MIPSpro^{TM} C and C++ Pragmas

Document Number 007–3587–003

New Features

MIPSpro™ C and C++ Pragmas 007–3587–003

This revision of *MIPSpro C and C++ Pragmas* supports the 7.3 release of the MIPSpro compiler. See the `cc`(1) man page for changes or additions to command-line options for the MIPSpro C and MIPSpro C++ compiler.

The following changes have been made to this document:

- Information has been added about the support for OpenMP directives based on the OpenMP C/C++ Application Program Interface (API) standard (Chapter 10, page 89).
- A new `#pragma` directive has been added, `#pragma unknown_control_flow` (Section 14.4, page 131).
- A new `#pragma` directive has been added, `#pragma pure` (Section 9.12, page 85).
- A new chapter was added about the Auto-Parallelizing Option (APO) (Chapter 15, page 133). This information was taken from the *MIPSpro Auto-Parallelizing Option Programmer's Guide*, which will no longer be revised.

Record of Revision

Version	*Description*
7.3	March 1999 This revision supports the 7.3 version of the MIPSpro compiler.

Contents

Page

About This Manual

This publication documents `#pragma` directives supported for the 7.3 release of the MIPSpro C and C++ compilers.

Related Publications

The following documents contain additional information that may be helpful:

- *C Language Reference Manual*
- *C++ Programmer's Guide*

Obtaining Publications

The *User Publications Catalog* describes the availability and content of all Cray Research hardware and software documents that are available to customers. Customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

To order a document, call +1 651 683 5907. Silicon Graphics employees may send electronic mail to `orderdsk@sgi.com` (UNIX system users).

Customers who subscribe to the CRInform program can order software release packages electronically by using the `Order Cray Software` option.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.

`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and part number of the document with your comments.

You can contact us in any of the following ways:

- Send electronic mail to the following address:

  `techpubs@sgi.com`

- Send a facsimile to the attention of “Technical Publications” at fax number +1 650 932 0801.

- Use the Suggestion Box form on the Technical Publications Library World Wide Web page:

  `http://techpubs.sgi.com/library/`

- Call the Technical Publications Group, through the Technical Assistance Center, using one of the following numbers:

  For Silicon Graphics IRIX based operating systems: 1 800 800 4SGI

  For UNICOS or UNICOS/mk based operating systems or CRAY Origin2000 systems: 1 800 950 2729 (toll free from the United States and Canada) or +1 651 683 5600

- Send mail to the following address:

  Technical Publications
  Silicon Graphics, Inc.
  1600 Amphitheatre Pkwy.
  Mountain View, California 94043–1351

We value your comments and will respond to them promptly.

Alphabetical Listing of Directives [1]

`#pragma` directives are used within the source program to request certain kinds of special processing. `#pragma` directives are part of the C and C++ languages, but the meaning of any `#pragma` directive is defined by the implementation. `#pragma` directives are expressed in the following form:

`#pragma` *identifier* [*arguments*]

Compiler directives can also be specified in the following form, which has the advantage in that it can appear inside macro definitions:

`_Pragma("`*identifier*`");`

This form has the same effect as using the `#pragma` form, except that everything that appeared on the line following the `#pragma` must now appear inside the double quotation marks and parentheses. The expression inside the parentheses must be a single string literal, but it cannot be a macro that expands into a string literal. `_Pragma` is a Silicon Graphics extension to the C and C++ standards.

The following is an example using the `#pragma` form:

```
#pragma ivdep
#pragma parallel local(i, j, k) \
                 shared(a, b, c)
```

The following is the same example using the alternative form:

```
_Pragma("ivdep")
_Pragma("parallel local(i, j, k) \
                  shared(a, b, c)")
```

Macro expansion occurs on the directive line after the directive name. (That is, macro expansion is applied only to arguments.) For example, if `NUM_CHUNKS` is a macro defined as the value 8, the original code is as follows:

```
#define NUM_CHUNKS 8
_Pragma("parallel numchunks(NUM_CHUNKS)")
```

Table 1, page 2, is an alphabetical list of Silicon Graphics supported `#pragma` directives, with a short description of each and a link to the chapter where the directive is discussed.

Table 1. Silicon Graphics `#pragma` Directives

`#pragma`	Short Description	Functional Group
`aggressive inner loop fission`	Fission inner loops into as many loops as possible.	Chapter 8, page 47
`align_symbol`	Specifies alignment of user variables, typically at cache-line or page boundaries.	Chapter 4, page 19
`blocking size`	Sets the blocksize of the specified loop that is involved in a blocking for the primary (secondary) cache.	Chapter 8, page 47
`can_instantiate`	Indicates that the specified declaration can be instantiated in the current compilation, but need not be.	Chapter 3, page 15
`concurrent`	Tells the compiler to ignore assumed dependences in the following loop.	Chapter 2, page 9
`concurrent call`	Tells the compiler that the function calls in the following loop are safe to execute in parallel.	Chapter 2, page 9
`concurrentize`	Tells the compiler to parallelize the next loop, overriding any `#pragma no concurrentize` directive that may apply to that loop.	Chapter 2, page 9
`copyin`	Copies the value from the master thread's version of an `-Xlocal`-linked global variable into the slave thread's version.	Chapter 9, page 59
`critical`	Protects access to critical statements.	Chapter 9, page 59
`distribute`	Specifies data distribution.	Chapter 5, page 23
`distribute_reshape`	Specifies data distribution with reshaping.	Chapter 5, page 23
`do_not_instantiate`	Prevents instantiation of the specific declaration in this compilation unit, even if that instance is used in the code.	Chapter 3, page 15
`dynamic`	Tells the compiler that the specified array may be redistributed in the program.	Chapter 5, page 23

#pragma	Short Description	Functional Group
`enter gate`	Indicates the point that all threads must clear before any threads are allowed to pass the corresponding `#pragma exit gate`.	Chapter 9, page 59
`exit gate`	Stops threads from passing this point until all threads have cleared the corresponding `#pragma enter gate`.	Chapter 9, page 59
`fill_symbol`	Tells the compiler to insert any necessary padding to ensure that the user variable does not share a cache-line with any other symbol.	Chapter 4, page 19
`fission`	Fission the enclosing specified levels of loops after this directive.	Chapter 8, page 47
`fissionable`	Disables validity testing.	Chapter 8, page 47
`fusable`	Disables validity testing.	Chapter 8, page 47
`fuse`	Fuse the following specified number of loops, which must be immediately adjacent.	Chapter 8, page 47
`hdrstop`	Indicates the point at which the precompiled header mechanism snapshots the headers. If `-pch` is off, `#pragma hdrstop` is ignored.	Chapter 11, page 121
`hidden`	Tells the compiler that the specified symbols are invisible to all executables or DSOs except the current one.	Chapter 7, page 39
`ident`	Adds a `.comment` section in the object file and puts the revision string inside the `.comment` section.	Chapter 14, page 129
`independent`	Tells the compiler to run an independent code section in parallel with the rest of the code in the parallel region.	Chapter 9, page 59

`#pragma`	Short Description	Functional Group
`inline {here|routine|global}]`	Tells the compiler to inline the named functions. Keywords: here (next statement only), routine (rest of routine or until corresponding `noinline` is found), and global (entire file, or until corresponding `noinline` is found).	Chapter 6, page 33
`instantiate`	Causes a specified instance of a template declaration to be immediately instantiated at that spot.	Chapter 3, page 15
`int_to_unsigned`	Identifies the specified function name as a function whose type was `int` in a previous release of the compilation system, but whose type is `unsigned int` in the MIPSpro compiler release.	Chapter 14, page 129
`internal`	Tells the compiler that the specified symbols are not referenced outside the current executable or DSO.	Chapter 7, page 39
`intrinsic`	Allows certain preselected functions from `math.h`, `stdio.h`, and `string.h` to be inlined at a callsite for execution efficiency.	Chapter 14, page 129
`ivdep`	Liberalizes dependence analysis. This applies only to inner loops. Given two memory references, where at least one is loop variant, ignore any loop-carried dependences between the two references.	Chapter 8, page 47
`local`	Tells the compiler the names of all the variables that must be local to each thread.	Chapter 9, page 59
`mips_frequency_hint {NEVER|INIT}`	Specifies the expected frequency of execution so the compiler can move exception code and initialization code into separate pages to minimize working set size.	Chapter 12, page 123
`no blocking`	Prevents the compiler from involving this loop in cache blocking.	Chapter 8, page 47

#pragma	Short Description	Functional Group
no concurrentize	Varies with placement. Tells the compiler to not parallelize any loops in a subroutine or file.	Chapter 2, page 9
no_delete	Inhibits deletion of functions that are never referenced.	Chapter 8, page 47
no fission	Keeps the following loop from being fissioned. Its innermost loops, however, are allowed to be fissioned.	Chapter 8, page 47
no fusion	Keeps the following loop from being fused with other loops.	Chapter 8, page 47
no interchange	Prevents the compiler from involving the loop directly following this directive (or any loop nested within this loop) in an interchange.	Chapter 8, page 47
no side effects	Tells the compiler to assume that all of the named functions are safe to execute concurrently.	Chapter 9, page 59
no_pch	Disables the precompiled header mechanism.	Chapter 11, page 121
noinline {here|routine|global}	Tells the compiler not to inline the named functions. Keywords: here (next statement only), routine (rest of routine or until corresponding `inline` is found), and global (entire file, or until corresponding `inline` is found).	Chapter 6, page 33
once	Ensures (in `-n32` and `-64` mode) that each `include` file is included at most one time in each compilation unit.	Chapter 11, page 121
one processor	Causes next statement to be executed on only one processor.	Chapter 9, page 59
optional	Tells the linker that the specified symbols are optional. This is the basic mechanism used for adding extensions to a library that can then be queried.	Chapter 7, page 39

`#pragma`	Short Description	Functional Group
`pack`	Controls the layout of structure offsets, such that the strictest alignment for any structure member will be *n* bytes, where *n* is 0, 1, 2, 4, 8, or 16. When *n* is 0, the compiler returns to default alignment for any subsequent `struct` definitions.	Chapter 4, page 19
`page_place`	Controls the placement of data on a DSM (distributed shared memory) machine.	Chapter 5, page 23
`permutation`	The specified array is a permutation array.	Chapter 2, page 9
`parallel`	Starts a parallel region.	Chapter 9, page 59
`pfor`	Marks a **for** loop to run in parallel.	Chapter 9, page 59
`prefer concurrent`	Tells the compiler to parallelize the following loop if it is safe.	Chapter 2, page 9
`prefer serial`	Tells the compiler not to parallelize the following loop.	Chapter 2, page 9
`prefetch`	Controls prefetching for each level of the cache.	Chapter 8, page 47
`prefetch_manual`	Specifies whether manual prefetches (through `#pragma` directives) should be respected or ignored.	Chapter 8, page 47
`prefetch_ref`	Generates a prefetch and connects it to the specified reference (if possible).	Chapter 8, page 47
`prefetch_ref_disable`	Explicitly disables prefetching for the specified reference.	Chapter 8, page 47
`protected`	Tells the compiler that the specified symbols are not preemptible.	Chapter 7, page 39
`pure`	Tells the compiler that a call to named functions has no side effects and its return value depends on the values of its arguments.	Chapter 9, page 59
`redistribute`	Specifies dynamic data redistribution.	Chapter 5, page 23
`reset woff`	Resets listed warnings to the state specified in the command line.	Chapter 13, page 125

`#pragma`	Short Description	Functional Group
`section_gp`	Causes an object to be placed in a gp_relative section.	Chapter 7, page 39
`section_non_gp`	Keeps an object from being placed in a gp_relative section.	Chapter 7, page 39
`serial`	Forces the loop immediately following it to be serial, and restricts optimization by forcing all enclosing loops to be serial also.	Chapter 2, page 9
`set chunksize`	Tells the compiler which values to use for `chunksize`.	Chapter 9, page 59
`set numthreads`	Tells the compiler which values to use for `numthreads`.	Chapter 9, page 59
`set schedtype`	Tells the compiler which values to use for `schedtype`.	Chapter 9, page 59
`set woff`	Suppresses listed compiler warnings.	Chapter 13, page 125
`shared`	Tells the compiler the names of all the variables that the threads must share.	Chapter 9, page 59
`synchronize`	Stops threads until all threads reach this point. This directive is a classic barrier construct.	Chapter 9, page 59
`unknown_control_flow`	Indicates which procedures have a nonstandard control flow behavior.	Chapter 14, page 129
`unroll`	Suggests to the compiler that *n*-1 copies of the loop body be added to the inner loop. If the loop following this directive is an inner loop, then it indicates standard unrolling (version 7.2 and later). If the loop following this directive is not innermost, then outer loop unrolling (unroll and jam) is performed (version 7.0 and later).	Chapter 8, page 47

`#pragma`	Short Description	Functional Group
`weak` *weak_symbol* = *strong_symbol*	Sets *weak_symbol* to be an alias for the function or data object denoted by *strong_symbol*, unless a defining declaration for *weak_symbol* is encountered at static link time. If encountered, the defining declaration preempts the weak denotation.	Chapter 7, page 39
`weak` *weak_symbol*	Tells the link editor not to issue a warning if it does not find a defining declaration of *weak_symbol*. Also allows the overriding of a current definition by a non-weak definition.	Chapter 7, page 39

Automatic Parallelization #pragma Directives [2]

Table 2, page 9, lists 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 2. IRIS Power C Analyzer `#pragma` Directives

#pragma	Short Description	Compiler Versions
`#pragma concurrent`	Tells the compiler to ignore assumed dependences in the next loop.	7.2 and later
`#pragma concurrent call`	Tells the compiler that the function calls in the next loop are safe to execute in parallel.	7.2 and later
`#pragma concurrentize`	Tells the compiler to parallelize the next loop, overriding any `#pragma no concurrentize` directive that may apply to that loop.	7.2 and later
`#pragma no concurrentize`	Varies with placement. Tells the compiler to not parallelize any loops in a function or file.	7.2 and later
`#pragma permutation`	The specified array is a permutation array.	7.2 and later
`#pragma prefer concurrent`	Tells the compiler to parallelize the next loop if it is safe.	7.2 and later
`#pragma prefer serial`	Tells the compiler to not parallelize the next loop.	7.2 and later
`#pragma serial`	Forces the loop immediately following it to be serial, and restricts optimization by forcing all enclosing loops to be serial also.	7.2 and later

2.1 #pragma concurrent

The `#pragma concurrent` directive instructs the compiler, when analyzing the loop immediately following this assertion, to ignore all dependences between two references to the same array.

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

```
#pragma concurrent
```

When using this directive, be aware of the following:

- If multiple loops in a nest can be parallelized, `#pragma concurrent` instructs the compiler to parallelize the loop immediately following the directive.
- Applying this directive to an inner loop may cause the loop to be made outermost by the compiler's loop interchange operations.
- `#pragma concurrent` does not affect how the compiler analyzes function calls. See Section 2.2, page 10.
- `#pragma concurrent` does not affect how the compiler analyzes dependences between two potentially aliased pointers.
- If there are real dependences between array references, `#pragma concurrent` may cause the compiler to generate incorrect code.

2.2 `#pragma concurrent call`

The `#pragma concurrent call` directive instructs the compiler to ignore the dependences of any function calls contained in the loop that follows the directive.

The syntax for `#pragma concurrent call` is as follows:

```
#pragma concurrent call
```

This directive applies to the loop that immediately follows it and to all loops nested inside that loop.

To prevent incorrect parallelization, make sure the following conditions are met when using `#pragma concurrent call`:

- A function inside the loop cannot read from a location that is written to during another iteration. This rule does not apply to a location that is a local variable declared inside the function.
- A function inside the loop cannot write to a location that is read from or written to during another iteration. This rule does not apply to a location that is a local variable declared inside the function.

2.2.1 Examples of #pragma concurrent call

2.2.1.1 Example 1

In this example the compiler ignores the dependences in the function `fred()` when it analyzes the following loop:

```
#pragma concurrent call
for (i = 0; i < N; i++0
{
fred(...)
...
}

void fred (...)
{
...
}
```

2.2.1.2 Example 2

The following code shows an illegal use of the assertion. Function `fred()` writes to variable *T,* which is also read by `wilma()` during other iterations.

```
float A[M], B[M];
int i, T;
#pragma concurrent call
for (i = 0; i < M; i++)
{
fred(B, i, &T);
wilma(A, i, &T);
}

void fred(float B[], int i, int* T)
{
*T = B[i];
}

void wilma(float A[], int i, int* T)
{
A[i] = *T;
}
```

By localizing the variable *T*, you can manually parallelize the preceding example safely. But the compiler is not instructed to localize *T*, and the loop is illegally parallelized because of the assertion.

2.3 `#pragma concurrentize`

The `#pragma concurrentize` directive instructs the compiler to parallelize an entire file or function.

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

```
#pragma concurrentize
```

Placing the `#pragma concurrentize` directive inside a function overrides a `#pragma no concurrentize` directive placed outside of it. In other words, this directive allows you to selectively parallelize functions in a file that has been made sequential with `#pragma no concurrentize`.

This directive works only with the MIPSpro Automatic Parallelizer.

2.4 `#pragma no concurrentize`

The `#pragma no concurrentize` directive prevents parallelization of a file or function.

The syntax of `#pragma no concurrentize` is as follows:

```
#pragma no concurrentize
```

The effect of `#pragma no concurrentize` depends on its placement:

- When placed inside a function, the directive prevent its parallelization.
- When placed outside of a function, `#pragma no concurrentize` prevents the parallelization of all functions in the file, even those that appear ahead of it in the file.

This directive works only with the MIPSpro Automatic Parallelizer.

2.5 `#pragma permutation`

When placed inside a function, the `#pragma permutation` directive instructs the compiler that the specified array is a permutation array.

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

`#pragma permutation (`*array*`)`

array is the name of a permutation array. Every element of *array* has a distinct value. The directive does not require the permutation array to be dense. In other words, while every *array[1]* must have a distinct value, there can be gaps between those values, such as *array[1]* = 1, *array[2]* = 4, *array[3]* = 9, and so on.

You can use this assertion to parallelize loops that use arrays for indirect addressing. Without this directive, the compiler cannot determine that the array elements used as indexes are distinct.

The `#pragma permutation` directive affects every loop in a function, even those that precede it.

2.6 #pragma prefer concurrent

The `#pragma prefer concurrent` directive instructs the compiler to parallelize the loop immediately following the directive, if it is safe to do so.

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

```
#pragma prefer concurrent
```

This pragma is always safe to use. The compiler parallelizes the loop only when it can determine that it is safe to do so.

When dealing with nested loops, the compiler follows these guidelines:

- If the loop specified by this directive is safe to parallelize, the compiler chooses it to parallelize, even if other loops are also candidates for parallelization.
- If the specified loop is not safe to parallelize, the compiler uses its heuristics to choose among loops that are safe.
- If this directive is applied to an inner loop, the compiler may make it the outermost loop.
- If this assertion is applied to more than one loop in a nest, the compiler uses its heuristics to choose one of the specified loops.

This directive works only with the MIPSpro Automatic Parallelizer.

2.7 `#pragma prefer serial`

The `#pragma prefer serial` directive instructs the compiler to not parallelize the loop that immediately follows it. It performs in the same way as the `#pragma serial` directive.

The syntax of `#pragma prefer serial` is as follows:

```
#pragma prefer serial
```

This directive works only with the MIPSpro Automatic Parallelizer.

2.8 `#pragma serial`

The `#pragma serial` directive instructs the compiler to not parallelize the loop following the assertion. However, the compiler may parallelize another loop in the same nest. The parallelized loop may be either inside or outside the designated sequential loop.

The syntax for this directive is as follows:

```
#pragma serial
```

This directive works only with the MIPSpro Automatic Parallelizer.

C++ Instantiation #pragma Directives [3]

Instantiation `#pragma` directives control the instantiation of specific template entities or sets of template entities.

Table 3, page 15, lists the C++ instantiation `#pragma` directives discussed in this chapter, along with a brief description of each and the compiler versions in which the directive is supported.

Table 3. C++ Template Instantiation `#pragma` Directives

#pragma	Short Description	Compiler Versions
`#pragma instantiate`	Causes a specified instance of a template declaration to be immediately instantiated at that spot.	7.1 and later
`#pragma can_instantiate`	Indicates that the specified *declaration* can be instantiated in the current compilation, but need not be.	7.0 and later
`#pragma do_not_instantiate`	Prevents instantiation of the specific *declaration* in this compilation unit, even if that instance is used in the code.	7.0 and later

3.1 `#pragma instantiate`

The `#pragma instantiate` directive causes a specific instance of a template declaration to be immediately instantiated.

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

`#pragma instantiate` *entity*

The *entity* argument can be any of the following:

A template class name	`A<int>`

A member function name	`A<int>::foo`
A member function declaration	`void A<int>::foo(int, char)`
A static data member name	`A<int>::name`
A template function declaration	`char* foo(int, float)`

The template definition of *entity* must be present in the compilation for an instantiation to occur. If you use `#pragma instantiate` to explicitly request the instantiation of a class or function for which no template definition is available, the compiler issues a warning.

