Chapter 10. Multiprocessing Directives

In addition to the usual interpretation performed by a C/C++ compiler, the multiprocessing C/C++ compiler can process explicit multiprocessing directives to produce code that can run concurrently on multiple processors.

Table 18, lists the multiprocessing `#pragma` directives to use when processing code in parallel regions. The multiprocessing compiler does not know whether an automatic parallelization tool, you the user, or a combination of the two put the directives in the code. The multiprocessing C/C++ compiler does not check for or warn against data dependencies or other restrictions that have been violated.

See the *MIPSpro C and C++ Pragmas.*

Table 18. Multiprocessing C/C++ Compiler Directives

#pragma	**Description**
`#pragma copyin`	Copies the value from the master thread's version of an `–Xlocal`–linked global vari into the slave thread's version.
`#pragma critical`	Protects access to critical statements.
`#pragma enter gate`	Indicates the point that all threads must clear before any threads are allowed to pass t corresponding exit gate.
`#pragma exit gate`	Stops threads from passing this point until all threads have cleared the corresponding gate.
`#pragma independent`	Starts an independent code section that executes in parallel with other code in the pa region.
`#pragma local`	Tells the compiler the names of all the variables that must be local to each thread.
`#pragma no side effects`	Tells the compiler to assume that all of the named functions are safe to execute concurrently.
`#pragma one processor`	Causes the next statement to be executed on only one processor.
`#pragma parallel`	Marks the start of a parallel region.
`#pragma pfor`	Marks a `for` loop to run in parallel.
`#pragma set chunksize`	Tells the compiler which values to use for `chunksize`.
`#pragma set numthreads`	Tells the compiler which values to use for `numthreads`.
`#pragma set schedtype`	Tells the compiler which values to use for `schedtype`.
`#pragma shared`	Tells the compiler the names of all the variables that the threads must share.
`#pragma synchronize`	Stops threads until all threads reach here.

10.1 OpenMP C/C++ API Multiprocessing Directives

The MIPSpro C and C++ compilers support OpenMP multiprocessing directives. These directives are based on the OpenMP C/C++ Application Program Interface (API) standard. Programs that use these directives are portable and can be compiled by other compilers that support the OpenMP standard.

To enable recognition of the OpenMP directives, specify `-mp` on the `cc` or `CC` command line.

For more information on how to use these directives, see the *MIPSpro C and C++ Pragmas* manual.

10.2 Using Parallel Regions

To understand how most of the multiprocessing C/C++ compiler directives work, consider the concept of a parallel region. On some systems, a parallel region is a single loop that runs in parallel. However, with the multi−processing C/C++ compiler, a parallel region can include several loops and/or independent code segments that execute in parallel.

A simple parallel region consists of only one work−sharing construct, usually a loop. (A parallel region consisting of only a serial section or independent code is a waste of processing resources.)

A parallel region of code can contain sections that execute sequentially as well as sections that execute concurrently. A single large parallel region has a number of advantages over a series of isolated parallel regions: each isolated region executes a single loop in parallel. At the very least, the single large parallel region can help reduce the overhead associated with moving from serial execution to parallel execution.

Large mixed parallel regions avoid the forced synchronization that occurs at the end of each parallel region. The large mixed parallel region also allows you to use `#pragma` directives that execute independent code sections that run concurrently.

Thus, if a thread finishes its work early, it can go on to execute the next section of code−provided that the next section of code is not dependent on the completion of the previous section. However, when you create parallel regions, you need more sophisticated synchronization methods than you need for isolated parallel loops.

10.2.1 Coding Rules of `#pragma` Directives

The `#pragma` directives are modeled after the Parallel Computing Forum (PCF) directives for parallel FORTRAN. The PCF directives define a broad range of parallel execution modes and provide a framework for defining corresponding C/C++ `#pragma` directives.

The following changes have been made to make the `#pragma` directives more C−like:

- Each `#pragma` directive starts with `#pragma` and follows the conventions of ANSI C for compiler directives. You can use white space before and after the `#`, and you must sometimes use white space to separate the words in a `#pragma` directive, as with C syntax. A line that contains a `#pragma` directive can contain nothing else (code or comments).
- `#pragma` directives apply to only one succeeding statement. If a directive applies to more than one statement, you must make a compound statement. C/C++ syntax lets you use curly braces, { }, to do this. Because of the differences between this syntax and FORTRAN, C/C++ can omit the PCF directives that indicate the end of a range (for example, `END PSECTIONS`).
- The `#pragma pfor` directive replaces the `PARALLEL DO` directive because the `for` statement in C is more loosely defined than the `FORTRAN DO` statement.

