# Intel® Fortran Compiler for Linux* Systems User's Guide Volume II: Optimizing Applications

Document Number: 253260-002

## Disclaimer and Legal Information

# Table Of Contents

# Optimizing Applications: Overview

This is the second volume in a two-volume Intel® Fortran Compiler User's Guide.

It covers the following topics:

Programming for high performance using the Intel Fortran Compiler:

- Setting Data Type and Alignment
- Using Arrays Efficiently
- Improving I/O Performance
- Improving Run-time Efficiency
- Using Intrinsics for Itanium-based Systems
- Coding Guidelines for Intel Architectures

Analyzing and timing your application:

- Using Intel Performance Analysis Tools
- Timing Your Application

Implementing Intel Fortran Compiler optimizations:

- Optimizing the Compilation Process
- Efficient Compilation
- Stack Options for Automatic Allocation and Checking
- Alignment Options
- Symbol Visibility Attribute Options
- Options to Optimize Different Application Types
- Floating Point Arithmetic Optimizations
- Optimizing for Specific Processors
- Interprocedural Optimizations
- Profile-guided Optimizations
- High-level Language Optimizations (HLO)

Parallel programming with Intel Fortran:

- Auto-vectorization (IA-32 Only)
- Auto-parallelization
- Parallelization with OpenMP*
- Debugging Multi-Threaded Programs

Optimization support features:

- Compiler Directives
- Optimizations and Debugging

- Optimizer Report Generation

For information on new features in this release, see the topic titled What's New in This Release, in Volume I. Also refer to the product Release Notes.

# How to Use This Document

This User's Guide explains how you can use the Intel® Fortran Compiler to enhance your application.

The optimizations provided by the Intel Fortran Compiler enable you to enhance the performance of your application. Each optimization is performed using a set of options discussed in the sections of this volume.

In addition to optimizations invoked by the compiler command line options, the compiler includes features that enhance your application performance such as directives, intrinsics, run-time library routines and various utilities. These features are discussed in the Optimization Support Features section.

**Note**
This document explains how information and instructions apply differently to targeted architectures. If there is no reference to a specific architecture, the description applies to all supported architectures.

This documentation assumes that you are familiar with the Fortran Standard programming language and with the Intel® processor architecture. You should also be familiar with the host computer's operating system.

## Notation Conventions

This manual uses the following conventions:

| | |
|---|---|
| Intel Fortran | The name of the common compiler language supported by the Intel® Fortran Compiler for Windows* and Intel® Fortran Compiler for Linux* products. |
| Fortran 95<br>Fortran 90<br>Fortran 77 | These terms are references to versions of the Fortran language. The default is "Fortran," which corresponds to all versions. |
| THIS TYPE STYLE | Statements, keywords, and directives are shown in all uppercase, in a normal font. . For example, “add the USE statement…”. |
| **This type style** | Bold, normal text indicates menu names, menu items, button names, dialog window names, and other user-interface items. |
| **File > Open** | Menu names and menu items joined by a greater than (>) sign indicate a sequence of actions. For example, "Click **File > Open**" indicates that in the |

| | |
|---|---|
| | **File** menu, click **Open** to perform this action. |
| `ifort` | The use of the compiler command in examples follows this general rule: when there is no usage difference between architectures, only one command is given. Whenever there is a difference in usage, the commands for each architecture are given. |
| `This type style` | Regular, monospaced text indicates an element of syntax, a reserved word, a keyword, a file name,  a variable, or a code example. The text appears in lowercase unless uppercase is required. |
| **`This type style`** | Bold, monospaced text indicates user input. It shows what you type as a command or input. |
| *`This type style`* | Italic, monospaced text indicates placeholders for information that you must supply. This style is also used to introduce new terms. |
| `[options]` | Items inside single square brackets are optional. (In some examples, square brackets are used to show arrays.) |
| `{value \| value}` | Braces and a vertical bar indicate a choice of items.  You must choose one of the items unless all of the items are also enclosed in square brackets. |
| `...` | In syntax examples, a horizontal ellipsis (three dots) following an item indicates that the item preceding the ellipsis can be repeated. In code examples, a horizontal ellipsis means that not all of the statements are shown. |
| Linux* systems | An asterisk at the end of a word or name indicates it is a third-party product trademark. |

# Programming for High Performance

## Programming for High Performance: Overview

This section provides information on the following:

- Programming Guidelines
  This section discusses programming guidelines that can enhance application performance and includes specific coding practices that use the Intel® architecture features.
- Analyzing and Timing Your Application
  This section discusses how to use the Intel performance analysis tools and how to time program execution to collect information about problem areas.

## Programming Guidelines

### Setting Data Type and Alignment

Data alignment considerations apply to the following kinds of variables:

- Those that are dynamically allocated
- Those that are members of a data structure
- Those that are global or local variables
- Those that are parameters passed on the stack

For best performance, align data as follows:

- Align 8-bit data at any address.
- Align 16-bit data to be contained within an aligned four byte word.
- Align 32-bit data so that its base address is a multiple of four.
- Align 64-bit data so that its base address is a multiple of eight.
- Align 80-bit data so that its base address is a multiple of sixteen.
- Align 128-bit data so that its base address is a multiple of sixteen.

#### Causes of Unaligned Data and Ensuring Natural Alignment

For optimal performance, make sure your data is aligned naturally. A natural boundary is a memory address that is a multiple of the data item's size. For example, a REAL (KIND=8) data item aligned on natural boundaries has an address that is a multiple of 8. An array is aligned on natural boundaries if all of its elements are so aligned.

All data items whose starting address is on a natural boundary are *naturally aligned*. Data not aligned on a natural boundary is called *unaligned data*.

Although the Intel® Fortran Compiler naturally aligns individual data items when it can, certain Fortran statements can cause data items to become unaligned.

You can use the command-line option `-align` to ensure naturally aligned data, but you should check and consider reordering data declarations of data items within common blocks, derived-type structures, and record structures as follows:

- Carefully specify the order and sizes of data declarations to ensure naturally aligned data.
- Start with the largest size numeric items first, followed by smaller size numeric items, and then non-numeric (character) data.

The following statements can cause unaligned data:

- Common blocks (COMMON statement)

  The order of variables in the COMMON statement determines their storage order. Unless you are sure that the data items in the common block will be naturally aligned, specify either the `-align commons` or `-align dcommons` option, depending on the largest data size used. See Alignment Options.

- Derived-type (user-defined) data

  Derived-type data items are declared after a TYPE statement.

  If your data includes derived-type data structures, you should use the `-align records` option, unless you are sure that the data items in the derived-type structures will be naturally aligned.

  If you omit the SEQUENCE statement, the `-align records` option (default) ensures all data items are naturally aligned.

  If you specify the SEQUENCE statement, the `-align records` option is prevented from adding necessary padding to avoid unaligned data (data items are packed) unless you specify the `-align sequence` option. When you use SEQUENCE , you should specify data declaration order so that all data items are naturally aligned.

- Record structures (RECORD and STRUCTURE statements)

  Intel Fortran record structures usually contain multiple data items. The order of variables in the STRUCTURE statement determines their storage order. The RECORD statement names the record structure. Record structures are an Intel Fortran language extension.

If your data includes record structures, you should use the `-align records` option, unless you are sure that the data items in the record structures will be naturally aligned.

- EQUIVALENCE statements

  EQUIVALENCE statements can force unaligned data or cause data to span natural boundaries. For more information, see the *Intel® Fortran Language Reference*.

To avoid unaligned data in a common block, derived-type structure, or record structure, use one or both of the following:

- For new programs or for programs where the source code declarations can be modified easily, plan the order of data declarations with care. For example, you should order variables in a COMMON statement such that numeric data is arranged from largest to smallest, followed by any character data (see the data declaration rules in Ordering Data Declarations to Avoid Unaligned Data below.
- For existing programs where source code changes are not easily done or for array elements containing derived-type or record structures, you can use command line options to request that the compiler align numeric data by adding padding spaces where needed.

Other possible causes of unaligned data include unaligned actual arguments and arrays that contain a derived-type structure or record structure:

- When actual arguments from outside the program unit are not naturally aligned, unaligned data access occurs. Intel Fortran assumes all passed arguments are naturally aligned and has no information at compile time about data that will be introduced by actual arguments during program execution.
- For arrays where each array element contains a derived-type structure or record structure, the size of the array elements may cause some elements (but not the first) to start on an unaligned boundary.
- Even if the data items are naturally aligned within a derived-type structure without the SEQUENCE statement or a record structure, the size of an array element might require use of the `-align records` option to supply needed padding to avoid some array elements being unaligned.
- If you specify `-align norecords` or specify `-vms` without `-align records`, no padding bytes are added between array elements. If array elements each contain a derived-type structure with the SEQUENCE statement, array elements are packed without padding bytes regardless of

the Fortran command options specified. In this case, some elements will be unaligned.
- When the `-align records` option is in effect, the number of padding bytes added by the compiler for each array element is dependent on the size of the largest data item within the structure. The compiler determines the size of the array elements as an exact multiple of the largest data item in the derived-type structure without the SEQUENCE statement or a record structure. The compiler then adds the appropriate number of padding bytes. For instance, if a structure contains an 8-byte floating-point number followed by a 3-byte character variable, each element contains five bytes of padding (16 is an exact multiple of 8). However, if the structure contains one 4-byte floating-point number, one 4-byte integer, followed by a 3-byte character variable, each element would contain one byte of padding (12 is an exact multiple of 4).

**Checking for Inefficient Unaligned Data**

During compilation, the Intel Fortran compiler naturally aligns as much data as possible. Exceptions that can result in unaligned data are described above.

Because unaligned data can slow run-time performance, it is worthwhile to:

- Double-check data declarations within common blocks, derived-type structures, or record structures to ensure all data items are naturally aligned (see the data declaration rules in the subsection below). Using modules to contain data declarations can ensure consistent alignment and use of such data.
- Avoid the EQUIVALENCE statement or use it in a way that cannot cause unaligned data or data spanning natural boundaries.
- Ensure that arguments passed from outside the program unit are naturally aligned.
- Check that the size of array elements containing at least one derived-type structure or record structure causes array elements to start on aligned boundaries (see the previous subsection).

There are two ways unaligned data might be reported:

- During compilation, warning messages are issued for any data items that are known to be unaligned, unless you specify the `-warn noalignments` (or `-W0`) option that suppresses all warnings.
- During program execution, warning messages are issued for any data that is detected as unaligned. The message includes the address of the unaligned access.

Consider the following run-time message:

```
Unaligned access pid=24821 <a.out> va=140000154,
pc=3ff80805d60, ra=1200017bc
```

This message shows that:

- The statement accessing the unaligned data (program counter) is located at 3ff80805d60
- The unaligned data is located at address 140000154

## Ordering Data Declarations to Avoid Unaligned Data

For new programs or when the source declarations of an existing program can be easily modified, plan the order of your data declarations carefully to ensure the data items in a common block, derived-type structure, record structure, or data items made equivalent by an EQUIVALENCE statement will be naturally aligned.

Use the following rules to prevent unaligned data:

- Always define the largest size numeric data items first.
- If your data includes a mixture of character and numeric data, place the numeric data first.
- Add small data items of the correct size (or padding) before otherwise unaligned data to ensure natural alignment for the data that follows.

When declaring data, consider using explicit length declarations, such as specifying a KIND parameter. For example, specify INTEGER(KIND=4) (or INTEGER(4)) rather than INTEGER. If you do use a default size (such as INTEGER, LOGICAL, COMPLEX, and REAL), be aware that the compiler options
`-integer_size{16|32|64}` or `-real_size{32|64|128}` can change the size of an individual field's data declaration size and thus can alter the data alignment of a carefully planned order of data declarations.

Using the suggested data declaration guidelines minimizes the need to use the `-align` *`keyword`* options to add padding bytes to ensure naturally aligned data. In cases where the `-align` *`keyword`* options are still needed, using the suggested data declaration guidelines can minimize the number of padding bytes added by the compiler.

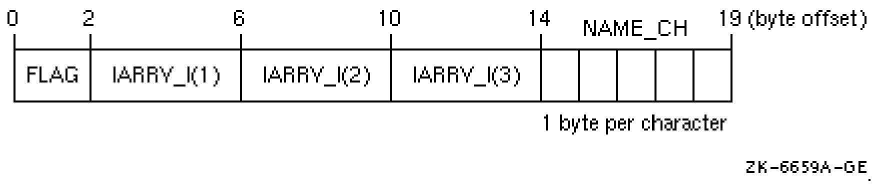
### Arranging Data Items in Common Blocks

The order of data items in a `common` statement determine the order in which the data items are stored. Consider the following declaration of a common block named `x`:

```
logical (kind=2) flag
integer          iarry_i(3)
character(len=5) name_ch
common /x/ flag, iarry_i(3), name_ch
```

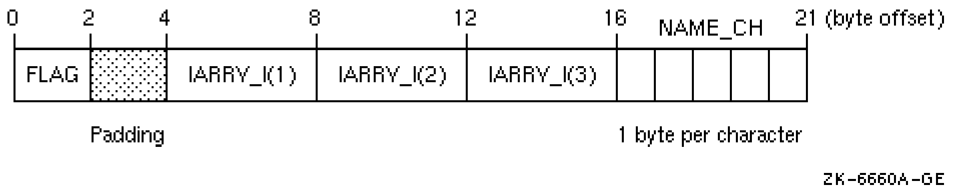
As shown in Figure 1-1, if you omit the appropriate Fortran command options, the common block will contain unaligned data items beginning at the first array element of `iarry_i`.

**Figure 1-1 Common Block with Unaligned Data**



As shown in Figure 1-2, if you compile the program units that use the common block with the
`-align commons` option, data items will be naturally aligned.

**Figure 1-2 Common Block with Naturally Aligned Data**



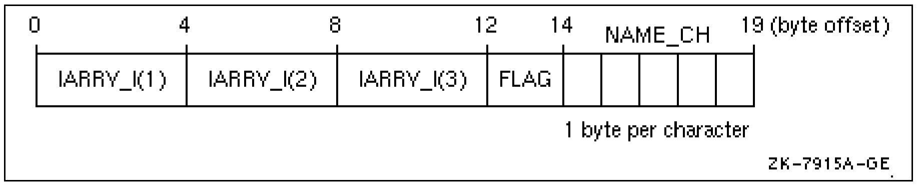
Because the common block x contains data items whose size is 32 bits or smaller, specify the
`-align commons` option. If the common block contains data items whose size might be larger than 32 bits (such as REAL (KIND=8) data), use the `-align commons` option.

If you can easily modify the source files that use the common block data, define the numeric variables in the COMMON statement in descending order of size and place the character variable last. This provides more portability, ensures natural alignment without padding, and does not require the command-line options `-align commons` or `-align dcommons` option:

```
logical (kind=2) flag
integer          iarry_i(3)
character(len=5) name_ch
common /x/ iarry_i(3), flag, name_ch
```

As shown in Figure 1-3, if you arrange the order of variables from largest to smallest size and place character data last, the data items will be naturally aligned.

**Figure 1-3 Common Block with Naturally Aligned Reordered Data**



When modifying or creating all source files that use common block data, consider placing the common block data declarations in a module so the declarations are consistent. If the common block is not needed for compatibility (such as file storage or Fortran 77 use), you can place the data declarations in a module without using a common block.

#### Arranging Data Items in Derived-Type Data

Like common blocks, derived-type structures can contain multiple data items (members).

Data item components within derived-type structures are naturally aligned on up to 64-bit boundaries, with certain exceptions related to the use of the SEQUENCE statement and Fortran options. See Options Controlling Alignment for information about these exceptions.

Intel Fortran stores a derived data type as a linear sequence of values, as follows:

- If you specify the SEQUENCE statement, the first data item is in the first storage location and the last data item is in the last storage location. The data items appear in the order in which they are declared. The Fortran options have no effect on unaligned data, so data declarations must be carefully specified to naturally align data. The `-align sequence` option specifically aligns data items in a SEQUENCE derived-type on natural boundaries.
- If you omit the SEQUENCE statement, Intel Fortran adds the padding bytes needed to naturally align data item components, unless you specify the `-align norecords` option.

Consider the following declaration of array `CATALOG_SPRING` of derived-type `PART_DT`:

```
module data_defs
type part_dt
```

```
integer           identifier
real              weight
character(len=15) description
end type part_dt
type(part_dt) catalog_spring(30)
.
.
.
end module data_defs
```

## Using Arrays Efficiently

Many of the array access efficiency techniques described in this section are applied automatically by the Intel Fortran loop transformations optimizations. Several aspects of array use can improve run-time performance:

- The fastest array access occurs when contiguous access to the whole array or most of an array occurs. Perform one or a few array operations that access all of the array or major parts of an array instead of numerous operations on scattered array elements. Rather than use explicit loops for array access, use elemental array operations, such as the following line that increments all elements of array variable a:

  ```
  a = a + 1
  ```

  When reading or writing an array, use the array name and not a DO loop or an implied DO-loop that specifies each element number. Fortran 95/90 array syntax allows you to reference a whole array by using its name in an expression. For example:

  ```
  real :: a(100,100)
  a = 0.0
  a = a + 1       ! Increment all elements
            !  of a by 1
  .
  .
  .
  write (8) a     ! Fast whole array use
  ```

  Similarly, you can use derived-type array structure components, such as:

  ```
  type x
  integer a(5)
  end type x
  .
  .
  .
  type (x) z
  write (8)z%a    ! Fast array structure
            !  component use
  ```

- Make sure multidimensional arrays are referenced using proper array syntax and are traversed in the *natural ascending storage order*, which is *column-major order* for Fortran. With column-major order, the leftmost subscript varies most rapidly with a stride of one. Whole array access uses column-major order.
  Avoid *row-major order*, as is done by C, where the rightmost subscript varies most rapidly.
  For example, consider the nested do loops that access a two-dimension array with the j loop as the innermost loop:

  ```
  integer x(3,5), y(3,5), i, j
  y = 0
  do i=1,3                ! I outer loop varies slowest
  do j=1,5                ! J inner loop varies fastest
  x (i,j) = y(i,j) + 1    ! Inefficient row-major storage
  order
  end do                  !  (rightmost subscript varies
  fastest)
  end do
  .
  .
  .
  end program
  ```

  Since j varies the fastest and is the second array subscript in the expression x (i,j), the array is accessed in row-major order.
  To make the array accessed in natural column-major order, examine the array algorithm and data being modified. Using arrays x and y, the array can be accessed in natural column-major order by changing the nesting order of the do loops so the innermost loop variable corresponds to the leftmost array dimension:

  ```
  integer x(3,5), y(3,5), i, j
  y = 0
  do j=1,5                ! J outer loop varies slowest
  do i=1,3                ! I inner loop varies fastest
  x (i,j) = y(i,j) + 1    ! Efficient column-major storage
  order
  end do                  !  (leftmost subscript varies
  fastest)
  end do
  .
  .
  .
  end program
  ```

  The Intel Fortran whole array access (x = y + 1) uses efficient column major order. However, if the application requires that J vary the fastest or if you cannot modify the loop order without changing the results, consider modifying the application program to use a rearranged order of array dimensions. Program modifications include rearranging the order of:

- Dimensions in the declaration of the arrays x(5,3) and y(5,3)
- The assignment of `x(j,i)` and `y(j,i)` within the do loops
- All other references to arrays x and y

In this case, the original DO loop nesting is used where `J` is the innermost loop:

```
integer x(3,5), y(3,5), i, j
y = 0
do i=1,3              ! I outer loop varies slowest
do j=1,5              ! J inner loop varies fastest
x (j,i) = y(j,i) + 1  ! Efficient column-major storage
order
end do                !  (leftmost subscript varies
fastest)
end do
.
.
.
end program
```

Code written to access multidimensional arrays in row-major order (like C) or random order can often make use of the CPU memory cache less efficient. For more information on using natural storage order during record, see Improving I/O Performance.

- Use the available Fortran 95/90 array intrinsic procedures rather than create your own.

  Whenever possible, use Fortran 95/90 array intrinsic procedures instead of creating your own routines to accomplish the same task. Fortran 95/90 array intrinsic procedures are designed for efficient use with the various Intel Fortran run-time components.

  Using the standard-conforming array intrinsics can also make your program more portable.

- With multidimensional arrays where access to array elements will be noncontiguous, avoid leftmost array dimensions that are a power of two (such as 256, 512).

  Since the cache sizes are a power of 2, array dimensions that are also a power of 2 may make less efficient use of cache when array access is noncontiguous. If the cache size is an exact multiple of the leftmost dimension, your program will probably make inefficient use of the cache.

This does not apply to contiguous sequential access or whole array access.

One work-around is to increase the dimension to allow some unused elements, making the leftmost dimension larger than actually needed. For example, increasing the leftmost dimension of A from 512 to 520 would make better use of cache:

```
 real a(512, 100)
do i= 2,511
do j = 2,99
a(i,j)=(a(i+1,j-1) + a(i-1, j+1)) * 0.5
end do
end do
```

In this code, array a has a leftmost dimension of 512, a power of two. The innermost loop accesses the rightmost dimension (row major), causing inefficient access. Increasing the leftmost dimension of a to 520 (`real a (520,100)`) allows the loop to provide better performance, but at the expense of some unused elements.

Because loop index variables I and J are used in the calculation, changing the nesting order of the do loops changes the results.

For more information on arrays and their data declaration statements, see the *Intel® Fortran Language Reference*.

**Passing Array Arguments Efficiently**

In Fortran, there are two general types of array arguments:

- Explicit-shape arrays used with Fortran 77.

  These arrays have a fixed rank and extent that is known at compile time. Other dummy argument (receiving) arrays that are not deferred-shape (such as assumed-size arrays) can be grouped with explicit-shape array arguments.

- Deferred-shape arrays introduced with Fortran 95/90.

  Types of deferred-shape arrays include array pointers and allocatable arrays. Assumed-shape array arguments generally follow the rules about passing deferred-shape array arguments.

When passing arrays as arguments, either the starting (base) address of the array or the address of an array descriptor is passed:

- When using explicit-shape (or assumed-size) arrays to receive an array, the starting address of the array is passed.
- When using deferred-shape or assumed-shape arrays to receive an array, the address of the array descriptor is passed (the compiler creates the array descriptor).

Passing an assumed-shape array or array pointer to an explicit-shape array can slow run-time performance. This is because the compiler needs to create an array temporary for the entire array. The array temporary is created because the passed array may not be contiguous and the receiving (explicit-shape) array requires a contiguous array. When an array temporary is created, the size of the passed array determines whether the impact on slowing run-time performance is slight or severe.

The following table summarizes what happens with the various combinations of array types. The amount of run-time performance inefficiency depends on the size of the array.

| **Actual Argument Array Types (choose one:)** | **Dummy Argument Array Types (choose one:)** | |
|---|---|---|
| | **Explicit-Shape Arrays** | **Deferred-Shape and Assumed-Shape Arrays** |
| **Explicit-shape arrays** | Result when using this combination: Very efficient. Does not use an array temporary. Does not pass an array descriptor. Interface block optional. | Result when using this combination: Efficient. Only allowed for assumed-shape arrays (not deferred-shape arrays). Does not use an array temporary. Passes an array descriptor. Requires an interface block. |
| **Deferred-shape and assumed-shape arrays** | Result when using this combination: When passing an allocatable array, very efficient. Does not use an array temporary. Does not pass an array descriptor. Interface block optional.<br><br>When not passing an allocatable array, not efficient. Instead use allocatable arrays whenever possible.<br><br>Uses an array temporary. | Result when using this combination: Efficient. Requires an assumed-shape or array pointer as dummy argument. Does not use an array temporary. Passes an array descriptor. Requires an interface block. |

| | | |
|---|---|---|
| | Does not pass an array descriptor. Interface block optional. | |

## Improving I/O Performance

Improving overall I/O performance can minimize both device I/O and actual CPU time. The techniques listed in this topic can significantly improve performance in many applications.

I/O flow problems limit the maximum speed of execution by being the slowest process in an executing program. In some programs, I/O is the bottleneck that prevents an improvement in run-time performance. The key to relieving I/O problems is to reduce the actual amount of CPU and I/O device time involved in I/O.

The problems can be caused by one or more of the following:

- A dramatic reduction in CPU time without a corresponding improvement in I/O time
- Such coding practices as:
  - Unnecessary formatting of data and other CPU-intensive processing
  - Unnecessary transfers of intermediate results
  - Inefficient transfers of small amounts of data
  - Application requirements

Improved coding practices can minimize actual device I/O, as well as the actual CPU time.

Intel offers software solutions to system-wide problems like minimizing device I/O delays.

### Use Unformatted Files Instead of Formatted Files

Use unformatted files whenever possible. Unformatted I/O of numeric data is more efficient and more precise than formatted I/O. Native unformatted data does not need to be modified when transferred and will take up less space on an external file.

Conversely, when writing data to formatted files, formatted data must be converted to character strings for output, less data can transfer in a single operation, and formatted data may lose precision if read back into binary form.

To write the array `A(25,25)` in the following statements, `S1` is more efficient than `S2`:

```
S1         WRITE (7) A

S2         WRITE (7,100) A
100   FORMAT (25(' ',25F5.21))
```

Although formatted data files are more easily ported to other systems, Intel Fortran can convert unformatted data in several formats; see Little-endian-to-Big-endian Conversion.