The declaration needs to be a complete declaration of a function or a static data member, exactly as if you had specified it for a specialization of the template.

The argument to an instantiation `#pragma` directive cannot be a compiler-generated function, an inline function, or a pure virtual function.

A member function name (for example, `A<int>::foo`) can be used as an argument for a `#pragma instantiate` directive only if it refers to a single, user-defined member function that is not an overloaded function. Compiler-generated functions are not considered, so a name can refer to a user-defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be instantiated by providing the complete member function declaration, as the following example shows:

```
char * A<int>::foo(int))
```

Note: Using the `#pragma instantiate` directive to instantiate a template class is equivalent to repeating the directive for each member function and static data member declared in the class. When instantiating an entire class, you can exclude a given member function or static data member by using the `#pragma do_not_instantiate` directive.

3.2 #pragma can_instantiate

The `#pragma can_instantiate` directive indicates that the specified entity can be instantiated in the current compilation, but need not be. It is used in conjunction with automatic instantiation to indicate potential sites for instantiation if the template entity is deemed to be required by the compiler.

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

```
#pragma can_instantiate entity
```

The argument, *entity*, can be any of the following:

A template class name	`A<int>`
A member function name	`A<int>::foo`
A member function declaration	`void A<int>::foo(int, char)`
A static data member name	`A<int>::name`
A template function declaration	`char* foo(int, float)`

The template definition of *entity* must be present in the compilation for an instantiation to occur. If you use `#pragma can_instantiate` to explicitly request the instantiation of a class or function for which no template definition is available, the compiler issues a warning.

The argument to a `#pragma can_instantiate` directive cannot be a compiler-generated function, an inline function, or a pure virtual function.

A member function name (for example, `A<int>::foo`) can be used as an argument for a `#pragma can_instantiate` directive only if it refers to a single, user-defined member function that is not an overloaded function. Compiler-generated functions are not considered, so a name can refer to a user-defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be instantiated by providing the complete member function declaration, as shown in the following example:

```
char * A<int>::foo(int)
```

3.3 #pragma do_not_instantiate

The `#pragma do_not_instantiate` directive suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition is supplied.

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

```
#pragma do_not_instantiate entity
```

The argument, *entity*, can be any of the following:

A template class name	`A<int>`

A member function name	`A<int>::foo`
A member function declaration	`void A<int>::foo(int, char)`
A static data member name	`A<int>::name`
A template function declaration	`char* foo(int, float)`

The argument to a `#pragma do_not_instantiate` directive cannot be a compiler-generated function, an inline function, or a pure virtual function.

A member function name (for example, `A<int>::foo`) can be used as an argument for the `#pragma do_not_instantiate` directive only if it refers to a single, user-defined member function that is not overloaded. Compiler-generated functions are not considered, so a name can refer to a user-defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be specified by providing the complete member function declaration, as the following example shows:

```
char * A<int>::foo(int)
```

Data Layout #pragma Directives [4]

Table 4, page 19, lists the #pragma directives discussed in this chapter, along with a short description of each and the compiler versions in which the directive is supported.

Table 4. Data Layout #pragma Directives

#pragma	Short Description	Compiler Versions
#pragma align_symbol	Specifies alignment of user variables, typically at cache-line or page boundaries.	7.2 and later
#pragma fill_symbol	Tells the compiler to insert any necessary padding to ensure that the user variable does not share a cache-line or page with any other symbol.	7.2 and later
#pragma pack	Controls the layout of structure offsets, such that the strictest alignment for any structure member will be *n* bytes, where *n* is 0, 1, 2, 4, 8, or 16. When *n* is 0, the compiler returns to default alignment for any subsequent struct definitions.	7.0 and later

4.1 #pragma align_symbol

The #pragma align_symbol directive specifies the alignment of user variables, typically at cache-line or page boundaries.

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

#pragma align_symbol (*symbol*, *size*)

The first argument to this directive is a *symbol*. The symbol can be a global or automatic variable, but it cannot be a formal parameter to a function, or an element of a structured type such as a structure or array.

The second argument, *size*, can be any one of the following:

- L1cacheline, a machine-specific first-level cache-line size, typically 32 bytes
- L2cacheline, a machine-specific second-level cache-line size, typically 128 bytes
- page, a machine specific page size, typically 16 Kilobytes
- a user-specified value, which must be a power of two

The `#pragma align_symbol` directive aligns the start of *symbol* at the specified alignment boundary.

For global variables, this directive must be specified where the variable is defined. The directive is optional where the variable is declared.

Caution: When using the `#pragma align_symbol` directive, there are two points to keep in mind:

- The `#pragma align_symbol` directive is ineffective for local variables of fixed-size symbols, such as simple scalars or arrays of known size. Theis directive is most effective for stack-allocated arrays of dynamically determined size.
- A variable cannot have both `#pragma fill_symbol` and `#pragma align_symbol` directives applied to it.

4.1.1 Example of `#pragma align_symbol`

The following code fragment illustrates the use of the `#pragma align_symbol` directive:

```
int x;                        /* x is a global variable */

#pragma align_symbol (x, 32)  /* x will start at a 32-byte boundary */

#pragma align_symbol (x, 2)   /* Error: cannot request an alignment
                                 lower than the natural alignment of the symbol. */
```

4.2 #pragma fill_symbol

The #pragma fill_symbol directive instructs the compiler to insert any necessary padding to ensure that the user variable does not share a cache-line, page, or other specified block of memory with any other symbol.

The syntax of the fill_symbol pragma is as follows:

#pragma fill_symbol (*symbol*, *size*)

The first argument to this pragma is a symbol. The symbol can be a global or automatic variable, but it cannot be a formal parameter to a function, or an element of a structured type such as a structure or array.

The second argument can be any one of the following:

- L1cacheline, a machine-specific first-level cache-line size, typically 32 bytes
- L2cacheline, a machine-specific second-level cache-line size, typically 128 bytes
- page, a machine specific page size, typically 16 kilobytes
- a user-specified value that must be a power of two

The #pragma fill_symbol directive pads the named *symbol* with additional storage so that the symbol is assured not to overlap with any other data item within the storage of the specified *size*. The additional padding required is heuristically divided between each end of the specified variable.

For instance, a #pragma fill_symbol directive for the L1cacheline guarantees that the specified *symbol* will not suffer from false-sharing (multiple, unrelated symbols sharing the same cache line) between multiple processors for the L1 cache line.

For global variables, this directive must be specified where the variable is defined. The directive is optional where the variable is declared.

A variable cannot have both #pragma fill_symbol and #pragma align_symbol directives applied to it.

4.2.1 Example of #pragma fill_symbol

The following code fragment illustrates the use of #pragma fill_symbol:

```
double y;                           /* y is a global or local variable */
#pragma fill_symbol (y, L2cacheline)  /* Allocates extra storage
```

```
both before and after y so that
y is within an L2cacheline (128
bytes) all by itself. */
```

4.3 #pragma pack

The `#pragma pack` directive controls the layout of structure offsets. The strictest alignment for any structure member is the specified number of bytes (1, 2, 4, 8, or 16).

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

`#pragma pack (`*n*`)`

The `#pragma pack` directive works according to the following rules:

- A struct type defined in the scope of a `#pragma pack` has up to *n* bytes of alignment, where *n* is 0, 1, 2, 4, 8, or 16. When *n* is 0, the compiler returns to default alignment for any subsequent structure definitions.
- The packed characteristics of the type apply wherever the type is used, even outside the scope of the pragma in which the type was declared.
- The scope of a `#pragma pack` ends with the next `#pragma pack`, hence this pragma does not nest. There is no way to "return" from one instance of the directive to a lexically earlier instance of the directive.

Caution:

- Silicon Graphics strongly discourages the use of `#pragma pack`, because it is a nonportable feature and the semantics of this directive may change in future compiler releases.
- A structure declaration must be subjected to identical instances of a `#pragma pack` directive in all files, or else misaligned memory accesses and erroneous structure member dereferencing may ensue.
- References to fields in packed structures may be less efficient than references to fields in unpacked structures.
- The `#pragma pack` directive is not supported for C++ in `-n32` and `-64` modes.

DSM Optimization #pragma Directives [5]

Table 5, page 23, lists the #pragma directives discussed in this chapter, along with a short description of each and the compiler versions in which the directive is supported. These directives are useful primarily on systems with distributed shared memory, such as Origin servers.

Table 5. Distributed Shared Memory #pragma Directives

#pragma	Short Description	Compiler Versions
`#pragma distribute`	Specifies data distribution.	7.2 and later
`#pragma distribute_reshape`	Specifies data distribution with reshaping.	7.2 and later
`#pragma dynamic`	Tells the compiler that the specified array may be redistributed in the program.	7.2 and later
`#pragma page_place`	Allows the explicit placement of data.	7.1 and later
`#pragma pfor` (Discussed in Chapter 9, page 59)	`affinity` clause allows data-affinity or thread-affinity scheduling; `nest` clause exploits nested concurrency. See Section 9.11, page 76	6.0 and later
`#pragma redistribute`	Specifies dynamic redistribution of data.	7.2 and later

5.1 #pragma distribute

The `#pragma distribute` directive specifies the distribution of data across the processors. It functions by influencing the mapping of virtual addresses to physical pages without affecting the layout of the data structure. Because the granularity of data allocation is a physical page (at least 16 KB), the achieved distribution is limited by the underlying page granularity. However, the advantages to using this directive are that it can be added to an existing program without any restrictions, and can be used for affinity scheduling. See Section 9.11.4, page 78, for more information about data affinity.

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

```
#pragma distribute array[dst1][[dst2]...] [onto (dim1, dim2[, dim3 ...])]
```

array is the name of the array you want to have distributed.

dst is the distribution specification for each dimension of the array. It can be any one of the following:

dst is the distribution specification for each dimension of the array. It can be any one of the following:

Value	Effect
`*`	Not distributed.
`block`	Partitions the elements of an array dimension into blocks equal to the size of the dimension (*N*) divided by the number of processors (*P*). The size of each block will be equal to *N*/*P*, rounded up to the nearest integer value (`ceiling (N/P)`).
`cyclic` [(*size_expr*)]	Partitions the elements of an array dimension into chunks and distributes the chunks sequentially across the processors. The size of the pieces is equal to the value of *size_expr*. If *size_expr* is not specified, the chunk size defaults to 1. A `cyclic` distribution with a chunk size that is either greater than 1 or is determined at run time is sometimes also called `block-cyclic`.

dim is the specification for partitioning the processors across the distributed dimensions (see Section 5.5.1, page 31, for more information).

The following is some additional information about `#pragma distribute`:

- You must specify the `#pragma distribute` directive in the declaration part of the program, along with the array declaration.
- You can specify a data distribution directive for any local or global array.
- Each dimension of a multi-dimensional array can be independently distributed.
- A distributed array is distributed across all of the processors being used in that particular execution of the program, as determined by the environment variable `MP_SET_NUMTHREADS`.

5.1.1 Example of #pragma distribute

The following code fragment demonstrates the use of #pragma distribute:

```
float A[200][300];
...
#pragma distribute A[cyclic][block];
...
```

On a machine with eight processors, the first dimension of array *A* is distributed across the processors in chunks of 1, and the second dimension is distributed in chunks of 25 for each processor.

5.1.2 onto Clause

If an array is distributed in more than one dimension, then by default the processors are apportioned as equally as possible across each distributed dimension. For instance, if an array has two distributed dimensions, then an execution with 16 processors assigns 4 processors to each dimension ($4 \times 4 = 16$), whereas an execution with 8 processors assigns 4 processors to the first dimension and 2 processors to the second dimension.

You can override this default and explicitly control the number of processors in each dimension by using the onto clause. The onto clause allows you to specify the processor topology when an array is being distributed in more than one dimension. For instance, if an array is distributed in two dimensions, and you want to assign more processors to the second dimension than to the first dimension, you can use the onto clause as in the following code fragment:

```
float A[100][200];

/* Assign to the second dimension twice as many processors as to
the first dimension. */

#pragma distribute A[block][block] onto (1, 2)
```

5.2 #pragma distribute_reshape

The #pragma distribute_reshape directive, like #pragma distribute, specifies the desired distribution of an array. In addition, however, the #pragma distribute_reshape directive declares that the program makes no assumptions about the storage layout of that array. The compiler performs aggressive optimizations for reshaped arrays that violate standard layout assumptions but guarantee the desired data distribution for that array.

For information about using data affinity with `#pragma redistribute-reshape`, see Section 9.11.4, page 78.

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

`#pragma distribute_reshape` *array*[*dst1*][[*dst2*]...]

The `#pragma distribute_reshape` directive accepts the same distributions as the `#pragma distribute` directive:

- *array* is the name of the array you want to have distributed.
- *dst* is the distribution specification for each dimension of the array. It can be any one of the following:

Value	Effect
`*`	Not distributed.
`block`	Partitions the elements of an array dimension into blocks equal to the size of the dimension (*N*) divided by the number of processors (*P*). The size of each block will be equal to *N/P*, rounded up to the nearest integer value (`ceiling (N/P)`).
`cyclic` (*size_expr*)	Partitions the elements of an array dimension into chunks and distributes the chunks sequentially across the processors. The size of the pieces is equal to the value of *size_expr*. If *size_expr* is not specified, the chunk size defaults to 1. A `cyclic` distribution with a chunk size that is either greater than 1 or is determined at run time is sometimes also called `block-cyclic`.

The following is some additional information about `#pragma distribute_reshape`:

- You must specify the `#pragma distribute_reshape` directive in the declaration part of the program, along with the array declaration.
- You can specify a data distribution directive for any local or global array.

- Each dimension of a multi-dimensional array can be independently distributed.
- A distributed array is distributed across all of the processors being used in that particular execution of the program, as determined by the environment variable `MP_SET_NUMTHREADS`.
- A reshaped array is passed as an actual parameter to a subroutine, in which case two possible scenarios exist:
  - The array is passed in its entirety (`func(A)` passes the entire array `A`, whereas `func(A([i][j])` passes a portion of `A`). The C compiler automatically clones a copy of the called function and compiles it for the incoming distribution. The actual and formal parameters must match in the number of dimensions, and the size of each dimension.

    The C++ compiler does not perform this cloning automatically, due to interactions in the compiler with the C++ template instantiation mechanism. For C++, therefore, the user has the following two options:

    1. The first option is to specify `#pragma distribute_reshape` directly on the formal parameter of the called function.
    2. The second option is to compile with `-MP:clone=on` to enable automatic cloning in C++.

    Caution: This option may not work for some programs that use templates.

  - You can restrict a function to accept a particular reshaped distribution on a parameter by specifying a `#pragma distribute_reshape` directive on the formal parameter within the function. All calls to this function with a mismatched distribution will lead to compile- or link-time errors.
  - A portion of the array can be passed as a parameter, but the callee must access only a single processor's portion. If the callee exceeds a single processor's portion, then the results are undefined. You can use intrinsics to access details about the array distribution (see the "Parallel Programming on Origin Servers" chapter in the *C Language Reference Manual* for more details).

Caution: Because the #pragma distribute_reshape directive specifies that the program does not depend on the storage layout of the reshaped array, restrictions on reshaping arrays include the following (for more details on reshaping arrays, see the *C Language Reference Manual*):

- The distribution of a reshaped array cannot be changed dynamically (that is, there is no #pragma redistribute_reshape directive).
- Initialized data cannot be reshaped.
- Arrays that are explicitly allocated through alloca/malloc and accessed through pointers cannot be reshaped. Use variable length arrays instead.
- An array that is equivalenced to another array cannot be reshaped.
- A global reshaped array cannot be linked -Xlocal. This user error is not caught by the compiler or linker.

5.2.1 Example of #pragma distribute_reshape

The following code fragment demonstrates the use of #pragma distribute_reshape:

```
float A[400][300];
...
#pragma distribute_reshape A[block][cyclic(3]);
...
```

On a machine with eight processors, the first dimension of array A is distributed in chunks of 50 for each processor, and the second dimension is distributed across the processors in chunks of 3.

5.3 #pragma dynamic

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.

The #pragma dynamic directive notifies the compiler that the named array may be dynamically redistributed at some point in the run. This tells the compiler that any data affinity for that array must be implemented at run time.

For information about using data affinity with `#pragma dynamic`, see Section 9.11.4, page 78.

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

`#pragma dynamic` *array*

array is the name of the array in question.

The `#pragma dynamic` directive informs the compiler that *array* may be dynamically redistributed. Data affinity for such arrays is implemented through a run-time lookup. Implementing data affinity in this manner incurs some extra overhead compared to a direct compile-time implementation, so you should use the `#pragma dynamic` directive only if it is actually necessary.

You must explicitly specify the `#pragma dynamic` declaration for a redistributed array under the following conditions:

- The function contains a `pfor` loop that specifies data affinity for the array.
- The distribution for the array is not known.

Under the following conditions, you can omit the `#pragma dynamic` directive and just supply the `#pragma distribute` directive with the particular distribution:

- The function contains data affinity for the redistributed array.
- The array has a specified distribution throughout the duration of the function.

Because reshaped arrays cannot be dynamically redistributed, this is an issue only for regular data distribution.

5.4 #pragma page_place

The `#pragma page_place` directive is useful for dealing with irregular data structures. It allows you to explicitly place data in the physical memory of a particular processor. This directive is often used in conjunction with thread affinity (see Section 9.11.4, page 78, for more information).

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

`#pragma page_place (`*object*`,` *size*`,` *threadnum*`)`

1. *object* is the object you want to place

2. *size* is the size in bytes
3. *threadnum* is the number of the destination processor

On a system with physically distributed shared memory, you can explicitly place all data pages spanned by the virtual address range [`&object, &object+ size-1`] in the physical memory of the processor corresponding to the specified thread. This directive is an executable statement; therefore, you can use it to place either statically or dynamically allocated data.

The function `getpagesize()` can be invoked to determine the page size. On the Origin2000™ server, the minimum page size is 16384 bytes.

5.4.1 Example of `#pragma page_place`

The following is an example of the use of `#pragma page_place`:

```
double A[8192];
#pragma page_place (A[0], 32768, 0)
#pragma page_place (A[4096], 16384, 1)
```

The first `#pragma page_place` directive causes the first half of the array to be placed in the physical memory associated with thread 0. The second causes the next quarter of the array to be placed in the physical memory associated with thread 1. The remaining portion of *A* is allocated based on the operating system's allocation policy (default is "first-touch").

5.5 `#pragma redistribute`

The `#pragma redistribute` directive allows you to dynamically redistribute previously distributed arrays. For information about using data affinity with `#pragma redistribute`, see Section 9.11.4, page 78.

The syntax of the `redistribute` pragma is as follows:

`#pragma redistribute` *array*[*dst1*][[*dst2*]`...`] `[onto (`*dim1*`,` *dim2*`[,` *dim3* `...])]`

- *array* is the name of the array you wish to have distributed.
- *dst* is the distribution specification for each dimension of the array. It can be any one of the following:

Value	Effect
`*`	Not distributed.
`block`	Partitions the elements of an array dimension into blocks equal to the size of the dimension (*N*) divided by the number of processors (*P*). The size of each block will be equal to N/P, rounded up to the nearest integer value (`ceiling (N/P)`).
`cyclic` (*size_expr*)	Partitions the elements of an array dimension into chunks and distributes the chunks sequentially across the processors. The size of the pieces is equal to the value of *size_expr*. If *size_expr* is not specified, the chunk size defaults to 1. A `cyclic` distribution with a chunk size that is either greater than 1 or is determined at run time is sometimes also called `block-cyclic`.

- *dim* is the specification for partitioning the processors across the distributed dimensions (see Section 5.5.1, page 31, for more information).

The following is some additional information about `#pragma redistribute`:

- It is an executable statement and can appear in any executable portion of the program.
- It changes the distribution permanently (or until another `redistribute` statement).
- It also affects subsequent affinity scheduling.

5.5.1 `onto` Clause

If an array is distributed in more than one dimension, then by default the processors are apportioned as equally as possible across each distributed dimension. For instance, if an array has two distributed dimensions, then an execution with 16 processors assigns 4 processors to each dimension ($4 \times 4 = 16$), whereas an execution with 8 processors assigns 4 processors to the first dimension and 2 processors to the second dimension.

You can override this default and explicitly control the number of processors in each dimension by using the `onto` clause. The `onto` clause allows you to specify the processor topology when an array is being distributed in more than one dimension. For instance, if an array is distributed in two dimensions, and you want to assign more processors to the second dimension than to the first dimension, you can use the `onto` clause as in the following code fragment:

```
float A[100][200];

/* Assign to the second dimension twice as many processors as to
the first dimension. */

#pragma redistribute A[block][block] onto (1, 2)
```

5.5.2 Example of `#pragma redistribute`

The following code fragment demonstrates the use of `#pragma redistribute`:

```
float A[500][300];
...
#pragma redistribute A[cyclic(1)][cyclic (5)];
...
```

After the `#pragma redistribute` directive, the first dimension of array *A* is distributed across the processors in chunks of 1, the second dimension in chunks of 5.

