SiliconGraphics
Computer Systems

MIPSpro™ C and C++ Pragmas

MIPSpro C and C++ Pragmas

SiliconGraphics
Computer Systems

007-3587-001

MIPSpro™ C and C++ Pragmas

Document Number 007-3587-001

CONTRIBUTORS

Written by Sandra Motroni
Edited by Christina Cary
Production by Carlos Miqueo
Engineering contributions by Chandrasekhar Murthy and Rohit Chandra

MIPSpro™ C and C++ Pragmas
Document Number 007-3587-001

Contents

List of Tables

List of Figures

Using This Guide

How This Book Is Organized

This book is organized for easy navigation. For an easy-to-use chart on navigating through this book, see Table i.

Chapter 1, “Alphabetical Listing of C and C++ Pragmas,” contains a table of all pragmas in this book, arranged in alphabetical order. For each pragma there is a brief description, along with a link to the chapter in which it is discussed.

Chapters 2 through 12 each contain descriptions of all of the pragmas within one functional group. At the beginning of each chapter is a table listing all pragmas discussed within the chapter (in alphabetical order), along with a short description, and a link to the longer description. The titles of the chapters are as follows:

- Chapter 2, “C++ Instantiation Pragmas”
- Chapter 3, “Data Layout Pragmas”
- Chapter 4, “Distributed Shared Memory (DSM) Optimization Pragmas”
- Chapter 5, “Inlining Pragmas”
- Chapter 6, “Loader Information Pragmas”
- Chapter 7, “Loop Nest Optimization Pragmas”
- Chapter 8, “Multiprocessing Pragmas”
- Chapter 9, “Precompiled Header Pragmas”
- Chapter 10, “Scalar Optimization Pragmas”
- Chapter 11, “Warning Suppression Control Pragmas”
- Chapter 12, “Miscellaneous Pragmas”

Navigating Through This Guide

Table i describes the best way to navigate through this guide.

Table i Navigating Through This Guide

If you...	Then start with...	Which contains...
Know the name of the pragma you wish to look up.	Chapter 1, "Alphabetical Listing of C and C++ Pragmas"	- An alphabetical table of all pragmas in this book. - A brief description of each pragma. - A link to the chapter containing more information.
Do not know the name of the pragma, but do know the functional group to which it belongs.	The table of contents	A list of all chapters in the book. Pragmas are broken down into functional groups, with one group covered in each chapter.
Just want to browse.	Chapter 1, "Alphabetical Listing of C and C++ Pragmas"	- An alphabetical table of all pragmas in this book. - A brief description of each pragma. - A link to the chapter containing more information.
	OR The table of contents	A list of all chapters in the book. Pragmas are broken down into functional groups with one group covered in each chapter.

Typographical Conventions Used in This Guide

The conventions used in this guide help make information easy to access and understand. The following list defines the notation and syntax conventions:

[ ] (brackets)	Enclose optional command arguments. Do not enter the brackets.
. . . (ellipses)	Indicates that the preceding optional items can appear more than once in succession.
( ) (parentheses)	Enclose items. Enter the text exactly as shown, including the parentheses.
{ } (braces)	Enclose items from which you must select exactly one. Do not enter the braces.
| (vertical bar)	Separates items from which you can choose one.
italic	Indicates arguments in a command line that you must replace with a valid value. In text, it is used to indicate document titles, filenames, and variables. In code examples, it is used to indicate variables.
`courier`	Indicates computer output and program listings.
Bold	Indicates command-line options.

Chapter 1

Alphabetical Listing of C and C++ Pragmas

Table 1-1 is an alphabetical list of Silicon Graphics® supported pragmas, with a short description of each and a link to the chapter where the pragma is discussed.

Table 1-1 Silicon Graphics Pragmas

#pragma	Short Description	Functional Group
aggressive inner loop fission	Fission inner loops into as many loops as possible.	Chapter 7, "Loop Nest Optimization Pragmas"
align_symbol	Specifies alignment of user variables, typically at cache-line or page boundaries.	Chapter 3, "Data Layout Pragmas"
blocking size	Sets the blocksize of the specified loop that is involved in a blocking for the primary (secondary) cache.	Chapter 7, "Loop Nest Optimization Pragmas"
can_instantiate	Indicates that the specified declaration can be instantiated in the current compilation, but need not be.	Chapter 2, "C++ Instantiation Pragmas"
copyin	Copies the value from the master thread's version of an **-Xlocal**-linked global variable into the slave thread's version.	Chapter 8, "Multiprocessing Pragmas"
critical	Protects access to critical statements.	Chapter 8, "Multiprocessing Pragmas"
distribute	Specifies data distribution.	Chapter 4, "Distributed Shared Memory (DSM) Optimization Pragmas"

Table 1-1 Silicon Graphics Pragmas

#pragma	Short Description	Functional Group
distribute_reshape	Specifies data distribution with reshaping.	Chapter 4, "Distributed Shared Memory (DSM) Optimization Pragmas"
do_not_instantiate	Prevents instantiation of the specific declaration in this compilation unit, even if that instance is used in the code.	Chapter 2, "C++ Instantiation Pragmas"
dynamic	Tells the compiler that the specified array may be redistributed in the program.	Chapter 4, "Distributed Shared Memory (DSM) Optimization Pragmas"
enter gate	Indicates the point that all threads must clear before any threads are allowed to pass the corresponding **#pragma exit gate**.	Chapter 8, "Multiprocessing Pragmas"
exit gate	Stops threads from passing this point until all threads have cleared the corresponding **#pragma enter gate**.	Chapter 8, "Multiprocessing Pragmas"
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 3, "Data Layout Pragmas"
fission	Fission the enclosing specified levels of loops after this pragma.	Chapter 7, "Loop Nest Optimization Pragmas"
fissionable	Disables validity testing.	Chapter 7, "Loop Nest Optimization Pragmas"
fusable	Disables validity testing.	Chapter 7, "Loop Nest Optimization Pragmas"
fuse	Fuse the following specified number of loops, which must be immediately adjacent.	Chapter 7, "Loop Nest Optimization Pragmas"

Table 1-1 Silicon Graphics Pragmas

#pragma	Short Description	Functional Group
hdrstop	Indicates the point at which the precompiled header mechanism snapshots the headers. If **–pch** is off, **#pragma hdrstop** is ignored.	Chapter 9, "Precompiled Header Pragmas"
hidden	Tells the compiler that the specified symbols are invisible to all executables or DSOs except the current one.	Chapter 6, "Loader Information Pragmas"
ident	Adds a *.comment* section in the object file and puts the revision string inside the *.comment* section.	Chapter 12, "Miscellaneous Pragmas"
independent	Tells the compiler to run an independent code section in parallel with the rest of the code in the parallel region.	Chapter 8, "Multiprocessing Pragmas"
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 5, "Inlining Pragmas"
instantiate	Causes a specified instance of a template declaration to be immediately instantiated at that spot.	Chapter 2, "C++ Instantiation Pragmas"
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 12, "Miscellaneous Pragmas"