## Write Whole Arrays or Strings

To eliminate unnecessary overhead, write whole arrays or strings at one time rather than individual elements at multiple times. Each item in an I/O list generates its own calling sequence. This processing overhead becomes most significant in implied-DO loops. When accessing whole arrays, use the array name (Fortran array syntax) instead of using implied-`DO` loops.

## Write Array Data in the Natural Storage Order

Use the natural ascending storage order whenever possible. This is column-major order, with the leftmost subscript varying fastest and striding by 1. (See Accessing Arrays Efficiently.) If a program must read or write data in any other order, efficient block moves are inhibited.

If the whole array is not being written, natural storage order is the best order possible.

If you must use an unnatural storage order, in certain cases it might be more efficient to transfer the data to memory and reorder the data before performing the I/O operation.

## Use Memory for Intermediate Results

Performance can improve by storing intermediate results in memory rather than storing them in a file on a peripheral device. One situation that may not benefit from using intermediate storage is when there is a disproportionately large amount of data in relation to physical memory on your system. Excessive page faults can dramatically impede virtual memory performance.

If you are primarily concerned with the CPU performance of the system, consider using a memory file system (mfs) virtual disk to hold any files your code reads or writes.

### Enable Implied-DO Loop Collapsing

DO loop collapsing reduces a major overhead in I/O processing. Normally, each element in an I/O list generates a separate call to the Intel Fortran run-time library (RTL). The processing overhead of these calls can be most significant in implied-DO loops.

Intel Fortran reduces the number of calls in implied-DO loops by replacing up to seven nested implied-DO loops with a single call to an optimized run-time library I/O routine. The routine can transmit many I/O elements at once.

Loop collapsing can occur in formatted and unformatted I/O, but only if certain conditions are met:

- The control variable must be an integer. The control variable cannot be a dummy argument or contained in an EQUIVALENCE or VOLATILE statement. Intel Fortran must be able to determine that the control variable does not change unexpectedly at run time.
- The format must not contain a variable format expression.

For information on the VOLATILE attribute and statement, see the *Intel® Fortran Language Reference*.

For loop optimizations, see Loop Transformations, Loop Unrolling, and Optimization Levels.

### Use of Variable Format Expressions

Variable format expressions (an Intel Fortran extension) are almost as flexible as run-time formatting, but they are more efficient because the compiler can eliminate run-time parsing of the I/O format. Only a small amount of processing and the actual data transfer are required during run time.

On the other hand, run-time formatting can impair performance significantly. For example, in the following statements, `S1` is more efficient than `S2` because the formatting is done once at compile time, not at run time:

```
S1        WRITE (6,400) (A(I), I=1,N)
400  FORMAT (1X, <N> F5.2)
```

```
     .
     .
     .
S2        WRITE (CHFMT,500) '(1X,',N,'F5.2)'
500  FORMAT (A,I3,A)
WRITE (6,FMT=CHFMT) (A(I), I=1,N)
```

**Efficient Use of Record Buffers and Disk I/O**

Records being read or written are transferred between the user's program buffers and one or more disk block I/O buffers, which are established when the file is opened by the Intel Fortran RTL. Unless very large records are being read or written, multiple logical records can reside in the disk block I/O buffer when it is written to disk or read from disk, minimizing physical disk I/O.

You can specify the size of the disk block physical I/O buffer by using the OPEN statement BLOCKSIZE specifier; the default size can be obtained from `fstat(2)`. If you omit the BLOCKSIZE specifier in the OPEN statement, it is set for optimal I/O use with the type of device the file resides on (with the exception of network access).

The OPEN statement BUFFERCOUNT specifier specifies the number of I/O buffers. The default for BUFFERCOUNT is 1. Any experiments to improve I/O performance should increase the BUFFERCOUNT value and not the BLOCKSIZE value, to increase the amount of data read by each disk I/O.

If the OPEN statement has BLOCKSIZE and BUFFERCOUNT specifiers, then the internal buffer size in bytes is the product of these specifiers. If the open statement does not have these specifiers, then the default internal buffer size is 8192 bytes. This internal buffer will grow to hold the largest single record, but will never shrink.

The default for the Fortran run-time system is to use unbuffered disk writes. That is, by default, records are written to disk immediately as each record is written instead of accumulating in the buffer to be written to disk later.

To enable buffered writes (that is, to allow the disk device to fill the internal buffer before the buffer is written to disk), use one of the following:

- The OPEN statement BUFFERED specifier
- The `-assume buffered_io` command-line option
- The `FORT_BUFFERED` run-time environment variable

The OPEN statement BUFFERED specifier takes precedence over the `-assume buffered_io` option. If neither one is set (which is the default), the `FORT_BUFFERED` environment variable is tested at run time.

The OPEN statement BUFFERED specifier applies to a specific logical unit. In contrast, the
`-assume nobuffered_io` option and the `FORT_BUFFERED` environment variable apply to all Fortran units.

Using buffered writes usually makes disk I/O more efficient by writing larger blocks of data to the disk less often. However, a system failure when using buffered writes can cause records to be lost, since they might not yet have been written to disk. (Such records would have been written to disk with the default unbuffered writes.)

When performing I/O across a network, be aware that the size of the block of network data sent across the network can impact application efficiency. When reading network data, follow the same advice for efficient disk reads, by increasing the BUFFERCOUNT. When writing data through the network, several items should be considered:

- Unless the application requires that records be written using unbuffered writes, enable buffered writes by a method described above.
- Especially with large files, increasing the BLOCKSIZE value increases the size of the block sent on the network and how often network data blocks get sent.
- Time the application when using different BLOCKSIZE values under similar conditions to find the optimal network block size.

When writing records, be aware that I/O records are written to unified buffer cache (UBC) system buffers. To request that I/O records be written from program buffers to the UBC system buffers, use the FLUSH library routine (see the *Intel® Fortran Libraries Reference*). Be aware that calling FLUSH also discards read-ahead data in user buffer.

**Specify RECL**

The sum of the record length (RECL specifier in an OPEN statement) and its overhead is a multiple or divisor of the blocksize, which is device-specific. For example, if the BLOCKSIZE is 8192 then RECL might be 24576 (a multiple of 3) or 1024 (a divisor of 8).

The RECL value should fill blocks as close to capacity as possible (but not over capacity). Such values allow efficient moves, with each operation moving as much data as possible; the least amount of space in the block is wasted. Avoid using values larger than the block capacity, because they create very inefficient moves for the excess data only slightly filling a block (allocating extra memory for the buffer and writing partial blocks are inefficient).

The RECL value unit for formatted files is always 1-byte units. For unformatted files, the RECL unit is 4-byte units, unless you specify the `-assume byterecl` option to request 1-byte units (see
`-assume byterecl`).

### Use the Optimal Record Type

Unless a certain record type is needed for portability reasons, choose the most efficient type, as follows:

- For sequential files of a consistent record size, the fixed-length record type gives the best performance.
- For sequential unformatted files when records are not fixed in size, the variable-length record type gives the best performance, particularly for BACKSPACE operations.
- For sequential formatted files when records are not fixed in size, the `Stream_LF` record type gives the best performance.

### Reading from a Redirected Standard Input File

Due to certain precautions that the Fortran run-time system takes to ensure the integrity of standard input, reads can be very slow when standard input is redirected from a file. For example, when you use a command such as `myprogram.exe < myinput.data>`, the data is read using the READ(*) or READ(5) statement, and performance is degraded. To avoid this problem, do one of the following:

- Explicitly open the file using the OPEN statement. For example:

  ```
  open(5, STATUS='OLD', FILE='myinput.dat')
  ```

- Use an environment variable to specify the input file.

To take advantage of these methods, be sure your program does not rely on sharing the standard input file.

For more information on Intel Fortran data files and I/O, see "Files, Devices, and I/O" in Volume I; on OPEN statement specifiers and defaults, see "Open Statement" in the *Intel® Fortran Language Reference*.

## Improving Run-time Efficiency

Follow these source coding guidelines to improve run-time performance. The amount of improvement in run-time performance is related to the number of times a statement is executed. For example, improving an arithmetic expression executed within a loop many times has the potential to improve performance, more than improving a similar expression executed once outside a loop.

**Avoid Small Integer and Small Logical Data Items**

Avoid using integer or logical data less than 32 bits. Accessing a 16-bit (or 8-bit) data type can make data access less efficient, especially on Itanium-based systems.

To minimize data storage and memory cache misses with arrays, use 32-bit data rather than 64-bit data, unless you require the greater numeric range of 8-byte integers or the greater range and precision of double precision floating-point numbers.

**Avoid Mixed Data Type Arithmetic Expressions**

Avoid mixing integer and floating-point (REAL) data in the same computation. Expressing all numbers in a floating-point arithmetic expression (assignment statement) as floating-point values eliminates the need to convert data between fixed and floating-point formats. Expressing all numbers in an integer arithmetic expression as integer values also achieves this. This improves run-time performance.

For example, assuming that `I` and `J` are both INTEGER variables, expressing a constant number (2.) as an integer value (2) eliminates the need to convert the data:

Inefficient Code:

```
INTEGER I, J
I = J / 2.
```

Efficient Code:

```
INTEGER I, J
I = J / 2
```

You can use different sizes of the same general data type in an expression with minimal or no effect on run-time performance. For example, using REAL, DOUBLE PRECISION, and COMPLEX floating-point numbers in the same floating-point arithmetic expression has minimal or no effect on run-time performance.

**Use Efficient Data Types**

In cases where more than one data type can be used for a variable, consider selecting the data types based on the following hierarchy, listed from most to least efficient:

- Integer (also see above example)
- Single-precision real, expressed explicitly as REAL, REAL (KIND=4), or REAL*4
- Double-precision real, expressed explicitly as DOUBLE PRECISION, REAL (KIND=8), or REAL*8
- Extended-precision real, expressed explicitly as REAL (KIND=16) or REAL*16

However, keep in mind that in an arithmetic expression, you should avoid mixing integer and floating-point (REAL) data (see example in the previous subsection).

**Avoid Using Slow Arithmetic Operators**

Before you modify source code to avoid slow arithmetic operators, be aware that optimizations convert many slow arithmetic operators to faster arithmetic operators. For example, the compiler optimizes the expression `H=J**2` to be `H=J*J`.

Consider also whether replacing a slow arithmetic operator with a faster arithmetic operator will change the accuracy of the results or impact the maintainability (readability) of the source code.

Replacing slow arithmetic operators with faster ones should be reserved for critical code areas. The following hierarchy lists the Intel Fortran arithmetic operators, from fastest to slowest:

- Addition (+), subtraction (-), and floating-point multiplication (*)
- Integer multiplication (*)
- Division (/)
- Exponentiation (**)

**Avoid Using EQUIVALENCE Statements**

Avoid using EQUIVALENCE statements. EQUIVALENCE statements can:

- Force unaligned data or cause data to span natural boundaries.
- Prevent certain optimizations, including:
    - Global data analysis under certain conditions (see -O2 in Setting Optimization with -On options).
    - Implied-`DO` loop collapsing when the control variable is contained in an `EQUIVALENCE` statement

### Use Statement Functions and Internal Subprograms

Whenever the Intel Fortran compiler has access to the use and definition of a subprogram during compilation, it may choose to inline the subprogram. Using statement functions and internal subprograms maximizes the number of subprogram references that will be inlined, especially when multiple source files are compiled together at optimization level `-O3`.

For more information, see Efficient Compilation.

### Code DO Loops for Efficiency

Minimize the arithmetic operations and other operations in a DO loop whenever possible. Moving unnecessary operations outside the loop will improve performance (for example, when the intermediate nonvarying values within the loop are not needed).

For more information on loop optimizations, see Pipelining for Itanium®-based Applications and Loop Unrolling; on the syntax of Intel Fortran statements, see the *Intel® Fortran Language Reference*.

## Using Intrinsics for Itanium®-based Systems

Intel® Fortran supports all standard Fortran intrinsic procedures and, in addition, provides  Intel-specific intrinsic procedures to extend the functionality of the language. Intel Fortran intrinsic procedures are provided in the library `libintrins.a`. For more information on intrinsic procedures, see the *Intel® Fortran Language Reference*.

This topic provides examples of the Intel-extended intrinsics that are helpful in developing efficient applications.

### CACHESIZE Intrinsic (Itanium® Compiler)

Intrinsic CACHESIZE (n)is used only with the Intel® Itanium® Compiler. CACHESIZE (n) returns the size in kilobytes of the cache at level n; 1 represents the first level cache. Zero is returned for a nonexistent cache level.

This intrinsic can be used in many scenarios where application programmers would like to tailor their algorithms for the target processor's cache hierarchy. For example, an application may query the cache size and use it to select block sizes in algorithms that operate on matrices.

```
subroutine foo (level)
integer level
if (cachesize(level) > threshold) then
call big_bar()
```

```
else
call small_bar()
end if
end subroutine
```

## Coding Guidelines for Intel® Architectures

This topic provides general guidelines for coding practices and techniques for:

- IA-32 architecture supporting MMX(TM) technology and Streaming SIMD Extensions (SSE) and Streaming SIMD Extensions 2 (SSE2)
- Itanium® architecture

This topic describes practices, tools, coding rules and recommendations associated with the architecture features that can improve the performance on IA-32 and Itanium processor families. For details about optimization for IA-32 processors, see the Intel® Architecture Optimization Reference Manual. For all details about optimization for Itanium processor family, see the Intel Itanium 2 Processor Reference Manual for Software Development and Optimization.

> **Note**
> If a guideline refers to a particular architecture only, this architecture is explicitly named. The default is for both IA-32 and Itanium architectures.

Performance of compiler-generated code may vary from one compiler to another. The Intel® Fortran Compiler generates code that is highly optimized for Intel architectures. You can significantly improve performance by using various compiler optimization options. In addition, you can help the compiler to optimize your Fortran program by following the guidelines described here.

To achieve optimum processor performance in your Fortran application, do the following:

- avoid memory access stalls
- ensure good floating-point performance
- ensure good SIMD integer performance
- use vectorization.

The coding practices, rules, and recommendations described here will contribute to optimizing the performance on Intel architecture-based processors.

### Memory Access

The Intel compiler lays out Fortran arrays in column-major order. For example, in a two-dimensional array, elements `A(22, 34)` and `A(23, 34)` are contiguous in memory. For best performance, code arrays so that inner loops access them in a contiguous manner. Consider the following examples.

The code in example 1 will likely have higher performance than the code in example 2.

**Example 1**

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

The code above illustrates access to arrays `A` and `B` in the inner loop `I` in a contiguous manner which results in good performance.

**Example 2**

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

The code above illustrates access to arrays `A` and `B` in inner loop `J` in a non-contiguous manner which results in poor performance.

The compiler itself can transform the code so that inner loops access memory in a contiguous manner. To do that, you need to use advanced optimization options, such as `-O3` for both IA-32 and Itanium architectures, and -O3 and `-ax{K|W|N|B|P}` for IA-32 only.

**Memory Layout**

Alignment is a very important factor in ensuring good performance. Aligned memory accesses are faster than unaligned accesses. If you use the interprocedural optimization on multiple files (the `-ipo` option), the compiler analyzes the code and decides whether it is beneficial to pad arrays so that they start from an aligned boundary. Multiple arrays specified in a single common block can impose extra constraints on the compiler. For example, consider the following COMMON statement:

```
COMMON /AREA1/ A(200), X, B(200)
```

If the compiler added padding to align `A(1)` at a 16-byte aligned address, the element `B(1)` would not be at a 16-byte aligned address. So it is better to split `AREA1` as follows.

```
COMMON /AREA1/ A(200)
COMMON /AREA2/ X
COMMON /AREA3/ B(200)
```

The above code provides the compiler maximum flexibility in determining the padding required for both `A` and `B`.

## Optimizing for Floating-point Applications

To improve floating-point performance, observe these general rules:

- Avoid exceeding representable ranges during computation since handling these cases can have a performance impact. Use REAL variables in single-precision format unless the extra precision obtained through DOUBLE or REAL*8 variables is required. Using variables with a larger precision formation will also increase memory size and bandwidth requirements.
- For IA-32 only: Avoid repeatedly changing rounding modes between more than two values, which can lead to poor performance when the computation is done using non-SSE instructions. Hence avoid using `FLOOR` and `TRUNC` instructions together when generating non-SSE code. The same applies for using `CEIL` and `TRUNC`.

  Another way to avoid the problem is to use the `-x{K|W|N|B|P}` options to do the computation using SSE instructions.

- Reduce the impact of denormal exceptions for both architectures as described below.

### Denormal Exceptions

Floating point computations with underflow can result in denormal values that have an adverse impact on performance.

For IA-32: Take advantage of the SIMD capabilities of Streaming SIMD Extensions (SSE), and Streaming SIMD Extensions 2 (SSE2) instructions. The `-x{K|W|N|B|P}` options enable the flush-to-zero (FTZ) mode in SSE and SSE2 instructions, whereby underflow results are automatically converted to zero, which improves application performance. In addition, the `-xP` option also enables the denormals-are-zero (DAZ) mode, whereby denormals are converted to zero on input, further improving performance. An application developer willing to trade pure IEEE-754 compliance for speed would benefit from these options. For more information on FTZ and DAZ, see Setting FTZ and DAZ Flags and "Floating-point Exceptions" in the Intel® Architecture Optimization Reference Manual.

For Itanium architecture: enable flush-to-zero (FTZ) mode with the -ftz option set by -O3 option.

### Auto-vectorization

Many applications significantly increase their performance if they can implement vectorization, which uses streaming SIMD SSE2 instructions for the main computational loops. The Intel Compiler turns vectorization on (auto-vectorization) or you can implement it with compiler directives.

See Auto-vectorization (IA-32 Only) section for complete details.

### Creating Multithreaded Applications

The Intel Fortran Compiler and the Intel® Threading Toolset have the capabilities that make developing multithreaded application easy. See Parallel Programming with Intel Fortran. Multithreaded applications can show significant benefit on multiprocessor Intel symmetric multiprocessing (SMP) systems or on Intel processors with Hyper-Threading technology.

# Analyzing and Timing Your Application

## Using Intel Performance Analysis Tools

Intel offers a variety of application performance tools that are optimized to take advantage of the Intel architecture-based processors. You can employ these tools for developing the most efficient programs without having to write assembly code.

The following performance tools help you analyze your application and find and resolve problem areas:

- Intel® Debugger (IDB)

  The IDB debugger provides extensive support for debugging programs through a command-line or graphical user interface.

- Intel® VTune(TM) Performance Analyzer

  The VTune analyzer collects, analyzes, and provides Intel architecture-specific software performance data from the system-wide view down to a specific module, function, and instruction in your code. For information, see http://www.intel.com/software/products/vtune/.

- Intel® Threading Tools. The Intel Threading Tools consist of the following:
    - Intel® Thread Checker
    - Intel® Thread Profiler

For general information, see http://www.intel.com/software/products/threadtool.htm.

## Timing Your Application

One of the performance indicators is your application timing. Use the `time` command to provide information about program performance. The following considerations apply to timing your application:

- Run program timings when other users are not active. Your timing results can be affected by one or more CPU-intensive processes also running while doing your timings.
- Try to run the program under the same conditions each time to provide the most accurate results, especially when comparing execution times of a previous version of the same program. Use the same CPU system (model, amount of memory, version of the operating system, and so on) if possible.
- If you do need to change systems, you should measure the time using the same version of the program on both systems, so you know each system's effect on your timings.
- For programs that run for less than a few seconds, run several timings to ensure that the results are not misleading. Overhead functions like loading shared libraries might influence short timings considerably.

Use the `time` command and specify  the name of the executable program to provide the following:

- The elapsed, real, or "wall clock" time, which will be greater than the total charged actual CPU time.
- Charged actual CPU time, shown for both system and user execution. The total actual CPU time is the sum of the actual user CPU time and actual system CPU time.

### Example

In the following example timings, the sample program being timed displays the following line:

```
Average of all the numbers is:    4368488960.000000
```

Using the Bourne* shell, the following program timing reports that the program uses 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use) and 2.46 seconds of elapsed time:

```
$ time a.out

Average of all the numbers is:
  4368488960.000000

real    0m2.46s

user    0m0.61s

sys     0m0.58s
```

Using the C shell, the following program timing reports 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use), about 4 seconds (0:04) of elapsed time, the use of 28% of available CPU time, and other information:

```
% time a.out

Average of all the numbers is:   4368488960.000000

0.61u 0.58s 0:04 28% 78+424k 9+5io 0pf+0w
```

Using the bash shell, the following program timing reports that the program uses 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use) and 2.46 seconds of elapsed time:

```
[user@system user]$ time ./a.out

Average of all the numbers is:   4368488960.000000

elapsed  0m2.46s

user     0m0.61s

sys      0m0.58s
```

Timings that indicate a large amount of system time is being used may suggest excessive I/O,  a condition worth investigating.

If your program displays a lot of text, you can redirect the output from the program on the time command line. Redirecting output from the program will change the times reported because of reduced screen I/O.

For more information, see `time(1)`.

In addition to the `time` command, you might consider modifying the program to call routines within the program to measure execution time. For example, use the Intel Fortran intrinsic procedures, such as SECNDS, DCLOCK, CPU_TIME, SYSTEM_CLOCK, TIME, and DATE_AND_TIME. See "Intrinsic Procedures" in the *Intel® Fortran Language Reference*.

# Compiler Optimizations

## Compiler Optimizations Overview

Intel® Fortran Compiler optimizations enable you to enhance the performance of your application. Optimization options are described in the following sections:

- Optimizing the compilation process (includes stack, alignment, and symbol visibility attribute options)
- Optimizing different application types
- Floating-point arithmetic operations
- Optimizing applications for specific processors
- Interprocedural optimizations (IPO)
- Profile-guided optimizations
- High-level language optimizations

In addition to optimizations invoked by the compiler command-line options, other performance-enhancing features such as directives, intrinsics, run-time library routines and various utilities are provided. These features are discussed in the Optimization Support Features section.

## Optimizing the Compilation Process

### Optimizing the Compilation Process Overview

This section describes the Intel® Fortran Compiler options that optimize the compilation process. By default, the compiler converts source code directly to an executable file. Appropriate options enable you not only to control the process and obtain desired output file produced by the compiler, but also make the compilation itself more efficient.

A group of options monitors the outcome of Intel compiler-generated code without interfering with the way your program runs. These options control some computation aspects, such as allocating the stack memory, setting or modifying variable settings, and defining the use of some registers.

The options in this section provide you with the following capabilities of efficient compilation:

- Automatic allocation of variables and stacks
- Aligning data
- Symbol visibility attribute options

### Efficient Compilation

Understandably, efficient compilation contributes to performance improvement. Before you analyze your program for performance improvement, and improve program performance, you should think of efficient compilation itself. Based on the analysis of your application, you can decide which Intel Fortran Compiler optimizations and command-line options can improve the run-time performance of your application.

## Efficient Compilation Techniques

The efficient compilation techniques can be used during the earlier stages and later stages of program development.

During the earlier stages of program development, you can use incremental compilation with minimal optimization. For example:

**`ifort -c -g -O0`** ***`sub2.f90`*** (generates object file of `sub2`)

**`ifort -c -g -O0`** ***`sub3.f90`*** (generates object file of `sub3`)

**`ifort -o`** ***`main`*** **`-g -O0`** ***`main.f90 sub2.o sub3.o`***

The above commands turn off all compiler default optimizations (for example, `-O2`) with `-O0.` You can use the `-g` option to generate symbolic debugging information and line numbers in the object code for all routines in the program for use by a source-level debugger. The `main` file created in the third command above contains symbolic debugging information as well.

During the later stages of program development, you should specify multiple source files together and use an optimization level of at least `-O2` (default) to allow more optimizations to occur. For instance, the following command compiles all three source files together using the default level of optimization, `-O2`:

**`ifort -o`** ***`main main.f90  sub2.f90  sub3.f90`***

Compiling multiple source files lets the compiler examine more code for possible optimizations, which results in:

- Inlining more procedures
- More complete data flow analysis
- Reducing the number of external references to be resolved during linking

For very large programs, compiling all source files together may not be practical. In such instances, consider compiling source files containing related routines together using multiple `ifort` commands, rather than compiling source files individually.

## Options That Improve Run-Time Performance

The table below lists the options in alphabetical order that can directly improve run-time performance. Most of these options do not affect the accuracy of the results, while others improve run-time performance but can change some numeric results. The Intel Fortran Compiler performs some optimizations by default unless you turn them off by corresponding command-line options. Additional optimizations can be enabled or disabled using command options.

| Option | Description |
|---|---|
| `-align` *`keyword`* | Analyzes and reorders memory layout for variables and arrays.<br>Controls whether padding bytes are added between data items within common blocks, derived-type data, and record structures to make the data items naturally aligned. |
| `-ax{K\|W\|N\|B\|P}`<br>IA-32 and Intel® Extended Memory 64 Technology (Intel® EM64T) systems only | Optimizes your application's performance for specific processors. Regardless of which `-ax` suboption you choose, your application is optimized to use all the benefits of that processor with the resulting binary file capable of being run on any Intel IA-32 processor. |
| `-fast` | Enables a collection of optimizations for run-time performance. |
| `-O1` | Optimizes to favor code size and code locality. See Setting Optimizations with -On Options. |
| -O2 | Optimizes for code speed. Sets performance-related options. Setting Optimizations with -On Options. |
| `-O3` | Activates loop transformation optimizations. Setting Optimizations with -On Options. |
| `-openmp` | Enables the parallelizer to generate multithreaded code based on the OpenMP* directives. |
| `-parallel` | Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel. |
| -qp | Requests profiling information, which you can use to identify those parts of your program where improving source code efficiency would most likely improve run-time performance. After you modify the appropriate source code, recompile the program and test the run-time performance. |
| `-tpp{n}` | Optimizes your application's performance for specific Intel processors. See Targeting a Processor, `-tpp{n}`. |
| `-unroll`*`n`* | Specifies the number of times a loop is unrolled (*n*) when specified with optimization level `-O3`. If you omit *n* in `-unroll`, the optimizer determines how many times loops can be unrolled. |

## Options That Slow Down the Run-time Performance

The table below lists options in alphabetical order that can slow down the run-time performance. Some applications that require floating-point exception handling or rounding might need to use the `-fpe`*n* dynamic option. Other applications might need to use the `-assume dummy_aliases` or `-vms` options for compatibility reasons. Other options that can slow down the run-time performance are primarily for troubleshooting or debugging purposes.