Inlining #pragma Directives [6]

Table 6, page 33, lists 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 6. Inlining #pragma Directives

#pragmas	Short Description	Compiler Versions
`#pragma inline` (see Section 6.1, page 33)	Tells the compiler to inline the named functions. Keywords: - here (next statement only) - routine (rest of routine or until corresponding `noinline` or `inline` is found) - global (entire file, or until corresponding `noinline` or `inline` is found)	7.1 and later
`#pragma noinline` (see Section 6.1, page 33)	Tells the compiler not to inline the named functions. Keywords: - here (next statement only) - routine (rest of routine or until corresponding `noinline` or `inline` is found) - global (entire file, or until corresponding `noinline` or `inline` is found)	7.1 and later

6.1 #pragma inline and #pragma noinline

The `#pragma inline` and `#pragma noinline` directives instruct the compiler whether or not to inline the named functions. These directives can have next-line, entire routine, or global scope.

The syntax of the `#pragma inline` and `#pragma noinline` directives is as follows:

```
#pragma [no] inline {here|routine|global} [(name1[,name2 ...])]
```

`here`, `routine`, and `global` are keywords (see Section 6.1.1, page 34).

The optional *name1* and *name2* are function names. If they are present, they follow these rules:

- If any functions are named in the directive, it applies only to them.
- If no function names are given, the pragma applies to all functions.
- If a specified function does not exist, a warning message is issued, and the pragma is ignored.

If the list of function names is empty, the parentheses around the function names are not required.

6.1.1 Keywords

The descriptions of the `here`, `routine`, and `global` keywords follow. These keywords must appear in lowercase, because function names are case sensitive.

`here`	The directive applies only to the next statement.
`routine`	The directive applies to the rest of the routine, or until a corresponding `#pragma noinline` appears. (Or, if the first directive was a `#pragma noinline`, until the corresponding `#pragma inline`.)
`global`	The directive applies to the entire file, or until toggled with a `#pragma noinline` directive. (Or, if the first directive was a `#pragma noinline`, until the corresponding `#pragma inline` directive.) Typically, `#pragma global` directives appear only at the top of the source file.
no keyword	The `#pragma inline` and `#pragma noinline` directives with no keyword have the same effect as using the `here` keyword, unless the directives appear at the top of the file, before any lines of source code. In that case, the `#pragma` directives

apply to the entire file, as if the `global` keyword had been used.

Caution: For C++ code, `#pragma inline` and `#pragma noinline` take C++ style function names. If you use mangled names, the results are undefined. The compiler gives a warning if it cannot find the supplied name.

6.1.2 Examples of `#pragma inline` and `#pragma noinline`

The following five examples illustrate different aspects of the `#pragma inline` and `#pragma noinline` directives.

Example 1: Using the `here` keyword with the `#pragma noinline` directive

This example illustrates the use of the `#pragma noinline` directive with the `here` keyword. All occurrences of `f1(int)` are marked for inlining, except the one directly following `#pragma noinline here.`

```
int ig = 0;
double dg = 0.;

inline void f1(int) {ig++;}
void f1(double){dg++;}

void main ()
{
int i;
double d;
f1(i);                 // f1(int) is marked for inlining
f1(d);

#pragma noinline here (void f1(int))
f1(i);                 // f1(int) is not marked for inlining
f1(d);
f1(i);                 // f1(int) is marked for inlining

printf(‘‘Result is %d\n’’, ig + (int) dg);
}
```

Example 2: Using the `here` keyword with the `#pragma inline` and `#pragma noinline` directives

This example illustrates the use of the `#pragma inline` and `#pragma noinline` directives with the `here` keyword. All occurrences of `f1(int)` are marked for inlining, except the one directly following `#pragma noinline here`. The only occurrence of `f1(double)` that is marked for inlining is the one directly following `#pragma inline here.`

```
int ig = 0;
double dg = 0.;

inline void f1(int) {ig++;}
void f1(double){dg++;}

void main ()
{
int i;
double d;

f1(i);                     // f1(int) is marked for inlining
f1(d);                     // f1(double) is not marked for inlining

#pragma noinline here (void f1(int))
f1(i);                     // f1(int) is not marked for inlining

#pragma inline here (void f1(double))
f1(d);                     // f1(double) is marked for inlining
f1(i);                     // f1(int) is marked for inlining

printf(‘‘Result is %d\n’’, ig + (int) dg);
}
```

Example 3: Using the `global` keyword with the `#pragma inline` directive

This example illustrates the use of the `#pragma inline` directive with the `global` keyword. All occurrences of `f1(int)` following the `#pragma inline global` are marked for inlining, except the one following the `#pragma noinline here`.

```
int ig = 0;
double dg = 0.;

void f1(int) {ig++;}
void f1(double){dg++;}

void main ()
{
#pragma inline global (void f1(int));
int i;
double d;
f1(i);                     // f1(int) is marked for inlining
f1(d);                     // f1(double) is not marked for inlining

#pragma noinline here (void f1(int))
f1(i);                     // f1(int) is not marked for inlining

#pragma inline here (void f1(double))
f1(d);                     // f1(double) is marked for inlining
f1(i);                     // f1(int) is marked for inlining

printf(‘‘Result is %d\n’’, ig + (int) dg);
}
```

Example 4: Using the `routine` keyword with the `#pragma inline` directive

This example illustrates the use of the `#pragma inline` directive with the `routine` keyword. All occurrences of `f1(int)` following `#pragma inline routine` are marked for inlining, except the one following `#pragma noinline here`.

```
int ig = 0;
double dg = 0.;

void f1(int) {ig++;}
void f1(double){dg++;}

void main ()
{
#pragma inline routine (void f1(int))
int i;
double d;
f1(i);                   // f1(int) is marked for inlining
f1(d);                   // f1(double) is not marked for inlining

#pragma noinline here (void f1(int))
f1(i);                   // f1(int) is not marked for inlining
#pragma inline here (void f1(double))
f1(d);                  // f1(double) is marked for inlining
f1(i);                  // f1(int) is marked for inlining

printf(‘‘Result is %d\n’’, ig + (int) dg);
}
```

Example 5: Using the `routine` keyword with the `#pragma noinline` directive

This example illustrates the use of the `#pragma noinline` directive with the `routine` keyword. None of the occurrences of `f1(int)` following `#pragma noinline routine` are marked for inlining, except the one following `#pragma inline here`.

```
int ig = 0;
double dg = 0.;

inline void f1(int) {ig++;}
void f1(double){dg++;}

void main ()
{
int i;
double d;

#pragma noinline routine (void f1(int))
f1(i);                         // f1(int) is not marked for inlining
f1(d);                         // f1(double) is not marked for inlining

#pragma inline here (void f1(int))
f1(i);                         // f1(int) is marked for inlining

#pragma noinline here (void f1(double))
f1(d);                         // f1(double) is not marked for inlining
f1(i);                         // f1(int) is not marked for inlining

printf(‘‘Result is %d\n’’, ig + (int) dg);
}
```

Loader Information #pragma Directives [7]

Table 7, page 39, lists 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 7. Loader Information `#pragma` Directives

#pragma	Short Description	Compiler Versions
`#pragma hidden`	Tells the compiler that the specified symbols are invisible to all executables or DSOs except the current one.	7.2 and later
`#pragma internal`	Tells the compiler that the specified symbols are not referenced outside the current executable or DSO.	7.2 and later
`#pragma no_delete`	Inhibits deletion of functions that are never referenced.	7.1 and later
`#pragma optional`	Tells the linker that the specified symbols are optional. This is the basic mechanism used for adding extensions to a library that can then be queried.	7.2.1 and later
`#pragma protected`	Tells the compiler that the specified symbols are not preemptible.	7.1 and later
`#pragma section_gp`	Causes an object to be placed in a gp_relative section.	7.2 and later
`#pragma section_non_gp`	Keeps an object from being placed in a gp_relative section.	7.2 and later

#pragma	Short Description	Compiler Versions
`#pragma weak`	Tells the link editor not to issue a warning if it does not find a defining declaration of the *weak_symbol*. Also allows the overriding of a current definition by a non-weak definition.	7.0 and later
`#pragma weak` *weak_symbol = strong_symbol*	Sets *weak_symbol* to be an alias for the function or data object denoted by *strong_symbol*, unless a defining declaration for *weak_symbol* is encountered at static link time. If encountered, the defining declaration preempts the weak denotation.	7.0 and later

7.1 `#pragma hidden`

The `#pragma hidden` directive tells the compiler that the specified symbols are invisible to all executables or DSOs except the current one. This allows hidden data objects to be placed in the small data area and accessed using the (fast) gp-relative load/store. Hidden symbols need not be put into the hash table of a DSO because they are not globally visible.

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

```
#pragma hidden symbol1 [, symbol2 ...]
```

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

All of the listed symbols are marked as `STO_HIDDEN.` This means that the symbol definition can be referenced only within an object, not from outside. Even though a hidden symbol cannot be directly referenced from outside a DSO, its address may be taken and passed, so it is possible to call a hidden function from another DSO.

7.2 #pragma internal

The #pragma internal directive tells the compiler that the specified functions are not referenced outside the current executable or DSO. Internal symbols are the same as hidden symbols, except that they are guaranteed not to be referenced from outside a DSO, even through pointers or weak bindings.

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

#pragma internal *func1* [, *func2* ...]

#pragma internal is not currently supported in C++, except for symbols marked extern ''C''.

The specified functions are marked STO_INTERNAL. This means that this function need not save, restore, or recalculate $gp (global pointer), because it is callable only from a location that has the same $gp (global pointer) value.

7.3 #pragma no_delete

The #pragma no_delete directive inhibits deletion of functions that are never referenced.

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

```
#pragma no_delete
```

7.4 #pragma optional

The #pragma optional directive tells the linker that the specified symbols are optional.

The static linker (ld), converts references to optional definitions (in another DSO) to optional references. Unresolved optional references are not reported as errors.

The run-time linker (rld) resolves any unresolved optional references to a special symbol in libc.so.1.

Programs can check for the existence of an optional symbol by use of macros defined in the header file /usr/include/optional_sym.h.

This is the basic mechanism used for adding extensions to a library that you can then query. For example, when new functions are added to the next revision of `libfoo.so`, they can be added as optional functions; then programs can check for their existence and use them only when the new revision of the library is available and avoid them on older systems, thus giving backwards and forwards compatibility across a series of releases.

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

`#pragma optional` *symbol1* `[,` *symbol2* `... ]`

The following rules apply to `#pragma optional`:

- `#pragma optional` must come after the declaration or definition of *symbol*.
- `#pragma optional` is not currently supported in C++, except for symbols marked `extern ‘‘C’’`.

7.5 `#pragma protected`

The `#pragma protected` directive tells the compiler that the specified symbols are not preemptible, but are visible from outside of a DSO.

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

`#pragma protected` *symbol1* `[,` *symbol2* `...]`

`#pragma protected` is not currently supported in C++, except for symbols marked `extern ‘‘C’’`.

The specified symbols are marked `STO_PROTECTED`. This means that the symbol definition cannot be preempted by another definition.

7.6 `#pragma section_gp`

MIPS binaries have a global pointer (gp) that can be used to reference global data more efficiently (by using gp + *offset*) than constructing the entire address when that variable is referenced. Only a limited set of elements can be referenced in this fashion because the size of *offset* is limited to 16 bits. The compiler heuristically places global data in either gp-relative or non-gp-relative sections. However, it is sometimes useful to manually control which variables go within the gp-relative section and which need to be addressed explicitly.

The `#pragma section_gp` directive causes an object to be placed in a gp_relative section, while the `#pragma section_non_gp` directive causes an object to be placed in a non-gp-relative section.

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

`#pragma section_gp (`*symbol1*`[,` *symbol2* `...])`

symbol must be a static or global variable.

7.7 #pragma section_non_gp

MIPS binaries have a global pointer (gp) that can be used to reference global data more efficiently (by using gp + *offset*) than constructing the entire address when that variable is referenced. Only a limited set of elements can be referenced in this fashion because the size of *offset* is limited to 16 bits. The compiler heuristically places global data in either gp-relative or non-gp-relative sections. However, it is sometimes useful to manually control which variables go within the gp-relative section and which need to be addressed explicitly.

The `#pragma section_gp` directive causes an object to be placed in a gp_relative section, while the `#pragma section_non_gp` directive causes an object to be placed in a non-gp-relative section.

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

`#pragma section_non_gp (`*symbol1*`[,` *symbol2* `...])`

symbol must be a static or global variable.

7.8 #pragma weak

The `#pragma weak` directive can be used in two ways. It can instruct the link editor to not issue a warning if it does not find a defining declaration of the specified weak symbol, or it can allow the overriding of a current definition by a non-weak definition.

Weak definitions behave as follows:

- A definition is weak if a symbol defined in an executable or DSO is marked as weak at the point of definition.
- A weak definition is preemptible and will be preempted by any strong global definition of the same name in the executable, the DSOs linked in at static link time, or the DSOs linked in at run time. Multiple weak definitions follow the same preemption rules as for global symbols except that they will all be preempted by any strong definition of their name.

- Multiple global weak definitions of a symbol may or may not result in an error:
  - At static link time, multiple global definitions of a weak symbol within a DSO or executable result in an error. For example, linking `a.o` and `b.o` when they both have definitions for the symbol *x* results in an error.
  - At run time, multiple global weak definitions of a symbol across the executable and its DSOs, result in the first definition preempting all others. No error message is generated. For example, if your executable, `j`, references the DSOs `k.so` and `l.so` that have weak definitions of the symbol *y*, the first definition encountered is used, and the other is ignored.
- Unresolved weak references do not cause a run-time error, even if the environment variable `LD_BIND_NOW` is set. They have a value of 0 (that is, the symbol address is taken as 0). Attempting a call of a weak undefined function symbol gets either a core dump (if `LD_BIND_NOW` is 1) or a fatal run-time linker error on an attempted address of an unresolved symbol (if `LD_BIND_NOW` is not 1). Attempting a load or store of an undefined weak symbol results in a core dump because the address is 0, and 0 is normally not a legal virtual address.
- Weak references do not trigger the loading of delay-loaded libraries. This implies that weak object references may go unresolved until some other event triggers the loading of the delay-load library.

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

`#pragma weak` *weak_symbol* [= *strong_symbol*]

When `#pragma weak` applies to a C++ function, *weak_symbol* and *strong_symbol* must be the mangled names.

The `#pragma weak` directive can be used in the following two ways:

- `#pragma weak`*weak_symbol*

  Used in this way, the `#pragma weak` directive tells the link editor to not issue a warning if it does not find a defining declaration of *weak_symbol*. References to the symbol use the appropriate lvalue if the symbol is defined; otherwise, it uses memory location zero (0).

- `#pragma weak` *weak_symbol* = *strong_symbol*

  In this case, the *weak_symbol* is an alias that denotes the same function or data object as that denoted by the *strong_symbol*, unless a defining declaration for the *weak_symbol* is encountered at static link time or in dynamically linked libraries. If encountered, the defining declaration preempts the weak denotation.

Observe the following conventions when using this form of the directive:

– Define the *strong_symbol* within the same compilation unit in which the directive occurs.

– Declare the weak and strong symbols with compatible types. When the strong symbol is a data object, its declaration must be initialized.

– Declare the *weak_symbol* with extern linkage in the same compilation unit. The extern declaration of the weak symbol is not required, unless the symbol is referenced within the compilation unit, but Silicon Graphics recommends it for type-checking purposes.

  Weak `extern` declarations are typically used to export non-ANSI C symbols from a library without polluting the ANSI C name-space. As an example, `libc` may export a weak symbol `read()`, which aliases a strong symbol `_read()`, where `_read()` is used in the implementation of the exported symbol `fread()`. You can either use the exported (weak) version of `read()`, or define your own version of `read()`, thereby preempting the weak denotation of this symbol. This will not alter the definition of `fread()`, because it depends only on the (strong) symbol `_read()`, which is outside of the ANSI C name-space.

  For example, the following code defines a new version of `read()` (which is a weak symbol in `libc.so.1`):

```
/* read() is a weak symbol in libc.so.1
     This program omits error checking and makes no
     attempt at good style!
*/
#include <stdio.h>
char *read(int);

int main(int argc, char **argv)
{
char *var;
int c;

c = getchar();

var = read(c);
printf(‘‘%s\n’’,var);
return c;
}

char *read(int val)
{
static char buf[100];
sprintf(buf,’’%d’’,val);
return buf;
}
```

This program reads a single character from standard input and prints the character’s decimal value. Even though `getchar()` uses the `libc.so` version of `fread()`, the redefinition of `read()` has no effect on the internal processing in `libc.so` because `fread()` uses the strong symbol `_read()`.

Caution: The `#pragma weak` directive is not supported in `-o32` C++.

Loop Nest Optimization `#pragma` Directives [8]

Table 8, page 47, 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 8. Loop Nest Optimization `#pragma` Directives

#pragma	Short Description	Compiler Versions
`#pragma aggressive inner loopfission`	Tells the compiler to fission inner loops into as many loops as possible.	7.0 and later
`#pragma blocking size`	Sets the blocksize of the specified loop, if it is involved in a blocking for the primary (or secondary) cache.	7.0 and later
`#pragma fission`	Tells the compiler to fission the enclosing specified levels of loops after this directive.	7.0 and later
`#pragma fissionable`	Disables validity testing.	7.0 and later
`#pragma fusable`	Disables validity testing.	7.0 and later
`#pragma fuse`	Tells the compiler to fuse the following *n* loops, which must be immediately adjacent.	7.0 and later
`#pragma ivdep`	Liberalizes dependence analysis. This applies only to inner loops. Given two memory references, where at least one is loop variant, ignore any loop-carried dependences between the two references.	6.0 and later
`#pragma no blocking`	Prevents the compiler from involving this loop in cache blocking.	7.0 and later
`#pragma no fission`	Keeps the following loop from being fissioned. Its innermost loops, however, are allowed to be fissioned.	7.0 and later
`#pragma no fusion`	Keeps the following loop from being fused with other loops.	7.0 and later

#pragma	Short Description	Compiler Versions
`#pragma no interchange`	Prevents the compiler from involving the loop directly following this directive (or any loop nested within this loop) in an interchange.	7.0 and later
`#pragma prefetch`	Specifies prefetching for each level of the cache. Scope: entire function containing the directive.	7.1 and later
`#pragma prefetch_manua`	Specifies whether manual prefetches (through `#pragma` directives) should be respected or ignored. Scope: entire function containing the directive.	7.1 and later
`#pragma prefetch_ref`	Generates a prefetch and connects it to the specified reference (if possible).	7.0 and later
`#pragma prefetch_ref_disable`	Disables prefetching for the specified reference in the current loop nest.	7.1 and later
`#pragma unroll`	Suggests to the compiler that a specified number of copies of the loop body be added to the inner loop. If the loop following this directive is an inner loop, then it indicates standard unrolling (version 7.2 and later). If the loop following this directive is not innermost, then outer loop unrolling (unroll and jam) is performed (version 7.0 and later).	7.0 and later

8.1 #pragma aggressive inner loop fission

The `#pragma aggressive inner loop fission` directive instructs the compiler to fission inner loops into as many loops as possible.

The syntax of the `#pragma aggressive inner loop fission` directive is as follows:

```
#pragma aggressive inner loop fission
```

The `#pragma aggressive inner loop fission` directive must be followed by an inner loop and has no effect if that loop is no longer inner after loop interchange.

8.2 #pragma blocking size

The #pragma blocking size directive sets the blocksize of the specified loop.

The syntax of the #pragma blocking size directive is as follows:

```
#pragma blocking size (n1, n2)
```

The loop specified, if it is involved in a blocking for the primary (secondary) cache, will have a blocksize of *n1* (*n2*). The compiler tries to include this loop within such a block. If a 0 blocking size is specified, then the loop is not stripped, but the entire loop is inside the block.

8.2.1 Example of #pragma blocking size

In the following code, the compiler makes 20 × 20 blocks when blocking:

```
void amat (double x, double y, double z, int n, int m, int mm)
{
int i, j, k;

for (k = 0; k < n; k++)
{
#pragma blocking size (20)
for (j = 0; j < m; j++)
{
#pragma blocking size (20)
for (i = 0; i < mm; i++)
z(i,k) = z(i,k) + x(i,j) * y(j,k)
}
}
}
```

8.3 #pragma no blocking

The #pragma no blocking directive prevents the compiler from involving this loop in cache blocking.

The syntax of the #pragma no blocking directive is as follows:

```
#pragma no blocking
```

8.4 #pragma fission

The `#pragma fission` directive instructs the compiler to fission the enclosing *n* levels of loops after this directive.

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

`#pragma fission [(`*n*`)]`

The default for *n* is 1. The compiler performs a validity test unless `#pragma fissionable` is also specified. The compiler does not reorder statements.

8.5 #pragma fissionable

The `#pragma fissionable` directive disables validity testing for loop fissioning.

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

```
#pragma fissionable
```

8.6 #pragma no fission

The `#pragma no fission` instructs the compiler to not fission the loop directly following this directive. Any inner loops, however, are allowed to be fissioned.

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

```
#pragma no fission
```

8.7 #pragma fuse

The `#pragma fuse` directive instructs the compiler to fuse the specified number of immediately adjacent loops.

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

`#pragma fuse [(`*num*`,` *level*`)]`

The loops to be fused must immediately follow the `#pragma fusion` directive.