Table 1-1 Silicon Graphics Pragmas

#pragma	Short Description	Functional Group
internal	Tells the compiler that the specified symbols are not referenced outside the current executable or DSO.	Chapter 6, "Loader Information Pragmas"
intrinsic	Allows certain preselected functions from *math.h*, *stdio.h*, and *string.h* to be inlined at a callsite for execution efficiency.	Chapter 12, "Miscellaneous Pragmas"
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 7, "Loop Nest Optimization Pragmas"
local	Tells the compiler the names of all the variables that must be local to each thread.	Chapter 8, "Multiprocessing Pragmas"
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 10, "Scalar Optimization Pragmas"
no blocking	Prevents the compiler from involving this loop in cache blocking.	Chapter 7, "Loop Nest Optimization Pragmas"
no_delete	Inhibits deletion of functions that are never referenced.	Chapter 7, "Loop Nest Optimization Pragmas"
no fission	Keeps the following loop from being fissioned. Its innermost loops, however, are allowed to be fissioned.	Chapter 7, "Loop Nest Optimization Pragmas"
no fusion	Keeps the following loop from being fused with other loops.	Chapter 7, "Loop Nest Optimization Pragmas"

Table 1-1 Silicon Graphics Pragmas

#pragma	Short Description	Functional Group
no interchange	Prevents the compiler from involving the loop directly following this pragma (or any loop nested within this loop) in an interchange.	Chapter 7, "Loop Nest Optimization Pragmas"
no side effects	Tells the compiler to assume that all of the named functions are safe to execute concurrently.	Chapter 8, "Multiprocessing Pragmas"
no_pch	Disables the precompiled header mechanism.	Chapter 9, "Precompiled Header Pragmas"
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 5, "Inlining Pragmas"
once	Ensures (in **-n32** and **–64** mode) that each *include* file is included at most one time in each compilation unit.	Chapter 9, "Precompiled Header Pragmas"
one processor	Causes next statement to be executed on only one processor.	Chapter 8, "Multiprocessing Pragmas"
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 6, "Loader Information Pragmas"

Table 1-1 Silicon Graphics Pragmas

#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 3, "Data Layout Pragmas"
page_place	Controls the placement of data on a DSM (distributed shared memory) machine.	Chapter 4, "Distributed Shared Memory (DSM) Optimization Pragmas"
parallel	Starts a parallel region.	Chapter 8, "Multiprocessing Pragmas"
pfor	Marks a *for* loop to run in parallel.	Chapter 8, "Multiprocessing Pragmas"
prefetch	Controls prefetching for each level of the cache.	Chapter 7, "Loop Nest Optimization Pragmas"
prefetch_manual	Specifies whether manual prefetches (through pragmas) should be respected or ignored.	Chapter 7, "Loop Nest Optimization Pragmas"
prefetch_ref	Generates a prefetch and connects it to the specified reference (if possible).	Chapter 7, "Loop Nest Optimization Pragmas"
prefetch_ref_disable	Explicitly disables prefetching for the specified reference.	Chapter 7, "Loop Nest Optimization Pragmas"
protected	Tells the compiler that the specified symbols are not preemptible.	Chapter 6, "Loader Information Pragmas"
redistribute	Specifies dynamic data redistribution.	Chapter 4, "Distributed Shared Memory (DSM) Optimization Pragmas"

Table 1-1 Silicon Graphics Pragmas

#pragma	Short Description	Functional Group
reset woff	Resets listed warnings to the state specified in the command line.	Chapter 11, "Warning Suppression Control Pragmas"
section_gp	Causes an object to be placed in a gp_relative section.	Chapter 6, "Loader Information Pragmas"
section_non_gp	Keeps an object from being placed in a gp_relative section.	Chapter 6, "Loader Information Pragmas"
set chunksize	Tells the compiler which values to use for *chunksize*.	Chapter 8, "Multiprocessing Pragmas"
set numthreads	Tells the compiler which values to use for *numthreads*.	Chapter 8, "Multiprocessing Pragmas"
set schedtype	Tells the compiler which values to use for *schedtype*.	Chapter 8, "Multiprocessing Pragmas"
set woff	Suppresses listed compiler warnings.	Chapter 11, "Warning Suppression Control Pragmas"
shared	Tells the compiler the names of all the variables that the threads must share.	Chapter 8, "Multiprocessing Pragmas"
synchronize	Stops threads until all threads reach this point. This pragma is a classic barrier construct.	Chapter 8, "Multiprocessing Pragmas"

Table 1-1 Silicon Graphics Pragmas

#pragma	Short Description	Functional Group
unroll	Suggests to the compiler that *n*-1 copies of the loop body be added to the inner loop. If the loop following this pragma is an inner loop, then it indicates standard unrolling (version 7.2 and later). If the loop following this pragma is not innermost, then outer loop unrolling (unroll and jam) is performed (version 7.0 and later).	Chapter 7, "Loop Nest Optimization Pragmas"
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 6, "Loader Information Pragmas"
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 6, "Loader Information Pragmas"

Chapter 2

C++ Instantiation Pragmas

Instantiation pragmas control the instantiation of specific template entities or sets of template entities.

Table 2-1 lists the C++ instantiation pragmas covered in this chapter, along with a brief description of each and the compiler versions in which the pragma is supported.

Table 2-1 C++ Template Instantiation

#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

#pragma instantiate

#pragma instantiate causes a specific instance of a template declaration to be immediately instantiated at that spot.

Using #pragma instantiate

The syntax of the **instantiate** pragma is as follows:

`#pragma 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 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 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 **instantiate** pragma 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.

#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.

Using #pragma can_instantiate

The syntax of the **can_instantiate** pragma 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 **can_instantiate** pragma 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)
```

#pragma do_not_instantiate

The **#pragma do_not_instantiate** directive is the opposite of **#pragma instantiate**: if this pragma is present, the compiler will not instantiate the specified declaration in this compilation unit, even if you use that instance in your code.

Using #pragma do_not_instantiate

The syntax of the **do_not_instantiate** pragma 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 **do_not_instantiate** pragma 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 Pragmas

Table 3-1 lists the pragmas covered in this chapter, along with a short description of each and the compiler versions in which the pragma is supported.

Table 3-1 Data Layout Pragmas

#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

#pragma align_symbol

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

Using #pragma align_symbol

The syntax of the **align_symbol** pragma is as follows:

```
#pragma align_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, which must be a power of two

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

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

Cautions for Using #pragma align_symbol

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

- The **align_symbol** pragma is ineffective for local variables of fixed-size symbols, such as simple scalars or arrays of known size. The pragma 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.