The following table lists the options that can slow down run-time performance.

| Option | Description |
|---|---|
| `-assume dummy_aliases` | Forces the compiler to assume that dummy (formal) arguments to procedures share memory locations with other dummy arguments or with variables shared through use association, host association, or common block use. These program semantics slow performance, so you should specify `-assume` *`dummy_aliases`* only for the called subprograms that depend on such aliases.<br><br>The use of dummy aliases violates the Fortran 77 and Fortran 95/90 standards but occurs in some older programs. |
| `-check bounds` | Generates extra code for array bounds checking at run time. |
| `-check overflow` | Generates extra code to check integer calculations for arithmetic overflow at run time. Once the program is debugged, omit this option to reduce executable program size and slightly improve run-time performance. |
| `-fpe 3` | Using this option enables certain types of floating-point exception handling, which can be expensive. |
| -g | Generate extra symbol table information in the object file. Specifying this option also reduces the default level of optimization to `-O0` or `-O0` (no optimization).<br><br>**Note**<br><br>The `-g` option only slows your program down when no optimization level is specified, in which case `-g` turns on `-O0`, which slows the compilation down. If `-g`, `-O2` are specified, the code runs very much the same speed as if `-g` were not specified. |

| | |
|---|---|
| `-O0` | Turns off optimizations. Can be used during the early stages of program development or when you use the debugger. |
| `-save` | Forces the local variables to retain their values from the last invocation terminated. This may change the output of your program for floating-point values as it forces operations to be carried out in memory rather than in registers, which in turn causes more frequent rounding of your results. |
| `-vms` | Controls certain VMS-related run-time defaults, including alignment. If you specify the `-vms` option, you may need to also specify the `-align records` option to obtain optimal run-time performance. |

## Little-endian-to-Big-endian Conversion

The Intel Fortran Compiler can write unformatted sequential files in big-endian format and also can read files produced in big-endian format by using the little-endian-to-big-endian conversion feature.

Both on IA-32-based processors and on Itanium®-based processors, Intel Fortran handles internal data in little-endian format. The little-endian-to-big-endian conversion feature is intended for Fortran unformatted input/output operations in unformatted sequential files. The feature enables:

- processing of the files developed on processors that accept big-endian data format
- producing big-endian files for such processors on little-endian systems.

The little-endian-to-big-endian conversion is accomplished by the following operations:

- The WRITE operation converts little-endian format to big-endian format.
- The READ operation converts big-endian format to little-endian format.

The feature enables the conversion of variables and arrays (or array subscripts) of basic data types. Derived data types are not supported.

### Little-to-Big Endian Conversion Environment Variable

In order to use the little-endian-to-big-endian conversion feature, specify the numbers of the units to be used for conversion purposes by setting the `F_UFMTENDIAN` environment variable. Then, the READ/WRITE statements that use these unit numbers, will perform relevant conversions. Other READ/WRITE statements will work in the usual way.

In the general case, the variable consists of two parts divided by a semicolon. No spaces are allowed inside the `F_UFMTENDIAN` value. The variable has the following syntax:

```
F_UFMTENDIAN=MODE | [MODE;] EXCEPTION
```

where:

```
MODE = big | little
EXCEPTION = big:ULIST | little:ULIST | ULIST
ULIST = U | ULIST,U
U = decimal | decimal -decimal
```

- `MODE` defines current format of data, represented in the files; it can be omitted.
  The keyword `little` means that the data have little endian format and will not be converted. This keyword is a default.
  The keyword `big` means that the data have big endian format and will be converted. This keyword may be omitted together with the colon.
- `EXCEPTION` is intended to define the list of exclusions for `MODE`; it can be omitted. `EXCEPTION` keyword (`little` or `big`) defines data format in the files that are connected to the units from the `EXCEPTION` list. This value overrides `MODE` value for the units listed.
- Each list member `U` is a simple unit number or a number of units. The number of list members is limited to 64.
  `decimal` is a non-negative decimal number less than $2^{32}$.

Converted data should have basic data types, or arrays of basic data types. Derived data types are disabled.

Command lines for variable setting with different shells:

Sh: **`export F_UFMTENDIAN=MODE;EXCEPTION`**

Csh: **`setenv F_UFMTENDIAN MODE;EXCEPTION`**

**Note**
Environment variable values should be enclosed in quotes if a semicolon is present.

**Another Possible Environment Variable Setting**

The environment variable can also have the following syntax:

```
F_UFMTENDIAN=u[,u] . . .
```

Command lines for the variable setting with different shells:

- Sh: **export F_UFMTENDIAN=u[,u] . . .**
- Csh: **setenv F_UFMTENDIAN u[,u] . . .**

See error messages that may be issued during the little endian – big endian conversion. They are all fatal. You should contact Intel if such errors occur.

**Usage Examples**

1. `F_UFMTENDIAN=big`

   All input/output operations perform conversion from big-endian to little-endian on READ and from little-endian to big-endian on WRITE.

2. `F_UFMTENDIAN="little;big:10,20"`
   or `F_UFMTENDIAN=big:10,20`
   or `F_UFMTENDIAN=10,20`

   In this case, only on unit numbers 10 and 20 the input/output operations perform big-little endian conversion.

3. `F_UFMTENDIAN="big;little:8"`

   In this case, on unit number 8 no conversion operation occurs. On all other units, the input/output operations perform big-little endian conversion.

4. `F_UFMTENDIAN=10-20`

   Define 10, 11, 12, ..., 19, 20 units for conversion purposes; on these units, the input/output operations perform big-little endian conversion.

5. Assume you set `F_UFMTENDIAN=10,100` and run the following program.

```
integer*4   cc4
integer*8   cc8
integer*4   c4
integer*8   c8
c4 = 456
c8 = 789

C  prepare a little endian representation of data

open(11,file='lit.tmp',form='unformatted')
write(11) c8
write(11) c4
close(11)

C  prepare a big endian representation of data
```

```
open(10,file='big.tmp',form='unformatted')
write(10) c8
write(10) c4
close(10)

C  read big endian data and operate with them on
C  little endian machine.

open(100,file='big.tmp',form='unformatted')
read(100) cc8
read(100) cc4

C  Any operation with data, which have been read

C    . . .
close(100)
stop
end
```

Now compare `lit.tmp` and `big.tmp` files with the help of od utility.

```
> od -t x4 lit.tmp

0000000 00000008 00000315 00000000 00000008
0000020 00000004 000001c8 00000004
0000034

> od -t x4 big.tmp

0000000 08000000 00000000 15030000 08000000
0000020 04000000 c8010000 04000000
0000034
```

You can see that the byte order is different in these files.

## Default Compiler Optimizations

If you invoke the Intel® Fortran Compiler without specifying any compiler options, the default state of each option takes effect. The following tables summarize the options whose default status is ON as they are required for Intel Fortran Compiler default operation. The tables group the options by their functionality.

For the default states and values of all options, see the Alphabetical Quick Reference Guide in the *Intel® Fortran Compiler Options Quick Reference*. The table provides links to the sections describing the functionality of the options. If an option has a default value, such value is indicated.

Depending on your application requirements, you can disable one or more options. For general methods of disabling optimizations, see Volume I.

The following tables list all options that compiler uses for its default optimizations.

**Data Setting and Fortran Language Conformance**

| Default Option | Description |
|---|---|
| `-align records` | Analyzes and reorders memory layout for variables and arrays. |
| `-align rec8byte` | Specifies 8-byte boundary for alignment constraint. |
| `-altparam` | Specifies that the alternate form of parameter constant declarations is recognized. |
| `-ansi_alias` | Enables assumption of the program's ANSI conformance. |
| `-assume cc_omp` | Enables OpenMP conditional compilation directives. |
| `-ccdefault default` | Specifies default carriage control for units 6 and `*`. |
| `-double_size 64` | Defines DOUBLE PRECISION declarations, constants, functions, and intrinsics as REAL*8. |
| `-dps` | Enables DEC* parameter statement recognition. |
| `-error_limit 30` | Specifies the maximum number of error-level or fatal-level compiler errors permissible. |
| `-fpe3` | Specifies floating-point exception handling at run time for the main program. |
| `-integer_size 32` | Makes default integer and logical variables 4 bytes long. INTEGER and LOGICAL declarations are treated as (KIND=4). |
| `-pad` | Enables changing variable and array memory layout. |
| `-pc80`<br>IA-32 only | `-pc{32\|64\|80}` enables floating-point significand precision control as follows: `-pc32` to 24-bit significand, `-pc64` to 53-bit significand, and `-pc80` to 64-bit significand. |
| `-real_size 64` | Specifies the size of `REAL` and `COMPLEX` declarations, constants, functions, and intrinsics. |

| | |
|---|---|
| `-save` | Saves all variables in static allocation. Disables `-auto`, that is, disables setting all variables AUTOMATIC. |
| `-Zp8` | `-Zp`*`n`* specifies alignment constraint for structures on 1-, 2-, 4-, 8-, or 16-byte boundary. To disable, use `-noalign or -Zp1`. |

**Optimizations**

| Default Option | Description |
|---|---|
| `-assume` *`cc_omp`* | Enables OpenMP conditional compilation directives. |
| `-fp`<br>IA-32 only | Disables the use of the ebp register in optimizations. Directs to use the ebp-based stack frame for all functions. |
| `-fpe3` | Specifies floating-point exception handling at run time for the main program. `-fpe0` disables the option. |
| `-IPF_fltacc-`<br>Itanium® compiler | Enables the compiler to apply optimizations that affect floating-point accuracy. |
| `-IPF_fma`<br>Itanium compiler | Enables the contraction of floating-point multiply and add/subtract operations into a single operation. |
| `-IPF_fp_speculation` *`fast`*<br>Itanium compiler | Sets the compiler to speculate on floating-point operations. `-IPF_fp_speculationoff` disables this optimization. |
| `-O, -O2` | Optimizes for maximum speed. |
| `-openmp_report1` | Indicates loops, regions, and sections parallelized. |
| `-opt_report_level`*`min`* | Specifies the minimal level of the optimizations report. |
| `-par_report1` | Indicates loops successfully auto-parallelized. |
| `-tpp2`<br>Itanium compiler | Optimizes code for the Intel® Itanium® 2 processor for Itanium-based applications. Generated code is compatible with the Itanium processor. |

| | |
|---|---|
| `-tpp7`<br>IA-32 only | Optimizes code for the Intel® Pentium® 4 and Intel® Xeon(TM) processor for IA-32 applications. |
| `-unroll` | `-unroll[`*n*`]`: omit *n* to let the compiler decide whether to perform unrolling or not (default).<br>Specify *n* to set maximum number of times to unroll a loop.<br>The Itanium compiler currently uses only<br>*n* = 0, `-unroll0` (disabled option) for compatibility. |
| `-vec_report1` | Indicates loops successfully vectorized. |

### Disabling Default Options

To disable an option, you can generally use one of the following:

- To disable one or a group of optimization options, use `-O0 option`. For example:

  **`ifort -O2 -O0`** ***`input_file(s)`***

  **Note**
  The `-O0` option is part of a mutually-exclusive group of options that includes `-O0`, `-O`, `-O1`, `-O2`, and `-O3`. The last of any of these options specified on the command line will override the previous options from this group.

- To disable options that include optional "-" shown as `[-]`, use that version of the option in the command line, for example: `-ftz-`.
- To disable options that have an `{n}` parameter, use n=0 version, for example: `-unroll0`.

  **Note**
  If there are enabling and disabling versions of options on the line, the last one takes precedence.

## Using Compilation Options

### Stacks: Automatic Allocation and Checking

The options in this group enable you to control the computation of stacks and variables in the compiler generated code.

**Automatic Allocation of Variables**

### `-auto`

The `-auto` option specifies that locally declared variables are allocated to the run-time stack rather than static storage. If variables defined in a procedure do not have the SAVE or ALLOCATABLE attribute, they are allocated to the stack. It does not affect variables that appear in an EQUIVALENCE or SAVE statement, or those that are in COMMON.

`-auto` is the same as `-automatic` and `-nosave`.

`-auto` may provide a performance gain for your program, but if your program depends on variables having the same value as the last time the routine was invoked, your program may not function properly. Variables that need to retain their values across routine calls should appear in a SAVE statement.

If you specify `-recursive` or `-openmp`, the default is `-auto`.

### `-auto_scalar`

The `-auto_scalar` option causes allocation of local scalar variables of intrinsic type INTEGER, REAL, COMPLEX, or LOGICAL to the stack. This option does not affect variables that appear in an EQUIVALENCE or SAVE statement, or those that are in COMMON.

`-auto_scalar` may provide a performance gain for your program, but if your program depends on variables having the same value as the last time the routine was invoked, your program may not function properly. Variables that need to retain their values across subroutine calls should appear in a SAVE statement. This option is similar to `-auto`, which causes all local variables to be allocated on the stack. The difference is that `-auto_scalar` allocates only scalar variables of the stated above intrinsic types to the stack.

`-auto_scalar` enables the compiler to make better choices about which variables should be kept in registers during program execution.

### `-save, -zero`

The `-save` option is opposite of `-auto`: the `-save` option saves all variables in static allocation except local variables within a recursive routine. If a routine is invoked more than once, this option forces the local variables to retain their values between the invocations. The `-save` option ensures that the final results on the exit of the routine is saved on memory and can be reused at the next occurrence of that routine. This may cause some performance degradation as it causes more frequent rounding of the results.

When the compiler optimizes the code, the results are stored in registers. `-save` is the same as `-noauto`.

The `-zero[-]` option initializes to zero all local scalar variables of intrinsic type INTEGER, REAL, COMPLEX, or LOGICAL, which are saved and not initialized yet. Used in conjunction with `-save`. The default is `-zero-`.

#### Summary

There are three choices for allocating variables: `-save`, `-auto`, and `-auto_scalar`. Only one of these three can be specified. The correlation among them is as follows:

- `-save` disables `-auto`, sets `-noautomatic`, and allocates all variables not marked AUTOMATIC to static memory.
- `-auto` disables `-save`, sets `-automatic`, and allocates all variables—scalars and arrays of all types—not marked SAVE to the stack.
- `-auto_scalar`:
    - It makes local scalars of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL automatic.
    - This is the default; there is no `-noauto_scalar`; however, `-recursive` or `-openmp` disables `-auto_scalar` and makes `-auto` the default.

#### Checking the Floating-point Stack State (IA-32 only), `-fpstkchk`

The `-fpstkchk` option (IA-32 only) checks whether a program makes a correct call to a function that should return a floating-point value. If an incorrect call is detected, the option places a code that marks the incorrect call in the program.

When an application calls a function that returns a floating-point value, the returned floating-point value is supposed to be on the top of the floating-point stack. If return value is not used, the compiler must pop the value off of the floating-point stack in order to keep the floating-point stack in correct state.

If the application calls a function, either without defining or incorrectly defining the function's prototype, the compiler does not know whether the function must return a floating-point value, and the return value is not popped off of the floating-point stack if it is not used. This can cause the floating-point stack overflow.

The overflow of the stack results in two undesirable situations:

- A NaN value gets involved in the floating-point calculations
- The program results become unpredictable; the point where the program starts making errors can be arbitrarily far away from the point of the actual error.

The `-fpstkchk` option marks the incorrect call and makes it easy to find the error.

**Note**

This option causes significant code generation after every function/subroutine call to insure a proper state of a floating-point stack and slows down compilation. It is meant only as a debugging aid for finding floating point stack underflow/overflow problems, which can be otherwise hard to find.

#### Aliases

### -common_args

The `-common_args` option assumes that the "by-reference" subprogram arguments may have aliases of one another.

#### Preventing CRAY* Pointer Aliasing

Option `-safe_cray_ptr` specifies that the CRAY* pointers do not alias with other variables. The default is OFF.

Consider the following example:

```
pointer (pb, b)
pb = getstorage()
do i = 1, n
b(i) = a(i) + 1
enddo
```

When `-safe_cray_ptr` is not specified (default), the compiler assumes that b and a are aliased. To prevent such an assumption, specify this option, and the compiler will treat `b(i)` and `a(i)` as independent of each other.

However, if the variables are intended to be aliased with CRAY pointers, using the `-safe_cray_ptr` option produces incorrect result. For the code example below, `-safe_cray_ptr` should not be used.

```
pb = loc(a(2))
do i=1, n
b(i) = a(i) +1
enddo
```

### -ansi_alias

The `-ansi_alias[-]` enables (default) or disables the compiler to assume that the program adheres to the ANSI Fortran type aliasablility rules. For example, an object of type real cannot be accessed as an integer. You should see the ANSI standard for the complete set of rules.

The option directs the compiler to assume the following:

- Arrays are not accessed out of arrays' bounds.
- Pointers are not cast to non-pointer types and vice-versa.
- References to objects of two different scalar types cannot alias. For example, an object of type INTEGER cannot alias with an object of type real or an object of type real cannot alias with an object of type double `precision`.

If your program satisfies the above conditions, setting the `-ansi_alias` option will help the compiler better optimize the program. However, if your program may not satisfy one of the above conditions, the option must be disabled, as it can lead the compiler to generate incorrect code.

The synonym of `-ansi_alias` is `-assume [no]dummy_aliases`.

**Alignment Options**

**`-align recnbyte` or `-Zp[n]`**

Use the `-align recnbyte` (or `-Zp[n]`) option to specify the alignment constraint for structures on n-byte boundaries (where n = 1, 2, 4, 8, or 16 with `-Zp[n]`).

When you specify this option, each structure member after the first is stored on either the size of the member type or n-byte boundaries (where n = 1, 2, 4, 8, or 16), whichever is smaller.

For example, to specify 2 bytes as the packing boundary (or alignment constraint) for all structures and unions in the file `prog1.f`, use the following command:

```
ifort -Zp2 prog1.f
```

The default for IA-32 and Itanium-based systems is `-align rec8byte` or `-Zp8`. The `-Zp16` option enables you to align Fortran structures such as common blocks. For information on Fortran record structures, see STRUCTURE statement in the *Intel® Fortran Language Reference*.

If you specify `-Zp` (omit n), structures are packed at 8-byte boundary.

**`-align` and `-pad`**

The `-align` option is a front-end option that changes alignment of variables in a common block.

Example:

```
common /block1/ch,doub,ch1,int
integer int
character(len=1) ch, ch1
double precision doub
end
```

The `-align` option enables padding inserted to ensure alignment of doub and `int` on natural alignment boundaries. The `-noalign` option disables padding.

The `-align` option applies mainly to structures. It analyzes and reorders memory layout for variables and arrays and basically functions as `-Zp{n}`. You can disable either option with `-noalign`.

For `-align` *`keyword`* options, see your User's Guide, Volume I.

The `-pad` option is effectively not different from `-align` when applied to structures and derived types. However, the scope of `-pad` is greater because it applies also to common blocks, derived types, sequence types, and VAX* structures.

**Recommendations on Controlling Alignment with Options**

The following options control whether the Intel Fortran compiler adds padding (when needed) to naturally align multiple data items in common blocks, derived-type structures, and Intel Fortran record structures:

- By default (with `-O2`), the `-align commons` option requests that data in common blocks be aligned on up to 4-byte boundaries, by adding padding bytes as needed.

  The `-align nocommons` arbitrarily aligns the bytes of common block data. In this case, unaligned data can occur unless the order of data items specified in the COMMON statement places the largest numeric data item first, followed by the next largest numeric data (and so on), followed by any character data.

- By default (with `-O2`), the `-align dcommons` option requests that data in common blocks be aligned on up to 8-byte boundaries, by adding padding bytes as needed.
  The `-align nodcommons` arbitrarily aligns the bytes of data items in a common data.

  Specify the `-align` *`dcommons`* option for applications that use common blocks, unless your application has no unaligned data or, if the application might have unaligned data, all data items are four bytes or smaller. For applications that use common blocks where all data items are four bytes or smaller, you can specify `-align commons` instead of `-align dcommons`.

- The `-align norecords` option requests that multiple data items in derived-type data and record structures (an Intel Fortran extension) be aligned arbitrarily on byte boundaries instead of being naturally aligned. The default is `-align records`.
- The `-align records` option requests that multiple data items in record structures (extension) and derived-type data without the SEQUENCE statement be naturally aligned, by adding padding bytes as needed.
- The `-align recnbyte` option requests that fields of records and components of derived types be aligned on either the size byte boundary specified or the boundary that will naturally align them, whichever is smaller. This option does not affect whether common blocks are naturally aligned or packed.
- The `-align sequence` option controls alignment of derived-type components declared with the SEQUENCE statement (sequenced components).

The `-align nosequence` option means that sequenced components are packed regardless of any other alignment rules. Note that `-align none` implies `-align nosequence`.

The `-align` *`sequence`* option means that sequenced components obey whatever alignment rules are currently in use. Consequently, since `-align record` is a default value, then `-align sequence` alone on the command line will cause the components of these derived types to be naturally aligned.

The default behavior is that multiple data items in derived-type structures and record structures *will* be naturally aligned; data items in common blocks will not (`-align records` with `-align nocommons` ). In derived-type structures, using the SEQUENCE statement prevents `-align records` from adding needed padding bytes to naturally align data items.

## Symbol Visibility Attribute Options

Applications that do not require symbol preemption or position-independent code can obtain a performance benefit by taking advantage of the generic ABI visibility attributes.

**Note**
The visibility options are supported by both IA-32 and Itanium compilers, but currently the optimization benefits are for Itanium-based systems only.

### Global Symbols and Visibility Attributes

A global symbol is a symbol that is visible outside the compilation unit in which it is declared (compilation unit is a single-source file with its include files). Each global symbol definition or reference in a compilation unit has a visibility attribute that controls how it may be referenced from outside the component in which it is defined. The values for visibility are defined in the table that follows.

| | |
|---|---|
| `EXTERN` | The compiler must treat the symbol as though it is defined in another component. This means that the compiler must assume that the symbol will be overridden (preempted) by a definition of the same name in another component. (See Symbol Preemption.) If a function symbol has external visibility, the compiler knows that it must be called indirectly and can inline the indirect call stub. |
| `DEFAULT` | Other components can reference the symbol. Furthermore, the symbol definition may be overridden (preempted) by a definition of the same name in another component. |
| `PROTECTED` | Other components can reference the symbol, but it cannot be preempted by a definition of the same name in another component. |
| `HIDDEN` | Other components cannot directly reference the symbol. However, its address might be passed to other components indirectly; for example, as an argument to a call to a function in another component, or by having its address stored in a data item referenced by a function in another component. |
| `INTERNAL` | The symbol cannot be referenced outside the component where it is defined, either directly or indirectly. |

**Note**
Visibility applies to both references and definitions. A symbol reference's visibility attribute is an assertion that the corresponding definition will have that visibility.

**Symbol Preemption and Optimization**

Sometimes programmers need to use some of the functions or data items from a shareable object, but at the same time, they need to replace other items with definitions of their own. For example, an application may need to use the standard run-time library shareable object, `libc.so`, but to use its own definitions of the heap management routines `malloc` and `free`. In this case it is important that calls to `malloc` and `free` within `libc.so` use the user's definition of the routines and not the definitions in `libc.so`. The user's definition should then override, or *preempt*, the definition within the shareable object.

This functionality of redefining the items in shareable objects is called symbol preemption. When the run-time loader loads a component, all symbols within the component that have default visibility are subject to preemption by symbols of the same name in components that are already loaded. Note that since the main program image is always loaded first, none of the symbols it defines will be preempted (redefined).

The possibility of symbol preemption inhibits many valuable compiler optimizations because symbols with default visibility are not bound to a memory address until run-time. For example, calls to a routine with default visibility cannot be inlined because the routine might be preempted if the compilation unit is linked into a shareable object. A preemptable data symbol cannot be accessed using GP-relative addressing because the name may be bound to a symbol in a different component; and the GP-relative address is not known at compile time.

Symbol preemption is a rarely used feature and has negative consequences for compiler optimization. For this reason, by default the compiler treats all global symbol definitions as non-preemptable (protected visibility). Global references to symbols defined in another compilation unit are assumed by default to be preemptable (default visibility). In those rare cases where all global definitions as well as references need to be preemptable, specify the `-fpic` option to override this default.