The default value for *num* is 2. Fusion is attempted on each pair of adjacent loops and the level, by default, is determined by the maximal perfectly nested loop levels of the fused loops, although partial fusion is allowed. Iterations may be peeled as needed during fusion; the limit of this peeling is 5 or the number specified by the `-LNO:fusion_peeling_limit` option. No fusion is done for non-adjacent outer loops.

When the `#pragma fusable` directive is present, no validity test is done and the fusion is done up to the maximal common levels.

8.8 `#pragma fusable`

The `#pragma fusable` directive disables validity testing for loop fusing.

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

```
#pragma fusable
```

8.9 `#pragma no fusion`

The `#pragma no fusion` directive instructs the compiler that the loop following this directive should not be fused with other loops.

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

```
#pragma no fusion
```

8.10 `#pragma no interchange`

The `#pragma no interchange` directive prevents the compiler from involving the next loop in an interchange. This directive also applies to any loop nested within the indicated loop.

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

```
#pragma no interchange
```

The pragma directive statement must immediately precede the loop to which it applies.

8.11 `#pragma ivdep`

The `#pragma ivdep` directive instructs the compiler to liberalize dependence analysis.

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

```
#pragma ivdep
```

Given two memory references, where at least one is loop variant, this directive instructs the compiler to ignore any loop-carried dependences between the two references. The `#pragma ivdep` directive applies only to inner loops. If `#pragma ivdep` is used on a loop that has an inner loop, the compiler ignores it.

8.11.1 Examples of `#pragma ivdep`

The following are some examples of the use of `#pragma ivdep`:

- `ivdep` does not break the dependence because `b(k)` is not loop variant:

  ```
  #pragma ivdep
  for (i = 0; i < n; i++)
  b(k) = b(k) +a(i);
  ```

- `ivdep` breaks the dependence, but the compiler warns the user that it is breaking an obvious dependence:

  ```
  #pragma ivdep
  for (i = 0; i < n; i++)
  a(i) = a(i-1) + 3.0;
  ```

- `ivdep` breaks the dependence:

  ```
  #pragma ivdep
  for (i = 0; i < n; i++)
  a(b(i)) = a(b(i)) + 3.0;
  ```

- ivdep does not break the dependence on a(i) because it is within an iteration:

  ```
  #pragma ivdep
  for (i = 0; i < n; i++)
  {
  a(i) = b(i);
  c(i) = a(i) + 3.0;
  }
  ```

If -OPT:cray_ivdep=TRUE is specified, ivdep instructs the compiler to use Cray semantics and break all backward dependences:

- ivdep breaks the dependence but the compiler warns the user that it is breaking an obvious dependence:

  ```
  #pragma ivdep
  for (i = 0; i < n; i++)
  {
  a(i) = a(i - 1) + 3.0;
  }
  ```

- ivdep does not break the dependence, because the it is from the load to the store, and the load comes lexically before the store:

  ```
  #pragma ivdep
  for (i = 0; i < n; i++)
  {
  a(i) = a(i + 1) + 3.0;
  }
  ```

To break all dependences, specify the following: -OPT:liberal_ivdep=TRUE.

8.12 #pragma prefetch

The #pragma prefetch directive specifies prefetching for each level of the cache.

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

#pragma prefetch [(*n1*, *n2*)]

n1 controls the level 1 cache; *n2* controls level 2. *n1* and *n2* can have the following values:

Value	Effect
0	Prefetching is off (default for all processors except R10000)
1	Prefetching is on but conservative (default at `-O3` when prefetch is on)
2	Prefetching on and aggressive

The scope of this directive is the entire function that contains it.

8.13 `#pragma prefetch_manual`

The `#pragma prefetch_manual` directive instructs the compiler as to whether manual prefetches (through `#pragma` directives) should be respected or ignored.

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

```
#pragma prefetch_manual[(n)]
```

Value	Effect
0	Compiler ignores manual prefetches (default for all processors except R10000)
1	Compiler respects manual prefetches (default at `-O3` for R10000 and beyond)

The scope of this directive is the entire function that contains it.

8.14 `#pragma prefetch_ref`

The `#pragma prefetch_ref` directive generates a prefetch and connects it to the specified reference (if possible).

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

```
pragma prefetch_ref = ref [, stride = num1 [, num2]]
[, level = [lev1][, lev2]]
[, kind = {rd|wr}]
[, size = sz]
```

ref is the object you want prefetched.

Table 9, page 55 describes each of the possible `#pragma prefetch_ref` clauses. These clauses are optional.

Table 9. Clauses for `#pragma prefetch_ref`

Clause	Effect	Default Value
`stride`	Prefetches every *num* iteration(s) of this loop.	1
`level`	Specifies the level in memory hierarchy to prefetch. The possible values for `level` are 1: prefetch from L2 to L1 cache 2: prefetch from memory to L1 cache	2
`kind`	Specifies **read** or **write**.	**write**
`size`	Specifies the size (in KB) of the object referenced in this loop. Must be a constant.	N/A

The `#pragma prefetch_ref` directive instructs the compiler to take the following actions:

- Generate a prefetch and connect to the specified object (if possible).
- Search for references in the current loop-nest that match the supplied object.
  - If such a reference is found, then the prefetch for that object is scheduled relative to the prefetch node, based on the miss latency for the specified level of the cache.
  - If no such reference is found, the prefetch is generated at the start of the loop body.
- Ignore all references by the automatic prefetcher (if enabled) to this variable in this loop-nest.
- Have the automatic prefetcher (if enabled) use the supplied size (if specified) in its volume analysis for this object.

This directive has no scope; it just generates a prefetch.

8.15 `#pragma prefetch_ref_disable`

The `#pragma prefetch_ref_disable` directive explicitly disables prefetching for the specified reference (in the current loop nest).

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

`#pragma prefetch_ref_disable =` *ref* `[, size =` *num*`]`

ref is the object for which you want to disable prefetching.

num specifies the size (in KB) of the object referenced in this loop (optional). The size must be a constant. This explicitly disables the prefetching of all references to object *ref* in the current loop nest. If enabled, the auto-prefetcher runs but ignores *ref*. The size is used for volume analysis.

The scope of this directive is the entire function containing it.

8.16 `#pragma unroll`

The `#pragma unroll` directive suggests to the compiler the type of unrolling that should be done.

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

`#pragma unroll (`*n*`)`

This directive instructs the compiler to add *n*-1 copies of the loop body to the inner loop. If the loop that this directive immediately precedes is an inner loop, then it indicates standard unrolling (version 7.2 and later). If the loop that this directive immediately precedes is not innermost, then outer loop unrolling (unroll and jam) is performed (version 7.0 and later).

The value of *n* must be at least 1. If it is 1, then unrolling is not performed.

Caution: The `#pragma unroll` directive works only on loops that are legal to unroll. Loops are often not unrollable in C because of potential aliasing. In these cases, you may want to use restrict pointers or the option `-OPT:alias=disjoint` (see the *C Language Reference Manual* for more information on restrict pointers). When `-OPT:alias=disjoint` is specified, distinct pointer expressions are assumed to point to distinct, non-overlapping objects.

`-OPT:alias=disjoint` is unsafe and may cause existing C programs to fail in obscure ways, so it should be used with extreme care.

8.16.1 Examples of #pragma unroll

The following code samples show the effect of using `#pragma unroll`. The code in Sample 1 becomes Sample 2, not Sample 3:

- Sample 1:

```
#pragma unroll (2)
for (i = 0; i < 10; i++)
{
for (j = 0; j < 10; j++)
{
a([i][j] = a[i][j] + b[i][j];
}
}
```

- Sample 2:

```
for (i = 0; i < 10; i + 2)
{
for (j = 0; j < 10; j++)
{
a[i][j] = a[i][j] + b[i][j];
a[i+1][j] = a[i+1][j] + b[i+1][j];
}
}
```

- Sample 3:

```
for (i = 0; i < 10; i + 2)
{
for (j = 0; j < 10; j++)
a[i][j] = a[i][j] + b[i][j];
for (j = 0; j < 10; j++)
a[i+1][j] = a[i+1][j] + b[i+1][j];
}
```

The `#pragma unroll` directive is attached to the given loop, so that if an interchange is performed, the corresponding loop is still unrolled. That is, Sample 1 is equivalent to the following:

```
#pragma interchange
for (j = 0; j < 10; j++)
{
#pragma unroll (2)
for (i = 0; i < 10; i++)
a[i][j] = a[i][j] + b[i][j];
}
```

Multiprocessing #pragma Directives [9]

Table 10, page 59, 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 later
`#pragma critical`	Protects access to critical statements.	6.0 and later
`#pragma enter gate` (see Section 9.3, page 63)	Indicates the point that all threads must clear before any threads are allowed to pass the corresponding `exit gate`.	6.0 and later
`#pragma exit gate` (see Section 9.3, page 63)	Stops threads from passing this point until all threads have cleared the corresponding `enter gate`.	6.0 and later
`#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 later
`#pragma local`	Tells the compiler the names of all the variables that must be local to each thread.	6.0 and later
`#pragma no side effects`	Tells the compiler to assume that all of the named functions are safe to execute concurrently.	7.1 and later
`#pragma one processor`	Causes the next statement to be executed on only one processor.	6.0 and later
`#pragma parallel` (see also Section 9.9, page 71)	Marks the start of a parallel region.	6.0 and later
`#pragma pfor` (see also Section 9.11, page 76)	Marks a `for` loop to run in parallel.	6.0 and later

#pragma	Short Description	Compiler Versions
`#pragma pure`	Tells the compiler to use only values of named functions to computer return value with no side effects.	7.3 and later
`#pragma set chunksize`	Tells the compiler which values to use for `chunksize`.	6.0 and later
`#pragma set numthreads`	Tells the compiler which values to use for `numthreads`.	6.0 and later
`#pragma set schedtype`	Tells the compiler which values to use for `schedtype`.	6.0 and later
`#pragma shared`	Tells the compiler the names of all the variables that the threads must share.	6.0 and later
`#pragma synchronize`	Stops threads until all threads reach this point.	6.0 and later

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, page 62, illustrates critical segment execution.

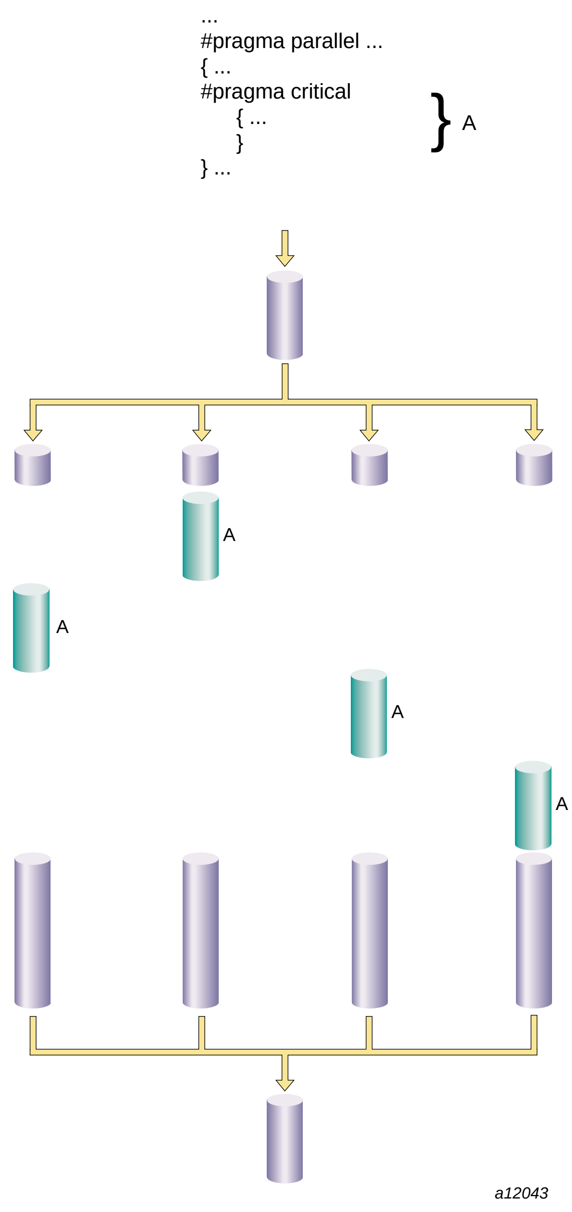

Figure 1. Critical Segment Execution

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, page 64, is a "time-lapse" sequence showing execution using enter and exit gates.

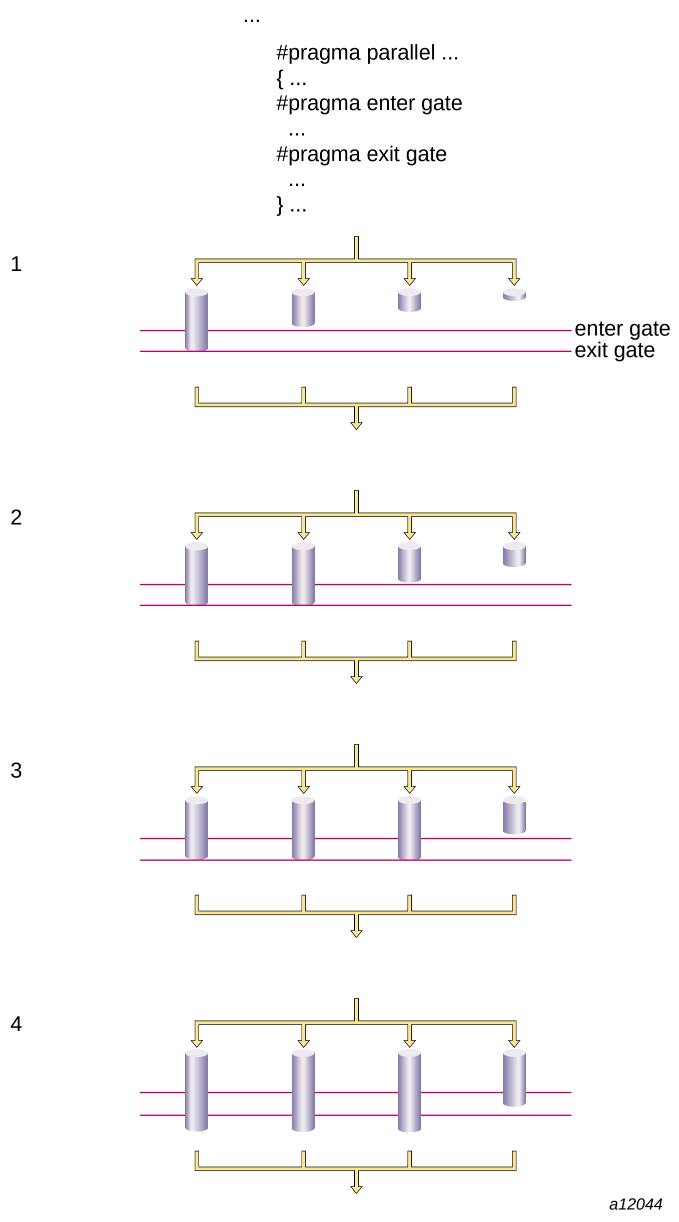

Figure 2. Execution Using Gates

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, page 67, shows an independent segment with execution by only one thread.

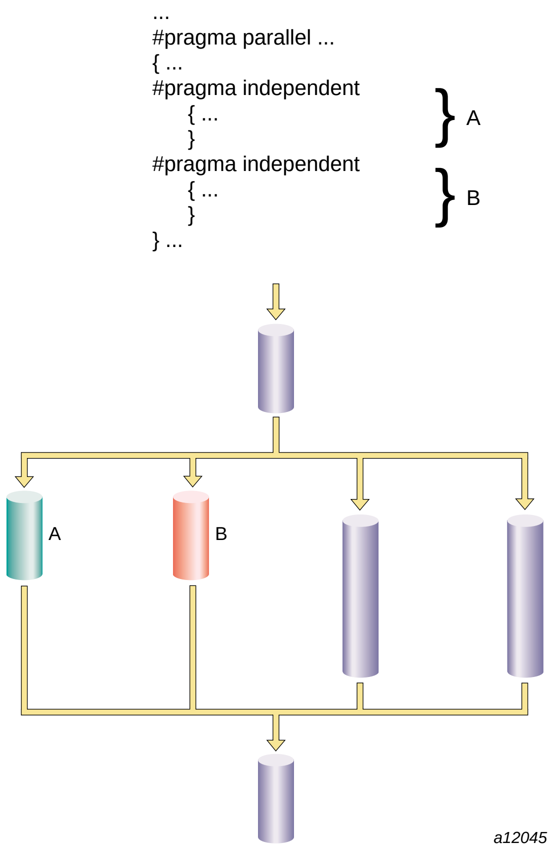

Figure 3. Independent Segment Execution

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, page 69, shows code executed by only one thread. No thread can proceed past this code until it has been executed.

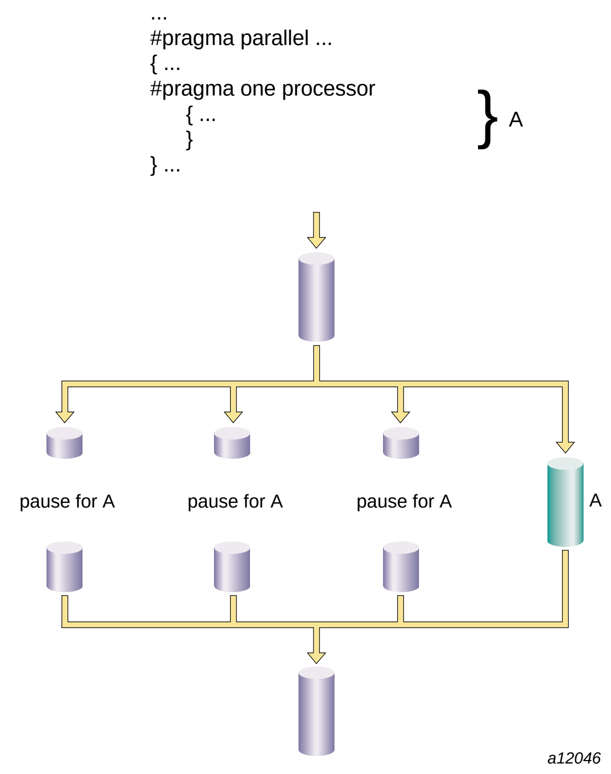

Figure 4. One Processor Segment

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, page 71). 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, page 70).

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, page 71 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, page 71, 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, page 75, shows parallel code segments using `#pragma pfor` running on four threads with simple scheduling.

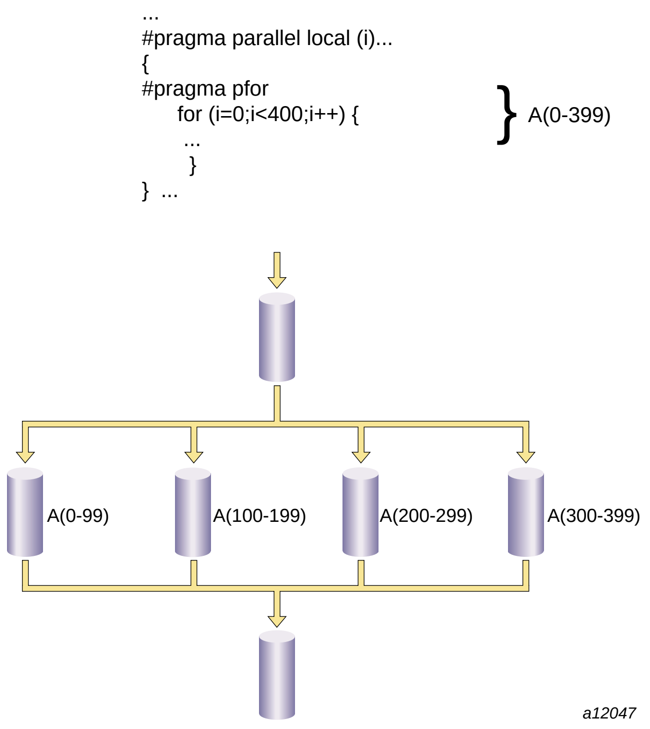

Figure 5. Parallel Code Segments Using #pragma pfor

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, page 77, 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, page 72).

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, page 29).

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, page 23 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, page 23, 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, page 28, 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, page 81. 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, page 84).

You can use the types in Table 12, page 82, 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. *chunksize* should be smaller than the number of total iterations divided by the number of threads. The advantage of dynamic over simple is that dynamic helps distribute the work more evenly than simple.
`interleave`	Tells the run-time scheduler to give each thread chunksize iterations of the loop, which are then assigned to the threads in an interleaved way.

Type	Function
`gss`(guided self-scheduling)	Tells the run-time scheduler to give each processor a varied number of iterations of the loop. This is like `dynamic`, but instead of a fixed *chunksize*, the *chunksize* iterations begin 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, page 83, shows how the iteration chunks are apportioned over the various processors by the different types of loop scheduling.