Example of #pragma align_symbol

The following code fragment illustrates the use of the **align_symbol** pragma:

```
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. */
```

#pragma fill_symbol

The **#pragma fill_symbol** directive tells 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.

Using #pragma fill_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 **fill_symbol** pragma 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 **fill_symbol** pragma 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 pragma must be specified where the variable is defined. The pragma is optional where the variable is declared.

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

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. */
```

#pragma pack

This pragma 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).

Using #pragma pack

The syntax of the **pack** pragma is as follows:

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

The **pack** pragma works according to the following rules:

- A **struct** type defined in the scope of a **#pragma pack** has at most an alignment of *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 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 pragma to a lexically earlier instance of the pragma.

Cautions for Using #pragma pack

- Silicon Graphics strongly discourages the use of **#pragma pack**, because it is a nonportable feature and the semantics of this pragma may change in future compiler releases.
- A structure declaration must be subjected to identical instances of a **#pragma pack** 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 **pack** pragma is not supported for C++ in **-n32** and **-64** modes.

Chapter 4

Distributed Shared Memory (DSM) Optimization Pragmas

Table 4-1 lists the pragmas discussed in this chapter, along with a short description of each and the compiler versions in which the pragma is supported. These pragmas are useful primarily on systems with distributed shared memory, such as Origin™ servers.

Table 4-1 Distributed Shared Memory Pragmas

#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 8, "Multiprocessing Pragmas.")	**affinity** clause allows data-affinity or thread-affinity scheduling; **nest** clause exploits nested concurrency. See "#pragma pfor clauses."	6.0 and later
"#pragma redistribute"	Specifies dynamic redistribution of data.	7.2 and later

#pragma distribute

The **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 "affinity: Specifying Thread and Data Affinity" in Chapter 8 for more information about data affinity.

Using #pragma distribute

The syntax of the **distribute** pragma is as follows:

`#pragma distribute` *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 "onto Clause" on page 23 for more information).

The following are some further points about **#pragma distribute**:

- You must specify the **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.

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.

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)
```

#pragma distribute_reshape

The **distribute_reshape** directive, like **#pragma distribute** specifies the desired distribution of an array. In addition, however, the **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 "affinity: Specifying Thread and Data Affinity" on page 104.

Using #pragma distribute_reshape

The syntax of the **distribute_reshape** pragma is as follows:

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

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

- *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**.

The following are some further points about **#pragma distribute_reshape**:

- You must specify the **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 two options.

    The first option is to specify the **distribute_reshape** pragma directly on the formal parameter of the called function.

    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 **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).

Cautions for Using #pragma distribute_reshape

Because the **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 **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.

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.

#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 "affinity: Specifying Thread and Data Affinity" on page 104.

Using #pragma dynamic

The syntax of the **dynamic** pragma is as follows:

```
#pragma dynamic array
```

array is the name of the array in question.

The **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 **dynamic** directive only if it is actually necessary.

You must explicitly specify the **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 **dynamic** directive and just supply the **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.

#pragma page_place

The **page_place** pragma is useful for dealing with irregular data structures. It allows you to explicitly place data in the physical memory of a particular processor. This pragma is often used in conjunction with thread affinity (see "affinity: Specifying Thread and Data Affinity" in Chapter 8 for more information).

Using #pragma page_place

The syntax of the **page_place** pragma is as follows:

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

The parameters for this pragma are as follows:

object	The object you wish to place
size	The size in bytes
threadnum	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.

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 **page_place** pragma 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").

#pragma redistribute

The **#pragma redistribute** directive allows you to dynamically redistribute previously distributed arrays. For information about using data affinity with **#pragma redistribute**, see "affinity: Specifying Thread and Data Affinity" on page 104.

Using #pragma redistribute

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

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

The **redistribute** directive accepts the same distributions as the **#pragma distribute** directive:

- *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 "onto Clause" on page 23 for more information).

The following are some further points 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.

onto Clause

The **onto** clause for the **redistribute** pragma is identical to the one for the **distribute** pragma. See "onto Clause" on page 23 for more information.

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

Chapter 5

Inlining Pragmas

Table 5-1 lists the pragmas covered in this chapter, along with a brief description of each.

Table 5-1 Inlining Pragmas

#pragmas	Short Description	Compiler Versions
#pragma inline (see "#pragma inline and #pragma noinline")	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 "#pragma inline and #pragma noinline")	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

#pragma inline and #pragma noinline

The **inline** and **noinline** pragmas tell the compiler whether or not to inline the named functions. These pragmas can have next-line, entire routine, or global scope.

Using #pragma inline and #pragma noinline

The syntax of the **inline** and **noinline** pragmas is as follows:

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

here, **routine**, and **global** are keywords (see "Keywords").

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 is not in the universe, 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.

Keywords

The keywords, **here**, **routine**, and **global** are described below. These keywords must appear in lowercase, because function names are case sensitive.

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

Caution for Using #pragma inline and #pragma noinline

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.

Examples of #pragma inline and #pragma noinline

The following five examples illustrate different aspects of the **inline** and **noinline** pragmas.

Example 5-1 Using the **here** keyword with the **noinline** pragma

This example illustrates the use of the **noinline** pragma with the **here** keyword. All occurrences of **f1(int)** are marked for inlining, except the one directly following the **#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 5-2 Using the **here** keyword with the **inline** and **noinline** pragmas

This example illustrates the use of the **inline** and **noinline** pragmas with the **here** keyword. All occurrences of **f1(int)** are marked for inlining, except the one directly following the **#pragma noinline here**. The only occurrence of **f1(double)** that is marked for inlining is the one directly following the **#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 5-3 Using the **global** keyword with the **inline** pragma

This example illustrates the use of the **inline** pragma 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 5-4 Using the **routine** keyword with the **inline** pragma

This example illustrates the use of the **inline** pragma with the **routine** keyword. All occurrences of **f1(int)** following the **#pragma inline routine** 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 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 5-5 Using the **routine** keyword with the **noinline** pragma

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

Chapter 6

Loader Information Pragmas

Table 6-1 lists the pragmas covered in this chapter, along with a brief description of each and the compiler versions in which the pragma is supported.

Table 6-1 Loader Information Pragmas

#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

Table 6-1 Loader Information Pragmas

#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

#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.