To make it easier to use `#pragma` directives, you can put several keywords on a single `#pragma` directive line, or spread the keywords over several lines. In either case, you must put the keywords in the correct order, and each directive must contain an initial keyword. For example, the following two code samples do the same thing:

Example 1:

```
#pragma parallel shared(a,b,c, n) local(i) pfor  
for (i=0; i<n; i++) a[i]=b[i]+c[i];
```

Example 2:

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

```
#pragma local( i )  
#pragma pfor  
   for (i=0; i<n; i++) a[i]=b[i]+c[i];
```

10.3 Parallel Regions

A parallel region consists of a number of work−sharing constructs. The C/C++ compiler supports the following work−sharing constructs:

· A loop executed in parallel

· "Local" code run (identically) by all threads

· An independent code section executed in parallel with the rest of the code in the parallel region

· Code executed by only one thread

· Code run in "protected mode" by all threads

In addition, the C/C++ compiler supports three types of explicit synchronization. To account for data dependencies, it is sometimes necessary for threads to wait for all other threads to complete executing an earlier section of code. Three sets of directives implement this coordination: `#pragma critical`, `#pragma synchronize`, and `#pragma enter gate` and `#pragma exit gate`.

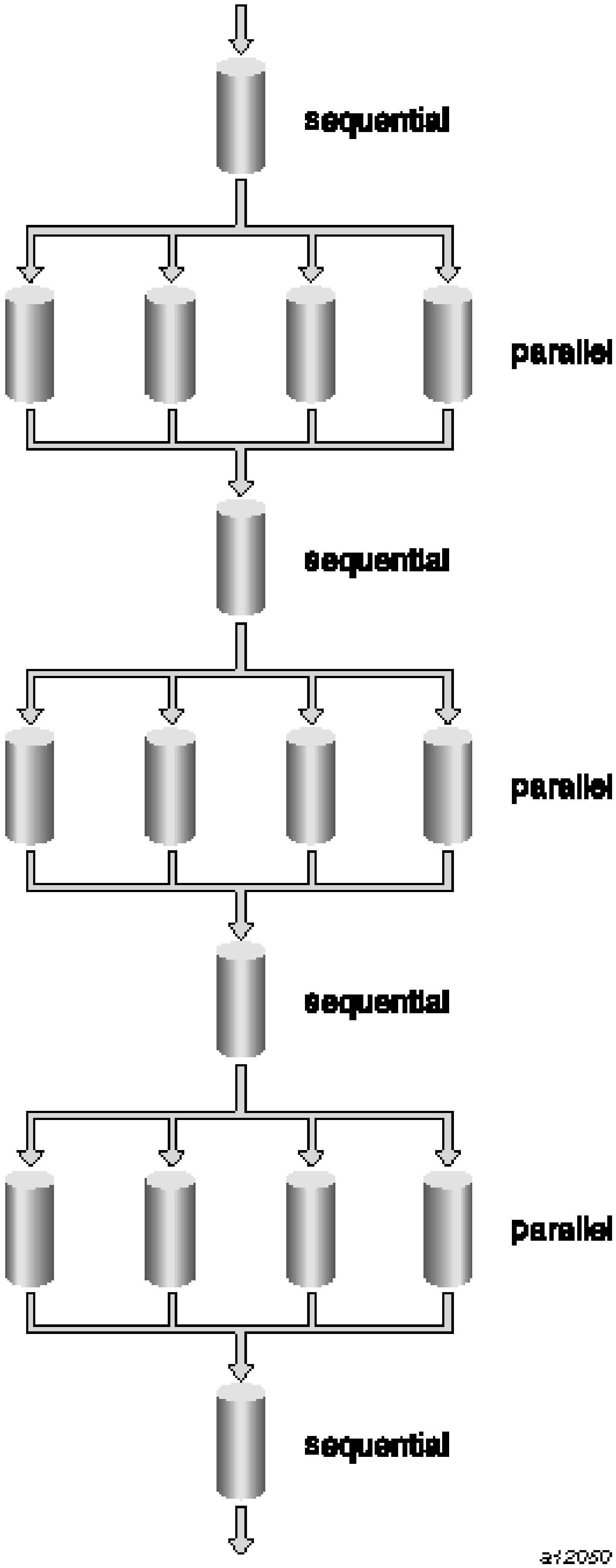
The parallel region should have a single entry at the top and a single exit at the bottom.