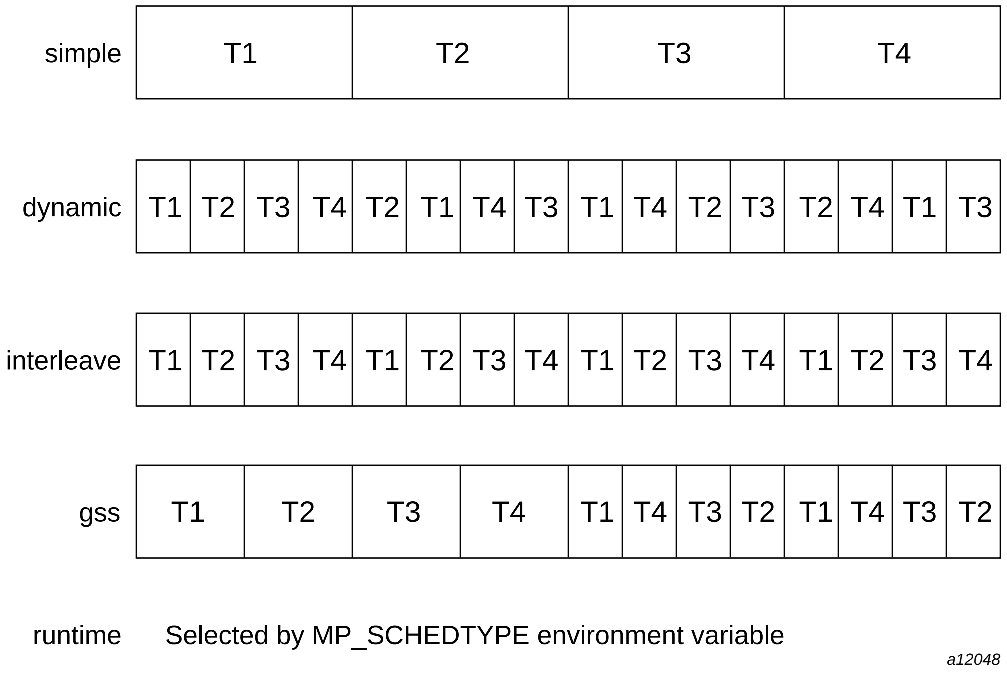
simple
T1
T2
T3
T4
dynamic
T1 T2 T3 T4 T2 T1 T4 T3 T1 T4 T2 T3 T2 T4 T1 T3
interleave
T1 T2 T3 T4 T1 T2 T3 T4 T1 T2 T3 T4 T1 T2 T3 T4
gss
T1 T2 T3 T4 T1 T4 T3 T2 T1 T4 T3 T2
runtime
Selected by MP_SCHEDTYPE environment variable
a12048

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, page 84, 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, page 82).

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, page 86, and Section 9.11, page 76, 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, page 76, 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, page 82, 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, page 71).

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, page 88, is a time-lapse sequence showing the synchronization of all threads.

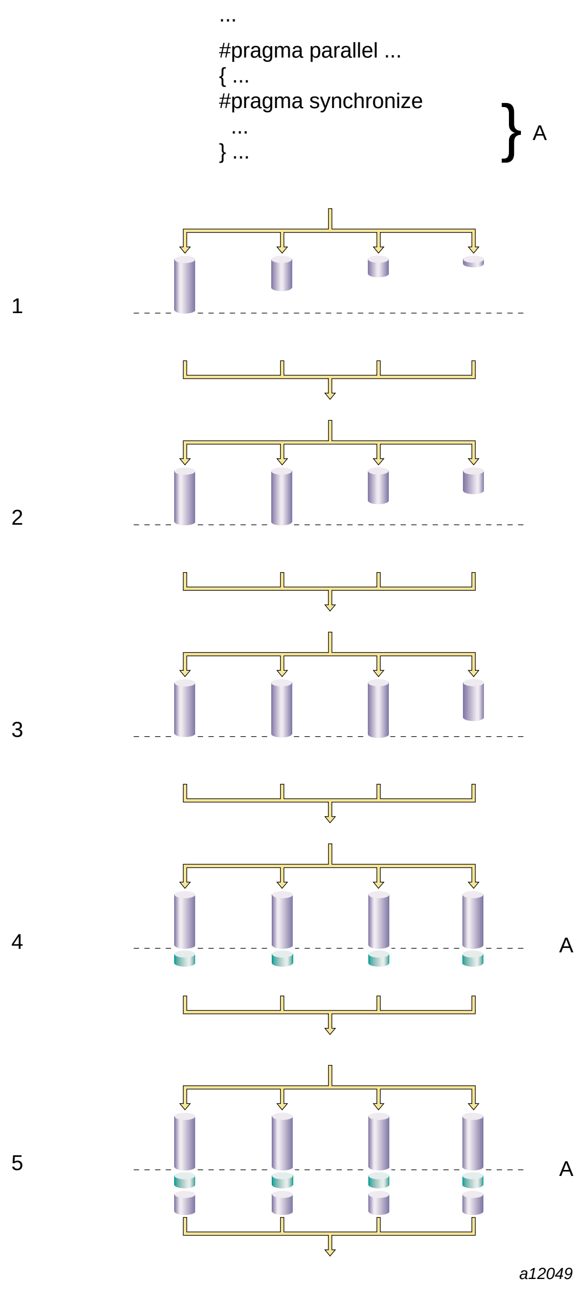

Figure 7. Synchronization

OpenMP C/C++ API Multiprocessing Directives [10]

This appendix discusses the multiprocessing directives that MIPSpro C and C++ compilers support. 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.

In addition to directives, the OpenMP C/C++ API describes several library functions and environment variables. Information on the library functions can be found on the `omp_lock`(3), `omp_nested`(3), and `omp_threads`(3) man pages. Information on the environment variables can be found on the `pe_environ`(5) man page.

This chapter contains the following sections:

- Section 10.1, page 90, describes using directives and the directive format.
- Section 10.2, page 90, describes conditional compilation.
- Section 10.3, page 91, describes the parallel region construct.
- Section 10.4, page 92, describes work-sharing constructs.
- Section 10.5, page 98, describes the combined parallel work-sharing constructs.
- Section 10.6, page 100, describes the synchronization constructs.
- Section 10.7, page 107, describes the data environment, which includes directives and clauses that affect the data environment.
- Section 10.8, page 116, describes directive binding.
- Section 10.9, page 117, describes directive nesting.

Note: The Silicon Graphics multiprocessing directives, including the Origin series distributed shared memory directives, are outmoded. Their preferred alternatives are the OpenMP C/C++ API directives described in this chapter.

10.1 Using Directives

Each OpenMP directive starts with #pragma omp, to reduce the potential for conflict with other #pragma directives with the same name. They have the following form:

```
#pragma omp directive-name [clause[ clause] ...] new-line
```

Except for starting with #pragma omp, the directive follows the conventions of the C and C++ standards for compiler directives.

Directives are case-sensitive. The order in which clauses appear in directives is not significant. Only one directive name can be specified per directive.

An OpenMP directive applies to at most one succeeding statement, which must be a structured block.

10.2 Conditional Compilation

The _OPENMP macro name is defined by OpenMP-compliant implementations as the decimal constant, *yyyymm*, which will be the year and month of the approved specification. This macro must not be the subject of a #define or a #undef preprocessing directive.

```
#ifdef _OPENMP
iam = omp_get_thread_num() + index;
#endif
```

If vendors define extensions to OpenMP, they may specify additional predefined macros.

If an implementation is not OpenMP-compliant, or if its OpenMP mode is disabled, it may ignore the OpenMP directives in a program. In effect, an OpenMP directive behaves as if it were enclosed within #ifdef _OPENMP and #endif. Thus, the following two examples are equivalent:

```
if(cond)
{
   #pragma omp flush (x)
}
X++;
```

```
if(cond)
   #ifdef )OPENMP
   #pragma omp flush (x)
   #endif
x++;
```

10.3 `parallel` Construct

The `#pragma omp parallel` directive defines a parallel region, which is a region of the program that is to be executed by multiple threads in parallel.

The `#pragma omp parallel` directive has the following syntax:

`#pragma omp parallel [`*clause*`[` *clause*`] ...]` *new-line structured-block*

clause is one of the following:

`if` (*scalar-expression*)

`private` (*list*)

`firstprivate` (*list*)

`default` (`shared | none`)

`shared` (*list*)

`copyin` (*list*)

`reduction` (*operator*: *list*)

For information on these data scope attribute clauses, see Section 10.7.2, page 109.

When a thread encounters a parallel construct and no if clause is present, or the if expression evaluates to a nonzero value, a team of threads is created. This thread becomes the master thread with a thread number of 0. The number of threads is controlled by environment variables and library calls. If the value of the if expression is zero, the region is serialized.

The number of threads remains constant while that parallel region is being executed. It can be changed either explicitly by the user or automatically by the runtime system from one parallel region to another. The `omp_set_dynamic`(3) library function and the `OMP_DYNAMIC` environment variable can be used to enable and disable the automatic adjustment of the number of threads. For more information on environment variables, see the `pe_environ`(5) man page.

If a thread in a team executing a parallel region encounters another parallel construct, it creates a new team, and it becomes the master of that new team. Nested parallel regions are seialized by default. By default, a nested parallel gregion is executed by a team composed of one threads. The default behavior can be changed by using either the `omp_set_nested` runtime library function or the `OMP_NESTED` environment variable.

The following restrictions apply to the `#pragma omp parallel` directive:

- Only one `if` clause can appear on the directive.
- It is unspecified whether any side-effects inside the `if` expression occur.
- A `throw` executed inside a parallel region must cause execution to resume within the dynamic extent of the same structured block, and it must be caught by the same thread that threw the exception.

The `parallel` directive can be used in coarse-grain parallel programs. In the following example, each thread in the parallel region decides what part of the global array `x` to work on, based on the thread number.

```
#pragma omp parallel shared(x, npoints) private(iam, np, ipoints)
{
   iam = omp_get_thread_num();
   np = omp_get_num_threads();
   ipoints = npoints / np;
   subdomain(x, iam, ipoints);
}
```

10.4 Work–sharing Constructs

A work-sharing construct distributes the execution of the associated statement among the members of the team that encounter it. The work-sharing directives do not launch new threads, and there is no implied barrier on entry to a work-sharing construct.

The sequence of work-sharing constructs and barrier directives encountered must be the same for every thread in a team.

OpenMP defines the following work-sharing constructs:

- `for` directive
- `sections` directive
- `single` directive

10.4.1 `for` Construct

The `#pragma omp for` directive identifies an iterative work-sharing construct that specifies a region in which the iterations of the associated loop should be executed in parallel. The iterations of the `for` loop are distributed across threads that already exist. The `#pragma omp for` directive has the following syntax:

```
#pragma omp for [clause[ clause] ... ] new-line for-loop
```

clause is one of the following:

`private` (*list*)

`firstprivate` (*list*)

`lastprivate` (*list*)

`reduction` (*operator*: *list*)

`ordered`

`schedule` (*kind*[, *chunk_size*])

`nowait`

For information on the `private`, `firstprivate`, `lastprivate`, and `reduction` clauses, see Section 10.7.2, page 109.

The `#pragma omp for` directive places restrictions on the structure of the corresponding `for` loop, which must have the following canonical shape:

```
for (init-expr; var logical-op b; incr-expr)
```

init-expr One of the following:

```
var = lb
integer-type var = lb
```

incr-expr One of the following:

```
++var
var++
--var
var--
var += incr
var -= incr
var = var + incr
var = incr + var
var = var - incr
```

`var`	One of the following:
	`<`
	`<=`
	`>`
	`>=`

`lb`, `b`, and `incr`	Loop invariant integer expressions. There is no synchronization during the evaluation of these expressions. Thus, any evaluated side effects produce indeterminate results.

The `schedule` clause specifies how iterations of the `for` loop are divided among threads of the team. The value of *chunk_size*, if specified, must be a log invariant integer expression with a positive value. Synchronization does not occur during the evaluation of this expression, therefore, any evaluated side effects produce indeterminate results. The schedule kind can be one of the following:

`static`	When `schedule(static,`*chunk_size*`)` is specified, iterations are divided into chunks specified by *chunk_size*. The chunks are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. When *chunk_size* is not specified, the iteration space is divided into chunks that are equal in size, with one chunk assigned to each thread.
`dynamic`	When `schedule(dynamic,`*chunk_size*`)` is specified, *chunk_size* iterations are assigned to each thread. When a thread finishes its chunk, it is dynamically assigned another until none remain. Default is 1.
`guided`	When `schedule(guided,`*chunk_size*`)` is specified, iterations are assigned to threads by decreasing sizes. When a thread finishes, it is dynamically assigned another chunk until none remain. Sizes decrease exponentially to 1. Default is 1.

`runtime`	When `schedule(runtime)` is specified, scheduling is deferred until runtime. Schedule kind and dize of chunks can be chosen by setting the `OMP_SCHEDULE` environment variable . If not set, the schedule is implementation-dependent.

The default schedule is implementation-dependent.

An OpenMP-compliant program should not rely on a particular schedule for correct execution. It is possible to have variations in the implementations of the same schedule kind across different compilers.

The `ordered` clause must be present when `ordered` directives are contained in the dynamic extent of the `for` construct.

There is an implicit barrier at the end of a `for` construct unless a `nowait` clause is specified.

The following restrictions apply to the `#pragma omp for` directive:

- The `for` loop iteration variable must have a `signed` integer type.
- The values of the loop control expressions of the `for` loop associated with a `for` directive must be the same for all the threads in the team.
- The `for` loop iteration variable must have a `signed` integer type.
- Only one `schedule` clause can appear on a `for` directive.
- Only one `ordered` clause can appear on a `for` directive.
- Only one `nowait` clause can appear on a `for` directive.
- It is unspecified if or how often any side effects within the *chunk_size*, `lb`, `b`, or `incr` expressions occur.
- The value of the *chunk_size* expression must be the same for all threads in the team.

If there are multiple independent loops within a parallel region, you can use the `nowait` clause to avoid the implied `barrier` at the end of the `for` directive, as follows:

```
#pragma omp parallel
{
   #pragma omp for nowait
      for (i=1; i<n; i++)
         b[i] = (a[i] + a[i-1]) / 2.0;
```

10.4.2 `sections` Construct

The `#pragma omp sections` directive identifies a non-iterative work-sharing construct that specifies a set of constructs that are to be divided among threads in a team. Each section is executed once by a thread in the team. Each section is preceded by a `sections` directive, although the `sections` directive is optional for the first section.

The `#pragma omp sections` directive has the following syntax:

```
#pragma omp sections [clause[ clause] ...] new-line
   {
   [#pragma omp section new-line] structured-block
   [#pragma omp section new-line structured-block
   .
   .
   .]
   }
```

clause is one of the following:

`firstprivate(`*list*`)`

`lastprivate(`*list*`)`

`reduction(`*operator: list*`)`

`nowait`

For information on `private`, `firstprivate`, `lastprivate`, and `reduction`, see Section 10.7.2, page 109.

There is an implicit barrier at the end of a `sections` construct, unless a `nowait` is specified.

The following restrictions apply to the `sections` construct:

- A `section` directive must not be outside the lexical extent of the `sections` directive.

- Only one `nowait` clause can appear on a `sections` directive.

10.4.3 `single` Construct

The `#pragma omp single` directive identifies a construct that specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread). The `#pragma omp single` directive has the following syntax:

```
#pragma omp single [clause[ clause] ...] new-line
  structured-block
```

clause is one of the following:

`private(`*list*`)`

`firstprivate(`*list*`)`

`nowait`

For information on the `private` and `firstprivate` clauses, see Section 10.7.2, page 109.

There is an implicit barrier after the `single` construct unless a `nowait` clause is specified.

The following restrictions apply to the `#pragma omp single` directive:

- Only one `nowait` clause can appear on a `single` directive.

In the following example, only one thread (usually the first thread that encounters the `single` directive) prints the progress message. The user must not make any assumptions as to which thread will execute the `single` section. All other threads will skip the `single` section and stop at the barrier at the end of the `single` construct. If other threads can proceed without waiting for the thread executing the `single` section, a `nowait` clause can be specified on the `single` directive.

```
#pragma omp parallel
{
  #pragma omp single
    printf("Beginning work1.\n");
  work1();
  #pragma omp single
    printf("Finishing work1.\n");
  #pragma omp single nowait
    printf("Finished work1 and beginning work2.\n");
  work2();
}
```

10.5 Combined Parallel Work-sharing Constructs

Combined parallel work-sharing constructs are short cuts for specifying a parallel region that contains only one work-sharing construct. The semantics of these directives are identical to that of explicitly specifying a `parallel` directive followed by a single work-sharing construct.

10.5.1 `parallel for` Construct

The `parallel for` directive is a shortcut for a `parallel` region that contains one `for` directive. It has the following syntax:

```
#pragma omp parallel for [clause[ clause] ...] new-line
  for-loop
```

clause is one of the following:

`if` (*scalar-expression*)

`private` (*list*)

`firstprivate` (*list*)

`lastprivate` (*list*)

`default` (`shared | none`)

`shared` (*list*)

`copyin` (*list*)

`reduction` (*operator*: *list*)

`ordered`

`schedule` (*kind*[, *chunk_size*])

`nowait`

The following restrictions apply to the `parallel for` directive:

- Only one `if` clause can appear on the directive.
- It is unspecified whether any side-effects inside the `if` expression occur.
- A `throw` executed inside a parallel region must cause execution to resume within the dynamic extent of the same structured block, and it must be caught by the same thread that threw the exception.
- The `for` loop iteration variable must have a `signed` integer type.
- The values of the loop control expressions of the `for` loop associated with a `for` directive must be the same for all the threads in the team.
- The `for` loop iteration variable must have a `signed` integer type.
- Only one `schedule` clause can appear on a `for` directive.
- Only one `ordered` clause can appear on a `for` directive.
- Only one `nowait` clause can appear on a `for` directive.
- It is unspecified if or how often any side effects within the *chunk_size*, `lb`, `b`, or `incr` expressions occur.
- The value of the *chunk_size* expression must be the same for all threads in the team.

10.5.2 `parallel sections` Construct

The `#pragma omp parallel sections` directive provides a shortcut form for specifying a parallel region containing one sections directive. The parallel sections directive has the following syntax:

```
#pragma omp parallel sections [clause[ clause] ...] new-line
   {
   [#pragma omp section new-line] structured-block
   [#pragma omp section new-line  structured-block
   .
   .
   .]
   }
```

clause is one of the following:

`if` (*scalar-expression*)

`private` (*list*)

`firstprivate` (*list*)

`lastprivate(`*list*`)`

`default` (`shared | none`)

`shared` (*list*)

`copyin` (*list*)

`reduction` (*operator*: *list*)

`nowait`

In the following example, functions `xaxis`, `yaxis`, and `zaxis` can be executed concurrently. The first section directive is optional. Note that all section directives must appear in the lexical extent of the parallel sections construct.

```
#pragma omp parallel sections
{
   #pragma omp section
     xaxis();
   #pragma omp section
     yaxis();
   #pragma omp section
     zaxis();
}
```

10.6 Master and Synchronization Constructs

The following sections describe the synchronization constructs:

- Section 10.6.1, page 101, describes the `#pragma omp master` directive.
- Section 10.6.2, page 101, describes the `#pragma omp critical` directive.
- Section 10.6.3, page 102, describes the `#pragma omp barrier` directive.
- Section 10.6.4, page 102, describes the `#pragma omp atomic` directive.
- Section 10.6.5, page 103, describes the `#pragma omp flush` directive.
- Section 10.6.6, page 106, describes the `#pragma omp ordered` directive.

10.6.1 `master` Construct

The `#pragma omp master` directive identifies a construct that specifies a structured block that is executed by the master thread of the team. It has the following syntax:

`#pragma omp master` *new-line structured-block*

Other threads in the team do not execute the associated statement. There is no implied barrier either on entry to or exit from the master section.

10.6.2 `critical` Construct

The `#pragma omp critical` directive identifies a construct that restricts execution of the associated structured block to one thread at a time. It has the following syntax:

`#pragma omp critical [(`*name*`)]` *new-line structured-block*

An optional name that has external linkage may be used to identify the critical region.

A thread waits at the beginning of a criical region until no other thread is executing a critical region with the same name. All unnamed `#pragma omp critical` directives map to the same unspecified name.

The following example includes several `#pragma omp critical` directives. It illustrates a queuing model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the same task, the dequeuing operation must be in a `#pragma omp critical` section. Because the two queues in this example are independent, they are protected by `#pragma omp critical` directives with different names, `xaxis` and `yaxis`.

```
#pragma omp parallel shared(x, y) private(x_next, y_next)
{
   #pragma omp critical ( xaxis )
     x_next = dequeue(x);
   work(x_next);
   #pragma omp critical ( yaxis )
     y_next = dequeue(y);
   work(y_next);
}
```

10.6.3 `barrier` Directive

The `#pragma omp barrier` directive synchronizes all the threads in a team, each thread waiting until all other threads have reached this point. After all threads have been synchronized, they begin executing the statements after the `barrier` directive in parallel. The `barrier` directive has the following syntax:

```
#pragma omp barrier new-line
```

10.6.4 `atomic` Construct

The `#pragma omp atomic` directive ensures that a specific memory location is updated atomically. The `atomic` directive has the following syntax:

```
#pragma omp atomic new-line expression-stmt
```

The *expression-stmt* must have one of the following forms:

```
x binop = expr
   x++
   ++x
   x--
   --x
```

Where:

x is an lvalue expression with scalar type.

expr is an expression with scalar type, and it does not reference the object designated by *x*.

binop is not an overloaded operator and one of +, *, —, /, &, ^, |, <<, or >>.