Using #pragma hidden

The syntax of the **hidden** pragma 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 can not 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.

#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.

Using #pragma internal

The syntax of the **internal** pragma 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.

#pragma no_delete

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

Using #pragma no_delete

The syntax of the **no_delete** pragma is as follows:

```
#pragma no_delete
```

#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 family of releases.

Using #pragma optional

The syntax of the **optional** pragma 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"`.

Caution for Using #pragma optional

The **optional** will not be fully implemented until the 7.2.1 compiler release. Until then, use it cautiously.

#pragma protected

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

Using #pragma protected

The syntax of the **protected** pragma 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 can not be preempted by another definition.

#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.

Using #pragma section_gp

The syntax of the **section_gp** pragma is as follows:

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

symbol must be a static or global variable.

#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.

Using #pragma section_non_gp

The syntax of the **section_non_gp** pragma is as follows:

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

symbol must be a static or global variable.

#pragma weak

The **#pragma weak** directive can be used in two ways. It can tell the link editor not to 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.

About Weak Definitions

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.

Using #pragma weak

The syntax of the **weak** pragma 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 **weak** pragma tells the link editor not to 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 in libraries. If encountered, the defining declaration preempts the weak denotation.

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

  – Define the *strong_symbol* within the same compilation unit in which the pragma 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 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 (which admittedly makes no attempt at good style) 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 for Using #pragma weak

#pragma weak is not supported in **-32** C++.

Chapter 7

Loop Nest Optimization Pragmas

Table 7-1 contains an alphabetical list of the pragmas covered in this chapter, along with a brief description of each and the compiler versions in which the pragma is supported.

Table 7-1 Loop Nest Optimization Pragmas

#pragma	Short Description	Compiler Versions
"#pragma aggressive inner loop fission"	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 pragma.	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

Table 7-1 (continued) Loop Nest Optimization Pragmas

#pragma	Short Description	Compiler Versions
"#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 no interchange"	Prevents the compiler from involving the loop directly following this pragma (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 pragma.	7.1 and later
"#pragma prefetch_manual"	Specifies whether manual prefetches (through pragmas) should be respected or ignored. Scope: entire function containing the pragma.	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 pragma is an inner loop, then it indicates standard unrolling (version 7.2 and later). If the loop following this pragma is not innermost, then outer loop unrolling (unroll and jam) is performed (version 7.0 and later).	7.0 and later

#pragma aggressive inner loop fission

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

Using #pragma aggressive inner loop fission

The syntax of the **aggressive inner loop fission** pragma is as follows:

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

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

#pragma blocking size

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

Using #pragma blocking size

The syntax of the **blocking size** pragma 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.

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)
        }
    }
}
```

#pragma no blocking

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

Using #pragma no blocking

The syntax of the **no blocking** pragma is as follows:

```
#pragma no blocking
```

#pragma fission

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

Using #pragma fission

The syntax of the **fission** pragma is as follows:

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

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

#pragma fissionable

The **fissionable** pragma disables validity testing for loop fissioning.

Using #pragma fissionable

The syntax of the **fissionable** pragma is as follows:

```
#pragma fissionable
```

#pragma no fission

The **#pragma no fission** tells the compile that the loop directly following this pragma should not be fissioned. Any inner loops, however, are allowed to be fissioned.

Using #pragma no fission

The syntax of the **no fission** pragma is as follows:

```
#pragma no fission
```

#pragma fuse

The **#pragma fuse** directive tells the compiler to fuse the specified number of immediately adjacent loops.

Using #pragma fuse

The syntax of the **fuse** pragma is as follows:

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

The loops to be fused must immediately follow the **fusion** pragma.

The default is 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.

#pragma fusable

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

Using #pragma fusable

The syntax of the **fusable** pragma is as follows:

```
#pragma fusable
```

#pragma no fusion

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

Using #pragma no fusion

The syntax of the **no fusion** pragma is as follows:

```
#pragma no fusion
```

#pragma no interchange

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

Using #pragma no interchange

The syntax of the **no interchange** pragma is as follows:

```
#pragma no interchange
```

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

#pragma ivdep

The **#pragma ivdep** directive causes the compiler to liberalize dependence analysis.

Using #pragma ivdep

The syntax of the **ivdep** pragma is as follows:

```
#pragma ivdep
```

Given two memory references, where at least one is loop variant, this pragma causes 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.

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's 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** tells the compiler to use Cray semantics and break all backward dependences:

- **ivdep** breaks the dependence but the compiler warns the user that it's 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 dependence 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 **–OPT:liberal_ivdep=TRUE**.

#pragma prefetch

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

Using #pragma prefetch

The syntax of the **prefetch** pragma 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 off (default for all processors except R10000)
1	prefetching on, but conservative (default at **–03** when prefetch is on)
2	prefetching on, and aggressive

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

#pragma prefetch_manual

The **#pragma prefetch_manual** directive tells the compiler whether manual prefetches (through pragmas) should be respected or ignored.

Using #pragma prefetch_manual

The syntax of the **prefetch_manual** pragma 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 **–03** for R10000 and beyond)

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

#pragma prefetch_ref

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

Using #pragma prefetch_ref

The syntax of the **prefetch_ref** pragma 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 7-2 describes each of the possible **#pragma prefetch_ref** clauses. These clauses are optional.

Table 7-2 Clauses for #pragma prefetch_ref

Clause	Effect	Default Value
`stride`	Prefetches every *num* iterations 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** pragma causes 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 pragma has no scope; it just generates a prefetch.

#pragma prefetch_ref_disable

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

Using #pragma prefetch_ref_disable

The syntax of the **prefetch_ref_disable** pragma 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 pragma is the entire function containing it.

#pragma unroll

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

Using #pragma unroll

The syntax of the **unroll** pragma is as follows:

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

This pragma suggests to the compiler that *n*-1 copies of the loop body be added to the inner loop. If the loop that this pragma directly precedes is an inner loop, then it indicates standard unrolling (version 7.2 and later). If the loop that this pragma directly 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 exactly 1, then no unrolling is performed.

Caution for Using #pragma unroll

#pragma unroll 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 wish to use restrict pointers (see the *C Language Reference Manual*) or the option **-OPT:alias=disjoint**. When **-OPT:alias=disjoint** is specified, distinct pointer expressions are assumed to point to distinct, non-overlapping objects.

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

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 **unroll** pragma 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];
}
```

Chapter 8

Multiprocessing Pragmas

Table 8-1 contains an alphabetical list of the pragmas covered in this chapter, along with a brief description of each and the compiler versions in which the pragma is supported.