#### Specifying Symbol Visibility Explicitly

The Intel Fortran Compiler has the visibility attribute options that provide command-line control of the visibility attributes as well as a source syntax to set the complete range of these attributes. The options ensure immediate access to the feature without depending on header file modifications. The visibility options cause all global symbols to get the visibility specified by the option. There are two variety of options to specify symbol visibility explicitly:

```
-fvisibility=keyword
-fvisibility-keyword=file
```

The first form specifies the default visibility for global symbols. The second form specifies the visibility for symbols that are in a file (this form overrides the first form).

The *`file`* is the pathname of a file containing the list of symbols whose visibility you want to set; the symbols are separated by whitespace (spaces, tabs, or newlines).

In both options, the `keyword` is: `extern`, `default`, `protected`, `hidden`, and `internal`, see definitions above.

**Note**
These two ways to explicitly set visibility are mutually exclusive: you may use the visibility attribute on the declaration, or specify the symbol name in a *`file`*, but not both.

The option `-fvisibility-keyword=`*`file`* specifies the same visibility attribute for a number of symbols using one of the five command line options corresponding to the `keyword`:

```
-fvisibility-extern=file
-fvisibility-default=file
-fvisibility-protected=file
-fvisibility-hidden=file
-fvisibility-internal=file
```

where *`file`* is the pathname of a file containing a list of the symbol names whose visibility you wish to set; the symbol names in the *`file`* are separated by either blanks, tabs, or newlines. For example, the command line option:

```
-fvisibility-protected=prot.txt
```

where file `prot.txt` contains symbols a, b, c, d, and e sets protected visibility for symbols a, b, c, d, and e. This has the same effect as declared attribute `visibility=protected` on the declaration for each of the symbols.

#### Specifying Visibility without Symbol File, `-fvisibility=keyword`

This option sets the visiblity for symbols not specified in a visibility list file and that do not have `visibilty` attribute in their declaration. If no symbol file option is specified, all symbols will get the specified attribute. Command line example:

```
ifort -fvisibility=protected a.f
```

You can set the default visibility for symbols using one of the following command line options:

```
-fvisibility=extern
-fvisibility=default
-fvisibility=protected
-fvisibility=hidden
-fvisibility=internal
```

The above options are listed in the order of precedence: explicitly setting the visibility to `extern`, by using either the attribute syntax or the command line option, overrides any setting to `default`, `protected`, `hidden`, or `internal`. Explicitly setting the visibility to `default` overrides any setting to `protected`, `hidden`, or `internal` and so on.

The visibility attribute `default` enables compiler to change the default symbol visibility and then set the default attribute on functions and variables that require the default setting. Since `internal` is a processor-specific attribute, it may not be desirable to have a general option for it.

In the combined command-line options

```
-fvisibility=protected -fvisibility-default=prot.txt
```

file `prot.txt` (see above) causes all global symbols except `a`, `b`, `c`, `d`, and `e` to have protected visibility. Those five symbols, however, will have default visibility and thus be preemptable.

**Visibility-related Options**

**`-fminshared`**

Directs to treat the compilation unit as a component of a main program and not to link it as a part of a shareable object.

Since symbols defined in the main program cannot be preempted, this enables the compiler to treat symbols declared with default visibility as though they have protected visibility. It means that
`-fminshared` implies `-fvisibility=protected`. The compiler need not generate position-independent code for the main program. It can use absolute addressing, which may reduce the size of the global offset table (GOT) and may reduce memory traffic.

**`-fpic`**

Specifies full symbol preemption. Global symbol definitions as well as global symbol references get default (that is, preemptable) visibility unless explicitly specified otherwise. Generates position-independent code. Required for building shared objects on Itanium-based systems.

## Optimizing Different Application Types

### Optimizing Different Application Types Overview

This section discusses the command-line options `-O0`, `-O1`, `-O2` (or `-O`), and -O3. The `-O0` option disables optimizations. Each of the other three turns on several compiler capabilities. To specify one of these optimizations, take into consideration the nature and structure of your application as indicated in the more detailed description of the options.

In general terms, `-O1`, `-O2` (or `-O`), and `-O3` optimize as follows:

`-O1` : code size and locality

`-O2` (or `-O)`: code speed; this is the default option

`-O3`: enables `-O2` with more aggressive optimizations.

`-fast`: enables `-O3` and `-ipo` to enhance speed across the entire program.

These options behave similarly on IA-32 and Itanium® architectures, with some specifics that are detailed in the sections that follow.

## Setting Optimizations with -On Options

The following table details the effects of the `-O0`, `-O1`, `-O2`, `-O3`, and `-fast` options. The table first describes the characteristics shared by both IA-32 and Itanium architectures and then explicitly describes the specifics (if any) of the `-On` and `-fast` options' behavior on each architecture.

| **Option** | **Effect** |
|---|---|
| `-O0` | Disables `-On` optimizations. On IA-32 systems, this option sets the `-fp` option. |
| `-O1` | Optimizes to favor code size and code locality.<br>Disables loop unrolling.<br>May improve performance for applications with very large code size, many branches, and execution time not dominated by code within loops.<br>In most cases, `-O2` is recommended over `-O1`.<br>On IA-32 systems:<br>Disables intrinsics inlining to reduce code size.<br>Enables optimizations for speed. Also disables intrinsic recognition and the `-fp` option.<br>On Itanium-based systems:<br>Disables software pipelining and global code scheduling. Enables optimizations for server applications (straight-line and branch-like code with flat profile). Enables optimizations for speed, while being aware of code size. For example, this option disables software pipelining and loop unrolling. |

| | |
|---|---|
| `-O2`, `-O` | This option is the default for optimizations. However, if `-g` is specified, the default is `-O0`.<br>Optimizes for code speed.<br>This is the generally recommended optimization level. However, if `-g` is specified, `-O2` is turned off and `-O0` is the default unless `-O2` (or `-O1` or `-O3`) is explicitly specified in the command line together with `-g`.<br><br>On IA-32 systems, this option is the same as the `-O1` option.<br><br>On Itanium-based systems:<br>Enables optimizations for speed, including global code scheduling, software pipelining, predication, and speculation.<br><br>On these systems, the `-O2` option enables inlining of intrinsics. It also enables the following capabilities for performance gain: constant propagation, copy propagation, dead-code elimination, global register allocation, global instruction scheduling and control speculation, loop unrolling, optimized code selection, partial redundancy elimination, strength reduction/induction variable simplification, variable renaming, exception handling optimizations, tail recursions, peephole optimizations, structure assignment lowering and optimizations, and dead store elimination. |

| | |
|---|---|
| `-O3` | Enables `-O2` optimizations and in addition, enables more aggressive optimizations such as prefetching, scalar replacement, and loop and memory access transformations. Enables optimizations for maximum speed, but does not guarantee higher performance unless loop and memory access transformation take place. The `-O3` optimizations may slow down code in some cases compared to `-O2` optimizations. Recommended for applications that have loops that heavily use floating-point calculations and process large data sets.<br>On IA-32 systems:<br>In conjunction with `-ax{K\|W\|N\|B\|P}` or `-x{K\|W\|N\|B\|P}` options, this option causes the compiler to perform more aggressive data dependency analysis than for `-O2`. This may result in longer compilation times.<br><br>On Itanium-based systems, enables optimizations for technical computing applications (loop-intensive code): loop optimizations and data prefetch. |
| `-fast` | This option is a single, simple method to enable a collection of optimizations for run-time performance. Sets the following options that can improve run-time performance:<br><br>`-O3`: maximum speed and high-level optimizations, see above<br><br>`-ipo`: enables interprocedural optimizations across files<br><br>`-static`: prevents linking with shared libraries<br><br>On IA-32 and Intel® EM64T systems, -fast sets these three options and also sets -xP.<br><br>Provides a shortcut that requests several important compiler optimizations. To override one of the options set by `-fast`, specify that option after the `-fast` option on the command line.<br><br>The options set by the `-fast` option may change from release to release.<br><br>On IA-32 systems: |

| | |
|---|---|
| | In conjunction with `-ax{K|W|N|B|P}` or `-x{K|W|N|B|P}` options, this option provides the best run-time performance. |

## Restricting Optimizations

The following options restrict or preclude the compiler's ability to optimize your program:

| | |
|---|---|
| `-O0` | Disables optimizations. Enables `-fp` option. |
| `-g` | Turns off the default `-O2` option and makes `-O0` the default unless `-O2` (or `-O1` or `-O3`) is explicitly specified in the command line together with `-g`. See Optimizations and Debugging. |
| `-mp` | Restricts optimizations that cause some minor loss or gain of precision in floating-point arithmetic to maintain a declared level of precision and to ensure that floating-point arithmetic more nearly conforms to the ANSI and IEEE* standards. See `-mp` option for more details. |
| `-nolib_inline` | Disables inline expansion of |

| | |
|---|---|
| | intrinsic functions. |

For more information on ways to restrict optimization, see Using `-ip` with -Qoption Specifiers.

## Floating-point Arithmetic Optimizations

### Options Used for Both IA-32 and Itanium® Architectures

The options described in this section all provide optimizations with varying degrees of precision in floating-point (FP) arithmetic for IA-32 and Itanium® architectures.

The `-mp1` (IA-32 only) and `-mp` options improve floating-point precision, but also affect the application performance. See more details about these options in Improving/Restricting FP Arithmetic Precision.

The FP options provide optimizations with varying degrees of precision in floating-point arithmetic. The option that disables these optimizations is `-O0`.

#### **`-mp` Option**

Use `-mp` to limit floating-point optimizations and maintain declared precision. For example, the Intel® Fortran Compiler can change floating-point division computations into multiplication by the reciprocal of the denominator. This change can alter the results of floating point division computations slightly. The `-mp` switch may slightly reduce execution speed. For more information, see Improving/Restricting FP Arithmetic Precision.

#### **`-mp1` Option**

Use the `-mp1` option to restrict floating-point precision to be closer to declared precision  with less impact to performance than with the `-mp` option. The option will ensure the out-of-range check of operands of transcendental functions and improve accuracy of floating-point compares.

#### **Flushing to Zero Denormal Values, `-ftz[-]`**

Option `-ftz[-]` flushes denormal results to zero when the application is in the gradual underflow mode. Flushing the denormal values to zero with `-ftz` may improve performance of your application.

The default status of is OFF (`-ftz-`). By default, the compiler lets results gradually underflow. With the default `-O2` option, `-ftz[-]` is OFF.

### `-ftz[-]` on Itanium-based systems

On Itanium-based systems only, the `-O3` option turns on `-ftz`.

If the `-ftz` option produces undesirable results of the numerical behavior of your program, you can turn the flush-to-zero (FTZ) mode off by using `-ftz-` in the command line while still benefiting from the `-O3` optimizations:

```
ifort -O3 -ftz- myprog.f
```

Usage:

- Use this option if the denormal values are not critical to application behavior.
- `-ftz[-]` only needs to be used on the source that contains the `main` program to turn the FTZ mode on. The initial thread, and any threads subsequently created by that process, will operate in FTZ mode.

The `-ftz[-]` option affects the results of floating underflow as follows:

- `-ftz-` results in *gradual* underflow to 0: the result of a floating underflow is a denormalized number or a zero.
- `-ftz` results in *abrupt* underflow to 0: the result of a floating underflow is set to zero and execution continues. `-ftz` also makes a denormal value used in a computation be treated as a zero so no floating invalid exception occurs. On Itanium-based systems, the `-O3` option sets the abrupt underflow to zero (`-ftz` is on). At lower optimization levels, gradual underflow to 0 is the default on the Itanium-based systems.

On IA-32, setting abrupt underflow by `-ftz` may improve performance of SSE/SSE2 instructions, while it does not affect either performance or numerical behavior of x87 instructions. Thus, `-ftz` will have no effect unless you select the `-x` or `-ax` options, which activate instructions of the more recent IA-32 Intel processors.

On Itanium-based processors, gradual underflow to 0 can degrade performance. Using higher optimization levels to get the default abrupt underflow or explicitly setting `-ftz` improves performance.
`-ftz` may improve performance on Itanium® 2 processor, even in the absence of actual underflow, most frequently for single-precision code.

### Using the Floating-point Exception Handling, `-fpe`*n*

Use the `-fpe`*n* option to control the handling of exceptions. The `-fpe`*n* option controls floating-point exceptions according to the value of *n*.

The following are the kinds of floating-point exceptions:

- Floating overflow: the result of a computation is too large for the floating-point data type. The result is replaced with the exceptional value Infinity with the proper "+" or "-" sign. For example, 1E30 * 1E30 overflows single-precision floating-point value and results in a +Infinity; -1E30 * 1E30 results in a -Infinity.
- Floating divide-by-zero: if the computation is 0.0 / 0.0, the result is the exceptional value NaN (Not a Number), a value that means the computation was not successful. If the numerator is not 0.0, the result is a signed Infinity.
- Floating underflow: the result of a computation is too small for the floating-poinit type. Each floating-point type (32-, 64-, and 128-bit) has a denormalized range where very small numbers can be represented with some loss of precision. For example, the lower bound for normalized single-precision floating-point value is approximately 1E-38; the lower bound for denormalized single-precision floating-point value is 1E-45. 1E-30 / 1E10 underflows the normalized range but not the denormalized range so the result is the denormal exceptional value 1E-40. 1E-30 / 1E30 underflows the entire range and the result is zero. This is known as gradual underflow to 0.
- Floating invalid: when the exceptional value (signed Infinities, NaN, denormal) is used as input to a computation, the result is also a NaN.

The `-fpe`*n* option allows some control over the results of floating-point exception handling at run time for the main program.

- `-fpe0` restricts floating-point exceptions as follows:
  - Floating overflow, floating divide-by-zero, and floating invalid cause the program to print an error message and abort.
  - If a floating underflow occurs, the result is set to zero and execution continues. This is called abrupt underflow to 0.
- `-fpe1` restricts only floating underflow:
  - Floating overflow, floating divide-by-zero, and floating invalid produce exceptional values (NaN and signed Infinities) and execution continues.
  - If a floating underflow occurs, the result is set to zero and execution continues.
- The default is `-fpe3` on both IA-32 and Itanium-based processors. This allows full floating-point exception behavior:
  - Floating overflow, floating divide-by-zero, and floating invalid produce exceptional values (NaN and signed Infinities) and execution continues.
  - Floating underflow is gradual: denormalized values are produced until the result becomes 0.

The `-fpe`*n* only affects the Fortran main program. The floating-point exception behavior set by the Fortran main program is in effect throughout the execution of the entire program. If the main program is not Fortran, you can use the Fortran intrinsic `FOR_SET_FPE` to set the floating-point exception behavior.

When compiling different routines in a program separately, you should use the same value of `n` in `-fpe`*n*.

For more information, refer to the Intel Fortran Compiler User's Guide for Linux* Systems, Volume I, section "Controlling Floating-point Exceptions."

## Floating-point Arithmetic Precision for IA-32 Systems

### `-prec_div` Option

The Intel® Fortran Compiler can change floating-point division computations into multiplication by the reciprocal of the denominator. Use `-prec_div` to disable floating point division-to-multiplication optimization resulting in more accurate division results. May have speed impact.

### `-pc{32|64|80}` Option

Use the `-pc{32|64|80}` option to enable floating-point significand precision control. Some floating-point algorithms, created for specific IA-32 and Itanium®-based systems, are sensitive to the accuracy of the significand or fractional part of the floating-point value. Use appropriate version of the option to round the significand to the number of bits as follows:

`-pc32`: 24 bits (single precision)

`-pc64`: 53 bits (double precision)

`-pc80`: 64 bits (extended precision)

The default version is `-pc80` for full floating-point precision.

This option enables full optimization. Using this option does not have the negative performance impact of using the `-mp` option because only the fractional part of the floating-point value is affected. The range of the exponent is not affected.

**Note**

This option only has an effect when the module being compiled contains the main program.

**Caution**

> A change of the default precision control or rounding mode (for example, by using the `-pc32` option or by user intervention) may affect the results returned by some of the mathematical functions.

### Rounding Control, `-rcd`, `-fp_port`

The Intel Fortran Compiler uses the `-rcd` option to disable changing of rounding mode for floating-point-to-integer conversions.

The system default floating-point rounding mode is round-to-nearest. This means that values are rounded during floating-point calculations. However, the Fortran language requires floating-point values to be truncated when a conversion to an integer is involved. To do this, the compiler must change the rounding mode to truncation before each floating-point conversion and change it back afterwards.

The `-rcd` option disables the change to truncation of the rounding mode for all floating-point calculations, including floating-point-to-integer conversions. Turning on this option can improve performance, but floating-point conversions to integer will not conform to Fortran semantics.

You can also use the `-fp_port` option to round floating-point results at assignments and casts. May cause some speed impact, but also makes sure that rounding to the user-declared precision at assignments is always done. The `-mp1` option implies `-fp_port`.

## Floating-point Arithmetic Precision for Itanium®-based Systems

The following Intel® Fortran Compiler options enable you to control the compiler optimizations for floating-point computations on Itanium®-based systems.

### Contraction of FP Multiply and Add/Subtract Operations

`-IPF_fma[-]` enables or disables the contraction of floating-point multiply and add/subtract operations into a single operations. Unless `-mp` is specified, the compiler tries to contract these operations whenever possible. The `-mp` option disables the contractions.

`-IPF_fma` and `-IPF_fma-` can be used to override the default compiler behavior. For example, a combination of `-mp` and `-IPF_fma` enables the compiler to contract operations:

```
ifort -mp -IPF_fma myprog.f
```

### FP Speculation

`-IPF_fp_speculation`*`mode`* sets the compiler to speculate on floating-point operations in one of the following *`mode`*s:

*`fast`*: sets the compiler to speculate on floating-point operations; this is the default.

*`safe`*: enables the compiler to speculate on floating-point operations only when it is safe.

*`strict`*: enables the compiler's speculation on floating-point operations preserving floating-point status in all situations. In the current version, this mode disables the speculation of floating-point operations (same as `off`).

*`off`*: disables the speculation on floating-point operations.

**FP Math Function Optimization**

-IPF_fp_relaxed[-] enables or disables use of faster, but slightly less accurate code sequences for math functions such as divide and sqrt. Compared to strict IEEE* precision, this option slightly reduces the accuracy of floating-point calculations performed by these functions (usually limited to the least significant digit). The default is `-QIPF_fp_relaxed-.`

**FP Operations Evaluation**

`-IPF_flt_eval_method{0|2}` directs the compiler to evaluate the expressions involving floating-point operands in the following way:

`-IPF_flt_eval_method0` directs the compiler to evaluate the expressions involving floating-point operands in the precision indicated by the variable types declared in the program.

`-IPF_flt_eval_method2` is not supported in the current version.

**Controlling Accuracy of the FP Results**

`-IPF_fltacc` disables the optimizations that affect floating-point accuracy. The default is
`-IPF_fltacc-` to enable such optimizations.

The Itanium® compiler may reassociate floating-point expressions to improve application performance. Use `-IPF_fltacc` or `-mp` to disable or restrict these floating-point optimizations.

## Improving/Restricting FP Arithmetic Precision

The `-mp` and `-mp1` options maintain and restrict, respectively, floating-point precision, but also affect the application performance. The `-mp1` option causes less impact on performance than the `-mp` option. `-mp1` ensures the out-of-range check of operands of transcendental functions and improve accuracy of floating-point compares. For IA-32 systems, the `-mp` option implies `-mp1`; `-mp1` implies `-fp_port`. `-mp` slows down performance the most of these three, `-fp_port` the least of these three.

The `-mp` option restricts some optimizations to maintain declared precision and to ensure that floating-point arithmetic conforms more closely to the ANSI and IEEE* standards. This option causes more frequent stores to memory, or disallow some data from being register candidates altogether. The Intel architecture normally maintains floating point results in registers. These registers are 80 bits long, and maintain greater precision than a double-precision number. When the results have to be stored to memory, rounding occurs. This can affect accuracy toward getting more of the "expected" result, but at a cost in speed. The `-pc{32|64|80}` option (IA-32 only) can be used to control floating point accuracy and rounding, along with setting various processor IEEE flags.

For most programs, specifying the `-mp` option adversely affects performance. If you are not sure whether your application needs this option, try compiling and running your program both with and without it to evaluate the effects on performance versus precision.

Specifying this option has the following effects on program compilation:

- On IA-32 systems, floating-point user variables declared as floating-point types are not assigned to registers.
- On Itanium®-based systems, floating-point user variables may be assigned to registers. The expressions are evaluated using precision of source operands. The compiler will not use Floating-point Multiply and Add (FMA) function to contract multiply and add/subtract operations in a single operation. The contractions can be enabled by using `-IPF_fma` option. The compiler will not speculate on floating-point operations that may affect the floating-point state of the machine. See Floating-point Arithmetic Precision for Itanium-based Systems.
- Floating-point arithmetic comparisons conform to IEEE 754.
- The exact operations specified in the code are performed. For example, division is never changed to multiplication by the reciprocal.
- The compiler performs floating-point operations in the order specified without reassociation.
- The compiler does not perform the constant folding on floating-point values. Constant folding also eliminates any multiplication by 1, division by 1, and addition or subtraction of 0. For example, code that adds 0.0 to a number is executed exactly as written. Compile-time floating-point

arithmetic is not performed to ensure that floating-point exceptions are also maintained.

On IA-32 systems, whenever an expression is spilled, it is spilled as 80 bits (extended precision), not 64 bits (DOUBLE PRECISION). Floating-point operations conform to IEEE 754. When assignments to type REAL and DOUBLE PRECISION are made, the precision is rounded from 80 bits (extended) down to 32 bits (REAL) or 64 bits (DOUBLE PRECISION). When you do not specify `-O0`, the extra bits of precision are not always rounded away before the variable is reused.

- Even if vectorization is enabled by the `-x{K|W|N|B|P}` options, the compiler does not vectorize reduction loops (loops computing the dot product) and loops with mixed precision types. Similarly, the compiler does not enable certain loop transformations. For example, the compiler does not transform reduction loops to perform partial summation or loop interchange.

## Optimizing for Specific Processors

### Optimizing for Specific Processors Overview

This section describes targeting a processor and processor dispatch and extensions support options.

The options `-tpp{5|6|7}` optimize for the IA-32 processors, and the options `-tpp{1|2}` optimize for the Itanium® processor family. The options `-x{K|W|N|B|P}` and `-ax{K|W|N|B|P}` generate code that is specific to processor-instruction extensions.

Note that you can run your application on the latest processor-based systems, like Intel® Pentium® M processor or Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3) instruction support and still gear your code to any of the previous processors specified by `N, W,` or `K` versions of the `-x` and `-ax` options.

### Targeting a Processor, -tpp{n}

The `-tpp{n}` optimizes your application's performance for specific Intel processors. This option generates code that is tuned for the processor associated with its version. For example, `-tpp7` generates code optimized for running on Intel® Pentium® 4, Intel® Xeon(TM), Intel® Pentium® M processors and Intel® Pentium® 4 processors with Streaming SIMD Extensions 3 (SSE3) instruction support, and `-tpp2` generates code optimized for running on Itanium® 2 processor.

The `-tpp{n}` option always generates code that is backwards compatible with Intel® processors of the same family. This means that code generated with `-tpp7` will run correctly on Pentium Pro or Pentium III processors, possibly just not quite as fast as if the code had been compiled with `-tpp6`. Similarly, code generated with `-tpp2` will run correctly on Itanium processor, but possibly not quite as fast as if it had been generated with `-tpp1`.

**Processors for IA-32 Systems**

The `-tpp5`, `-tpp6`, and `-tpp7` options optimize your application's performance for a specific Intel IA-32 processor as listed in the table below. The resulting binaries will also run correctly on any of the processors mentioned in the table.

| Option | Optimizes your application for... |
|---|---|
| `-tpp5` | Intel® Pentium® and Pentium® with MMX(TM) technology processor |
| `-tpp6` | Intel® Pentium® Pro, Pentium® II and Pentium® III processors |
| `-tpp7` (default) | Intel Pentium 4 processors, Intel® Xeon(TM) processors, Intel® Pentium® M processors, and Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3) instruction support |

**Example**

The invocations listed below each result in a compiled binary of the source program `prog.f` optimized for Pentium 4 and Intel Xeon processors by default. The same binary will also run on Pentium, Pentium Pro, Pentium II, and Pentium III processors.

```
ifort prog.f
```

```
ifort -tpp7 prog.f
```

However if you intend to target your application specifically to the Intel Pentium and Pentium with MMX technology processors, use the `-tpp5` option:

```
ifort -tpp5 prog.f
```

**Processors for Itanium®-based Systems**

The `-tpp1` and `-tpp2` options optimize your application's performance for a specific Intel Itanium® processor as listed in the table below. The resulting binaries will also run correctly on both processors mentioned in the table.

| Option | Optimizes your application for... |
|---|---|

| | |
|---|---|
| `-tpp1` | Intel® Itanium® processor |
| `-tpp2` (default) | Intel® Itanium® 2 processor |

**Example**

The following invocation results in a compiled binary of the source program `prog.f` optimized for the Itanium 2 processor by default. The same binary will also run on Itanium processors.

```
ifort prog.f
```

```
ifort -tpp2 prog.f
```

However if you intend to target your application specifically to the Intel Itanium processor, use the `-tpp1` option:

```
ifort -tpp1 prog.f
```

## Processor-specific Optimization (IA-32 only)

The `-x{K|W|N|B|P}` options target your program to run on a specific Intel processor. The resulting code might contain unconditional use of features that are not supported on other processors.

| **Option** | **Optimizes for...** |
|---|---|
| `-xK` | Intel Pentium® III and compatible Intel processors. |
| `-xW` | Intel Pentium 4 and compatible Intel processors. |
| `-xN` | Intel Pentium 4 and compatible Intel processors. When the main program is compiled with this option, it will detect non-compatible processors and generate an error message during execution. This option also enables new optimizations in addition to Intel processor specific-optimizations. |
| `-xB` | Intel® Pentium® M and compatible Intel processors. When the main program is compiled with this option, it will detect non-compatible processors and generate an error message during execution. This option also enables new optimizations in addition to Intel processor-specific optimizations. |
| `-xP` | Intel® Pentium® 4 processors with Streaming SIMD Extensions 3 (SSE3) instruction support. When the main program is compiled with this option, it will detect non-compatible processors and generate an error message during execution. This option also enables new |

| | |
|---|---|
| | optimizations in addition to Intel processor-specific optimizations. |

To execute a program on x86 processors not provided by Intel Corporation, do not specify the
`-x{K|W|N|B|P}` option.