Although a conforming implementation can replace all `#pragma omp atomic` directives with critical directives that have the same unique name, the `#pragma omp atomic` directive permits better optimization. Often hardware instructions are available that can perform the atomic update with the least overhead.

Only the load and store of the object designated by *x* are atomic. To avoid race conditions, all updates of the location in parallel should be protected with the atomic directive, unless they are known to be free of race conditions.

The following restrictions apply to the `#pragma omp atomic` directive:

- All atomic references to the storage location *x* throughout the program are required to have a compatible type.

Examples:

```
extern float a[], *p = a, b;
/* Protect against races among multiple updates.*/
#pragma omp atomic
a[index[i] += b;
/* Protect against races with updates through a.*/
#pragma omp atomic
p[i] -= 1.0f;
extern union {int n; float x;} u;
/* ERROR - References through incompatible types.*/
#pragma omp atomic
u.n++;
#pragma omp atomic
u.x -= 1.0f;
```

10.6.5 `flush` Directive

The `#pragma omp flush` directive, explicit or implied, identifies precise synchronization points at which the implementation is required to provide a consistent view of certain objects in memory. This means that previous evaluations of expressions that reference those objects are complete and subsequent evaluations have not yet begun.

The `flush` directive has the following syntax:

```
#pragma omp flush [(list)] new-line
```

list	Objects that require synchronization that can be designated by variables. If a pointer is present in the list, the pointer itself is flushed, not the object to which the pointer refers.

If no list is specified, all shared objects except inaccessible objects with automatic storage duration are synchronized. A `flush` directive without a *list* is implied for the following directives:

- `barrier`

- At entry to and exit from `critical`
- At entry to and exit from `ordered`
- At exit from `parallel`
- At exit from `for`
- At exit from `sections`
- At exit from `single`

The directive is not implied if a `nowait` clause is present.

A reference that accesses the value of an object with a volatile-qualified type behaves as if there were a `flush` directive specifying that object at the previous sequence point. A reference that modifies the value of an object with a volatile-qualified type behaves as if there were a `flush` directive specifying that object at the subsequent sequence point.

The following restriction applies to the `flush` directive:

- A variable specified in a `flush` directive must not have a reference type.

The following example uses the `flush` directive for point-to-point synchronization of specific objects between pairs of threads:

```
#pragma omp parallel private(iam,neighbor) shared(work,sync)
{

  iam = omp_get_thread_num();
  sync[iam] = 0;
  #pragma omp barrier

  /*Do computation into my portion of work array */
  work[iam] = ...;

  /*  Announce that I am done with my work
   *  The first flush ensures that my work is made visible before sync.
   *  The second flush ensures that sync is made visible.
   */
  #pragma omp flush(work)
  sync[iam] = 1;
  #pragma omp flush(sync)

  /*Wait for neighbor*/
  neighbor = (iam>0 ? iam : omp_get_num_threads()) - 1;
  while (sync[neighbor]==0) {
    #pragma omp flush(sync)
  }

  /*Read neighbor’s values of work array */
  ... = work[neighbor];
}
```

The following example distinguishes the shared objects affected by a `flush` directive with no list from the shared objects that are not affected:

```
int x, *p = &x;

void f1(int *q)
{
  *q = 1;
  #pragma omp flush
  // x, p, and *q are flushed
  //   because they are shared and accessible
}

void f2(int *q)
{
  *q = 2;
  #pragma omp barrier
  // a barrier implies a flush
  // x, p, and *q are flushed
  //   because they are shared and accessible
}

int g(int n)
{
  int i = 1, j, sum = 0;
  *p = 1;
  #pragma omp parallel reduction(+: sum)
  {
    f1(&j);
    // i and n were not flushed
    //   because they were not accessible in f1
    // j was flushed because it was accessible
    sum += j;
    f2(&j);
    // i and n were not flushed
    //   because they were not accessible in f2
    // j was flushed because it was accessible
    sum += i + j + *p + n;
  }
  return sum;
}
```

10.6.6 `ordered` Construct

A `#pragma omp ordered` directive must be within the dynamic extent of a `for` or `parallel for` construct that has an `ordered` clause. The *structured-block* following an `ordered` directive is executed in the same order as iterations in a sequential loop. It has the following syntax:

```
#pragma omp ordered new-line structured-block
```

The following restrictions apply to an `ordered` directive:

- It must not be in the dynamic extent of a for directive that does not have the ordered clause specified.
- An iteration of a loop with a `for` construct must not execute the same `ordered` directive more than once, and it must not execute more than one `ordered` directive.

Ordered sections are useful for sequentially ordering the output from work that is done in parallel. The following program prints out the indexes in sequential order:

```
#pragma omp for ordered schedule(dynamic)
  for (i=lb; i<ub; i+=st)
    work(i);


void work(int k)
{
  #pragma omp ordered
    printf(" %d", k);
}
```

10.7 Data Environment

The `#pragma omp threadprivate` directive and data scope attribute clauses control the data environment during the execution of parallel regions.

10.7.1 `threadprivate` Directive

The `#pragma omp threadprivate` directive makes named common blocks private to a thread but global within the thread. In other words, each thread executing a `threadprivate` directive receives its own private copy of the named common blocks, which are then available to it in any routine within the scope of an application.

The `threadprivate` directive has the following syntax:

```
#pragma omp threadprivate(list) new-line
```

A thread must not reference another thread's copy of a `threadprivate` object. During serial regions and master regions of the program, references will be to the master thread's copy of the object.

On entry to the first parallel region, data in the `threadprivate` common blocks should be assumed to be undefined unless a `copyin` clause is specified on the `parallel` directive. When a common block that is initialized using data statements appears in a `threadprivate` directive, each thread's copy is initialized once prior to its first use. For subsequent parallel regions, the data in the `threadprivate` common blocks are guaranteed to persist only if the dynamic threads mechanism has been disabled and if the number of threads are the same for all the parallel regions. For more information on dynamic threads, see the `omp_set_dynamic`(3) library function and the `OMP_DYNAMIC` environment variable on the `pe_environ`(5) man page.

The following restrictions apply to the `threadprivate` directive:

- The `threadprivate` directive must appear at file scope or namespace scope, must appear outside of any definition or declaration, and must lexically precede all references to any of the variables in its *list*.
- Each variable in the list of a `threadprivate` directive must have a file-scope or namespace-scope declaration that lexically precedes the directive.

- If a variable is specified in a `threadprivate` directive in one translation unit, it must be specified in a `threadprivate` directive in every translation unit in which it is declared.
- A `threadprivate` variable may appear only in the `copyin`, `schedule`, or the `if` clause. It is not permitted in the `private`, `firstprivate`, `lastprivate`, `shared`, or `reduction` clauses. They are not affected by the `default` clause.
- The address of a `threadprivate` variable is not an address constant.
- A `threadprivate` variable must not have an incomplete type or a reference type.
- A `threadprivate` variable with non-POD class type must have an accessible, unambiguous copy constructor if it is declared with an explicit initializer (in case the initialization is implemented using a temporary shared object).
- The following example shows how modifying a variable that appears in an initializer can cause unspecified behavior, and also how to avoid this problem by using an auxiliary object and a copy-constructor:

```
int x = 1;
T a(x);
const T baux(x); /*Capture value of x = 1 */
T b(b_aux);
#pragma omp threadprivate(a, b)

void f(int n) {
  x++;
  #pragma omp parallel for
  /* In each thread:
   * Object a is constructed from x (with value 1 or 2?)
   * Object b is copy-constructed from b_aux
   */
   for (int i=0; i<n; i++) {
      g(a, b); /* Value of a is unspecified. */
   }
}
```

10.7.2 Data Scope Attribute Clauses

Several directives accept clauses that allow a user to control the scope attributes of variables for the duration of the construct. Not all of the clauses in this section are allowed on all directives, but the clauses that are valid on a particular directive are included with the description of the directive. Usually, if no data scope clauses are specified for a directive, the default scope for variables affected by the directive is share.

The following sections describe the data scope attribute clauses:

- Section 10.7.2.1, page 109, describes the `private` clause.
- Section 10.7.2.2, page 110, describes the `firstprivate` clause.
- Section 10.7.2.3, page 111, describes the `lastprivate` clause.
- Section 10.7.2.4, page 112, describes the `shared` clause.
- Section 10.7.2.5, page 112, describes the `default` clause.
- Section 10.7.2.6, page 113, describes the `reduction` clause.
- Section 10.7.2.7, page 116, describes the `copyin` clause.

10.7.2.1 `private` Clause

The `private` clause declares the variables in list to be private to each thread in a team.

This clause has the following syntax:

```
private(list)
```

The behavior of a variable declared in a `private` clause is as follows:

- A new object of the same type is declared once for each thread in the team. The new object is no longer storage associated with the storage location of the original object.

  All references to the original object in the lexical extent of the directive construct are replaced with references to the private object.

  Variables defined as private are undefined for each thread on entering the construct and the corresponding shared variable is undefined on exit from a parallel construct.

Contents, allocation state, and association status of variables defined as private are undefined when they are referenced outside the lexical extent (but inside the dynamic extent) of the construct, unless they are passed as actual arguments to called functions.

The following restrictions apply to the `private` clause:

- A variable with a class type that is specified in a `private` clause must have an accessible, unambiguous default constructor.
- Unless it has a class type with a `mutable` member, a variable specified in a `private` clause must not have a const-qualified type.
- A variable specified in a `private` clause must not have an incomplete type or a reference type.
- Variables that are private within a parallel region cannot be specified in a `private` clause on an enclosed work-sharing or `parallel` directive. As a result, variables that are specified private on a work-sharing or `parallel` directive must be shared in the enclosing parallel region.

Example: The values of `i` and `j` in the following example are undefined on exit from the parallel region:

```
int i, j;
i = 1;
j = 2;
#pragma omp parallel private(i) firstprivate(j)
{
  i = 3;
  j = j + 2;
}
printf("%d %d\n", i, j);
```

10.7.2.2 `firstprivate` Clause

The `firstprivate` clause provides a superset of the functionality provided by the `private` clause.

This clause has the following syntax:

```
firstprivate(list)
```

In addition to the `private` clause semantics, each new private object is initialized as if there were an implied declaration inside the structured block, and the initializer is the value of the variable's original object. A copy constructor is invoked for a class object, if necessary.

The following restrictions apply to the `firstprivate` clause:

- All restrictions for `private` apply, except for the restrictions about default constructors and about const-qualified types.
- A variable with a class type that is specified as `firstprivate` must have an accessible, unambiguous copy constructor.

10.7.2.3 `lastprivate` Clause

The `lastprivate` clause provides a superset of the functionality provided by the `private` clause.

This clause has the following syntax:

```
lastprivate(list)
```

When a `lastprivate` clause appears on the directive that identifies a work-sharing construct, the value of each variable from the sequentially last iteration of the associated loop, or the lexically last `section` directive, is assigned to the variable's original object. Variables that are not assigned a value by the last iteration of the `for` or `parallel for`, or by the lexically last section of the `sections` or `parallel sections` directive, have indeterminate values after the construct. Unassigned subobjects also have an indeterminate value after the construct.

The following restrictions apply to the `lastprivate` clause:

- All restrictions for private apply.
- A variable that is specified as `lastprivate` must have an accessible, unambiguous copy assignment operator.

Example: Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. Such programs must list all such variables as arguments to a `lastprivate` clause so that the values of the variables are the same as when the loop is executed sequentially. In the following example, the value of `i` at the end of the parallel region will equal `n-1`, as in the sequential case.

```
#pragma omp parallel
{
   #pragma omp for lastprivate(i)
     for (i=0; i<n; i++)
       a[i] = b[i] + b[i+1];
}
a[i]=b[i];
```

10.7.2.4 `shared` Clause

This clause shares variables that appear in the *list* among all the threads in a team. All threads within a team access the same storage area for `shared` variables.

This clause has the following syntax:

```
shared(list)
```

10.7.2.5 `default` Clause

The `default` clause allows the user to specify a `shared` or `none` default scope attribute for all variables in the lexical extent of any parallel region. Variables in `threadprivate` common blocks are not affected by this clause.

This clause has the following syntax:

```
default(shared | none)
```

`shared`	Specifying `default(shared)` is equivalent to explicitly listing each currently visible variable in a `shared` clause. It is the default.
`none`	Specifying default(none) declares that there is no implicit default as to whether variables are shared. In this case, the `private`, `shared`, `firstprivate`, `lastprivate`, or `reduction` attribute of each variable used in the lexical extent of the parallel region must be specified.

Only one `default` clause can be specified on a `parallel` directive.

The following example shows how variables can be exceptioned from a defined default using the `private`, `shared`, `firstprivate`, `lastprivate`, or `reduction` clauses:

```
#pragma omp parallel for default(shared) firstprivate(i) private(x)/private(r) lastprivate(i)
```

The following example distinguishes the variables that are affected by the `default(none)` clause from those that are not:

```
int x, y, z[1000];
#pragma omp threadprivate(x)

void fun(int a) {
  const int c = 1;
  int i = 0;

  #pragma omp parallel default(none) private(a) shared(z)
  {
     int j = omp_get_num_thread();
                // O.K.  - j is declared within parallel region
     a = z[j];   // O.K.  - a is listed in private clause
                //       - z is listed in shared clause
     x = c;     // O.K.  - x is threadprivate
                 //       - c has const-qualified type
     z[i] = y;   // Error - cannot reference i or y here

     #pragma omp for firstprivate(y)
     for (i=0; i<10 ; i++) {
         z[i] = y;  // O.K. - i is the loop control variable
                    //      - y is listed in firstprivate clause
     }
     z[i] = y;   // Error - cannot reference i or y here
  }
}
```

10.7.2.6 `reduction` Clause

This clause performs a reduction on the variables specified, with the operator or the intrinsic specified. This clause has the following syntax:

`reduction(`*op*`:`*list*`)`

A `reduction` is typically used in a statement with one of the following forms:

```
x = x op expr
x <binop> = expr
x = expr op x (except for subtration)
x++
++x
x--
--x
```

Where:

`x`	One of the reduction variables specified in the list.
list	A comma-separated list of reduction variables.
expr	An expression with scalar type that does not reference x.
op	One of +, *, —, &, ^, `|`, &&, or `||`.
binop	One of +, *, — &, ^, or `|`.

The following example shows how to use the `reduction` clause:

```
#pragma omp parallel for reduction(+: a, y) reduction(||: am) for (i=0; i<n; i++) {
   a += b[i];
   y = sun(y, c[i];
   am = am || b[i] == c[i];
}
```

Because the operator may be hidden inside a function call, ensure that the operator specified in the `reduction` clause matches the reduction operation.

The following table lists the operators and intrinsics that are valid and their canonical initialization values. The actual initialization value will be consistent with the data type of the reduction variable:

Operator	Initialization
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

Any number of reduction clauses can be specified on the directive, but a variable can appear in at most one reduction clause for that directive.

The following example shows how variables that appear in the `reduction` clause must be shared in the enclosing context:

```
#pragma omp parallel private(y)
{ /* ERROR - private variable y cannot be specified in a reduction clause */
   #pragma omp for reduction(+: y)
   for (i=0; i<n; i++)
     y += b[i];
}

/* ERROR - variable x cannot be specified in both a shared and a reduction clause */
#pragma omp parallel for shared(x) reduction(+: x)
```

The following restrictions apply to the `reduction` clause:

- The type of the variables in the `reduction` clause must be valid for the reduction operator except that pointer types and reference types are never permitted.
- A variable that is specified in the `reduction` clause must not be const-qualified.
- A variable that is specified in the `reduction` clause must be shared in the enclosing context.

10.7.2.7 `copyin` Clause

The `copyin` clause lets you assign the same value to `threadprivate` variables for each thread in the team executing the parallel region. For each variable specified, the value of the variable in the master thread of the team is copied to the `threadprivate` copies at the beginning of the parallel region.

This clause has the following syntax:

```
copyin(list)
```

The following restrictions apply to the `copyin` clause:

- A variable that is specified in the `copyin` clause must have an accessible, unambiguous copy assignment operator.
- A variable that is specified in the `copyin` clause must be a `threadprivate` variable.

10.8 Directive Binding

Some directives are bound to other directives. A binding specifies the way in which one directive is related to another. For instance, a directive is bound to a second directive if it can appear in the dynamic extent of that second directive. The following rules apply with respect to the dynamic binding of directives:

- The `for`, `sections`, `single`, `master`, and `barrier` directives bind to the dynamically enclosing `parallel` directive, if one exists. If no parallel region is currently being executed, the directives have no effect.
- The `ordered` directive binds to the dynamically enclosing `for` directive.
- The `atomic` directive enforces exclusive access with respect to atomic directives in all threads, not just the current team.
- The `critical` directive enforces exclusive access with respect to critical directives in all threads, not just the current team.
- A directive cannot bind to a directive outside the closest enclosing `parallel` directive.

The directive binding rules call for a `barrier` directive to bind to the closest enclosing `parallel` directive. In the following example, the calls in `main`, to `sub1` and `sub2`, are both valid, and the `barrier` in `sub3` binds to the `parallel` region in `sub2` in both cases. The effect is different, however, because in the call to `sub1`, the `barrier` affects only a subteam. The number of threads in a subteam is implementation-dependent if nested parallelism is enabled (with the `OMP_NESTED` environment variable), and otherwise is one (in which case the `barrier` has no real effect).

```
int main()
{
  sub1(2);
  sub2(2);
}

void sub1(int n)
{
  int i;
  #pragma omp parallel private(i) shared(n)
  {
    #pragma omp for
    for (i=0; i<n; i++)
      sub2(i);
  }
}

void sub2(int k)
{
  #pragma omp parallel shared(k)
    sub3(k);
}

void sub3(int n)
{
  work1(n);
  #pragma omp barrier
  work2(n);
}
```

10.9 Directive Nesting

Dynamic nesting of directives must adhere to the following rules:

- A `parallel` directive dynamically inside another `parallel` directive logically establishes a new team, which is composed of only the current thread, unless nested parallelism is enabled.

- `for`, `sections`, and `single` directives that bind to the same `parallel` directive are not allowed to be nested inside each other.
- `critical` directives with the same name are not allowed to be nested inside each other.
- `for`, `sections`, and `single` directives are not permitted in the dynamic extent of `critical`, `ordered`, and `master` regions.
- `barrier` directives are not permitted in the dynamic extent of `for`, `ordered`, `sections`, `single`, `master`, and `critical` regions.
- `master` directives are not permitted in the dynamic extent of `for`, `sections`, and `single` directives.
- `ordered` directives are not allowed in the dynamic extent of `critical` regions.
- Any directive that is permitted when executed dynamically inside a `parallel` region is also permitted when executed outside a `parallel` region. When executed dynamically outside a user-specified parallel region, the directive is executed with respect to a team composed of only the master thread.

The following program is correct because the inner and outer `for` directives bind to different `parallel` regions:

```
#pragma omp parallel default(shared)
{
  #pragma omp for
    for (i=0; i<n; i++) {
      #pragma omp parallel shared(i, n)
      {
        #pragma omp for
          for (j=0; j<n; j++)
            work(i, j);
      }
   }
}
```

A following variation of the preceding example is also correct:

```
#pragma omp parallel default(shared)
{
  #pragma omp for
    for (i=0; i<n; i++)
```

```
      work1(i, n);
}


}

void work1(int i, int n)
{
  int j;
  #pragma omp parallel default(shared)
  {
    #pragma omp for
      for (j=0; j<n; j++)
        work2(i, j);
  }
  return;
}
```

Precompiled Header #pragma Directives [11]

Table 14, page 121, lists the precompiled header #pragmas directives, along with a short description of each and the compiler versions in which the directive is supported.

Table 14. Precompiled Header #pragma Directives

#pragma	Short Description	Compiler Versions
#pragma hdrstop	Indicates the point at which the precompiled header mechanism snapshots the headers. If -pch is off, #pragma hdrstop is ignored.	7.2 and later
#pragma no_pch	Disables the precompiled header mechanism.	7.2 and later
#pragma once	Ensures (in -n32 and -64 mode) that an include file is included at most one time in each compilation unit.	7.0 and later

11.1 #pragma hdrstop

The #pragma hdrstop directive indicates the point at which the precompiled header mechanism snapshots the headers.

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

```
#pragma hdrstop
```

If -pch is on, #pragma hdrstop indicates the point at which the precompiled header mechanism snapshots the headers.

If -pch is off, #pragma hdrstop is ignored.

See the *MIPSpro Compiling and Performance Tuning Guide* for details on the precompiled header mechanism.

11.2 #pragma no_pch

The `#pragma no_pch` directive disables the precompiled header mechanism.

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

```
#pragma no_pch
```

11.3 #pragma once

The `#pragma once` directive ensures (in `-n32` and `-64` mode) that each `include` file is included one time in each compilation unit.