Table 8-1 Multiprocessing Pragmas

#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 "#pragma enter gate and #pragma exit gate")	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 "#pragma enter gate and #pragma exit gate")	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 "#pragma parallel clauses")	Marks the start of a parallel region.	6.0 and later

Table 8-1 Multiprocessing Pragmas

#pragma	Short Description	Compiler Versions
"#pragma pfor" (see also "#pragma pfor clauses")	Marks a *for* loop to run in parallel.	6.0 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

#pragma copyin

It is occasionally desirable to be able to copy values from the master thread's version of an **-Xlocal**-linked global variable into the slave thread's version. The special pragma **copyin** allows this.

Using #pragma copyin

#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 pragma inside a parallel region.

Example of #pragma copyin

The following line of code demonstrates the use of the **copyin** pragma:

```
#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 pragma is translated into executable code, so in this example *i* is evaluated at the time this statement is executed.

#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 sometimes use a lock on the statement to preserve the integrity of the data.

Using #pragma critical

The syntax of the **critical** pragma is as follows:

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

The statements after the **critical** pragma are executed by all threads, one at a time.

In the multiprocessing C/C++ compiler, you can use the **critical** pragma 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 **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 don't 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 for Using #pragma critical

This pragma 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.

Diagram of #pragma critical

Figure 8-1 illustrates critical segment execution.

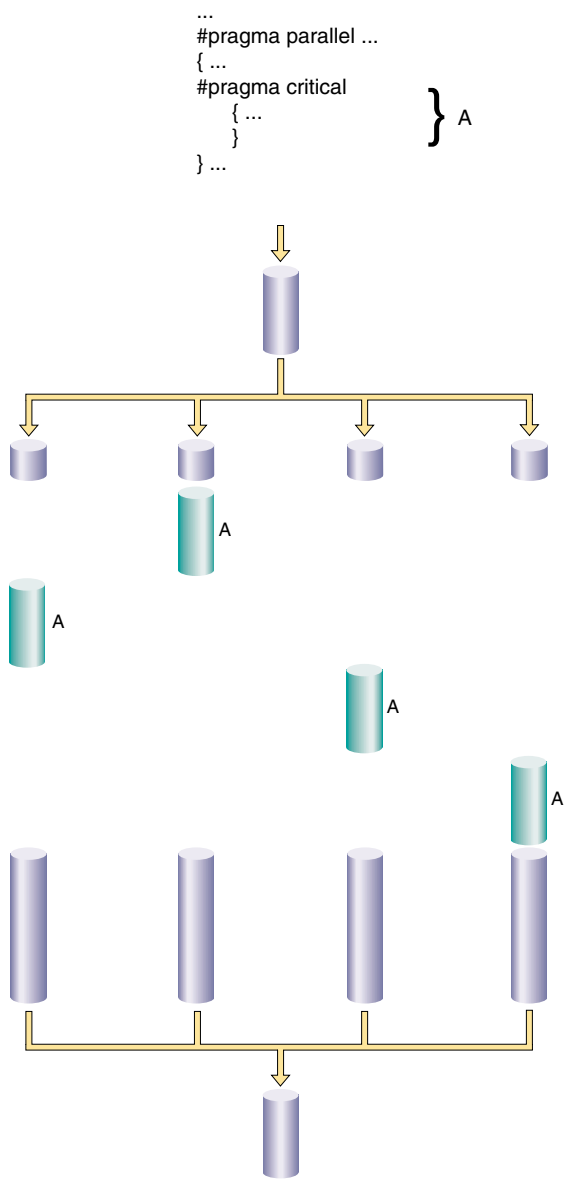

Figure 8-1 Critical Segment Execution

#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 pragmas 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.

Using #pragma enter gate and #pragma exit gate

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

#pragma enter gate

The syntax of the **enter gate** pragma is as follows:

```
#pragma enter gate
```

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

#pragma exit gate

The syntax of the **exit gate** pragma is as follows:

```
#pragma exit gate
```

Put this pragma 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 **enter gate** and **exit gate** pragmas is not supported.

Caution for Using #pragma enter gate and #pragma exit gate

These pragmas 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

Diagram of #pragma enter gate and #pragma exit gate

Figure 8-2 is a "time-lapse" sequence showing execution using enter and exit gates.

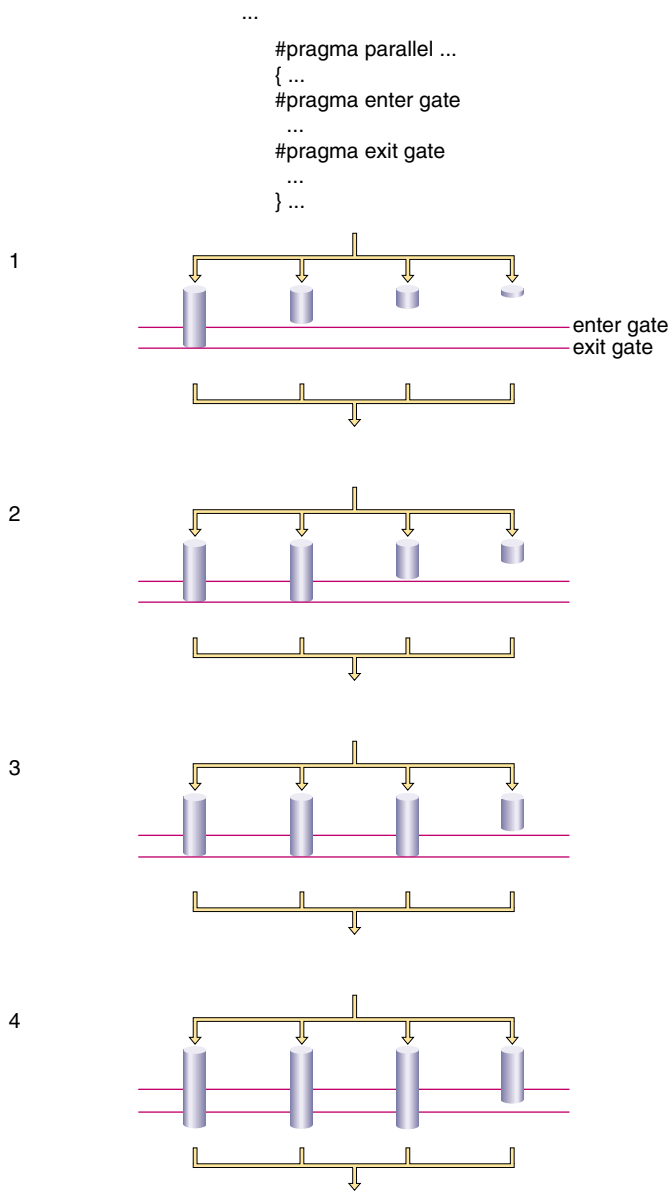

Figure 8-2 Execution Using Gates

Example of #pragma enter gate and #pragma exit gate

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

For example, suppose you have a parallel region consisting of the work-sharing constructs A, B, C, D, E, and so forth. A dependency may exist between B and E such that you can not execute E until all the work on B has completed (see code at right).

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

One option is to put a **synchronize** before E. But this pragma 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 dependency, put **#pragma enter gate** after B and **#pragma exit gate** before E. Putting the **enter gate** after B tells the system to note which threads have completed the B work-sharing construct. Putting the **exit gate** pragma prior to the E work sharing construct tells the system to allow no thread into E until all threads have cleared B.

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

#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.)

Using #pragma independent

The syntax of the **independent** pragma is as follows:

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

The **independent** pragma has no modifiers. Use this pragma 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.

Diagram of #pragma independent

Figure 8-3 shows an independent segment with execution by only one thread.

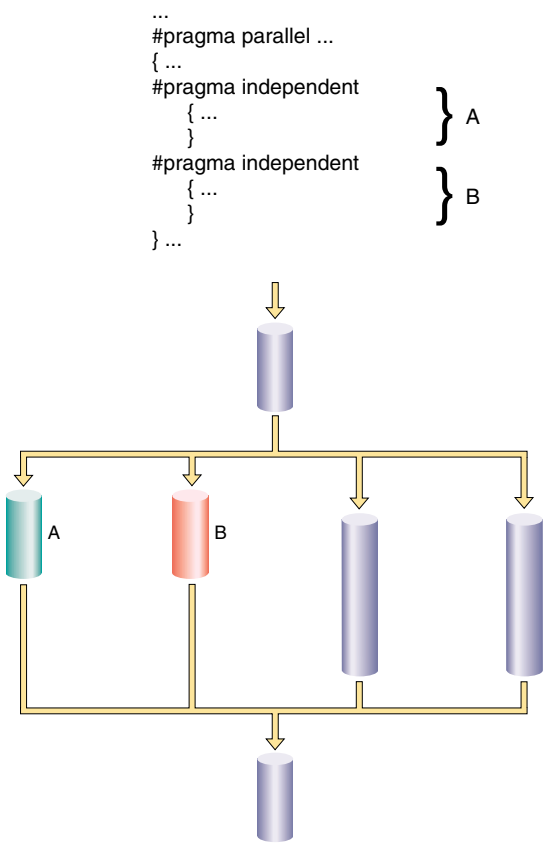

Figure 8-3 Independent Segment Execution

#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.

Using #pragma local

The syntax of the **local** pragma 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.

#pragma no side effects

C functions frequently produce more information than just the returned value. Changing values of arguments via pointers or arrays, changing global data, and I/O can make a function unsafe to run concurrently.

The **#pragma no side effects** pragma tells the compiler to assume that all of the named functions are safe to execute concurrently. This means that the functions perform no I/O and that they modify only local variables.

Using #pragma no side effects

The syntax of the **no side effects** pragma 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"`.

If you use this directive and you pass pointers or array names to any listed function, the compiler assumes that the memory locations represented by those parameters are not modified.

#pragma one processor

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

Using #pragma one processor

The syntax of the **one processor** pragma is as follows:

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

If a thread is executing the statement enclosed by this pragma, 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 pragma, then all threads encountering this statement skip the statement and continue without pause.

Diagram of #pragma one processor

Figure 8-4 shows code executed by only one thread. No thread can proceed past this code until it has been executed.

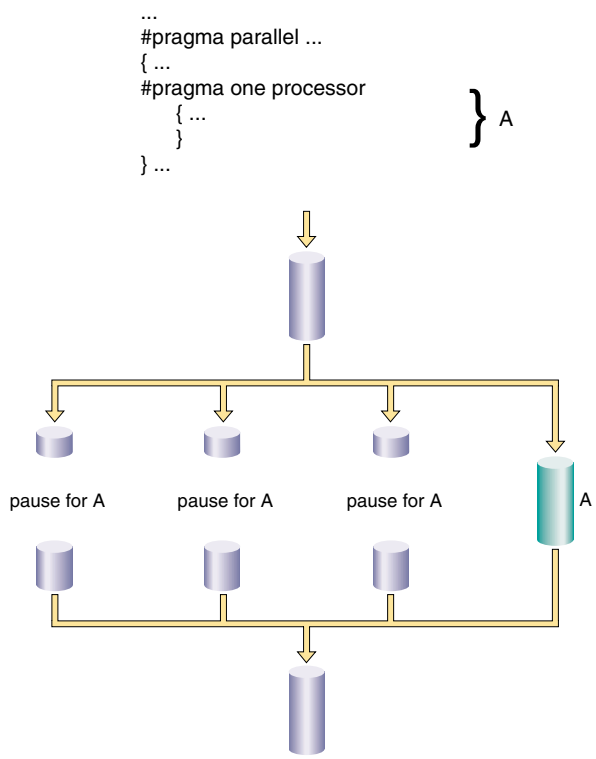

Figure 8-4 One Processor Segment

#pragma parallel

The **parallel** pragma 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 further information on how to run the block of code (see "#pragma parallel clauses" on page 95). These clauses can either be listed on the same line as the parallel pragma, or can be broken out into separate pragmas (see "Example of #pragma parallel" on page 94).

Using #pragma parallel

The syntax of the **parallel** pragma is as follows:

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

Use the **parallel** pragma to start a parallel region. This pragma has a number of clauses (see "#pragma parallel clauses" on page 95 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 **parallel** pragma 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 for Using #pragma parallel

This pragma 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

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];
}
```