**Example**

The invocation below compiles `myprog.f` for Intel Pentium 4 and compatible processors. The resulting binary might not execute correctly on Pentium, Pentium Pro, Pentium II, Pentium III, or Pentium with MMX technology processors, or on x86 processors not provided by Intel corporation.

```
ifort -xN myprog.f
```

Caution

If a program compiled with `-x{K|W|N|B|P}` is executed on a non-compatible processor, it might fail with an illegal instruction exception, or display other unexpected behavior. Executing programs compiled with `-xN` , `-xB`, or `-xP` on unsupported processors (see table above) will display the following run-time error:

```
Fatal error: This program was not built to run on the
processor in your system.
```

## Automatic Processor-specific Optimization (IA-32 only)

The `-ax{K|W|N|B|P}` options direct the compiler to find opportunities to generate separate versions of functions that take advantage of features that are specific to the specified Intel processor. If the compiler finds such an opportunity, it first checks whether generating a processor-specific version of a function is likely to result in a performance gain. If this is the case, the compiler generates both a processor-specific version of a function and a generic version of the function. The generic version will run on any IA-32 processor.

At run time, one of the versions is chosen to execute, depending on the Intel processor in use. In this way, the program can benefit from performance gains on more advanced Intel processors, while still working properly on older IA-32 processors.

The disadvantages of using `-ax{K|W|N|B|P}` are:

- The size of the compiled binary increases because it contains processor-specific versions of some of the code, as well as a generic version of the code.

- Performance is affected slightly by the run-time checks to determine which code to use.

**Note**
Applications that you compile to optimize themselves for specific processors in this way will execute on any Intel IA-32 processor. If you specify both the `-x` and `-ax` options, the `-x` option forces the generic code to execute only on processors compatible with the processor type specified by the `-x` option.

| **Option** | **Optimizes Your Code for...** |
|---|---|
| `-axK` | Intel Pentium® III and compatible Intel processors. |
| `-axW` | Intel Pentium 4 and compatible Intel processors. |
| `-axN` | Intel Pentium 4 and compatible Intel processors. This option also enables new optimizations in addition to Intel processor-specific optimizations. |
| `-axB` | Intel Pentium M and compatible Intel processors. This option also enables new optimizations in addition to Intel processor-specific optimizations. |
| `-axP` | Intel® Pentium® 4 processors with Streaming SIMD Extensions 3 (SSE3) instruction support. This option also enables new optimizations in addition to Intel processor-specific optimizations. |

**Example**

The compilation below generates a single executable that includes:

- A generic version for use on any IA-32 processor
- A version optimized for Intel Pentium 4 processors, as long as there is a performance benefit.
- A version optimized for Intel Pentium M processors, as long as there is a performance benefit.

```
ifort -axNB prog.f90
```

## Processor-specific Run-time Checks, IA-32 Systems

The Intel Fortran Compiler optimizations take effect at run-time. For IA-32 systems, the compiler enhances processor-specific optimizations by inserting in the main routine a code segment that performs run-time checks described below.

### Check for Supported Processor with `-xB` , `-xB`, or `-xP`

To prevent from execution errors, the compiler inserts code in the main routine of the program to check for proper processor usage. Programs compiled with options `-xN`, `-xB,` or `-xP` check at run-time whether they are being executed on the Intel Pentium® 4, Intel® Pentium® M processor or the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3) instruction support, respectively, or a compatible Intel processor. If the program is not executed on one of these processors, the program terminates with an error.

**Example**

To optimize a program `foo.f90` for an Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3) instruction support, issue the following command:

```
ifort -xP foo.f90 -o foo.exe
```

`foo.exe` aborts if it is executed on a processor that is not validated to support the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3) instruction support to account for the fact that this processor may have some additional feature enabling.

If you intend to run your programs on multiple IA-32 processors, do not use the -x options that optimize for processor-specific features; consider using -ax to attain processor-specific performance and portability among different processors.

**Setting FTZ and DAZ Flags**

Previously, the default status of the flags flush-to-zero (FTZ) and denormals-are-zero (DAZ) for IA-32 processors were off by default. However, even at the cost of losing IEEE compliance, turning these flags on significantly increases the performance of programs with denormal floating-point values in the gradual underflow mode run on the most recent IA-32 processors. Hence, for the Intel Pentium III, Pentium 4, Pentium M, Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3) instruction support, and compatible IA-32 processors, the compiler's default behavior is to turn these flags on. The compiler inserts code in the program to perform a run-time check for the processor on which the program runs to verify it is one of the afore-listed Intel processors.

- Executing a program on a Pentium III processor enables the FTZ flag, but not DAZ.
- Executing a program on an Intel Pentium M processor or Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3) instruction support enables both the FTZ and DAZ flags.

These flags are only turned on by Intel processors that have been validated to support them.

For non-Intel processors, the flags can be set manually by calling the following Intel Fortran intrinsic:

```
RESULT = FOR_SET_FPE (FOR_M_ABRUPT_UND)
```

## Interprocedural Optimizations (IPO)

### Overview of Interprocedural Optimizations

Use `-ip` and `-ipo` to enable interprocedural optimizations (IPO), which enable the compiler to analyze your code to determine where you can benefit from the optimizations listed in tables that follow.

#### IA-32 and Itanium®-based applications

| Optimization | Affected Aspect of Program |
|---|---|
| Inline function expansion | Calls, jumps, branches, and loops |
| Interprocedural constant propagation | Arguments, global variables, and return values |
| Monitoring module-level static variables | Further optimizations and loop invariant code |
| Dead code elimination | Code size |
| Propagation of function characteristics | Call deletion and call movement |
| Multifile optimization | The same aspects as `-ip`, but across multiple files |

#### IA-32 applications only

| Optimization | Affected Aspect of Program |
|---|---|
| Passing arguments in registers | Calls and register usage |
| Loop-invariant code motion | Further optimizations and loop invariant code |

Inline function expansion is one of the main optimizations performed by the interprocedural optimizer. For function calls that the compiler believes are frequently executed, the compiler might decide to replace the instructions of the call with code for the function itself.

With `-ip`, the compiler performs inline function expansion for calls to procedures defined within the current source file. However, when you use `-ipo` to specify multifile IPO, the compiler performs inline function expansion for calls to procedures defined in separate files.

To disable the IPO optimizations, use the `-O0` option.

**Caution**

The `-ip` and `-ipo` options can in some cases significantly increase compile time and code size.

### Option `-auto_ilp32` for Itanium®-based Systems

On Itanium-based systems, the `-auto_ilp32` option requires interprocedural analysis over the whole program. This optimization allows the compiler to use 32-bit pointers whenever possible as long as the application does not exceed a 32-bit address space. Using the `-auto_ilp32` option on programs that exceed 32-bit address space might cause unpredictable results during program execution.

Because this optimization requires interprocedural analysis over the whole program, you must use the
`-auto_ilp32` option with the `-ipo` option.

On Intel® EM64T systems, `-auto_ilp32` has no effect unless `-xP` or `-axP` is also specified.

## IPO Compilation Model

For the topics in this section, the term IPO generally refers to multi-file IPO.

When you use the `-ipo` option, the compiler collects information from individual program modules of a program. Using this information, the compiler performs optimizations across modules. In order to do this, the `-ipo` option is applied to both the compilation phase and the link phase.

One of the main benefits of IPO is that it enables more inlining. For information on inlining and the minimum inlining criteria, see Criteria for Inline Function Expansion and Controlling Inline Expansion of User Functions. Inlining and other optimizations are improved by profile information. For a description of how to use IPO with profile information for further optimization, see Example of Profile-Guided Optimization.

### Compilation Phase

When using IPO, as each source file is compiled, the compiler stores an intermediate representation (IR) of the source code in the object file, which includes summary information used for optimization.

By default, the compiler produces "mock" object files during the compilation phase of IPO. Generating mock files instead of real object files reduces the time spent in the IPO compilation phase. Each mock object file contains the IR for its corresponding source file, but no real code or data. These mock objects must be linked using the `-ipo` option in `ifort` or using the `xild` tool. (See Creating a Multifile IPO Executable with xild.)

**Note**
Failure to link "mock" objects with `ifort` and `-ipo` or `xild` will result in linkage errors. There are situations where mock object files cannot be used. See Compilation with Real Object Files for more information.

**Linkage Phase**

When you invoke the linker, adding `-ipo` to the command line causes the compiler to be invoked a final time before the linker. The compiler performs IPO across all object files that have an IR. The compiler first analyzes all of the summary information, and then finishes compiling the pieces of the application for which it has IR. Having global information about the application while it is compiling individual pieces can improve the quality of optimization.

**Note**
The compiler does not support multifile IPO for static libraries (`.a` files). See Compilation with Real Object Files for more information.

`-ipo` enables the driver and compiler to attempt detecting a whole program automatically. If a whole program is detected, the interprocedural constant propagation, stack frame alignment, data layout and padding of common blocks perform more efficiently, while more dead functions get deleted. This option is safe.

## Command Line for Creating an IPO Executable

The command line options to enable IPO for compilations targeted for both the IA-32 and Itanium® architectures are identical.

To produce mock object files containing IR, compile your source files with `-ipo` as follows:

```
ifort -ipo -c a.f b.f c.f
```

This produces `a.o`, `b.o`, and `c.o` object files. These files contain Intel® compiler intermediate representation (IR) corresponding to the compiled source files `a.f`, `b.f`, and `c.f`. Using `-c` to stop compilation after generating `.o` files is required.

You can now optimize interprocedurally by adding `-ipo` to your link command line. The following example produces an executable named `app`:

```
ifort -oapp -ipo a.o b.o c.o
```

This command invokes the compiler on the objects containing IR and creates a new list of object(s) to be linked. The command then calls GCC `ld` to link the specified object files and produce `app`, as specified by the `-o` option. IPO is applied only to the object files that contain IR; otherwise the object file passes to link stage.

**Note**
For the above step, you can use the `xild` tool instead of `ifort`.

The two steps described above can be combined, as shown in the following:

```
ifort -ipo -oapp a.f b.f c.f
```

## Generating Multiple IPO Object Files

For the most part, IPO generates a single object file for the link-time compilation. This can be clumsy for very large applications, perhaps even making it impossible to use `-ipo` on the application. The compiler provides two ways to avoid this problem. The first way is a size-based heuristic, which automatically causes the compiler to generate multiple object files for large link-time compilations. The second way is using one of two explicit command line controls for that tell the compiler to do multi-object IPO:

- `-ipo`*N*, where *N* indicates the number of object files to generate
- `-ipo_separate`, which tells the compiler to generate a separate IPO object file for each source file.

These options are alternatives to the `-ipo` option, that is, they indicate an IPO compilation. Explicitly requesting a multi-object IPO compilation turns the size-based heuristic off.

The number of files generated by the link-time compilation is invisible to the user unless either the `-ipo_c` or `-ipo_S` option is used. In this case the compiler appends a number to the file name. For example, consider this command line:

```
ifort a.o b.o c.o -ipo_separate -ipo_c
```

In this command line, `a.o`, `b.o`, and `c.o` all contain IR, so the compiler will generate `ipo_out.o`, `ipo_out1.o`, `ipo_out2.o`, and `ipo_out3.o`.

The first object file contains global symbols. The other object files correspond to the source files.

This naming convention is also applied to user-specified names. For example:

```
ifort a.o b.o c.o -ipo_separate -ipo_c -o appl.o
```

This will generate `appl.o`, `appl1.o`, `appl2.o`, and `appl3.o`.

## Capturing Intermediate Outputs of IPO

The `-ipo_c` and `-ipo_S` options are useful either for analyzing the effects of IPO, or when using IPO on modules that do not make up a complete program.

Use the `-ipo_c` option to optimize across files and produce an object file. This option performs optimizations as described for `-ipo`, but stops prior to the final link stage, leaving an optimized object file. The default name for this file is `ipo_out.o`. You can use the `-o` option to specify a different name. For example:

```
ifort -tpp6 -ipo_c -ofilename a.f b.f c.f
```

Use the `-ipo_S` option to optimize across files and produce an assembly file. This option performs optimizations as described for `-ipo`, but stops prior to the final link stage, leaving an optimized assembly file. The default name for this file is `ipo_out.s`. You can use the `-o` option to specify a different name. For example:

```
ifort -tpp6 -ipo_S -ofilename a.f b.f c.f
```

The `-ipo_c` and `-ipo_S` options generate multiple outputs if multi-object IPO is being used. The name of the first file is taken from the value of the `-o` option. The name of subsequent files is derived from this file by appending a numeric value to the file name. For example, if the first object file is named `foo.o`, the second object file will be named `foo1.o`.

The compiler generates a message indicating the name of each object or assembly file it is generating. These files can be added to the real link step to build the final application.

## Creating an IPO Executable Using xild

Use the Intel® linker, `xild` instead of step 2 in Command Line for Creating an IPO Executable. The linker `xild` performs the following steps:

1. Invokes the compiler to perform IPO if objects containing `IR` are found.

2. Invokes GCC `ld` to link the application.

The command-line syntax for `xild` is the same as that of the GCC linker:

**`xild [<options>] <`*`LINK_commandline`*`>`**

where:

- `[<options>]` (optional) may include any GCC linker options or options supported only by `xild`.
- `<`*`LINK_commandline`*`>` is your linker command line containing a set of valid arguments to the `ld`.

To create `app` using IPO, use the option `-o`*`filename`* as shown in the following example:

**`xild -oapp a.o b.o c.o`**

`xild` calls the compiler to perform IPO for objects containing `IR` and creates a new list of object(s) to be linked. Then `xild` calls `ld` to link the object files that are specified in the new list and produce `app`.

**Note**

The `-ipo` option can reorder object files and linker arguments on the command line. Therefore, if your program relies on a precise order of arguments on the command line, `-ipo` can affect the behavior of your program.

The `xild` command recognizes all three spellings the IPO switch (`-ipo`, `-ipo`*`N`*, and `-ipo_separate`).

**Usage Rules**

You must use the Intel linker `xild` to link your application if:

- Your source files were compiled with IPO enabled. IPO is enabled by specifying the `-ipo` command-line option
- You normally would invoke the GCC linker (`ld`) to link your application.

**The `xild` Options**

The additional options supported by `xild` may be used to examine the results of IPO. These options are described in the following table.

| | |
|---|---|
| `-qipo_fa[`*`file.s`*`]` | Produces an assembly listing for the |

| | |
|---|---|
| | IPO compilation. You can specify an optional name for the listing file, or a directory (with the backslash) in which to place the file. The default listing name is `ipo_out.s`. |
| `-qipo_fo[`*`file.o`*`]` | Produces an object file for the IPO compilation. You can specify an optional name for the object file, or a directory (with the backslash) in which to place the file. The default object file name is `ipo_out.o`. |
| `-ipo_fcode-asm` | Adds code bytes to the assembly listing. |
| `-ipo_fsource-asm` | Adds high-level source code to the assembly listing. |
| `-ipo_fverbose-asm`, `-ipo_fnoverbose-asm` | Enables and disables, respectively, inserting comments containing version and options used in the assembly listing for `xild`. |

If the `xild` invocation leads to an IPO multi-object compilation (either because the application is big, or because the user explicity asked for multiple objects), the first `.s` file takes its name from the `-qipo_fa` option. The compiler derives the names of subsequent `.s` files by appending a number to the name, for example, `foo.s` and `foo1.s` for `-qipo_fafoo.s`. The same is true for the `-qipo_fo` option.

## Code Layout and Multi-Object IPO

One of the optimizations performed during an IPO compilation is code layout. IPO analysis determines a layout order for all of the routines for which it has IR. If a single object is being generated, the compiler generates the layout simply by compiling the routines in the desired order.

For a multi-object IPO compilation, the compiler must tell the linker about the desired order. The compiler first puts each routine in a named text section (the first routine in `.text00001`, the second in `.text00002`, and so forth). It then generates a linker script that tells the linker to first link contributions from `.text00001`, then `.text00002`. This happens transparently when the same `ifort` (or `xild`) invocation is used for both the link-time compilation and the final link.

However, the linker script must be taken into account by the user if `-ipo_c` or `-ipo_S` is used. With these switches, the IPO compilation and actual link are done by different invocations of `ifort`. When this occurs, `ifort` will issue an informational message indicating that it is generating an explicit linker script, `ipo_layout.script`.

When `ipo_layout.script` is generated, the typical response is to modify your link command to use this script:

```
--script=ipo_layout.script
```

If your application already requires a custom linker script, you can place the necessary contents of `ipo_layout.script` in your script. The layout-specific content of `ipo_layout.script` is at the beginning of the description of the `.text` section. For example, to describe the layout order for 12 routines:

```
.text     :
{
*(.text00001) *(.text00002) *(.text00003) *(.text00004)
*(.text00005)
*(.text00006) *(.text00007) *(.text00008) *(.text00009)
*(.text00010)
*(.text00011) *(.text00012)
...
```

For applications that already require a linker script, you can add this section of the `.text` section description to the customized linker script. If you add these lines to your linker script, it is desirable to add additional entries to account for future development. This is harmless, since the "*(…)" syntax makes these contributions optional.

If you choose to not use the linker script your application will still build, but the layout order will be random. This may have an adverse affect on application performance, particularly for large applications.

## Compilation with Real Object Files

In certain situations you might need to generate real object files with `-ipo`. To force the compiler to produce real object files instead of "mock" ones with IPO, you must specify `-ipo_obj` in addition to
`-ipo`.

Use of `-ipo_obj` is necessary under the following conditions:

- The objects produced by the compilation phase of `-ipo` will be placed in a static library without the use of `xiar`. The compiler does not support multifile IPO for static libraries, so all static libraries are passed to the linker. Linking with a static library that contains "mock" object files will result in linkage errors because the objects do not contain real code or data. Specifying

`-ipo_obj` causes the compiler to generate object files that can be used in static libraries.
- Alternatively, if you create the static library using `xiar`, then the resulting static library will work as a normal library.
- The objects produced by the compilation phase of `-ipo` might be linked without the `-ipo` option and without the use of `xiar`.
- You want to generate an assembly listing for each source file (using `-S`) while compiling with `-ipo`. If you use `-ipo` with `-S`, but without `-ipo_obj`, the compiler issues a warning and an empty assembly file is produced for each compiled source file.

### Implementing the `.il` Files with Version Numbers

An IPO compilation consists of two parts: the compile phase and the link phase. In the compile phase, the compiler produces an intermediate language (IL) version of the users' code. In the link phase, the compiler reads the IL and completes the compilation, producing a real object file or executable.

Generally, different compiler versions produce IL based on different definitions, and therefore the ILs from different compilations can be incompatible. Intel Fortran Compiler assigns a unique version number with each compiler's IL definition. If a compiler attempts to read IL in a file with a version number other than its own, the compilation proceeds, but the IL is discarded and not used in the compilation. The compiler then issues a warning message about an incompatible IL detected and discarded.

The IL produced by the Intel compiler is stored in file with a suffix `.il`. Then the `.il` file is placed in the library. If this library is used in an IPO compilation invoked with the same compiler as produced the IL for the library, the compiler can extract the `.il` file from the library and use it to optimize the program. For example, it is possible to inline functions defined in the libraries into the users' source code.

## Creating a Library from IPO Objects

Normally, libraries are created using a library manager such as `ar`. Given a list of objects, the library manager will insert the objects into a named library to be used in subsequent link steps.

```
xiar cru user.a a.o b.o
```

The above command creates a library named `user.a` that contains the `a.o` and `b.o` objects.

If, however, the objects have been created using `-ipo -c`, then the objects will not contain a valid object but only the intermediate representation (`IR`) for that object file. For example:

```
ifort -ipo -c a.f b.f
```

will produce `a.o` and `b.o` that only contains `IR` to be used in a link time compilation. The library manager will not allow these to be inserted in a library.

In this case, you must use the Intel library driver `xild -ar`. This program will invoke the compiler on the `IR` saved in the object file and generate a valid object that can be inserted in a library.

```
xild -lib cru user.a a.o b.o
```

See Creating a Multifile IPO Executable Using `xild`.

## Using -ip with -Qoption Specifiers

You can adjust the Intel® Fortran Compiler's optimization for a particular application by experimenting with memory and interprocedural optimizations.

Enter the `-Qoption` option with the applicable keywords to select particular inline expansions and loop optimizations. The option must be entered with an `-ip` or `-ipo` specification, as follows:

```
-ip[-Qoption,tool,opts]
```

where `tool` is Fortran (`f`) and `opts` are `-Qoption` specifiers (see below). Also refer to Criteria for Inline Function Expansion to see how these specifiers may affect the inlining heuristics of the compiler.

For more information about passing options to other tools, see (`/Qoption,`*`tool`*`,`*`opts`*).

### -Qoption Specifiers

If you specify `-ip` or `-ipo` without any `-Qoption` qualification, the compiler does the following:

- Expands functions in line
- Propagates constant arguments
- Passes arguments in registers
- Monitors module-level static variables.

You can refine interprocedural optimizations by using the following `-Qoption` specifiers. To have an effect, the `-Qoption` option must be entered with either `-ip` or `-ipo` also specified, as in this example:

```
-ip -Qoption,f,ip_specifier
```

where `ip_specifier` is one of the `-Qoption` specifiers described in the following table:

| **-Qoption Specifiers** | |
|---|---|
| `-ip_args_in_regs=0` | Disables the passing of arguments in registers. By default, external functions can pass arguments in registers when called locally. Normally, only static functions can pass arguments in registers, provided the address of the function is not taken and the function does not use a variable number of arguments. |
| `-ip_ninl_max_stats=n` | Sets the valid number of intermediate language statements for a function that is expanded in line. The number n is a positive integer. The number of intermediate language statements usually exceeds the actual number of source language statements. The default value for n is 230. |
| `-ip_ninl_min_stats=n` | Sets the valid `min` number of intermediate language statements for a function that is expanded in line. The number n is a positive integer. The default value for `ip_ninl_min_stats` is:<br>IA-32 compiler:<br>`ip_ninl_min_stats` = 7<br>Itanium® compiler:<br>`ip_ninl_min_stats` = 15 |
| `-ip_ninl_max_total_stats=n` | Sets the maximum increase in size of a function, measured in intermediate language statements, due to inlining. The |

| | |
|---|---|
| | number n is a positive integer. The default value for n is 2000. |

The following command activates procedural and interprocedural optimizations on source.f and sets the maximum increase in the number of intermediate language statements to five for each function:

```
ifort -ip -Qoption,f,-ip_ninl_max_stats=5 source.f
```

## Inline Expansion of Functions

### Criteria for Inline Function Expansion

For a call to be considered for inlining, it has to meet certain minimum criteria. There are three main components of a call:

*Call-site* is the site of the call to the function that might be inlined.

*Caller* is the function that contains the call-site.

*Callee* is the function being called that might be inlined.

**Minimum call-site criteria:**

- The number of actual arguments must match the number of formal arguments of the callee.
- The number of return values must match the number of return values of the callee.
- The data types of the actual and formal arguments must be compatible.
- No multilingual inlining is permitted. Caller and callee must be written in the same source language.

**Minimum criteria for the caller:**

- At most 2000 intermediate statements will be inlined into the caller from all the call-sites being inlined into the caller. You can change this value by specifying the option

  **-Qoption,*f*,-*ip_ninl_max_total_stats*=new value**

- The function must be called if it is declared as static. Otherwise, it will be deleted.

**Minimum criteria for the callee:**

- Does not have variable argument list.

- Is not considered infrequent due to the name. Routines which contain the following substrings in their names are not inlined: `abort`, `alloca`, `denied`, `err`, `exit`, `fail`, `fatal`, `fault`, `halt`, `init`, `interrupt`, `invalid`, `quit`, `rare`, `stop`, `timeout`, `trace`, `trap`, and `warn`.
- Is not considered unsafe for other reasons.

### Selecting Routines for Inlining with or without PGO

Once the above criteria are met, the compiler picks the routines whose inline expansions will provide the greatest benefit to program performance. This is done using the default heuristics. The inlining heuristics used by the compiler differ based on whether you use profile-guided optimizations (`-prof_use`) or not.

