information).

The syntax of the `#pragma set chunksize` directive is as follows:

```
#pragma set chunksize (n)
```

Silicon Graphics recommends using the following formula:

```
(number of iterations)/X
```

X should be between twice and ten times the number of threads. Select twice the number of threads when iterations vary slightly. Reduce the chunk size to reflect the increasing variance in the iterations. Performance gains may diminish after increasing X to ten times the number of threads.

9.14 `#pragma set numthreads`

The `#pragma set numthreads` directive sets the value for `numthreads,` which tells the multiprocessing C/C++ compiler how many of the available threads to use when running this region in parallel. The default is all the available threads.

If you want to run a loop in parallel while you run some other code, you can use this option to tell the compiler to use only some of the available threads.

9.14.1 Using `#pragma set numthreads`

The syntax of the `#pragma set numthreads` directive is as follows:

```
#pragma set numthreads (n)
```

n can range from 1 to 255. If if *n* is greater than 255, the compiler assumes the maximum and generates a warning message. If *n* is less than 1, the compiler generates a warning message and ignores the directive.

In general, you should never have more threads of execution than you have processors, and you should specify `numthreads` with the `MP_SET_NUMTHREADS` environment variable at run time (see the *C Language Reference Manual* for more information).

9.15 `#pragma set schedtype`

The `#pragma set schedtype` directive sets the value of `schedtype`, which tells the multiprocessing C compiler how to share the loop iterations among the processors. The `schedtype` chosen depends on the type of system you are using and the number of programs executing (see Section 9.11, for more information on `schedtype`).

The syntax of the `#pragma set schedtype` directive is as follows:

```
#pragma set schedtype (type)
```

The schedtype *types* are

- `simple`
- `dynamic`
- `interleave`
- `gss`

· runtime

See Table 12, for a description of each type.

9.16 #pragma shared

The `#pragma shared` directive tells the multiprocessing C/C++ compiler the names of all the variables that the threads must share. This directive must be used in conjunction with the `#pragma parallel` directive. `#pragma shared` can also be used as a clause for the `#pragma parallel` directive (see Section 9.9).

The syntax of `#pragma shared` is as follows:

```
#pragma shared (variable1, [, variable2...])
```

Note: A variable in a shared clause cannot be an array element or a field within a class, structure, or union.

9.17 #pragma synchronize

The `#pragma synchronize` directive tells the multiprocessing C/C++ compiler that within a parallel region, no thread can execute the statement that follows this directive until all threads have reached it. This directive is a classic barrier construct.

The syntax of `#pragma synchronize` is as follows:

```
#pragma synchronize
```

9.17.1 Diagram of #pragma synchronize

Figure 7, is a time−lapse sequence showing the synchronization of all threads.

Figure 7. Synchronization

```
...

#pragma parallel ...
{ ...
#pragma synchronize
  ...
} ...
```

A