Then, you must 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 this:

```
#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.

#pragma parallel clauses

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

- **shared**
- **local**
- **if**
- **numthreads**

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.

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

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's 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 doesn't 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 runs serially. Otherwise, the statements run in parallel.

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 (see the "Multiprocessing Advanced Features" chapter in the *C Language Reference Manual* for more details). 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.

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.

#pragma pfor

#pragma pfor marks a *for* loop to run in parallel. This pragma must follow a **parallel** pragma and be contained within a parallel region. **pfor** takes several clauses (see "#pragma parallel clauses" on page 95 for more details), which control various aspects such as the following:

- how the work load are 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

Using #pragma pfor

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

- The **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. A quick test for independence is if the loop can be run backwards, then chances are good the iterations are 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 for Using #pragma pfor

This pragma 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

Diagram of #pragma pfor

Figure 8-5 shows parallel code segments using **#pragma pfor** running on four threads with simple scheduling.

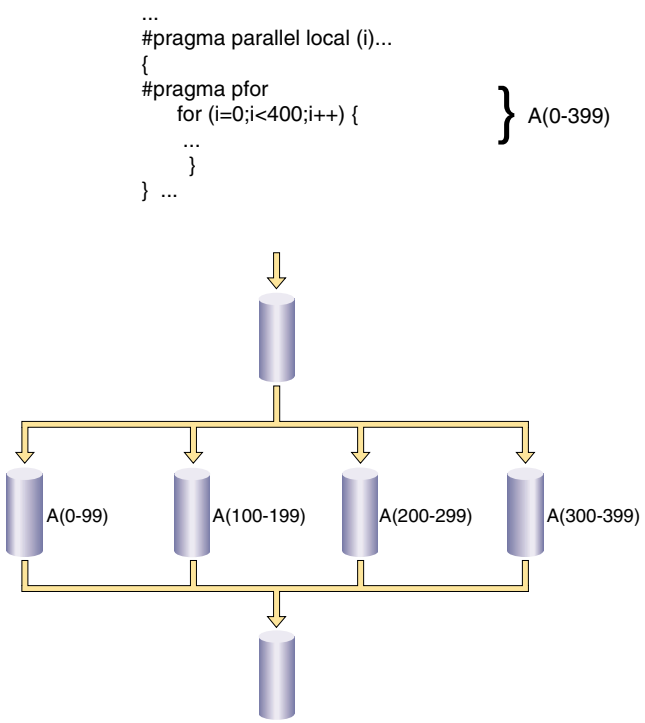

Figure 8-5 Parallel Code Segments Using #pragma pfor

C++ Multiprocessing Considerations With #pragma pfor

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

```
int i = 0;
```

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

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

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

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

#pragma pfor clauses

The **pfor** pragma has seven clauses:

iterate	Gives 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.

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 gives 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 8-2 describes the components of the **iterate** clause.

Table 8-2 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.

iterate Example

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.

local and lastlocal: Specifying Local Variables

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

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

local 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 "local: Specifying Local Variables" on page 95).

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

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

lastlocal 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.

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 "Parallel Reduction Operations in C and C++" section in the "Multiprocessing C/C++ Compiler Directives" chapter of 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.

affinity: Specifying Thread and Data Affinity

The **affinity** clause can be used for thread or data affinity. Thread affinity assigns particular iterations to a particular thread. Data affinity applies to distributed arrays (and is useful only on Origin systems).

Thread Affinity

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 "#pragma page_place.".

Data Affinity

Data affinity applies only to distributed arrays and is useful only on Origin systems. See Chapter 4, "Distributed Shared Memory (DSM) Optimization Pragmas," 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; and *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. For example,

```
#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];
```

In this example, the loop is scheduled based on the block distribution of the first dimension. See Chapter 4, “Distributed Shared Memory (DSM) Optimization Pragmas,” for more information about distribution directives.

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.

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 "#pragma dynamic" on 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.

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.)

Data Affinity and the #pragma pfor nest Clause

The **nest** clause for **#pragma pfor** is described in "nest: Exploiting Nested Concurrency" on page 107. 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;
```

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 wish 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*.

schedtype: Sharing Loop Iterations Among Processors

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

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

schedtype 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 8-4).

Valid Types for schedtype

You can use the types in Table 8-3 to modify **schedtype**.

Table 8-3 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.
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 "Run-time Environment Variables" section in the "Multiprocessing Advanced Features" chapter of the *C Language Reference Manual* for more information).

Loop Scheduling

Figure 8-6 shows how the iteration chunks are apportioned over the various processors by the different types of loop scheduling.

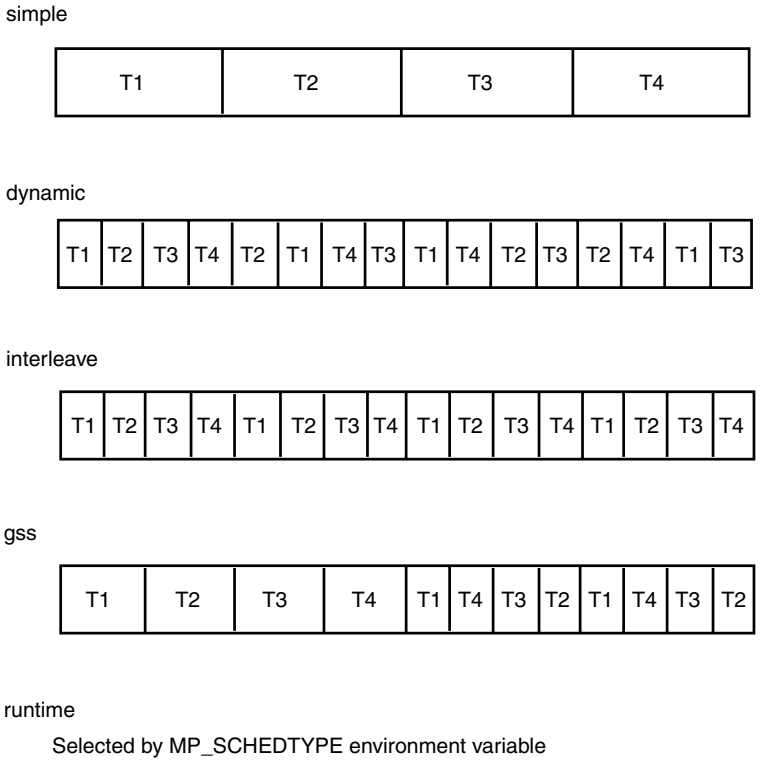

Figure 8-6 Loop Scheduling Types

Choosing a schedtype

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 8-4 describes how to choose a **schedtype**.

Table 8-4 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*.

chunksize: Specifying the Number of Iterations in a Chunk

chunksize tells the multiprocessing C compiler how many iterations to define as a chunk when using the **dynamic** or **interleave** clause (see "schedtype: Sharing Loop Iterations Among Processors" on page 108).

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.

#pragma set chunksize

#pragma set chunksize 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 "#pragma set schedtype" on page 114 and "#pragma pfor clauses" on page 101 for more information).

Using #pragma set chunksize

The syntax of the **set chunksize** pragma 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.

#pragma set numthreads

#pragma set numthreads 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.

Using #pragma set numthreads

The syntax of the **set numthreads** pragma 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 pragma.

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 "Run-time Environment Variables" in the "Multiprocessing Advanced Features" chapter of the *C Language Reference Manual* for more information).

#pragma set schedtype

#pragma set schedtype 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 "#pragma pfor clauses" on page 101 for more information on **schedtype**).