When you use profile-guided optimizations with `-ip` or `-ipo`, the compiler uses the following heuristics:

- The default heuristic focuses on the most frequently executed call sites, based on the profile information gathered for the program.
- By default, the compiler does not inline functions with more than 230 intermediate statements. You can change this value by specifying the option `-Qoption,`*`f,-ip_ninl_max_stats`*`=new value`.
- The default inline heuristic will stop inlining when direct recursion is detected.
- The default heuristic always inlines very small functions that meet the minimum inline criteria.
  - Default for Itanium®-based applications: `ip_ninl_min_stats = 15`.
  - Default for IA-32 applications: `ip_ninl_min_stats = 7`.
- These limits can be modified with the option `-Qoption,`*`f,-ip_ninl_min_stats`*`=new value`.

  See -Qoption Specifiers and Profile-Guided Optimization (PGO).

When you do not use profile-guided optimizations with `-ip` or `-ipo`, the compiler uses less aggressive inlining heuristics: it inlines a function if the inline expansion does not increase the size of the final program.

### Inlining and Preemption

Preemption of a function means that the code, which implements that function at run-time, is replaced by different code. When a function is preempted, the new version of this function is executed rather than the old version. Preemption can be used to replace an erroneous or inferior version of a function with a correct or improved version.

The compiler assumes that when `-ip` is on, any externally visible function might be preempted and therefore cannot be inlined. Currently, this means that all Fortran subprograms, except for internal procedures, are not inlinable when `-ip` is on.

However, if you use `-ipo` and `-ipo_obj` on a file-by-file basis, the functions can be inlined. See Compilation with Real Object Files.

### Controlling Inline Expansion of User Functions

The compiler enables you to control the amount of inline function expansion, with the options shown in the following summary.

| Option | Effect |
|---|---|
| `-ip_no_inlining` | This option is only useful if `-ip` or `-ipo` is also specified. In this case, `-ip_no_inlining` disables inlining that would result from the `-ip` interprocedural optimizations, but has no effect on other interprocedural optimizations. |
| `-inline_debug_info` | Preserve the source position of inlined code instead of assigning the call-site source position to inlined code. |
| `-ip_no_pinlining` | Disables partial inlining; can be used if `-ip` or `-ipo` is also specified. |

### Inline Expansion of Library Functions

By default, the compiler automatically expands (inlines) a number of standard and math library functions at the point of the call to that function, which usually results in faster computation.

However, the inlined library functions do not set the `errno` variable when being expanded inline. In code that relies upon the setting of the `errno` variable, you should use the `-nolib_inline` option. Also, if one of your functions has the same name as one of the compiler-supplied library functions, then when this function is called, the compiler assumes that the call is to the library function and replaces the call with an inlined version of the library function.

So, if the program defines a function with the same name as one of the known library routines, you must use the `-nolib_inline` option to ensure that the user-supplied function is used.
`-nolib_inline` disables inlining of all intrinsics.

**Note**
Automatic inline expansion of library functions is not related to the inline expansion that the compiler does during interprocedural optimizations. For example, the following command compiles the program `sum.f` without expanding the math library functions:

```
ifort -ip -nolib_inline sum.f
```

## Profile-guided Optimizations

### Profile-guided Optimizations Overview

Profile-guided optimizations (PGO) tell the compiler which areas of an application are most frequently executed. By knowing these areas, the compiler is able to be more selective and specific in optimizing the application. For example, the use of PGO often enables the compiler to make better decisions about function inlining, thereby increasing the effectiveness of interprocedural optimizations.

#### Instrumented Program

Profile-guided Optimization creates an instrumented program from your source code and special code from the compiler. Each time this instrumented code is executed, the instrumented program generates a dynamic information file. When you compile a second time, the dynamic information files are merged into a summary file. Using the profile information in this file, the compiler attempts to optimize the execution of the most heavily travelled paths in the program.

Unlike other optimizations such as those strictly for size or speed, the results of IPO and PGO vary. This is due to each program having a different profile and different opportunities for optimizations. The guidelines provided help you determine if you can benefit by using IPO and PGO. You need to understanding the principles of the optimizations and the unique aspects of your source code.

#### Added Performance with PGO

In this version of the Intel® Fortran Compiler, PGO is improved in the following ways:

- Register allocation uses the profile information to optimize the location of spill code.
- For indirect function calls, branch prediction is improved by identifying the most likely targets. With the Intel® Pentium® 4 and Intel® Xeon(TM) processors' longer pipeline, improving branch prediction translates into high performance gains.

- The compiler detects and does not vectorize loops that execute only a small number of iterations, reducing the run time overhead that vectorization might otherwise add.

## Profile-guided Optimizations Methodology and Usage Model

PGO works best for code with many frequently executed branches that are difficult to predict at compile time. An example is the code with intensive error-checking in which the error conditions are false most of the time. The "cold" error-handling code can be placed such that the branch is hardly ever mispredicted. Minimizing "cold" code interleaved into the "hot" code improves instruction cache behavior.

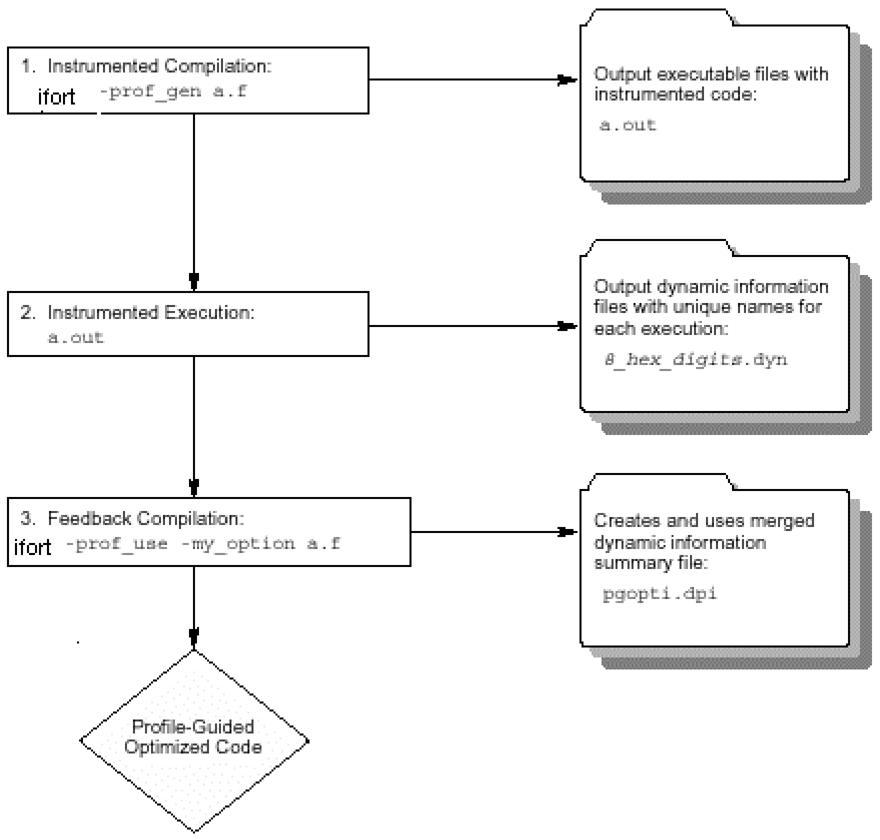
### PGO Phases

The PGO methodology requires three phases and options:

1. **Instrumentation compilation** and linking with `-prof_gen`
2. **Instrumented execution** by running the executable; as a result, the dynamic-information files (`.dyn`) are produced.
3. **Feedback compilation** with `-prof_use`

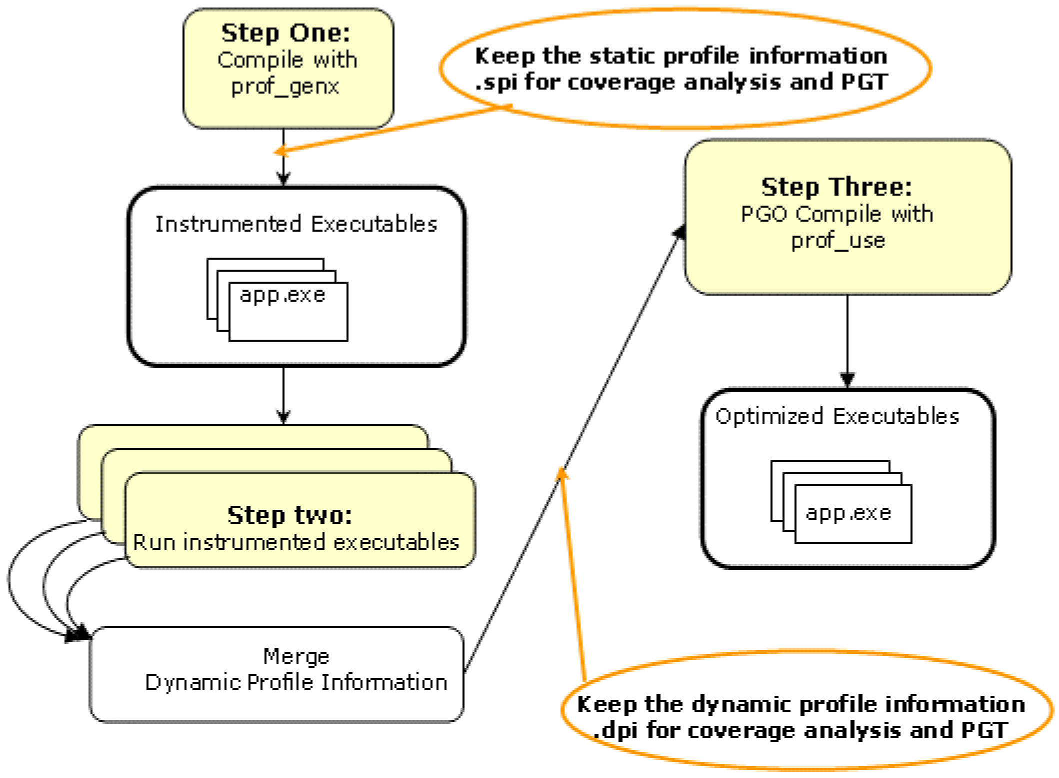
The flowcharts below illustrate this process for IA-32 compilation and Itanium®-based compilation . A key factor in deciding whether you want to use PGO lies in knowing which sections of your code are the most heavily used. If the data set provided to your program is very consistent and it elicits a similar behavior on every execution, then PGO can probably help optimize your program execution. However, different data sets can elicit different algorithms to be called. This can cause the behavior of your program to vary from one execution to the next.

### Phases of Basic Profile-Guided Optimization


1. Instrumented Compilation:
ifort -prof_gen a.f
Output executable files with instrumented code:
a.out
2. Instrumented Execution:
a.out
Output dynamic information files with unique names for each execution:
8_hex_digits.dyn
3. Feedback Compilation:
ifort -prof_use -my_option a.f
Creates and uses merged dynamic information summary file:
pgopti.dpi
Profile-Guided Optimized Code

## PGO Usage Model

The chart that follows presents PGO usage model.


Step One:
Compile with
prof_genx
Keep the static profile information
.spi for coverage analysis and PGT
Instrumented Executables
app.exe
Step two:
Run instrumented executables
Merge
Dynamic Profile Information
Step Three:
PGO Compile with
prof_use
Optimized Executables
app.exe
Keep the dynamic profile information
.dpi for coverage analysis and PGT

Here are the steps for a simple example (`myApp.f90`) for IA-32 systems.

1. Set the following:

   **`PROF_DIR=c:/myApp/prof_dir`**

2. Issue the following command:

   **`ifort -prof_genx myApp.f90`**

   This command compiles the program and generates instrumented binary `myApp.exe` as well as the corresponding static profile information `pgopti.spi`.

3. Execute `myApp.`

   Each invocation of `myApp` runs the instrumented application and generates one or more new dynamic profile information files that have an extension `.dyn` in the directory specified by `PROF_DIR`.

4. Issue the following command:

   **`ifort -prof_use myApp.f90`**

At this step, the compiler merges all the `.dyn` files into one `.dpi` file representing the total profile information of the application and generates the optimized binary. The default name of the `.dpi` file is `pgopti.dpi`.

## Basic PGO Options

The options used for basic PGO optimizations are:

- `-prof_gen` to generate instrumented code
- `-prof_use` to generate a profile-optimized executable
- `-prof_format_32` to produce 32-bit counters for `.dyn` and `.dpi` files

In cases where your code behavior differs greatly between executions, you have to ensure that the benefit of the profile information is worth the effort required to maintain up-to-date profiles. In the basic profile-guided optimization, the following options are used in the phases of the PGO:

### Generating Instrumented Code, `-prof_gen`

The `-prof_gen` option instruments the program for profiling to get the execution count of each basic block. It is used in phase 1 of the PGO to instruct the compiler to produce instrumented code in your object files in preparation for instrumented execution. Parallel `make` is automatically supported for `-prof_gen` compilations.

### Generating a Profile-optimized Executable, `-prof_use`

The `-prof_use` option is used in phase 3 of the PGO to instruct the compiler to produce a profile-optimized executable and merges available dynamic-information (`.dyn`) files into a `pgopti.dpi` file.

**Note**

The dynamic-information files are produced in phase 2 when you run the instrumented executable.

If you perform multiple executions of the instrumented program, `-prof_use` merges the dynamic-information files again and overwrites the previous `pgopti.dpi` file.

### Using 32-bit Counters, `-prof_format_32`

The Intel Fortran compiler by default produces profile data with 64-bit counters to handle large numbers of events in the `.dyn` and `.dpi` files. The `-prof_format_32` option produces 32-bit counters for compatibility with the earlier compiler versions. If the format of the `.dyn` and `.dpi` files is incompatible with the format used in the current compilation, the compiler issues the following message:

```
Error:  xxx.dyn has old or incompatible file format - delete
file and redo instrumentation compilation/execution.
```

The 64-bit format for counters and pointers in `.dyn` and `.dpi` files eliminate the incompatibilities on various platforms due to different pointer sizes.

**Disabling Function Splitting, `-fnsplit-`**

`-fnsplit-` disables function splitting on Itanium®-based systems. Function splitting is enabled by `-prof_use` in phase 3 to improve code locality by splitting routines into different sections: one section to contain the cold or very infrequently executed code and one section to contain the rest of the code (hot code).

You can use `-fnsplit-` to disable function splitting for the following reasons:

- Most importantly, to get improved debugging capability. In the debug symbol table, it is difficult to represent a split routine, that is, a routine with some of its code in the hot code section and some of its code in the cold code section. The `-fnsplit-` option disables the splitting within a routine but enables function grouping, an optimization in which entire routines are placed either in the cold code section or the hot code section. Function grouping does not degrade debugging capability.

- Another reason can arise when the profile data does not represent the actual program behavior, that is, when the routine is actually used frequently rather than infrequently.

  **Note**

  For Itanium®-based applications, if you intend to use the `-prof_use` option with optimizations at the `-O3` level, the `-O3` option must be on. If you intend to use the `-prof_use` option with optimizations at the `-O2` level or lower, you can generate the profile data with the default options.

See an example of using PGO.

## Advanced PGO Options

The options controlling advanced PGO optimizations are:

- `-prof_dir`*`dirname`*
- `-prof_file`*`filename`*.

Use the `-prof_dir`*`dirname`* option to specify the directory in which you intend to place the dynamic information (`.dyn`) files to be created. The default is the directory where the program is compiled. The specified directory must already exist.

You should specify `-prof_dir`*`dirname`* option with the same directory name for both the instrumentation and feedback compilations. If you move the `.dyn` files, you need to specify the new path.

The `-prof_file`*`filename`* option specifies file name for profiling summary file.

### Guidelines for Using Advanced PGO

When you use PGO, consider the following guidelines:

- Minimize the changes to your program after instrumented execution and before feedback compilation. During feedback compilation, the compiler ignores dynamic information for functions modified after that information was generated.

  **Note**
  The compiler issues a warning that the dynamic information does not correspond to a modified function.

- Repeat the instrumentation compilation if you make many changes to your source files after execution and before feedback compilation.
- Specify the name of the profile summary file using the `-prof_file`*`filename`* option

See PGO Environment Variables.

## PGO Environment Variables

The environment variables determine the directory in which to store dynamic information files or whether to overwrite `pgopti.dpi`. The PGO environment variables are described in the table below.

| **Variable** | **Description** |
|---|---|
| `PROF_DIR` | Specifies the directory in which dynamic information files are created. This variable applies to all three phases of the profiling process. |
| `PROF_DUMP_INTERVAL` | Initiates interval profile dumping in an instrumented user application. |

| | |
|---|---|
| `PROF_NO_CLOBBER` | Alters the feedback compilation phase slightly. By default, during the feedback compilation phase, the compiler merges the data from all dynamic information files and creates a new `pgopti.dpi` file, even if one already exists. When this variable is set, the compiler does not overwrite the existing `pgopti.dpi` file. Instead, the compiler issues a warning and you must remove the `pgopti.dpi` file if you want to use additional dynamic information files. |

See the documentation for your operating system for instructions on how to specify environment variables and their values.

## Example of Profile-Guided Optimization

The following is an example of the basic PGO phases:

1. **Instrumentation Compilation and Linking**—Use `-prof_gen` to produce an executable with instrumented information. Use also the `-prof_dir` option as recommended for most programs, especially if the application includes the source files located in multiple directories. `-prof_dir` ensures that the profile information is generated in one consistent place. For example:

   ```
   ifort -prof_gen -prof_dir/usr/profdata -c a1.f a2.f
   a3.f
   ifort -oa1 a1.o a2.o a3.o
   ```

   In place of the second command, you could use the linker (`ld`) directly to produce the instrumented program. If you do this, make sure you link with the `libirc.a` library.

2. **Instrumented Execution**—Run your instrumented program with a representative set of data to create a dynamic information file.

   ```
   prompt> a1
   ```

   The resulting dynamic information file has a unique name and `.dyn` suffix every time you run `a1`. The instrumented file helps predict how the program runs with a particular set of data. You can run the program more than once with different input data.

3. **Feedback Compilation**—Compile and link the source files with `-prof_use` to use the dynamic information to optimize your program according to its profile:

**`ifort -prof_use -prof_dir/usr/profdata -ipo a1.f a2.f a3.f`**

Besides the optimization, the compiler produces a `pgopti.dpi` file. You typically specify the default optimizations (`-O2`) for phase 1, and specify more advanced optimizations (`-ip` or `-ipo`) for phase 3. This example used `-O2` in phase 1 and the `-ipo` in phase 3.

**Note**
The compiler ignores the `-ip` or the `-ipo` options with `-prof_gen`.

See Basic PGO Options.

## Merging the .dyn Files

To merge the `.dyn` files, use the `profmerge` utility. The compiler executes `profmerge` automatically during the feedback compilation phase when you specify `-prof_use`.

The command-line usage for `profmerge` is as follows:

**`profmerge [-nologo] [-prof_dir`*`dirname`*`]`**

where **`-prof_dir`*`dirname`*** is a `profmerge` utility option.

This merges all `.dyn` files in the current directory or the directory specified by `-prof_dir`, and produces the summary file `pgopti.dpi`.

The `-prof_file`*`filename`* option enables you to specify the name of the `.dpi` file.

The command-line usage for `profmerge` with `-prof_file`*`filename`* is as follows:

**`profmerge [-nologo] [-prof_file`*`filename`*`]`**

where **`/prof_file`*`filename`*** is a `profmerge` utility option.

**Note**
The `profmerge` tool merges all the `.dyn` files that exist in the given directory. It is very important to make sure that unrelated `.dyn` files, oftentimes from previous runs, are not present in that directory. Otherwise, profile information will be based on invalid profile data. This can negatively impact the performance of optimized code as well as generate misleading coverage information.

**Note**
The `.dyn` files can be merged to a `.dpi` file by the `profmerge` tool without recompiling the application.

**Dumping Profile Data**

This subsection provides an example of how to call the C PGO API routines from Fortran. For complete description of the PGO API support routines, see PGO API: Profile Information Generation Support.

As part of the instrumented execution phase of profile-guided optimization, the instrumented program writes profile data to the dynamic information file (`.dyn` file). The file is written after the instrumented program returns normally from `main()` or calls the standard exit function. Programs that do not terminate normally, can use the `_PGOPTI_Prof_Dump` function. During the instrumentation compilation
(`-prof_gen`) you can add a call to this function to your program. Here is an example:

```
INTERFACE
SUBROUTINE PGOPTI_PROF_DUMP()
!DEC$ ATTRIBUTES C,
ALIAS:'PGOPTI_Prof_Dump'::PGOPTI_PROF_DUMP
END SUBROUTINE
END INTERFACE
CALL PGOPTI_PROF_DUMP()
```

**Note**
You must remove the call or comment it out prior to the feedback compilation with `-prof_use`.

## Using profmerge to Relocate the Source Files

The compiler uses the full path to the source file for each routine to look up the profile summary information associated with that routine. By default, this prevents you from:

- Using the profile summary file (`.dpi`) if you move your application sources.
- Sharing the profile summary file with another user who is building identical application sources that are located in a different directory.

To enable the movement of application sources, as well as the sharing of profile summary files, use the `profmerge` with `-src_old` and `-src_new` options. For example:

```
prompt>profmerge -prof_dir c:/work -src_old c:/work/sources
-src_new d:/project/src
```

The above command will read the `c:/work/pgopti.dpi` file. For each routine represented in the `pgopti.dpi` file, whose source path begins with the `c:/work/sources` prefix, `profmerge` replaces that prefix with `d:/project/src`. The `c:/work/pgopti.dpi` file is updated with the new source path information.

The following rules apply:

- You can execute profmerge more than once on a given pgopti.dpi file. You may need to do this if the source files are located in multiple directories. For example:

```
profmerge -src_old "c:/program files" -src_new
"e:/program files"

profmerge -src_old c:/proj/application -src_new d:/app
```

- In the values specified for `-src_old` and `-src_new`, uppercase and lowercase characters are treated as identical. Likewise, forward slash (`/`) and backward slash (`\`) characters are treated as identical.
- Because the source relocation feature of `profmerge` modifies the `pgopti.dpi` file, you may wish to make a backup copy of the file prior to performing the source relocation.

## Code-coverage Tool

The Intel® Compilers Code-coverage tool can be used for both IA-32 and Itanium® architectures, in a number of ways to improve development efficiency, reduce defects, and increase application performance. The major features of the Intel Compilers code-coverage tool are:

- Visual presentation of the application's code coverage information with the code-coverage coloring scheme
- Display of the dynamic execution counts of each basic block of the application
- Differential coverage, or comparison of the profiles of the application's two runs

The syntax for this tool is as follows:

```
codecov [-codecov_option]
```

where `-codecov_option` is a tool option you choose to run the code coverage with. If you do not use any option, the tool will provide the top level code coverage for your whole program.

The tool uses options that are listed in the table that follows.

| Option | Description | Default |
|---|---|---|
| `-help` | Prints all the options of the code-coverage tool. | |
| `-spi` *file* | Sets the path name of the static profile information file `.spi`. | `pgopti.spi` |
| `-dpi` *file* | Sets the path name of the dynamic profile information file `.dpi`. | `pgopti.dpi` |
| `-prj` | Sets the project name. | |
| `-counts` | Generates dynamic execution counts. | |
| `-nopartial` | Treats partially covered code as fully covered code. | |
| `-comp` | Sets the filename that contains the list of files of interest. | |
| `-ref` | Finds the differential coverage with respect to ref_dpi_file. | |
| `-demang` | Demangles both function names and their arguments. | |
| `-mname` | Sets the name of the web-page owner. | |
| `-maddr` | Sets the email address of the web-page owner. | |
| `-bcolor` | Sets the html color name or code of the uncovered blocks. | `#ffff99` |
| `-fcolor` | Sets the html color name or code of the uncovered functions. | `#ffcccc` |
| `-pcolor` | Sets the html color name or code of the partially covered code. | `#fafad2` |
| `-ccolor` | Sets the html color name or code of the covered code. | `#ffffff` |
| `-ucolor` | Sets the html color name or code of the unknown code. | `#ffffff` |

**Visual Presentation of the Application's Code Coverage**

Based on the profile information collected from running the instrumented binaries when testing an application, Intel® Compiler creates HTML files using a code-coverage tool. These HTML files indicate portions of the source code that were or were not exercised by the tests. When applied to the profile of the performance workloads, the code-coverage information shows how well the training workload covers the application's critical code. High coverage of performance-critical modules is essential to taking full advantage of the profile-guided optimizations.