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

```
#pragma once
```

This directive has no effect in `-o32` mode, but will ensure idempotent `include` files in `-n32` and `-64` mode (that is, that an `include` file is included at most one time in each compilation unit).

Silicon Graphics recommends enclosing the contents of an `afile.h` include file with an `#ifdef` directive similar to the following:

```
#ifndef afile_INCLUDED
#define afile_INCLUDED
<contents of afile.h>
#endif
```

Scalar Optimization #pragma Directives [12]

Table 15, page 123, lists the `#pragma` directives discussed in this chapter, along with a short description of each and the compiler versions in which the directive is supported.

Table 15. Scalar Optimization `#pragma` Directives

#pragma	Short Description	Compiler Versions
`#pragma mips_frequency_hint`	Specifies the expected frequency of execution so that cord2 can move exception code and initialization code into separate pages to minimize working set size.	7.2 and later
`#pragma section_gp` (in Chapter 7, page 39)	Causes an object to be placed in a gp_relative section.	7.2 and later
`#pragma section_non_gp` (in Chapter 7, page 39)	Keeps an object from being placed in a gp_relative section.	7.2 and later
`#pragma unroll` (in Chapter 8, page 47)	Suggests to the compiler that a specified number of copies of the loop body be added to the inner loop. If the loop following this directive is an inner loop, then it indicates standard unrolling. If the loop following this directive is not innermost, then outer loop unrolling (unroll and jam) is performed.	7.2 and later

12.1 `#pragma mips_frequency_hint`

This directive allows you to specify the expected frequency of execution of the named function so the compiler can move exception code and initialization code into separate pages to minimize working-set size.

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

`#pragma mips_frequency_hint {NEVER|INIT}` [*function_name*]

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

This directive provides a mechanism for you to specify information about execution frequency for certain regions in the code. You can provide the following frequency specifications:

`NEVER`	This region of code is never or rarely executed. The compiler might move this region of the code away from the normal path. This movement might either be at the end of the procedure or at some point to an entirely separate section.
`INIT`	This region of code is executed only during initialization or startup of the program. The compiler might try to put all regions under "INIT" together to provide better locality during startup of a program.

You can use this directive in two ways:

1. You can specify it with a function declaration. The directive then applies everywhere the function is called.

   ```
   extern void Error_Routine();
   #pragma mips_frequency_hint NEVER Error_Routine
   ```

   Note: In this case, the directive must appear after the function declaration.

2. You can specify it without a function declaration. In this case, you can place the directive anywhere in the body of a procedure. It then applies to the statement directly following the directive.

   ```
   if (some_condition)
   {
   #pragma mips_frequency_hint NEVER
   Error_Routine ();
   ...
   }
   ```

Caution: This is directive is supported on compiler version 7.2 only, and it does not work for `-o32` because it requires an ELF object file with `.MIPS.content` sections.

Warning Suppression Control #pragma Directives [13]

Table 16, page 125, lists the #pragma directives discussed in this chapter, along with a brief description and the compiler versions in which the directive is supported.

Table 16. Warning Suppression Control #pragma Directives

#pragma	Short Description	Compiler Versions
`#pragma set woff`	Suppresses compiler warnings (either all, or by warning number).	7.2 and later
`#pragma reset woff`	Resets listed warnings to the state specified in the command line.	7.2 and later

13.1 #pragma set woff

The `#pragma set woff` directive suppresses compiler warnings individually by warning number.

The syntax of `#pragma set woff` is as follows:

#pragma set woff (*warning_list*)

warning_list is a list of the warning numbers that you want suppressed. Ranges are allowed. Only the specified compiler warnings are suppressed.

For example, the following directive turns off warnings 1, 2, 300 through 310, and 8:

```
#pragma set woff 1,2,300-310,8
```

`#pragma set woff` does not nest. That is, any `#pragma reset woff` on a given number resets the value to that implied by the command line.

13.1.1 Example of `#pragma set woff`

The following code illustrates the use of `#pragma set woff`:

```
cc -woff 300,302

/*  example.c */
#pragma set woff 400
/* warnings 300,302, and 400  are off  in example.c */

#include ‘‘example.h’’
/* You would expect that warnings 300,302,and 400 would be off
in example.h. However, the #pragma set woff does not travel
into #includes properly. In MIPSpro7.2 300 and 302 are off, but
400 is on in example.h. In a future release 400 may be off in
example.h
*/

#pragma reset woff 400
/*  400 is reset to command line state; that is, 400 is on. */

#pragma reset woff 300
/* 300 is reset to command line state; that is, 300 is still off */
```

13.2 `#pragma reset woff`

The `#pragma reset woff` directive resets listed warnings to the state specified in the command line.

The syntax of `#pragma reset woff` is as follows:

`#pragma reset woff (`*warning_list*`)`

warning_list consists of a list of the warning numbers that you want reset to the state specified in the command line. Ranges are allowed. Only the specified compiler warnings are reset.

For example, the following directive sets warnings 1, 2, 300 through 310, and 8 back to the command-line setting:

```
#pragma set woff 1,2,300-310,8
```

This directive does not nest.

13.2.1 Example of `#pragma reset woff`

The following code illustrates the use of `#pragma reset woff`:

```
cc -woff 300,302

/*  example.c */
#pragma set woff 400
/* warnings 300,302, and 400 are off in example.c */

#include ‘‘example.h’’
/* You would expect that warnings 300,302,and 400 would be off
in example.h. However, the #pragma set woff does not travel
into #includes properly. In MIPSpro7.2 300 and 302 are off,
but 400 is on in example.h. In a future release 400 may be off
in example.h
*/

#pragma reset woff 400
/*  400 is reset to command line state; that is, 400 is on. */

#pragma reset woff 300
/* 300 is reset to command line state; that is, 300 is still off */
```

Miscellaneous #pragma Directives [14]

Table 17, page 129, lists the #pragma directives described in this chapter, along with a brief description of each and the compiler version in which they are supported.

Table 17. Miscellaneous #pragma Directives

#pragma	Short Description	Compiler Versions
#pragma ident	Adds a .comment section to the object file and puts the supplied string inside the .comment section.	6.0 and later (-o32 only)
#pragma int_to_unsigned	Identifies *identifier* as a function whose type was int in a previous release of the compilation system, but whose type is unsigned int in the MIPSpro compiler release.	7.0 and later
#pragma intrinsic	Allows certain preselected functions from math.h, stdio.h, and string.h to be inlined at a call site. Can also enable the compiler to get additional information about the function to improve execution efficiency.	7.0 and later
#pragma unknown_control_flow	Indicates user level functions that have behavior similar to setjmp and getcontext.	7.3 and later

14.1 #pragma ident

The #pragma ident directive adds a .comment section to the object file and puts the supplied string inside the .comment section.

The syntax of #pragma ident is as follows:

```
#pragma ident ''string''
```

string is the string you want to add to the .comment section in the object file. The string must be enclosed in double quotation marks.

Caution: The `#pragma ident` directive is only available in `-o32` mode.

14.2 #pragma int_to_unsigned

The `#pragma int_to_unsigned` directive tells the compiler that the named function has a different type (`unsigned int`) in the MIPSpro compiler release than it did in previous releases (`int`).

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

`#pragma int_to_unsigned` *function_name*

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

This directive identifies *function_name* as a function whose type was `int` in a previous release of the compilation system, but whose type is `unsigned int` in the MIPSpro compiler release. The declaration of the identifier must precede the directive:

```
unsigned int strlen(const char*);
#pragma int_to_unsigned strlen
```

This declaration makes it possible for the compiler to identify where the changed type may affect the evaluation of expressions.

14.3 #pragma intrinsic

The `#pragma intrinsic` directive allows certain preselected functions from `math.h`, `stdio.h`, and `string.h` to be inlined at a call site for execution efficiency.

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

`#pragma intrinsic (`*function_name*`)`

Caution:

- This directive has no effect on functions other than the preselected ones.
- Exactly which functions may be inlined, how they are inlined, and under what circumstances inlining occurs is implementation-defined and may vary from one release of the compilers to the next.
- The inlining of intrinsics may violate some aspect of the ANSI C standard (for example, the errno setting for `math.h` functions).
- All intrinsics are activated through directives in the respective standard header files and only when the preprocessor symbol `__INLINE_INTRINSICS` is defined and the appropriate include files are included. `__INLINE_INTRINSICS` is predefined by default only in `-cckr` and `-xansi` mode.

14.4 #pragma unknown_control_flow

The `#pragma unknown_control_flow` directive indicates that the procedures listed as *func1*, *func2*, etc. have a nonstandard control flow behavior, such as `setjmp` or `getcontext`. This type of behavior interferes with optimizations such as tail call optimization.

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

```
#pragma unknown_control_flow (func1 [, func2 ...])
```

This directive should appear after the external declaration of the function(s).

The Auto-Parallelizing Option (APO) [15]

Note: APO is licensed and sold separately from the MIPSpro C/C++ compilers. APO features in your code are ignored unless you are licensed for this product. For sales and licensing information, contact your sales representative.

The Auto-Parallelizing Option (APO) enables the MIPSpro C/C++ compilers to optimize parallel codes and enhances performance on multiprocessor systems. APO is controlled with command line options and source directives.

APO is integrated into the compiler; it is not a source-to-source preprocessor. Although run-time performance suffers slightly on single-processor systems, parallelized programs can be created and debugged with APO enabled.

Parallelization is the process of analyzing sequential programs for parallelism and restructuring them to run efficiently on multiprocessor systems. The goal is to minimize the overall computation time by distributing the computational workload among the available processors. Parallelization can be automatic or manual.

During *automatic parallelization*, the compiler analyzes and restructures the program with little or no intervention by you. With APO, the compiler automatically generates code that splits the processing of loops among multiple processors. An alternative is *manual parallelization*, in which you perform the parallelization using compiler directives and other programming techniques.

APO integrates automatic parallelization with other compiler optimizations, such as interprocedural analysis (IPA), optimizations for single processors, and loop nest optimization (LNO). In addition, run-time and compile-time performance is improved.

15.1 C/C++ Command Line Options That Affect APO

Several `cc`(1) and `CC`(1) command line options control APO’s effect on your program. For example, the following command line invokes APO and requests aggressive optimization:

```
CC -apo -O3 zebra.c
```

The following subsections describe the effects that various C/C++ command line options have on APO.

> **Note:** If you invoke the loader separately, you must specify the `-apo` option on the `ld`(1) command line.

15.1.1 `-apo`

The `-apo` option invokes APO. When this option is enabled, the compiler automatically converts sequential code into parallel code by inserting parallel directives where it is safe and beneficial to do so. Specifying `-apo` also enables the `-mp` option, which enables recognition of the parallel directives inserted into your code.

15.1.2 `-apokeep` and `-apolist`

The `-apokeep` and `-apolist` options control output files. Both options generate *file*`.list`, which is a listing file that contains information on the loops that were parallelized and explains why others were not parallelized.

When `-apokeep` is specified, the compiler writes *file*`.list`, and in addition, it retains *file*`.anl` and *file*`.m`. The ProMPF tools use *file*`.anl`. For more information on ProMPF, see the *ProDev ProMP User's Guide*. *file*`.m` is an annotated version of your source code that shows the insertion of multiprocessing directives.

When `-IPA` is specified with the `-apokeep` option, the default settings for IPA suboptions are used with the exception of `-IPA:inline`, which is set to `OFF`.

For more information on the content of *file*`.list`, *file*`.anl`, and *file*`.m`, see Section 15.2, page 137.

> **Note:** Because of data conflicts, do not specify the `-mplist` or `-CLIST` options when `-apokeep` is specified.

15.1.3 `-CLIST:...`

This option generates a C/C++ listing and directs the compiler to write an equivalent parallelized program in *file*`.w2c.c`. For more information on the content of *file*`.w2c.c`, see Section 15.2, page 137.

15.1.4 -IPA:...

Interprocedural analysis (IPA) is invoked by the `-ipa` or `-IPA` command line option. It performs program optimizations that can only be done by examining the whole program, not parts of a program.

When APO is invoked with IPA, only those loops whose function calls were determined to be safe by the APO are optimized.

If IPA expands functions inline in a calling routine, the functions are compiled with the options of the calling routine. If the calling routine is not compiled with `-apo`, none of its inlined functions are parallelized. This is true even if the functions are compiled separately with `-apo` because with IPA, automatic parallelization is deferred until link time.

When `-apokeep` or `-pcakeep` are specified in conjunction with `-ipa` or `-IPA`, the default settings for IPA suboptions are used with the exception of the `inline=`*setting* suboption, which is set to `OFF`.

For more information on the effect of IPA, see Section 15.5.1.2, page 148. For more information on IPA itself, see the `ipa`(5) man page.

15.1.5 -LNO:...

The `-LNO` options control the Loop Nest Optimizer (LNO). LNO performs loop optimizations that better exploit caches and instruction-level parallelism. The following LNO options are of particular interest to APO users:

- `-LNO:auto_dist=on`. This option requests that APO insert data distribution directives to provide the best memory utilization on Origin2000 systems.
- `-LNO:ignore_pragmas=`*setting*. This option directs APO to ignore all of the directives and assertions described in Section 15.4, page 141.
- `-LNO:parallel_overhead=`*num_cycles*. This option allows you to override certain compiler assumptions regarding the efficiency to be gained by executing certain loops in parallel rather than serially. Specifically, changing this setting changes the default estimate of the cost to invoke a parallel loop in your run-time environment. This estimate varies depending on your particular run-time environment, but it is typically several thousand machine cycles.

You can view the transformed code in the original source language after LNO performs its transformations. Two translators, integrated into the compiler, convert the compiler's internal representation into the original source language. You can invoke the desired translator by using the `CC -CLIST:=on` option. For example, the following command creates an `a.out` object file and the C/C++ file `test.w2c.c`:

```
CC -O3 -CLIST:=on test.c
```

Because it is generated at a later stage of the compilation, this `.w2c.c` file differs somewhat from the `.w2c.c` file generated by the `-apokeep` option (see Section 15.1.2, page 134). You can read the `.w2c.c` file, which is a compilable C/C++ representation of the original program after the LNO phase. Because the LNO is not a preprocessor, recompiling the *file*`.w2c.c` can result in an executable that differs from the original compilation of the `.c` file.

15.1.6 `-O3`

To obtain maximum performance, specify `-O3` when compiling with APO enabled. The optimization at this level maximizes code quality even if it requires extensive compile time or relaxes the language rules. The `-O3` option uses transformations that are usually beneficial but can sometimes hurt performance. This optimization may cause noticeable changes in floating-point results due to the relaxation of operation-ordering rules. Floating-point optimization is discussed further in Section 15.1.7, page 136.

15.1.7 `-OPT:...`

The `-OPT` command line option controls general optimizations that are not associated with a distinct compiler phase.

The `-OPT:roundoff=`*n* option controls floating-point accuracy and the behavior of overflow and underflow exceptions relative to the source language rules.

When `-O3` is in effect, the default rounding setting is `-OPT:roundoff=2`. This setting allows transformations with extensive effects on floating-point results. It allows associative rearrangement across loop iterations and the distribution of multiplication over addition and subtraction. It disallows only transformations known to cause overflow, underflow, or cumulative round-off errors for a wide range of floating-point operands.

At `-OPT:roundoff=2` or `3`, APO can change the sequence of a loop's floating-point operations in order to parallelize it. Because floating-point operations have finite precision, this change can cause slightly different results.

If you want to avoid these differences by not having such loops parallelized, you must compile with `-OPT:roundoff=0` or `-OPT:roundoff=1`.

Example. APO parallelizes the following loop when compiled with the default settings of `-OPT:roundoff=2` and `-O3`:

```
float a, b[100];
for(i=0; i<100; i++)
    a = a + b[i];
```

At the start of the loop, each processor gets a private copy of `a` in which to hold a partial sum. At the end of the loop, the partial sum in each processor's copy is added to the total in the original, global copy. This value of `a` can be different from the value generated by a version of the loop that is not parallelized.

15.1.8 `-pca`, `-pcakeep`, `-pcalist`

The `-pca` option invokes APO. For the O32 ABI, the `-pca` option invokes Power C. The `-pcakeep` and `-pcalist` options control output files.

When `-IPA` is specified with the `-pcakeep` option, the default settings for IPA suboptions are used with the exception of `-IPA:inline`, which is set to `OFF`.

Note: These options are outmoded. The preferred way of invoking APO is through the `-apo` option, and the preferred way to obtain a listing is through the `-apolist` option. For more information on these options, see Section 15.1.1, page 134, and Section 15.1.2, page 134.

15.1.9 *file*

Your input file.

For information on files used and generated when APO is enabled, see Section 15.2.

15.2 Files

APO provides a number of options to generate listings that describe where parallelization failed and where it succeeded. You can use these listings to identify constructs that inhibit parallelization. When you remove these constructs, you can often improve program performance dramatically.

When looking for loops to run in parallel, focus on the areas of the code that use most of the execution time. To determine where the program spends its execution time, you can use tools such as SpeedShop and the WorkShop Pro MPF Parallel Analyzer View described in *ProDev ProMP User's Guide*.

The following sections describe the content of the files generated by APO.

15.2.1 The *file*`.list` File

The `-apolist` and `-apokeep` options generate files that list the original loops in the program along with messages indicating if the loops were parallelized. For loops that were not parallelized, an explanation is provided.

Example. The following function resides in file `testl.c`:

```
void sub(double arr[], int n)
{
    extern void foo(double);
    int i;
    for(i=1; i<n; i++) {
      arr[i] += arr[i-1];
    }
    for(i=0; i<n; i++) {
      arr[i] += 7.0;
      foo(arr[i]);
    }
    for(i=0; i<n; i++) {
      arr[i] += 7.0;
    }
}
```

File `testl.c` is compiled with the following command:

```
cc -O3 -n32 -mips4 -apolist -c testl.c
```

APO produces file `testl.list`:

```
Parallelization Log for Subprogram sub
    5: Not Parallel
         Array dependence from arr on line 6 to arr on line 6.

    8: Not Parallel
         Call foo on line 10.

   12: PARALLEL (Auto) __mpdo_sub1
```

Note the message for line 12. Whenever a loop is run in parallel, the parallel version of the loop is put in its own function. The MIPSpro profiling tools attribute all the time spent in the loop to this function. The last line indicates that the name of the function is `__mpdo_sub1`.

15.2.2 The *file*`.w2f.c` File

File *file*`.w2c.c` contains code that mimics the behavior of programs after they undergo automatic parallelization. The representation is designed to be readable so that you can see what portions of the original code were not parallelized. You can use this information to change the original program.

The compiler creates *file*`.w2c.c` by invoking the appropriate translator to turn the compiler's internal representations into C/C++. In most cases, the files contain valid code that can be recompiled, although compiling *file*`.w2c.c` without APO enabled does not produce object code that is exactly the same as that generated when APO is enabled on the original source.

The `-apolist` option generates *file*`.w2c.c`. Because it is generated at an earlier stage of the compilation, *file*`.w2c.c` from `-apolist` is more easily understood than *file*`.w2c.c` generated from `-CLIST:=on` option. On the other hand, the `-CLIST` option shows more of the optimizations that were performed. The parallelized program in *file*`.w2c.c` uses OpenMP directives.

Example. File `testw2.c` is compiled with the following command:

```
cc -O3 -n32 -mips4 -c -apo -apolist -c testw2.c
```

```
void trivial(float a[])
{
    int i;
    for(i=0; i<10000; i++) {
      a[i] = 0.0;
    }
}
```

Compiling `testw2.c` generates an object file, `testw2.o`, and listing file `testw2.w2c.c`, which contains the following code:

```
/*******************************************************
 * C file translated from WHIRL Wed Oct 28 14:03:23 1998
 *******************************************************/
/* Include file-level type and variable decls */
```

```
#include "testw2.w2c.h"


void trivial(
  _IEEE32(*a0)[])
{
  register _INT32 i0;

  /* PARALLEL DO will be converted to SUBROUTINE __mpdo_trivial1 */;
#pragma parallel
  {
#pragma pfor
#pragma local(i0)
#pragma shared(a0)
    for(i0 = 0; i0 <= 9999; i0 = i0 + 1)
    {
      (*a0)[i0] = 0.0F;
    }
  }
  return;
} /* trivial */
```

Note: `WHIRL` is the name for the compiler's intermediate representation.

As explained in Section 15.2.1, page 138, parallel versions of loops are put in their own functions. In this example, that function is `__mpdo_trivial_1`. `#pragma omp parallel` is an OpenMP directive that specifies a parallel region containing a single `DO` directive.

15.2.3 About the `.m` and `.anl` Files

The `-apokeep` option generates *file*`.list`. It also generates *file*`.m` and *file*`.anl`, which are used by Workshop Pro MPF.

file`.m` is similar to the *file*`.w2c.c` file but is more like original source code; it is based on OpenMP and mimics the behavior of the program after automatic parallelization.

WorkShop Pro MPF is a Silicon Graphics product that provides a graphical interface to aid in both automatic and manual parallelization for C/C++. The WorkShop Pro MPF Parallel Analyzer View helps you understand the structure and parallelization of multiprocessing applications by providing an interactive, visual comparison of their original source with transformed, parallelized code.