Using #pragma schedtype

The syntax of the **schedtype** pragma is as follows:

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

The schedtype *types* are

- simple
- dynamic
- interleave
- gss
- runtime

See Table 8-3 for a description of each type.

#pragma shared

The **#pragma shared** directive tells the multiprocessing C/C++ compiler the names of all the variables that the threads must share. This pragma must be used in conjunction with the **parallel** pragma. **shared** can also be used as a clause for the **parallel** pragma (see "#pragma parallel clauses" on page 95).

Using #pragma shared

The syntax of the **shared** pragma 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.

#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 pragma until all threads have reached it. This pragma is a classic barrier construct.

Using #pragma synchronize

The syntax of the **synchronize** pragma is as follows:

```
#pragma synchronize
```

Diagram of #pragma synchronize

Figure 8-7 is a "time-lapse" sequence showing the synchronization of all threads.

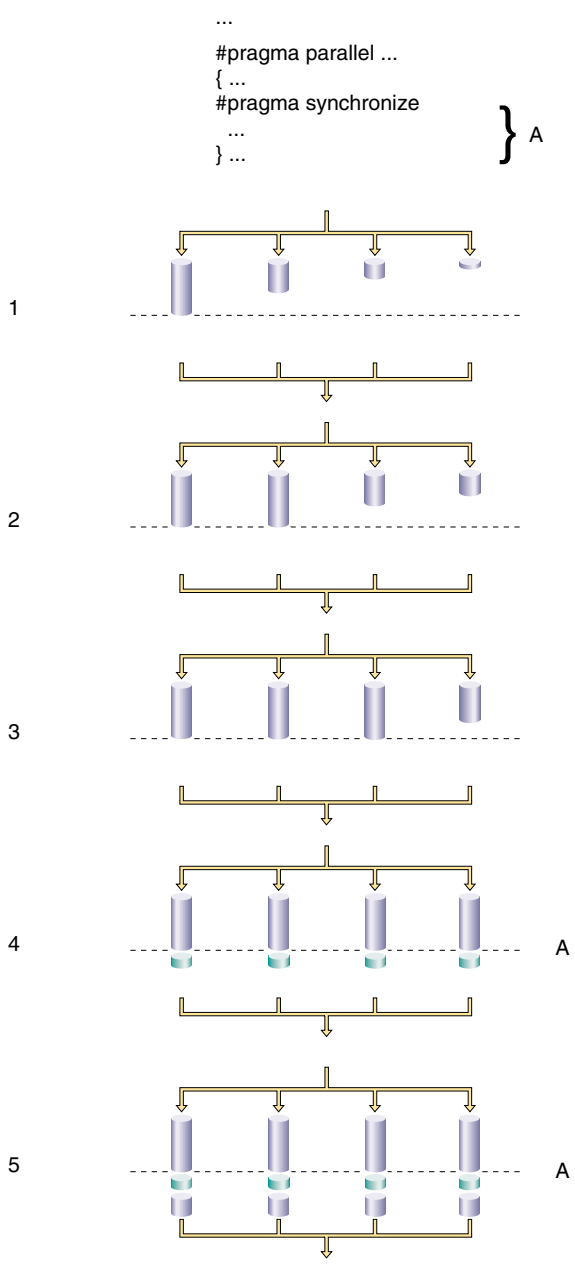

Figure 8-7 Synchronization

Precompiled Header Pragmas

Table 9-1 lists the precompiled header pragmas, along with a short description of each and the compiler versions in which the pragma is supported.

Table 9-1 Precompiled Header Pragmas

#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

#pragma hdrstop

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

Using #pragma hdrstop

The syntax of the **hdrstop** pragma 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.

#pragma no_pch

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

Using #pragma no_pch

The syntax of the **no_pch** pragma is as follows:

```
#pragma no_pch
```

#pragma once

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

Using #pragma once

The syntax of the **once** pragma is as follows:

```
#pragma once
```

This pragma has no effect in **–32** 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 include file *afile.h* with an **#ifdef** directive similar to the following:

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

Chapter 10

Scalar Optimization Pragmas

Table 10-1 lists the pragmas covered in this chapter, along with a short description of each and the compiler versions in which the pragma is supported.

Table 10-1 Scalar Optimization Pragmas

#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 6, "Loader Information Pragmas")	Causes an object to be placed in a gp_relative section.	7.2 and later
"#pragma section_non_gp" (in Chapter 6, "Loader Information Pragmas")	Keeps an object from being placed in a gp_relative section.	7.2 and later
"#pragma unroll" (in Chapter 7, "Loop Nest Optimization Pragmas")	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 pragma is an inner loop, then it indicates standard unrolling. If the loop following this pragma is not innermost, then outer loop unrolling (unroll and jam) is performed.	7.2 and later

#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.

Using #pragma mips_frequency_hint

The syntax of the **mips_frequency_hint** pragma 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 pragma provides a mechanism for you to give 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 to 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 pragma in two ways:

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

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

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

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

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

Cautions for Using #pragma mips_frequency_hint

This is for compiler version 7.2 only, and does not work for **-32**, because it requires an ELF object file with *.MIPS.content* sections.

Warning Suppression Control Pragmas

Table 11-1 lists the pragmas discussed in this chapter, along with a brief description and the compiler versions in which the pragma is supported.

Table 11-1 Warning Suppression Control Pragmas

#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

#pragma set woff

The **#pragma set woff** directive suppresses complier warnings individually by warning number.

Using #pragma set woff

The syntax of the **set woff** pragma is as follows:

```
#pragma set woff (warning_list)
```

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

For example, the following pragma 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.

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 */
```

#pragma reset woff

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

Using #pragma reset woff

The syntax of the **reset woff** pragma 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 pragma sets warnings 1, 2, 300 through 310, and 8 back to the command-line setting:

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

This pragma does not nest.

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 */
```

Chapter 12

Miscellaneous Pragmas

Table 12-1 lists the pragmas described in this chapter, along with a brief description of each.

Table 12-1 Miscellaneous Pragmas

#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 (**-32** 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 ident

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

Using #pragma ident

The syntax of the **ident** pragma is as follows:

```
#pragma ident "string"
```

string is the string you wish to add to the *.comment* section in the object file. The string must be enclosed in double quotes.

Caution for Using #pragma ident

The **ident** pragma is not available with **-n32** or **-64** mode. It is only available with **-32** mode.

#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**).

Using #pragma int_to_unsigned

The syntax of the **int_to_unsigned** pragma 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 pragma 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 pragma:

```
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.

#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.

Using #pragma intrinsic

The syntax of the **intrinsic** pragma is as follows:

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

Cautions for Using #pragma intrinsic

- This pragma 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 pragmas 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.

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3587-001.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389