The code-coverage tool can create two levels of coverage:

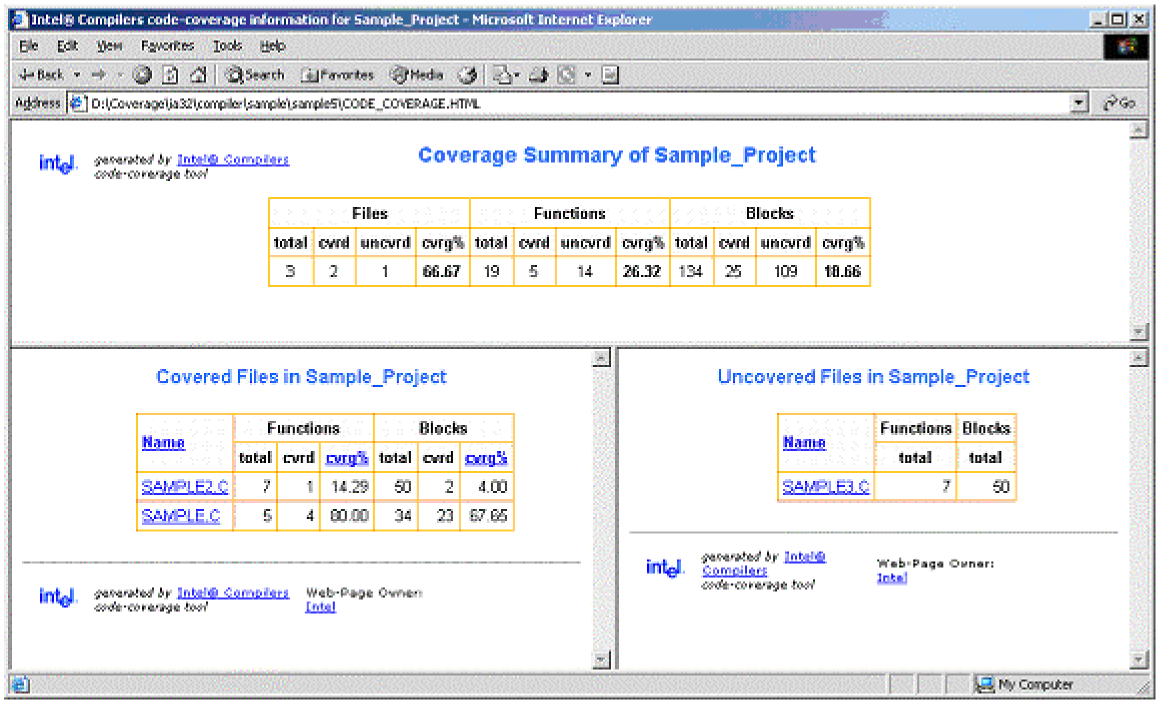
- Top level: for a group of selected modules
- Individual module source view

### Top Level Coverage

The top-level coverage reports the overall code coverage of the modules that were selected. The following options are provided:

- You can select the modules of interest
- For the selected modules, the tool generates a list with their coverage information. The information includes the total number of functions and blocks in a module and the portions that were covered.
- By clicking on the title of columns in the reported tables, the lists may be sorted in ascending or descending order based on:
    - basic block coverage
    - function coverage
    - function name.

The screenshot that follows shows a sample top-level coverage summary for a project. By clicking on a module name (for example, `SAMPLE.C`), the browser will display the coverage source view of that particular module.


Intel® Compilers code-coverage information for Sample_Project - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address D:\Coverage\ia32\compiler\sample\sample5\CODE_COVERAGE.HTML
Go
My Computer

generated by Intel® Compilers code-coverage tool

Coverage Summary of Sample_Project

| Files | | | | Functions | | | | Blocks | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| total | cvrd | uncvrd | cvrg% | total | cvrd | uncvrd | cvrg% | total | cvrd | uncvrd | cvrg% |
| 3 | 2 | 1 | 66.67 | 19 | 5 | 14 | 26.32 | 134 | 25 | 109 | 18.66 |

Covered Files in Sample_Project

| Name | Functions | | | Blocks | | |
|---|---|---|---|---|---|---|
| | total | cvrd | cvrg% | total | cvrd | cvrg% |
| SAMPLE2.C | 7 | 1 | 14.29 | 50 | 2 | 4.00 |
| SAMPLE.C | 5 | 4 | 80.00 | 34 | 23 | 67.65 |

generated by Intel® Compilers code-coverage tool
Web-Page Owner: Intel

Uncovered Files in Sample_Project

| Name | Functions | Blocks |
|---|---|---|
| | total | total |
| SAMPLE3.C | 7 | 50 |

generated by Intel® Compilers code-coverage tool
Web-Page Owner: Intel

### Browsing the Frames

The coverage tool creates frames that facilitate browsing through the code to identify uncovered code. The top frame displays the list of uncovered functions while the bottom frame displays the list of covered functions. For uncovered functions, the total number of basic blocks of each function is also displayed. For covered functions, both the total number of blocks and the number of covered blocks as well as their ratio (that is, the coverage rate) are displayed.

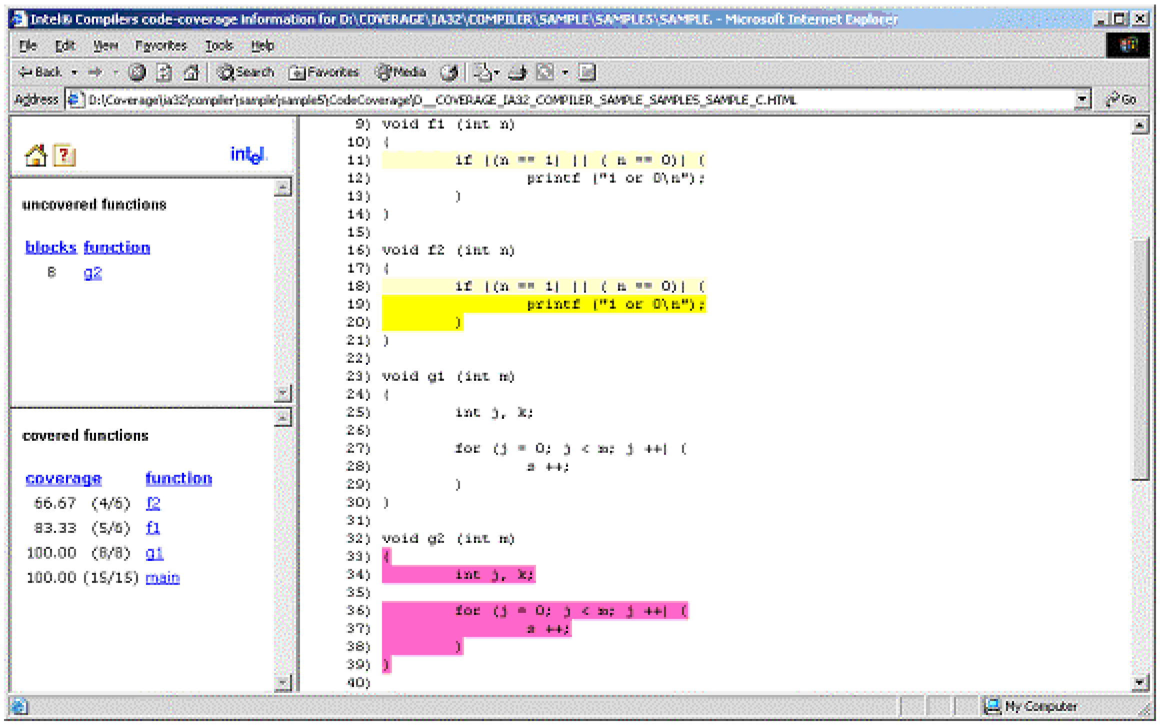
For example, 66.67(4/6) indicates that four out of the six blocks of the corresponding function were covered. The block coverage rate of that function is thus 66.67%. These lists can be sorted based on the coverage rate, number of blocks, or function names.  Function names are linked to the position in source view where the function body starts. So, just by one click, the user can see the least-covered function in the list and by another click the browser displays the body of the function. The user can then scroll down in the source view and browse through the function body.

**Individual Module Source View**

Within the individual module source views, the tool provides the list of uncovered functions as well as the list of covered functions. The lists are reported in two distinct frames that provide easy navigation of the source code. The lists can be sorted based on:

- the number of blocks within uncovered functions
- the block coverage in the case of covered functions
- the function names.

The following screen shows the coverage source view of `SAMPLE.C`.


Intel® Compilers code-coverage information for D:\COVERAGE\IA32\COMPILER\SAMPLE\SAMPLES\SAMPLE. - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address D:\Coverage\ia32\compiler\sample\sample5\CodeCoverage\D__COVERAGE_IA32_COMPILER_SAMPLE_SAMPLES_SAMPLE_C.HTML
Go
My Computer

uncovered functions

| blocks | function |
|---|---|
| 8 | g2 |

covered functions

| coverage | | function |
|---|---|---|
| 66.67 | (4/6) | f2 |
| 83.33 | (5/6) | f1 |
| 100.00 | (8/8) | g1 |
| 100.00 | (15/15) | main |

```
void f1 (int n)
{
        if ((n == 1) || ( n == 0)) {
                printf ("1 or 0\n");
        }
}

void f2 (int n)
{
        if ((n == 1) || ( n == 0)) {
                printf ("1 or 0\n");
        }
}

void g1 (int m)
{
        int j, k;

        for (j = 0; j < m; j ++) {
                k ++;
        }
}

void g2 (int m)
{
        int j, k;

        for (j = 0; j < m; j ++) {
                k ++;
        }
}

```

**Setting the Coloring Scheme for the Code Coverage**

The tool provides a visible coloring distinction of the following coverage categories:

- covered code
- uncovered basic blocks
- uncovered functions
- partially covered code
- unknown.

The default colors that the tool uses for presenting the coverage information are shown in the tables that follows.

| This color | Means |
|---|---|
| Covered code | The portion of code colored in this color was exercised by the tests. The default color can be overridden with the `-ccolor` option. |
| 🟨 Uncovered basic block | Basic blocks that are colored in this color were not exercised by any of the tests.<br><br>They were, however, within functions that were executed during the tests.<br><br>The default color can be overridden with the `-bcolor` option. |
| 🟪 Uncovered | Functions that are colored in this color were never called |

| | |
|---|---|
| function | during the tests. The default color can be overridden with the -`fcolor` option. |
| Partially covered code | More than one basic block was generated for the code at this position.<br><br>Some of the blocks were covered while some were not. The default color can be overridden with the `-pcolor` option. |
| Unknown | No code was generated for this source line. Most probably, the source at this position is a comment, a header-file inclusion, or a variable declaration. The default color can be overridden with the `-ucolor` option. |

The default colors can be customized to be any valid HTML by using the options mentioned for each coverage category in the table above.

For code-coverage colored presentation, the coverage tool uses the following heuristic. Source characters are scanned until reaching a position in the source that is indicated by the profile information as the beginning of a basic block. If the profile information for that basic block indicates that a coverage category changes, then the tool changes the color corresponding to the coverage condition of that portion of the code, and the coverage tool inserts the appropriate color change in the HTML files.

**Note**
You need to interpret the colors in the context of the code. For instance, comment lines that follow a basic block that was never executed would be colored in the same color as the uncovered blocks. Another example is the closing brackets in C/C++ applications.

**Coverage Analysis of a Modules Subset**

One of the capabilities of the Intel Compilers code-coverage tool is efficient coverage analysis of an application' s subset of modules. This analysis is accomplished based on the selected option `-comp` of the tool's execution.

You can generate the profile information for the whole application, or a subset of it, and then break the covered modules into different components and use the coverage tool to obtain the coverage information of each individual component. If only a subset of the application modules is compiler with the `-prof_genx` option, then the coverage information is generated only for those modules that are involved with this compiler option, thus avoiding the overhead incurred for profile generation of other modules.

To specify the modules of interest, use the tool's `-comp` option. This option takes the name of a file as its argument. That file must be a text file that includes the name of modules or directories you would like to analyze. Here is an example:

**`codecov -prj Project_Name -comp`** ***`component1`***

> **Note**
> Each line of component file should include one, and only one, module name.

Any module of the application whose full path name has an occurrence of any of the names in the component file will be selected for coverage analysis. For example, if a line of file `component1` in the above example contains `mod1.f90`, then all modules in the application that have such a name will be selected. The user can specify a particular module by giving more specific path information. For instance, if the line contains `/cmp1/mod1.f90`, then only those modules with the name mod1.c will be selected that are in a directory named `cmp1`. If no component file is specified, then all files that have been compiled with `-prof_genx` are selected for coverage analysis.

**Dynamic Counters**

This feature displays the dynamic execution count of each basic block of the application, and as such it is useful for both coverage and performance tuning.

The coverage tool can be configured to generate the information about the dynamic execution counts. This configuration requires using the `-counts` option. The counts information is displayed under the code after a **^** sign precisely under the source position where the corresponding basic block begins. If more than one basic block is generated for the code at a source position (for example, for macros), then the total number of such blocks and the number of the blocks that were executed are also displayed in front of the execution count.

For example, line 11 in the code is an `IF` statement:

```
11   IF ((N .EQ. 1).OR. (N .EQ. 0))
^ 10 (1/2)
12      PRINT N
^ 7
```

The coverage lines under code lines 11 and 12 contain the following information:

- The `IF` statement in line 11 was executed 10 times.
- Two basic blocks were generated for the `IF` statement in line 11.
- Only one of the two blocks was executed, hence the partial coverage color.
- Only seven out of the ten times variable n had a value of 0 or 1.

In certain situations, it may be desirable to consider all the blocks generated for a single source position as one entity. In such cases, it is necessary to assume that all blocks generated for one source position are covered when at least one of the blocks is covered. This assumption can be configured with the `-nopartial` option. When this option is specified, decision coverage is disabled, and the related statistics are adjusted accordingly. The code lines 11 and 12 indicate that the PRINT statement in line 12 was covered. However, only one of the conditions in line 11 was ever true. With the `-nopartial` option, the tool treats the partially covered code (like the code on line 11) as covered.

**Differential Coverage**

Using the code-coverage tool, you can compare the profiles of the application's two runs: a reference run and a new run identifying the code that is covered by the new run but not covered by the reference run. This feature can be used to find the portion of the application's code that is not covered by the application's tests but is executed when the application is run by a customer. It can also be used to find the incremental coverage impact of newly added tests to an application's test space.

The dynamic profile information of the reference run for differential coverage is specified by the `-ref` option.  such as in the following command:

```
codecov -prj Project_Name -dpi customer.dpi -ref
appTests.dpi
```

The coverage statistics of a differential-coverage run shows the percentage of the code that was exercised on a new run but was missed in the reference run. In such cases, the coverage tool shows only the modules that included the code that was uncovered.

The coloring scheme in the source views also should be interpreted accordingly. The code that has the same coverage property (covered or not covered) on both runs is considered as covered code. Otherwise, if the new run indicates that the code was executed while in the reference run the code was not executed, then the code is treated as uncovered. On the other hand, if the code is covered in the reference run but not covered in the new run, the differential-coverage source view shows the code as covered.

**Running for Differential Coverage**

To run the Intel Compilers code-coverage tool for differential coverage, the following files are required:

- The application sources
- The `.spi` file generated by Intel Compilers when compiling the application for the instrumented binaries with the `-prof_genx` option.
- The `.dpi` file generated by Intel Compilers `profmerge` utility as the result of merging the dynamic profile information `.dyn` files or the `.dpi` file

generated implicitly by Intel Compilers when compiling the application with the `-prof_use` option.

See Usage Model of the Profile-guided Optimizations.

Once the required files are available, the coverage tool may be launched from this command line:

```
codecov -prj Project_Name -spi pgopti.spi -dpi pgopti.dpi
```

The `-spi` and `-dpi` options specify the paths to the corresponding files.

The coverage tool also has the following additional options for generating a link at the bottom of each HTML page to send an electronic message to a named contact by using -`mname` and `-maddr` options.

```
codecov -prj Project_Name -mname John_Smith -maddr
js@company.com
```

## Test Prioritization Tool

The Intel® Compilers Test-prioritization tool enables the profile-guided optimizations to select and prioritize application's tests based on prior execution profiles of the application. The tool offers a potential of significant time saving in testing and developing large-scale applications where testing is the major bottleneck. The tool can be used for both IA-32 and Itanium® architectures.

This tool enables the users to select and prioritize the tests that are most relevant for any subset of the application's code. When certain modules of an application are changed, the test-prioritization tool suggests the tests that are most probably affected by the change. The tool analyzes the profile data from previous runs of the application, discovers the dependency between the application's components and its tests, and uses this information to guide the process of testing.

### Features and Benefits

The tool provides an effective testing hierarchy based on the application's code coverage. The advantages of the tool usage can be summarized as follows:

- Minimizing the number of tests that are required to achieve a given overall coverage for any subset of the application: the tool defines the smallest subset of the application tests that achieve exactly the same code coverage as the entire set of tests.
- Reducing the turn-around time of testing: instead of spending a long time on finding a possibly large number of failures, the tool enables the users to quickly find a small number of tests that expose the defects associated with the regressions caused by a change set.

- Selecting and prioritizing the tests to achieve certain level of code coverage in a minimal time based on the data of the tests' execution time.

### Command-line Syntax

The syntax for this tool is as follows:

**`tselect -dpi_list`** ***`file`***

where `-dpi_list` is a required tool option that sets the path to the DPI list *`file`* that contains the list of the `.dpi` files of the tests you need to prioritize.

### Tool Options

The tool uses options that are listed in the table that follows.

| Option | Description |
|---|---|
| `-help` | Prints all the options of the test-prioritization tool. |
| `-spi` *`file`* | Sets the path name of the static profile information file `.spi`. The default is `pgopti.spi`. |
| `-dpi_list` *`file`* | Sets the path name of the file that contains the name of the dynamic profile information (`.dpi`) files. Each line of the file should contain one `.dpi` name optionally followed by its execution time. The name must uniquely identify the test. |
| `-prof_dpi` *`file`* | Sets the path name of the output report file. |
| `-comp` | Sets the filename that contains the list of files of interest. |
| `-cutoff` *`value`* | Terminates when the cumulative block coverage reaches *`value`*% of pre-computed total coverage. *`value`* must be greater than 0.0 (for example, 99.00). It may be set to 100. |
| `-nototal` | Does not pre-compute the total coverage. |
| `-mintime` | Minimizes testing execution time. The execution time of each test must be provided on the same line of `dpi_list` file after the test name in `dd:hh:mm:ss` format. |
| `-verbose` | Generates more logging information about the program progress. |

### Usage Requirements

To run the test-prioritization tool on an application's tests, the following files are required:

- The `.spi` file generated by Intel Compilers when compiling the application for the instrumented binaries with the `-prof_genx` option.
- The `.dpi` files generated by Intel Compilers `profmerge` tool as a result of merging the dynamic profile information `.dyn` files of each of the application tests. The user needs to apply the `profmerge` tool to all `.dyn` files that are generated for each individual test and name the resulting `.dpi` in a fashion that uniquely identifies the test. The profmerge tool merges all the `.dyn` files that exist in the given directory.

  **Note**
  It is very important that the user makes sure that unrelated `.dyn` files, oftentimes from previous runs or from other tests, are not present in that directory. Otherwise, profile information will be based on invalid profile data. This can negatively impact the performance of optimized code as well as generate misleading coverage information.

- User-generated file containing the list of tests to be prioritized.

  For successful tool execution, you should:

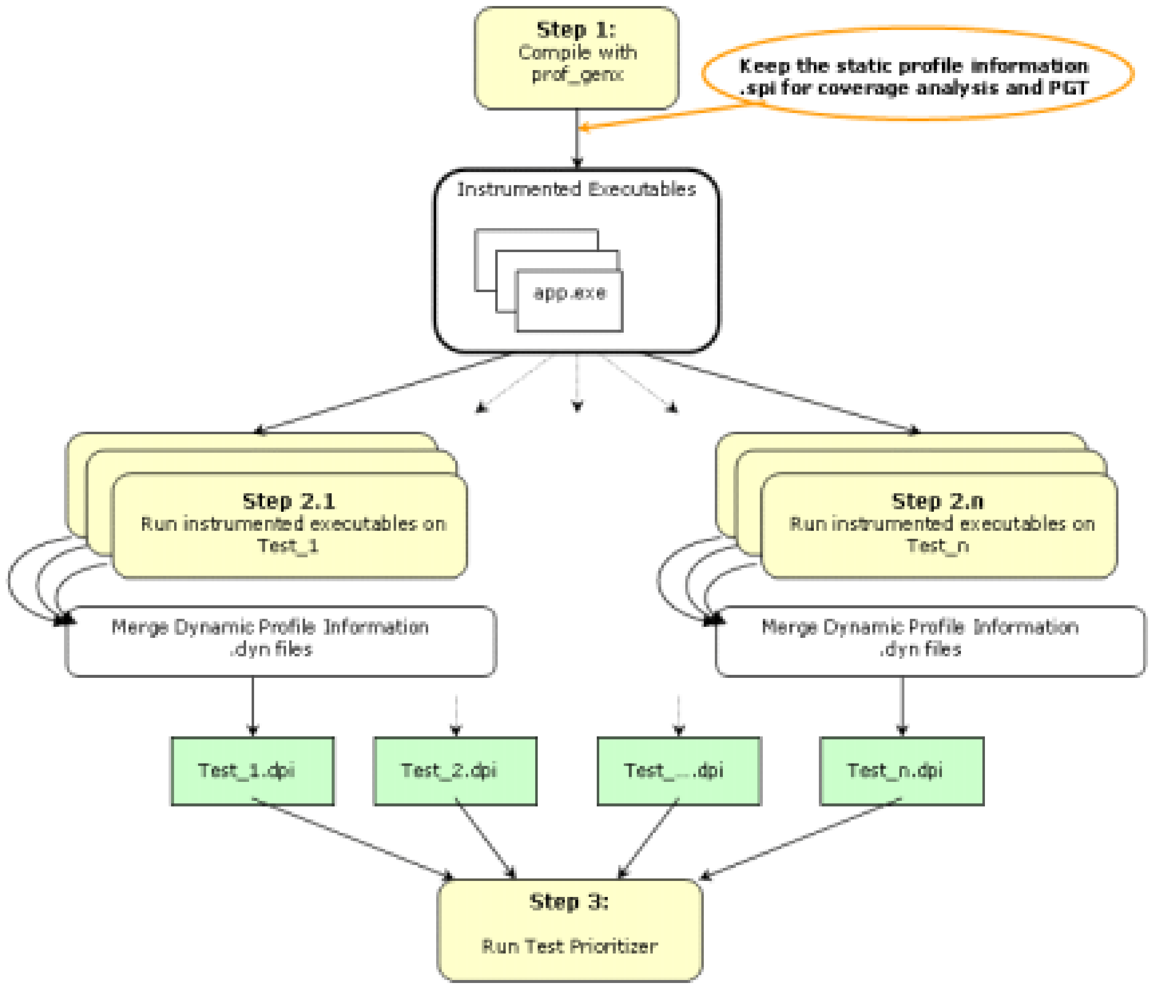
  - Name each test `.dpi` file so that the file names uniquely identify each test.
  - Create a DPI list file: a text file that contains the names of all `.dpi` test files. The name of this file serves as an input for the test-prioritization tool execution command. Each line of the DPI list file should include one, and only one, `.dpi` file name. The name can optionally be followed by the duration of the execution time for a corresponding test in the `dd:hh:mm:ss` format.

  For example: `Test1.dpi 00:00:60:35` informs that Test1 lasted 0 days, 0 hours, 60 minutes and 35 seconds.

  The execution time is optional. However, if it is not provided, then the tool will not prioritize the test for minimizing execution time. It will prioritize to minimize the number of tests only.

### Usage Model

The chart that follows presents the test-prioritization tool usage model.


Step 1:
Compile with prof_genx
Keep the static profile information .spi for coverage analysis and PGT
Instrumented Executables
app.exe
Step 2.1
Run instrumented executables on Test_1
Step 2.n
Run instrumented executables on Test_n
Merge Dynamic Profile Information .dyn files
Merge Dynamic Profile Information .dyn files
Test_1.dpi
Test_2.dpi
Test_....dpi
Test_n.dpi
Step 3:
Run Test Prioritizer

Here are the steps for a simple example (`myApp.f90`) for IA-32 systems.

1. Set the following:

   **`PROF_DIR=c:/myApp/prof_dir`**

2. Issue the following command:

   **`ifort -prof_genx`** ***`myApp.f90`***

   This command compiles the program and generates instrumented binary `myApp` as well as the corresponding static profile information `pgopti.spi`.

3. Issue the following command:

   **`rm PROF_DIR /*.dyn`**

   Make sure that there are no unrelated `.dyn` files present.

4. Issue the following command:

   ```
   myApp < data1
   ```

   Invocation of this command runs the instrumented application and generates one or more new dynamic profile information files that have an extension `.dyn` in the directory specified by `PROF_DIR`.

5. Issue the following command:

   ```
   profmerge -prof_dpi Test1.dpi
   ```

   At this step, the `profmerge` tool merges all the `.dyn` files into one file (`Test1.dpi`) that represents the total profile information of the application on Test1.

6. Issue the following command:

   ```
   rm PROF_DIR /*.dyn
   ```

   Make sure that there are no unrelated `.dyn` files present.

7. Issue the following command:

   ```
   myApp < data2
   ```

   This command runs the instrumented application and generates one or more new dynamic profile information files that have an extension `.dyn` in the directory specified by `PROF_DIR`.

8. Issue the following command:

   ```
   profmerge -prof_dpi Test2.dpi
   ```

   At this step, the `profmerge` tool merges all the `.dyn` files into one file (`Test2.dpi`) that represents the total profile information of the application on Test2.

9. Issue the following command:

   ```
   rm PROF_DIR /*.dyn
   ```

   Make sure that there are no unrelated `.dyn` files present.

10. Issue the following command:

    ```
    myApp < data3
    ```

This command runs the instrumented application and generates one or more new dynamic profile information files that have an extension `.dyn` in the directory specified by `PROF_DIR`.

11. Issue the following command:

    **`profmerge -prof_dpi Test3.dpi`**

    At this step, the `profmerge` tool merges all the `.dyn` files into one file (`Test3.dpi`) that represents the total profile information of the application on Test3.