To start a parallel region, use the `#pragma parallel` directive. To mark a `for` loop to run in parallel, use the `#pragma pfor` directive. To start an independent code section that executes in parallel with the rest of the code in the parallel region, use the `#pragma independent`.

When you execute a program, nothing actually runs in parallel until it reaches a parallel region. Then multiple threads begin (or continue, if this is not the first parallel region), and the program runs in parallel mode. When the program exits a parallel region, only a single thread continues (sequential mode) until the program again enters a parallel region and the process repeats.

Figure 1, shows the execution of a typical parallel program with parts running in sequential mode and other parts running in parallel mode.

Figure 1. Program Execution


sequential
parallel
sequential
parallel
sequential
parallel
sequential
a12050

10.4 Parallel Reduction Operations in C and C++

A reduction operation applies to an array of values and reduces (combines) the array values into a single value.

Consider the following example:

```
int a[10000];  
int i;  
int sum = 0;  
for (i = 0; i < 10000; i++)  
sum = sum + a[i];
```

The loop computes the cumulative sum of the elements of the array. Because the value of a sum computed in one iteration is used in the next iteration, the loop as written cannot be executed in parallel directly on multiprocessors.

However, you can rewrite the above loop to compute the local sum on each processor by introducing a local variable. This breaks the iteration dependency of sum and the loop can be executed in parallel on multiprocessors. This loop computes the local sum of the elements on each processor, which can subsequently be serially added to yield the final sum.

The multiprocessing C/C++ compiler provides a `reduction` clause as a modifier for a `pfor` statement. Using this clause, the above loop can be parallelized as follows:

```
int a[10000];  
int i;  
int sum = 0  
#pragma parallel shared(a, sum) local(i)  
#pragma pfor reduction(sum)  
for i=0; i<10000; i++)  
    sum = sum + a[i];
```

10.4.1 Restrictions on the Reduction Clause

The following restrictions are imposed on the `reduction` clause:

- You can specify only variables of integer types (`int`, `short`, and so forth) or of floating point types (`float`, `double`, and so forth).
- You can use the `reduction` clause only with the primitive operations plus (+), and times (*), which satisfy the associativity property as illustrated in the following example:

  ```
  a op (b op c) == (a op b) op c.
  ```

The preceding example that uses a `reduction` clause has the same semantics as the following code that uses local variables and explicit synchronization. In this code, because *sum* is shared, the computation of the final sum has to be done in a critical region to allow each processor exclusive access to *sum*:

```
int a[10000];  
int i;  
int sum,localsum;  
sum = 0;  
#pragma parallel shared(a,sum) local(i,localsum)  
{  
    localsum = 0;  
#pragma pfor iterate(;;)  
    for (i = 0; i < 10000; i++) localsum +=a[i];  
#pragma critical  
    sum = sum + localsum;  
}
```

The general case of reduction of a binary operation, `op,` on an array a1,a2,a3,...a*n* involves the following computation:

```
a1 op a2 op a3 op.... op an
```

When the various operations are performed in parallel, they can be invoked in any order. In order for the reduction to produce a unique result, the binary operation, **op**, must therefore satisfy the associativity property, as shown below:

```
a op (b op c) == (a op b) op c
```

10.4.2 Performance Considerations

The reduction example in Section 10.4.1, has the drawback that when the number of processors increases, there is more contention for the lock in the critical region.

The following example uses a shared array to record the result on individual processors. The array entries are *CacheLine* apart to prevent write contention on the cache line (128 bytes in this example. The array permits recording results for up to *NumProcs* processors. Both these variables *CacheLine* and *NumProcs* can be tuned for a specific platform:

```
#define CacheLine 128  
int a[10000];  
int i, sum;  
int *localsum = malloc(NumProcs * CacheLine);  
  
for (i = 0; i < NumProcs; i++)  
localsum [i] = 0;  
  
sum = 0;  
#pragma parallel shared (a, sum, localsum) local (i) local (myid)  
{  
myid = mp_my_threadnum();  
  
#pragma pfor  
for (i = 0; i < 10000; i++)  
localsum [myid] += a [i];  
}  
for (i = 0; i < numprocs; i++)  
sum += localsum[i];
```

The only operation in the critical region is the computation of the final result from the local results on individual processors.

In the case when the reduction applies to an array of integers, the reduction function can be specialized by using an intrinsic operation __fetch_and_<op> rather than the more expensive critical region. (See Section 11.4.)