For more information, see the *ProDev ProMP User's Guide* and the *Developer Magic: Performance Analyzer User's Guide*.

SpeedShop, another Silicon Graphics product, allows you to run experiments and generate reports to track down the sources of performance problems. SpeedShop includes a set of commands and a number of libraries to support the commands. For more information, see the *SpeedShop User's Guide*.

15.3 Running Your Program

Running a parallelized version of your program is no different from running a sequential one. The same binary output file can be executed on various numbers of processors. The default is to have the run-time environment select the number of processors to use based on how many are available.

You can change the default behavior by setting the `OMP_NUM_THREADS` environment variable, which tells the system to use an explicit number of processors. The following statement causes the program to create two threads regardless of the number of processors available:

```
setenv OMP_NUM_THREADS 2
```

The `OMP_DYNAMIC` environment variable allows you to control whether the run-time environment should dynamically adjust the number of threads available for executing parallel regions to optimize system resources. The default value is `ON`. If `OMP_DYNAMIC` is set to `OFF`, dynamic adjustment is disabled.

For more information on these and other environment variables, see the `pe_environ`(5) man page.

15.4 Compiler Directives

APO works in conjunction with the OpenMP C/C++ API directives and with the Origin series directives. You can use these directives to manually parallelize some loop nests, while leaving others to APO.This approach has the following positive and negative aspects:

- As a positive aspect, the OpenMP and Origin series directives are well defined and deterministic. If you use a directive, the specified loop is run in parallel. This assumes that the trip count is greater than one and that the specified loop is not nested in another parallel loop.

- The negative side to this is that you must carefully analyze the code to determine that parallelism is safe. In particular, you may need to specify special attributes for some variables, such as `private` or `reduction`, or specify explicit synchronizations, such as a `barrier` or a `critical` section.

In addition to the OpenMP and Origin series directives, you can also use the APO-specific directives described in this section. These directives give APO more information about your code.

> **Note:** APO also recognizes the Silicon Graphics multiprocessing directives. These directives are outmoded, and you must include the `-mp` option on the `CC`(1) command line in order for the compiler to recognize them. The OpenMP directive set is the preferred directive set for multiprocessing.

The APO directives can affect certain optimizations, such as loop interchange, during the compiling process. To direct the compiler to disregard any of the preceding directives, specify the `-x`*dirlist* option.

The APO directives are as follows:

- `#pragma concurrent call`. This directive directs APO to ignore dependencies in function calls that would inhibit parallelization. For more information on this directive, see Section 15.4.1, page 143.
- `#pragma concurrent`. This directive asserts that APO should not let perceived dependencies between two references to the same array inhibit parallelizing. For more information on this directive, see Section 15.4.2, page 144.
- `#pragma serial`. This directive requests that the following loop be executed in serial mode. For more information on this directive, see Section 15.4.3, page 145.
- `#pragma prefer concurrent`. This directive parallelizes the following loop if it is safe. For more information on this directive, see Section 15.4.4, page 145.
- `#pragma permutation (`*array_name*`)`. Asserts that array *array_name* is a permutation array. For more information on this directive, see Section 15.4.5, page 146.
- `#pragma no concurrentize` and `#pragma concurrentize`. The `#pragma no concurrentize` directive inhibits either parallelization of all loops in a function or parallelization of all loops in a file. The `#pragma concurrentize` directive overrides the `#pragma no concurrentize` directive, and its effect varies with its

placement. For more information on these directives, see Section 15.4.6, page 147.

Note: The compiler honors the following APO directives even if the `-apo` option is not included on your command line:

- `#pragma concurrent call`
- `#pragma prefer concurrent`
- `#pragma permutation (`*array_name*`)`

15.4.1 `#pragma concurrent call`

The `#pragma concurrent call` directive instructs APO to ignore the dependencies of function and function calls contained in the loop that follows the assertion. The directive applies to the loop that immediately follows it and to all loops nested inside that loop. Other points to be aware of are the following:

Note: The directive affects the compilation even when `-apo` is not specified.

APO ignores potential dependencies in function `fred()` when it analyzes the following loop:

```
#pragma concurrent call
for(i=0; i<n; i++) {
    fred();
    ...
}
```

To prevent incorrect parallelization, make sure the following conditions are met when using `#pragma concurrent call`:

- A function inside the loop cannot read from a location that is written to during another iteration. This rule does not apply to a location that is a local variable declared inside the function.
- A function inside the loop cannot write to a location that is read from or written to during another iteration. This rule does not apply to a location that is a local variable declared inside the function.

Example. The following code shows an illegal use of the directive. Function `fred()` writes to variable `x`, which is also read from by `wilma()` during other iterations, and the directive instructs APO to ignore this dependence.

```
void fred(float *b, int i, float *t) {
    *t = b[i];
}
void wilma(float *a, int i, float *t){
    a[i] = *t;
}

    #pragma concurrent call
    for(i=0; i<m; i++) {
       fred(b, i, &x);
       wilma(a, i, &x);
    }
```

The following example shows how you can manually parallelize the preceding example safely by 'localizing' variable `x` with a declaration `float x;` at the top of the loop body.

```
#pragma concurrent call
for (i=0, i<m, i++) {
   float x;
   fred(b, i, &x);
   wilma(a,i, &x);
}
```

15.4.2 `#pragma concurrent`

The `#pragma concurrent` directive instructs APO, when analyzing the loop immediately following this directive, to ignore all dependencies between two references to the same array. If there are real dependencies between array references, the `#pragma concurrent` directive can cause APO to generate incorrect code.

Note: This directive affects the compilation even when `-apo` is not specified.

The following example shows correct use of this directive when `m > n`:

```
#pragma concurrent
for(i=0; i<n; i++)
    a[i] = a[i+m];
```

Be aware of the following points when using this directive:

- If multiple loops in a nest can be parallelized, `#pragma concurrent` causes APO to parallelize the loop immediately following the assertion.

- Applying this directive to an inner loop can cause the loop to be made outermost by APO's loop interchange operations.
- This directive does not affect how APO analyzes function calls. For more information on APO's interaction with function calls, see Section 15.4.1, page 143.
- This directive does not affect how APO analyzes dependencies between two potentially aliased pointers.
- The compiler may find some obvious real dependencies. If it does so, it ignores this directive.

15.4.3 `#pragma serial`

The `#pragma serial` instructs APO not to parallelize the loop following the assertion; the loop is executed in serial mode. APO can, however, parallelize another loop in the same nest. The parallelized loop can be either inside or outside the designated sequential loop.

Example. The following code fragment contains a directive that requests that loop `j` be run serially:

```
for(i=0; i<m; i++) {
    #pragma serial
    for(j=0; j<n; j++)
        a[i][j] = b[i][j];
    ...
}
```

The directive applies only to the loop that immediately follows it. For example, APO still tries to parallelize loop `i`. This directive is useful in cases like this when the value of `n` is known to be very small.

15.4.4 `#pragma prefer concurrent`

The `#pragma prefer concurrent` directive instructs APO to parallelize the loop immediately following the directive if it is safe to do so.

Example. The following code fragment encourages APO to run loop `i` in parallel:

```
#pragma prefer concurrent
for(i=0; i<m; i++) {
    for(j=0; j<n; j++)
        a[i][j] = b[i][j];
    ...
}
```

When dealing with nested loops, APO follows these guidelines:

- If the loop specified by the `#pragma prefer concurrent` directive is safe to parallelize, APO parallelizes the specified loop even if other loops in the nest are safe.
- If the specified loop is not safe to parallelize, APO parallelizes a different loop that is safe.
- If this directive is applied to an inner loop, APO can interchange the loop and make the specified loop the outermost loop.
- If this directive is applied to more than one loop in a nest, APO parallelizes one of the specified loops.

15.4.5 #pragma permutation

When placed inside a function, the `#pragma permutation (`*array_name*`)` directive informs APO that *array_name* is a *permutation* array. A permutation array is one in which every element of the array has a distinct value.

The directive does not require the permutation array to be *dense*. That is, within the array, every `b[i]` must have a distinct value, but there can be gaps between the values, such as `b[1]` = 1, `b[2]` = 4, `b[3]` = 9, and so on.

Note: This directive affects compilation even when `-apo` is not specified.

Example. In the following code fragment, array `b` is declared to be a permutation array for both loops in `sub1()`:

```
void sub1(int n) {
    int i;
    extern int a[], b[];
    for(i=0; i<n; i++) {
         a[b[i]] = i;
    }
    #pragma permutation (b)
    for(i=0; i<n; i++) {
        a[b[i]] = i;
    }
}
```

Note the following points about this directive:

- As shown in the example, you can use this directive to parallelize loops that use arrays for indirect addressing. Without this directive, APO cannot determine that the array elements used as indexes are distinct.
- `#pragma permutation (`*array_name*`)` affects every loop in a function, even those that appear before it.

15.4.6 `#pragma no concurrentize`, `#pragma concurrentize`

The `#pragma no concurrentize` directive inhibits parallelization. Its effect depends on its placement.

- When placed inside functions, this directive inhibits parallelization. In the following example, no loops inside `sub1()` are parallelized:

  ```
  void sub1() {
  #pragma no concurrentize
      ...
  }
  ```

- When placed outside of a function, `#pragma no concurrentize` prevents the parallelization of all functions in the file, even those that appear ahead of it in the file. Loops inside functions `sub2()` and `sub3()` are not parallelized in the following example:

  ```
  void sub2() {
      ...
  }
  #pragma no concurrentize
  void sub3() {
      ...
  }
  ```

The `#pragma concurrentize` directive, when placed inside a function, overrides a `#pragma no concurrentize` directive that is placed outside of it. Thus, this directive allows you to selectively parallelize functions in a file that has been made sequential with a `#pragma no concurrentize` directive.

15.5 Troubleshooting Incomplete Optimizations

Some loops cannot be safely parallelized and others are written in ways that inhibit APO's efficiency. The following subsections describe the steps you can take to make APO more effective. The sections that follow, and the topics they discuss, are as follows:

- Section 15.5.1, page 148, describes constructs that inhibit parallelization.
- Section 15.5.2, page 151, describes constructs that reduce performance of parallelized code.

15.5.1 Constructs That Inhibit Parallelization

A program's performance can be severely constrained if APO cannot recognize that a loop is safe to parallelize. APO analyzes every loop in a program. If a loop does not appear safe, it does not parallelize that loop. The following sections describe constructs that can inhibit parallelization:

- Section 15.5.1.1, page 148, describes basic data dependencies.
- Section 15.5.1.2, page 148, describes function calls.
- Section 15.5.1.3, page 149, describes `goto` statements.
- Section 15.5.1.4, page 149, describes problematic array subscripts.
- Section 15.5.1.5, page 150, describes conditionally assigned local variables.

In many instances, loops containing the previous constructs can be parallelized after minor changes. Reviewing the information generated in program *file*`.list`, described in Section 15.2.1, page 138, can show you if any of these constructs are in your code.

15.5.1.1 Loops Containing Data Dependencies

Generally, a loop is safe if there are no data dependencies, such as a variable being assigned in one iteration of a loop and used in another. APO does not parallelize loops for which it detects data dependencies.

15.5.1.2 Loops Containing Function Calls

By default, APO does not parallelize a loop that contains a function call because the function in one iteration of the loop can modify or depend on data in other iterations.

You can, however, use interprocedural analysis (IPA) to provide the MIPSpro APO with enough information to parallelize some loops containing function calls. IPA is specified by the `-ipa` command line option. For more information on IPA, see `ipa`(5) and the *MIPSpro Compiling and Performance Tuning Guide*.

You can also direct APO to ignore function call dependencies when analyzing the specified loops by using the `#pragma concurrent call` directive described in Section 15.4.1, page 143.

15.5.1.3 Loops Containing `goto` Statements

`goto` statements are unstructured control flows. APO converts most unstructured control flows in loops into structured flows that can be parallelized. However, `goto` statements in loops can still cause the following problems:

- Unstructured control flows. APO is unable to restructure all types of flow control in loops. You must either restructure these control flows or manually parallelize the loops containing them.
- Early exits from loops. Loops with early exits cannot be parallelized, either automatically or manually.

For improved performance, remove `goto` statements from loops to be considered candidates for parallelization.

15.5.1.4 Loops Containing Problematic Array Constructs

The following array constructs inhibit parallelization and should be removed whenever APO is used:

- Arrays with subscripts that are indirect array references. APO cannot analyze indirect array references. The following loop cannot be run safely in parallel if the indirect reference `b[i]` is equal to the same value for different iterations of `i`:

  ```
  for(i=0; i<n; i++)
      a[b[i]] = ...
  ```

  If every element of array `b` is unique, the loop can safely be made parallel. To achieve automatic parallelism in such cases, use the `#pragma permutation(b)` directive, as discussed in Section 15.4.5, page 146.

- Arrays with unanalyzable subscripts. APO cannot parallelize loops containing arrays with unanalyzable subscripts. Allowable subscripts can contain the following elements:

  – Literal constants (`1`, `2`, `3`, ...)

  – Variables (`i`, `j`, `k`, ...)

  – The product of a literal constant and a variable, such as `n*5` or `k*32`

  – A sum or difference of any combination of the first three items, such as `n*21+k-251`

  In the following example, APO cannot analyze the division operator (`/`) in the array subscript and cannot reorder the loop:

  ```
  for(i=0; i<n; i+=2)
      a[i/2] = ...;
  ```

- Unknown information. In the following example there may be hidden knowledge about the relationship between variables `m` and `n`:

  ```
  for(i=0; i<n; i++)
      a[i] = a[i+m];
  ```

  The loop can be run in parallel if `m > n` because the array reference does not overlap. However, APO does not know the value of the variables and therefore cannot make the loop parallel. You can use the `#pragma concurrent` directive to have APO automatically parallelize this loop. For more information on this directive, see Section 15.4.2, page 144.

15.5.1.5 Loops Containing Local Variables

When parallelizing a loop, APO often localizes (privatizes) temporary scalar and array variables by giving each processor its own non-shared copy of them. In the following example, array `tmp` is used for local scratch space:

```
for(i=0; i<n; i++) {
    for(j=0; j<n; j++)
      tmp[j] = i+j;
    for(j=0; j<n; j++)
      a[i][j] = a[i][j] + tmp[j];
}
```

To successfully parallelize the outer loop (`i`), APO must give each processor a distinct, private copy of array `tmp`. In this example, it is able to localize `tmp` and, thereby, to parallelize the loop.

APO cannot parallelize a loop when a conditionally assigned temporary variable might be used outside of the loop, as in the following example:

```
extern int t;
for(i=0; i<n; i++) {
    if(b[i]) {
        t = ...;
        a[i] += t;
    }
}
s2();
```

If the loop were to be run in parallel, a problem would arise if the value of `t` were used inside function `s2()` because it is not known which processor's private copy of `t` should be used by `s2()`. If `t` were not conditionally assigned, the processor that executed iteration `i == n-1` would be used. Because `t` is conditionally assigned, APO cannot determine which copy to use.

The solution comes with the realization that the loop is inherently parallel if the conditionally assigned variable `t` is localized. If the value of `t` is not used outside the loop, replace `t` with a local variable. Unless `t` is a local variable, APO assumes that `s2()` might use it.

15.5.2 Constructs That Reduce Performance of Parallelized Code

APO parallelizes a loop by distributing its iterations among the available processors. Loop nesting, loops with low trip counts, and other program characteristics can affect the efficiency of APO. The following subsections describe the effect that these and other programming constructs can have on APO's ability to parallelize:

- Section 15.5.2.1, page 152, describes parallelizing nested loops.
- Section 15.5.2.2, page 153, describes parallelizing loops with small or indeterminate trip counts.
- Section 15.5.2.3, page 154, describes parallelizing loops that exhibit poor data locality.

15.5.2.1 Parallelizing Nested Loops

APO can parallelize only one loop in a loop nest. In these cases, the most effective optimization usually occurs when the outermost loop is parallelized. The effectiveness derives from that fact that more processors end up processing larger sections of the program. This saves synchronization and other overhead costs.

Example 1. Consider the following simple loop nest:

```
for(i=0; i<n; i++)
    for(j=0; j<m; j++)
        for(k=0; k<l; k++)
           ...
```

When parallelizing nested loops `i`, `j`, and `k`, APO parallelizes only one of the loops. Effective loop nest parallelization depends on the loop that APO chooses, but it is possible for APO to choose an inferior loop to be parallelized. APO may attempt to interchange loops to make a more promising one the outermost. If the outermost loop attempt fails, APO attempts to parallelize an inner loop.

Section 15.2.1, page 138, describes *file*`.list`. This output file contains information that tells you which loop in a nest was parallelized. Because of the potential for improved performance, it is useful for you to modify your code so that the outermost loop is the one parallelized.

For every loop that is parallelized, APO generates a test to determine whether the loop is being called from within either another parallel loop or from within a parallel region. In some cases, you can minimize the extra testing that APO must perform by inserting directives into your code to inhibit parallelization testing. The following example demonstrates this:

Example 2:

```
void sub(int i, int n) {
   int j;
   #pragma serial
   for(j=0; j<n; j++) {
       ...
   }
}
void caller(int n) {
    int i;
    #pragma concurrent call
    for(i=0; i<n; i++) {
        sub(i, n);
    }
}
```

Assume that `sub()` is called only from within `caller()`. The loop in `caller()` is parallelized, so the loop in `sub()` can never be run in parallel. In this case, the test is avoided by using the `#pragma serial` directive, as shown, to force the sequential execution of the loop.

For more information on this compiler directive, see Section 15.4.3, page 145.

15.5.2.2 Parallelizing Loops With Small Or Indeterminate Trip Counts

The *trip count* is the number of times a loop is executed. Loops with large trip counts are the best candidates for parallelization. The following paragraphs show how to modify your program if your program contains loops with small trip counts or loops with indeterminate trip counts:

- Loops with small trip counts generally run faster when they are not parallelized. Consider the following loop nest:

  ```
  #pragma prefer serial
  for(i=0; i<m; i++) {
    for(j=0; j<n; j++) {
      ...
    }
  }
  ```

  Without the directive, APO would attempt to parallelize loop `i` because it is outermost. If `m` is very small, it would be better to interchange the loops and make loop `j` outermost, so that it would be parallelized. If that is not possible, and if APO cannot determine that `m` is small, you can use a `#pragma prefer serial` directive, as shown, to indicate to APO that it is better to parallelize loop `j`.

- Loops with large trip counts run faster if they are unconditionally parallelized. Consider the following loop:

  ```
  #pragma prefer concurrent
  for(j=0; j<n; j++)
    ...
  ```

Without the directive, if the trip count is not known (and sometimes even if it is), APO parallelizes the loop conditionally. It generates code for both a parallel and a sequential version of the loop, plus code to select the version to use, based on the trip count, the code inside the loop's body, the number of processors available, and an estimate of the cost to invoke a parallel loop in that run-time environment.

You can avoid the overhead of conditional parallelization by using the `#pragma prefer concurrent` directive, as shown, to indicate to APO that only the parallel version of the loop should be generated.

15.5.2.3 Parallelizing Loops With Poor Data Locality

Computer memory has a hierarchical organization. Higher up the hierarchy, memory becomes closer to the CPU, faster, more expensive, and more limited in size. Cache memory is at the top of the hierarchy, and main memory is further down in the hierarchy. In multiprocessor systems, each processor has its own cache memory. Because it is time consuming for one processor to access another processor's cache, a program's performance is best when each processor has the data it needs in its own cache.

Programs, especially those that include extensive looping, often exhibit *locality of reference*, which means that if a memory location is referenced, it is probable that it or a nearby location will be referenced in the near future. Loops designed to take advantage of locality do a better job of concentrating data in memory, increasing the probability that a processor will find the data it needs in its own cache.

The following examples show the effect of locality on parallelization. Assume that the loops are to be parallelized and that there are *p* processors.

Example 1. Distribution of Iterations.

```
for(i=0; i<n; i++) {
    ...a[i]...
}
for(i=n-1; i>=0; i--) {
    ...a[i]...
}
```

In the first loop, the first processor accesses the first `n/`*p* elements of `a`; the second processor accesses the next `n/`*p* elements; and so on. In the second loop, the distribution of iterations is reversed. That is, the first processor accesses the last `n/`*p* elements of `a`, and so on. Most elements are not in the cache of the processor needing them during the second loop. This code fragment would run more efficiently, and be a better candidate for parallelization, if you reverse the direction of one of the loops.

Example 2. Two Nests in Sequence.

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
      a[i][j] = b[j][i] + ...;

for(i=0; i<n; i++)
    for(j=0; j<n; j++)
      b[i][j] = a[j][i] + ...;
```

In example 2, APO may parallelize the outer loop of each member of a sequence of nests. If so, while processing the first nest, the first processor accesses the first `n/`*p* rows of `a` and the first `n/`*p* columns of `b`. In the second nest, the first processor accesses the first `n/`*p* columns of `a` and the first `N/`*p* rows of `B`. This example runs much more efficiently if you parallelize the `i` loop in one nest and the `j` loop in the other. You can instruct APO to do this by inserting a `#pragma prefer serial` directive just prior to the `i` loop that contains the `j` loop that you want to be parallelized.

Index