For example, to add an array of integers, the critical region can be replaced by the following call:

```
__fetch_and_add(&sum, localsum);
```

The intrinsic __fetch_and_<op> is defined only for the following operations: add, sub, or, xor, nand, mpy, min, and max; and for the type integers together with their size and signed variants. Therefore, it cannot be used in the general case.

10.4.3 Reduction on User−Defined Types in C++

In C++ a generalized reduction function can be written for any user−defined binary operator op that satisfies the associative property.

10.4.3.1 Reduction Example

The following generic function performs reduction on an array of elements of type ElemType, with array indices of type IndexType, and a binary operation op that takes two arguments of type ElemType and produces a result of type ElemType. The type IndexType is assumed to have operators <, −, and ++ defined on it. The use of a function object plus is in keeping with the spirit of generic programming as in the Standard Template Library (STL). A function object is preferred over a function pointer because it permits inlining:

```
template <class ElemType, class IndexType, class BinaryOp>  
ElemType reduction(IndexType first, IndexType last,  
ElemType zero, ElemType ar[],  
BinaryOp op) {  
  
ElemType result = zero;  
IndexType i;  
#pragma parallel shared (result, ar) local (i) byvalue(zero, first, last)  
{  
    ElemType localresult = zero;  
  
#pragma pfor  
{  
for (i = first; i < last - first; i++)  
localresult = op(localresult,ar[i]);  
}  
  
#pragma critical  
     result = op(result,localresult);  
}  
  
return result;  
}
```

With the preceding definition of reduction, you can perform the following reduction:

```
adsum = reduction(0,size,0,ad,plus<double>());  
acsum = reduction(0,size,czero,ac,plus<Complex>());
```

10.5 Restrictions for the C++ Compiler

This section summarizes some restrictions that are relevant only for the C++ compiler. It also lists some restrictions that result from the interaction between pragmas and C++ semantics.

10.5.1 Restrictions on `pfor`

If you are writing a `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 this 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.

10.5.2 Restrictions on Exception Handling

The following restrictions apply to exception handling by the multiprocessing C++ compiler:

- A throw cannot cross an multiprocessing parallel region boundary; it must be caught within the multiprocessing region.

  A thread that throws an exception must catch the exception as well. For example, the following program is valid. Each thread throws and catches an exception:

```
extern "C" printf(char *,...);  
extern "C" int mp_my_threadnum();  
main() {  
int localmax,n;  
  
#pragma parallel local (localmax,n)  
    {  
      localmax = 0;  
  
try {  
          throw 10;  
      }  
      /* .... */  
      catch (int) {  
        printf("!!!!exception caught in process \n");  
        printf("My thread number is %d\n",mp_my_threadnum());  
      } /* end of try block */  
    } /* end of parallel region */  
}
```

- An attempt to throw (propagate) an exception past the end of a parallel program region results in a runtime abort. All other threads abort.

  For example, if the following program is executed, all threads abort:

```
extern "C" printf(char *,...);  
void ehfn() {  
try {  
          throw 10;  
      }  
catch (double)  // not a handler for throw 10  
      {  
        printf("exception caught in process \n");  
      }  
}  
  
main() {  
#pragma parallel  
    {  
      ehfn();  
    }  
}
```

  The program aborts even if a handler is present in `main()`, as in the following example:

```
main() {  
#pragma parallel  
    {  
     try {  
          ehfn();  
         }  
     catch (...) {};  
     }
```

  The reason this program aborts is that the throw propagates past the multiprocessing region.

10.5.3 Scoping Restrictions

The following default scope rules apply for the C++ multiprocessing compiler.

- Class objects or structures that have constructors [that is, non−pods (plain old data structures)] cannot be placed on the local list of `#pragma parallel`.

  The following is invalid:

```
class C {  
```

```
 ....  
};  
  
main() {  
  
C c;  
#pragma parallel local (c) // Class object c cannot be in local list  
{  
....  
}  
}
```

Instead, declaring such objects within the parallel region allows the default rule to be used to indicate that they are local (as the following example illustrates):

```
main() {  
#pragma parallel  
{  
C c;  
....  
}  
}
```

- Structure fields and class object members cannot be placed on the local list. Instead, the entire class object must be made local.
- Values of variables in the local list are not copied into each processor's local variables; instead, initialize locals within the parallel program text. For example,

```
main() {  
int i;  
  
i = 0;  
#pragma parallel local(i)  
{  
// Here i is not 0.  
// Explicit initialization of i within the parallel region  
// is necessary  
}  
}
```