12. Create a file named `tests_list` with three lines. The first line contains `Test1.dpi`, the second line contains `Test2.dpi`, and the third line contains `Test3.dpi`.

    When these items are available, the test-prioritization tool may be launched from the command line in `PROF_DIR` directory as described in the following examples.

Note that in all examples, the discussion references the same set of data.

**Example 1 Minimizing the Number of Tests**

**`tselect -dpi_list tests_list -spi pgopti.spi`**

where the `/spi` option specifies the path to the `.spi` file.

Here is a sample output from this run of the test-prioritization tool.

```
Total number of tests   =  3

Total block coverage    ~  52.17

Total function coverage ~  50.00
```

| Num | %RatCvrg | %BlkCvrg | %FncCvrg | Test Name @ Options |
|---|---|---|---|---|
| 1 | 87.50 | 45.65 | 37.50 | Test3.dpi |
| 2 | 100.00 | 52.17 | 50.00 | Test2.dpi |

In this example, the test-prioritization tool has provided the following information:

- By running all three tests, we achieve 52.17% block coverage and 50.00% function coverage.

- Test3 by itself covers 45.65% of the basic blocks of the application, which is 87.50% of the total block coverage that can be achieved from all three tests.
- By adding Test2, we achieve a cumulative block coverage of 52.17% or 100% of the total block coverage of Test1, Test2, and Test3.
- Elimination of Test1 has no negative impact on the total block coverage.

**Example 2 Minimizing Execution Time**

Suppose we have the following execution time of each test in the `tests_list` file.

```
Test1.dpi 00:00:60:35

Test2.dpi 00:00:10:15

Test3.dpi 00:00:30:45
```

The following command executes the test-prioritization tool to minimize the execution time with the
`-mintime` option:

```
tselect -dpi_list tests_list -spi pgopti.spi -mintime
```

Here is a sample output.

```
Total number of tests   =  3

Total block coverage    ~  52.17

Total function coverage ~  50.00

Total execution time    =  1:41:35
```

| num | elapsedTime | %RatCvrg | %BlkCvrg | %FncCvrg | Test Name @ Options |
|---|---|---|---|---|---|
| 1 | 10:15 | 75.00 | 39.13 | 25.00 | Test2.dpi |
| 2 | 41:00 | 100.00 | 52.17 | 50.00 | Test3.dpi |

In this case, the results indicate that the running all tests sequentially would require one hour, 45 minutes, and 35 seconds, while the selected tests would achieve the same total block coverage in only 41 minutes.

**Note**
The order of tests when prioritization is based on minimizing time (first Test2, then Test3) could be different than when prioritization is done based on minimizing the number of tests. See example above: first Test3, then

Test2. In Example 2, Test2 is the test that gives the highest coverage per execution time. So, it is picked as the first test to run.

### Using Other Options

The `-cutoff` option enables the test-prioritization tool to exit when it reaches a given level of basic block coverage.

```
tselect -dpi_list tests_list -spi pgopti.spi -cutoff 85.00
```

If the tool is run with the cutoff value of 85.00 in the above example, only Test3 will be selected, as it achieves 45.65% block coverage, which corresponds to 87.50% of the total block coverage that is reached from all three tests.

The test-prioritization tool does an initial merging of all the profile information to figure out the total coverage that is obtained by running all the tests. The `-nototal` option. enables you to skip this step. In such a case, only the absolute coverage information will be reported, as the overall coverage remains unknown.

## PGO API: Profile Information Generation Support

### PGO API Support Overview

The Profile Information Generation Support (Profile IGS) enables you to control the generation of profile information during the instrumented execution phase of profile-guided optimizations.

Normally, profile information is generated by an instrumented application when it terminates by calling the standard `exit()` function.

To ensure that profile information is generated, the functions described in this section may be necessary or useful in the following situations:

- The instrumented application exits using a non-standard exit routine.
- The instrumented application is a non-terminating application: `exit()` is never called.
- The application requires control of when the profile information is generated.

A set of functions and an environment variable comprise the Profile IGS.

#### The Profile IGS Functions

The Profile IGS functions are available to your application by inserting a header file at the top of any source file where the functions may be used.

```
#include "pgouser.h"
```

**Note**
The Profile IGS functions are written in C language. Fortran applications need to call C functions.

The rest of the topics in this section describe the Profile IGS functions.

**Note**
Without instrumentation, the Profile IGS functions cannot provide PGO API support.

#### The Profile IGS Environment Variable

The environment variable for Profile IGS is `PROF_DUMP_INTERVAL`. This environment variable may be used to initiate Interval Profile Dumping in an instrumented user application. For more information, see the recommended usage of `_PGOPTI_Set_Interval_Prof_Dump()`.

### Dumping Profile Information

The `_PGOPTI_Prof_Dump()` function dumps the profile information collected by the instrumented application and has the following prototype:

```
void _PGOPTI_Prof_Dump(void);
```

The profile information is generated in a `.dyn` file (generated in phase 2 of the PGO).

#### Recommended usage

Insert a single call to this function in the body of the function which terminates the user application. Normally, `_PGOPTI_Prof_Dump()` should be called just once.

It is also possible to use this function in conjunction with the `_PGOPTI_Prof_Reset()` function to generate multiple `.dyn` files (presumably from multiple sets of input data).

**Example:**

```
! selectively collect profile information
!  for the portion of the application
!  involved in processing input data

input_data = get_input_data()
do while (input_data)
call _PGOPTI_Prof_Reset()
call process_data(input_data)
call _PGOPTI_Prof_Dump();
```

```
input_data = get_input_data();
end do
```

### Resetting the Dynamic Profile Counters

The `_PGOPTI_Prof_Reset()` function resets the dynamic profile counters and has the following prototype:

```
void _PGOPTI_Prof_Reset(void);
```

#### Recommended usage

Use this function to clear the profile counters prior to collecting profile information on a section of the instrumented application. See the example under `_PGOPTI_Prof_Dump()`.

### Dumping and Resetting Profile Information

The `_PGOPTI_Prof_Dump_And_Reset()` function dumps the profile information to a new `.dyn` file and then resets the dynamic profile counters. Then the execution of the instrumented application continues. The prototype of this function is:

```
void _PGOPTI_Prof_Dump_And_Reset(void);
```

This function is used in non-terminating applications and may be called more than once.

#### Recommended usage

Periodic calls to this function enables a non-terminating application to generate one or more profile information files (`.dyn` files). These files are merged during the feedback phase (phase 3) of profile-guided optimizations. The direct use of this function enables your application to control precisely when the profile information is generated.

### Interval Profile Dumping

The `_PGOPTI_Set_Interval_Prof_Dump()` function activates Interval Profile Dumping and sets the approximate frequency at which dumps occur. The prototype of the function call is:

```
void _PGOPTI_Set_Interval_Prof_Dump(int interval);
```

This function is used in non-terminating applications.

The `interval` parameter specifies the time interval at which profile dumping occurs and is measured in milliseconds. For example, if interval is set to 5000, then a profile dump and reset will occur approximately every 5 seconds. The interval is approximate because the time-check controlling the dump and reset is only performed upon entry to any instrumented function in your application.

**Note**

1. Setting interval to zero or a negative number will disable interval profile dumping.
2. Setting a very small value for interval may cause the instrumented application to spend nearly all of its time dumping profile information. Be sure to set interval to a large enough value so that the application can perform actual work and substantial profile information is collected.

**Recommended usage**

This function may be called at the start of a non-terminating user application, to initiate Interval Profile Dumping. Note that an alternative method of initiating Interval Profile Dumping is by setting the environment variable, `PROF_DUMP_INTERVAL`, to the desired interval value prior to starting the application.

The intention of Interval Profile Dumping is to allow a non-terminating application to be profiled with minimal changes to the application source code.

## High-level Language Optimizations (HLO)

### HLO Overview

High-level optimizations exploit the properties of source code constructs (for example, loops and arrays) in the applications developed in high-level programming languages, such as Fortran and C++. The high-level optimizations include loop interchange, loop fusion, loop unrolling, loop distribution, unroll-and-jam, blocking, data prefetch, scalar replacement, data layout optimizations and loop unrolling techniques.

The option that turns on the high-level optimizations is `-O3`. The scope of optimizations turned on by `-O3` is different for IA-32 and Itanium®-based applications. See Setting Optimization Levels.

#### IA-32 and Itanium®-based Applications

The `-O3` option enables the `-O2` option and adds more aggressive optimizations; for example, loop transformation and prefetching. `-O3` optimizes for maximum speed, but may not improve performance for some programs.

### IA-32 Applications

In conjunction with the vectorization options, `-ax{K|W|N|B|P}` and `-x{K|W|N|B|P}`, the `-O3` option causes the compiler to perform more aggressive data dependency analysis than for default `-O2`. This may result in longer compilation times.

### Itanium-based Applications

The `-ivdep_parallel` option asserts there is no loop-carried dependency in the loop where `IVDEP` directive is specified. This is useful for sparse matrix applications.

### Key Techniques to Tune Your Itanium-based Applications

Follow these steps to tune applications on Itanium-based systems:

1. Compile your program with `-O3` and `-ipo`. Use profile guided optimization whenever possible.
2. Identify hot spots in your code.
3. Turn on Optimization reporting.
4. Check why loops are not software pipelined.

    - Use `CDEC$ ivdep` to tell the compiler there is no dependency. You may also need the option `-ivdep_parallel` to indicate there is no loop carried dependency.
    - Use `CDEC$ swp` to enable software pipelining (useful for lop-sided control and unknown loop count).
    - Use `CDEC$ loop count(n)` when needed.
    - If cray pointers are used, use `-safe_cray_ptr` to indicate there is no aliasing.
    - Use `CDEC$ distribute point` to split large loops (normally, this is automatically done).

5. Check that the prefetch distance is correct. Use `CDEC$ prefetch` to override the distance when it is needed.

## Loop Transformations

The loop transformation techniques include:

- loop normalization

- loop reversal
- loop interchange and permutation
- loop distribution
- loop fusion
- scalar replacement
- absence of loop-carried memory dependency with the IVDEP directive
- runtime data dependencies checking (Itanium®-based systems only)

The loop transformations listed above are supported by data dependence. The loop transformation techniques also include:

- induction variable elimination
- constant propagation
- copy propagation
- forward substitution
- and dead code elimination.

In addition to the loop transformations listed for both IA-32 and Itanium® architectures above, the Itanium architecture enables implementation of the collapsing techniques.

## Scalar Replacement (IA-32 Only)

The goal of scalar replacement is to reduce memory references. This is done mainly by replacing array references with register references.

While the compiler replaces some array references with register references when `-O1` or `-O2` is specified, more aggressive replacement is performed when `-O3` (`-scalar_rep`) is specified. For example, with `-O3` the compiler attempts replacement when there are loop-carried dependences or when data-dependence analysis is required for memory disambiguation.

| | |
|---|---|
| `-scalar_rep[-]` | Enables (default) or disables scalar replacement performed during loop transformations (requires `-O3`). |

## Loop Unrolling with -unroll[n]

The `-unroll[`*n*`]` option is used in the following way:

- `-unroll`*n* specifies the maximum number of times you want to unroll a loop. The following example unrolls a loop at most four times:

  **`ifort -unroll4`** ***`a.f`***

To disable loop unrolling, specify *n* as 0. On IA-32 systems, specifying 0 also disables the vectorizer's unroller, except for the unrolling required to resolve cache line splits penalties. The following example disables loop unrolling:

**`ifort -unroll0`** *a.f*

- `-unroll` (*n* omitted) lets the compiler decide whether to perform unrolling or not. This is the default; the compiler uses default heuristics or defines *n*.
- `-unroll0` (*n* = 0) disables the unroller.

The Itanium® compiler currently uses only *n* = 0; any other value is NOP.

### Benefits and Limitations of Loop Unrolling

The benefits are:

- Unrolling eliminates branches and some of the code.
- Unrolling enables you to aggressively schedule (or pipeline) the loop to hide latencies if you have enough free registers to keep variables live.
- The Intel® Pentium® 4 or Intel® Xeon(TM) processors can correctly predict the exit branch for an inner loop that has 16 or fewer iterations, if that number of iterations is predictable and there are no conditional branches in the loop. Therefore, if the loop body size is not excessive, and the probable number of iterations is known, unroll inner loops for:
  - Pentium 4 or Intel Xeon processor, until they have a maximum of 16 iterations
  - Pentium III or Pentium II processors, until they have a maximum of 4 iterations

The potential cost: excessive unrolling, or unrolling of very large loops can lead to increased code size.

For more information on how to optimize with `-unroll[n]`, refer to the *Intel® Pentium® 4 and Intel® Xeon(TM) Processor Optimization Reference Manual*.

## Memory Dependency with IVDEP Directive

For Itanium®-based applications, the `-ivdep_parallel` option indicates there is absolutely no loop-carried memory dependency in the loop where `IVDEP` directive is specified. This technique is useful for some sparse matrix applications.

For example, the following loop requires `-ivdep_parallel` in addition to the directive IVDEP to ensure there is no loop-carried dependency for the store into `a()`.

```
!DIR$ IVDEP
do j=1,n
a(b(j)) = a(b(j))+1
enddo
```

See also Vectorization Support.

## Prefetching

The goal of `-prefetch` insertion is to reduce cache misses by providing hints to the processor about when data should be loaded into the cache. The prefetching optimizations implement the following options:

| | |
|---|---|
| `-prefetch[-]` | Enables or disables (`-prefetch-`) prefetch insertion. This option requires that `-O3` be specified. The default with `-O3` is `-prefetch`. |

To facilitate compiler optimization:

- Minimize use of global variables and pointers.
- Minimize use of complex control flow.
- Choose data types carefully and avoid type casting.

For more information on how to optimize with `-prefetch[-]`, refer to the *Intel® Pentium® 4 and Intel® Xeon(TM) Processor Optimization Reference Manual*.

In addition to the `-prefetch` option, an intrinsic subroutine, MM_PREFETCH and compiler directive PREFETCH are also available. The subroutine MM_PREFETCH prefetches data from the specified address on one memory cache line. The compiler directive PREFETCH enables a data prefetch from memory.

The following example is for Itanium®-based systems only:

```
do j=1,lastrow-firstrow+1
  i = rowstr(j)
 iresidue = mod( rowstr(j+1)-i, 8 )
 sum = 0.d0
CDEC$ NOPREFETCH a,p,colidx
do k=i,i+iresidue-1
  sum = sum +  a(k)*p(colidx(k))
enddo
CDEC$ NOPREFETCH colidx
CDEC$ PREFETCH a:1:40
CDEC$ PREFETCH p:1:20
 do k=i+iresidue, rowstr(j+1)-8, 8
```

```
  sum = sum + a(k  )*p(colidx(k  ))
&      + a(k+1)*p(colidx(k+1)) + a(k+2)*p(colidx(k+2))
&      + a(k+3)*p(colidx(k+3)) + a(k+4)*p(colidx(k+4))
&      + a(k+5)*p(colidx(k+5)) + a(k+6)*p(colidx(k+6))
&      + a(k+7)*p(colidx(k+7))
  enddo
 q(j) = sum
enddo
```

For details, refer to the *Intel® Fortran Language Reference.*

# Parallel Programming with Intel® Fortran

## Parallelism: an Overview

This section discusses the three major features of parallel programming supported by the Intel® Fortran compiler: OpenMP*, Auto-parallelization, and Auto-vectorization. Each of these features contributes to the application performance depending on the number of processors, target architecture (IA-32 or Itanium® architecture), and the nature of the application. The three features OpenMP, Auto-parallelization and Auto-vectorization, can be combined arbitrarily to contribute to the application performance.

Parallel programming can be *explicit*, that is, defined by a programmer using OpenMP directives. Parallel programming can be *implicit*, that is, detected automatically by the compiler. Implicit parallelism is exploited by either Auto-parallelization of outer-most loops or Auto-vectorization of innermost loops (or both).

Parallelism defined with OpenMP and Auto-parallelization directives is based on thread-level parallelism (TLP). Parallelism defined with Auto-vectorization techniques is based on instruction-level parallelism (ILP).

The Intel Fortran compiler supports OpenMP and Auto-parallelization on both IA-32 and Itanium architectures for multiprocessor systems as well as on single IA-32 processors with Hyper-Threading Technology (for Hyper-Threading Technology, refer to the IA-32 Intel® Architecture Optimization Reference Manual). Auto-vectorization is supported on the families of the Pentium®, Pentium with MMX(TM) technology, Pentium II, Pentium III, and Pentium 4 processors. To enhance the compilation of the code with Auto-vectorization, the users can also add vectorizer directives to their program. A closely related technique that is available on the Itanium-based systems is software pipelining (SWP).

The table below summarizes the different ways in which parallelism can be exploited with the Intel Fortran compiler.

| **Parallelism** | | |
|---|---|---|
| **Explicit** | **Implicit** | |
| Parallelism programmed by the user | Parallelism generated by the compiler and by user-supplied hints | |
| **OpenMP*(TLP)** | **Auto-parallelization (TLP)** | **Auto-vectorization (ILP)** |

| | | |
|---|---|---|
| IA-32 and Itanium architectures | of outer-most loops<br><br>IA-32 and Itanium architectures | of inner-most loops<br><br>IA-32 only<br><br>Software pipelining for Itanium architecture |
| **Supported on:** | | **Supported on:** |
| IA-32 or Itanium-based Multiprocessor systems;<br><br>IA-32 Hyper-Threading Technology-enabled systems. | | Pentium®, Pentium with MMX™ Technology, Pentium II, Pentium III, and Pentium 4 processors |

## Parallel Program Development

The Intel Fortran Compiler supports the OpenMP Fortran version 2.0 API specification available from the www.openmp.org web site. The OpenMP directives relieve the user from having to deal with the low-level details of iteration space partitioning, data sharing, and thread scheduling and synchronization.

The Auto-parallelization feature of the Intel Fortran Compiler automatically translates serial portions of the input program into semantically equivalent multithreaded code. Automatic parallelization determines the loops that are good worksharing candidates, performs the dataflow analysis to verify correct parallel execution, and partitions the data for threaded code generation as is needed in programming with OpenMP directives. The OpenMP and Auto-parallelization applications provide the performance gains from shared memory on multiprocessor systems and IA-32 processors with the Hyper-Threading Technology.

Auto-vectorization detects low-level operations in the program that can be done in parallel, and then converts the sequential program to process 2, 4, 8 or up to 16 elements in one operation, depending on the data type. In some cases auto-parallelization and vectorization can be combined for better performance results. For example, in the code below, TLP can be exploited in the outermost loop, while ILP can be exploited in the innermost loop.

```
DO I = 1, 100            ! execute groups of iterations in
different
! threads (TLP)
DO J = 1, 32          ! execute in SIMD style with
multimedia
! extension (ILP)
A(J,I) = A(J,I) + 1
ENDDO
ENDDO
```

Auto-vectorization can help improve performance of an application that runs on the systems based on Pentium®, Pentium with MMX(TM) technology, Pentium II, Pentium III, and Pentium 4 processors.

The following table lists the options that enable Auto-vectorization, Auto-parallelization, and OpenMP support.

| **Auto-vectorization, IA-32 only** | |
|---|---|
| `-x{K|W|N|B|P}` | Generates specialized code to run exclusively on processors with the extensions specified by `{K|W|N|B|P}`. |
| `-ax{K|W|N|B|P}` | Generates, in a single binary, code specialized to the extensions specified by `{K|W|N|B|P}` and also generic IA-32 code. The generic code is usually slower. |
| `-vec_report{0|1|2|3|4|5}` | Controls the diagnostic messages from the vectorizer, see subsection that follows the table. |
| **Auto-parallelization, IA-32 and Itanium architectures** | |
| `-parallel` | Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel. Default: OFF. |
| `-par_threshold{n}` | Sets a threshold for the auto-parallelization of loops based on the probability of profitable execution of the loop in parallel, n=0 to 100. n=0 implies "always." Default: n=100. |
| `-par_report{0|1|2|3}` | Controls the auto-parallelizer's diagnostic levels. Default: `-par_report1`. |
| **OpenMP, IA-32 and Itanium architectures** | |
| `-openmp` | Enables the parallelizer to generate multithreaded code based on the OpenMP directives. Default: OFF. |
| `-openmp_report{0|1|2}` | Controls the OpenMP parallelizer's diagnostic levels. Default: `/Qopenmp_report1`. |
| `-openmp_stubs` | Enables compilation of OpenMP programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked. Default: OFF. |

**Note**
When both `-openmp` and `-parallel` are specified on the command line, the `-parallel` option is only honored in routines that do not contain

OpenMP Directives. For routines that contain OpenMP directives, only the `-openmp` option is honored.

With the right choice of options, the programmers can:

- Increase the performance of your application with minimum effort
- Use compiler features to develop multithreaded programs faster

With a relatively small effort of adding the OpenMP directives to their code, the programmers can transform a sequential program into a parallel program. The following are examples of the OpenMP directives within the code:

```
!OMP$ PARALLEL PRIVATE(NUM), SHARED (X,A,B,C)
!Defines a parallel region
!OMP$ PARALLEL DO ! Specifies a parallel region that
! implicitly contains a single DO directive
DO I = 1, 1000
NUM = FOO(B(i), C(I))
X(I) = BAR(A(I), NUM)
! Assume FOO and BAR have no side effects
ENDDO
```

See examples of the Auto-parallelization and Auto-vectorization directives in the respective sections.

# Auto-vectorization (IA-32 Only)

## Vectorization Overview

The vectorizer is a component of the Intel® Fortran Compiler that automatically uses SIMD instructions in the MMX(TM), SSE, SSE2 and SSE3 instruction sets. The vectorizer detects operations in the program that can be done in parallel, and then converts the sequential operations like one SIMD instruction that processes 2, 4, 8 or up to 16 elements in parallel, depending on the data type.

This section provides options description, guidelines, and examples for Intel Fortran Compiler vectorization implemented by IA-32 compiler only. For additional information, see Publications on Compiler Optimizations.

The following list summarizes this section contents.

- Descriptions of compiler options to control vectorization
- Vectorization Key programming guidelines
- Discussion and general guidelines on vectorization levels:
    - Automatic vectorization
    - Vectorization with user intervention
- Examples demonstrating typical vectorization issues and resolutions

The Intel Fortran compiler supports a variety of directives that can help the compiler to generate effective vector instructions. See compiler directives supporting vectorization.

## Vectorizer Options

Vectorization is an IA-32-specific feature and can be summarized by the command line options described in the following tables. Vectorization depends upon the compiler's ability to disambiguate memory references. Certain options may enable the compiler to do better vectorization. These options can enable other optimizations in addition to vectorization. When an `-x{K|W|N|B|P}` or `-ax{K|W|N|B|P}` is used and `-O2` (which is ON by default) is also in effect, the vectorizer is enabled. The `-x{K|W|N|B|P}` or `-ax{K|W|N|B|P}` options enable vectorizer with `-O1` and `-O3` options also.

| | |
|---|---|
| `-x{K\|W\|N\|B\|P}` | Generates specialized code to run exclusively on the processors supporting the extensions indicated by `{K\|W\|N\|B\|P}`. See Processor-specific Optimization (IA-32 only) for details. |
| `-ax{K\|W\|N\|B\|P}` | Generates, in a single binary, code specialized to the extensions specified by `{K\|W\|N\|B\|P}` and also generic IA-32 code. The generic code is usually slower. See Automatic Processor-specific Optimization (IA-32 only) for details. |
| `-vec_report {0\|1\|2\|3\|4\|5}` Default: `-vec_report1` | Controls the diagnostic messages from the vectorizer, see the subsection that follows the table. |

### Vectorization Reports

The `-vec_report{0|1|2|3|4|5}` options directs the compiler to generate the vectorization reports with different level of information as follows:

`-vec_report0`: no diagnostic information is displayed

`-vec_report1`: display diagnostics indicating loops successfully vectorized (default)

`-vec_report2`: same as `-vec_report1`, plus diagnostics indicating loops not successfully vectorized

`-vec_report3`: same as `-vec_report2`, plus additional information about any proven or assumed dependences
`-vec_report4`: indicate non-vectorized loops
`-vec_report5`: indicate non-vectorized loops and the reason why they were not vectorized.

If you specify `-vec_report` without a number, the default of `-vec_report1` is used.

**Usage with Other Options**

The vectorization reports are generated in the final compilation phase when executable is generated. Therefore if you use the `-c` option and a `-vec_report{n}` option in the command line, no report will be generated.

If you use `-c`, `-ipo` and `-x{K|W|N|B|P}` or `-ax{K|W|N|B|P}` and `-vec_report{n}`, the compiler issues a warning and no report is generated.

To produce a report when using the above mentioned options, you need to add the `-ipo_obj` option. The combination of `-c` and `-ipo_obj` produces a single file compilation, and hence does generate object code, and eventually a report is generated.

The following commands generate vectorization report:

```
ifort -x{K|W|N|B|P} -vec_report3 file.f

ifort -x{K|W|N|B|P} -ipo -ipo_obj -vec_report3 file.f

ifort -c -x{K|W|N|B|P} -ipo -ipo_obj -vec_report3 file.f
```

## Loop Parallelization and Vectorization

Combining the `-parallel` and `-x{K|W|N|B|P}` options instructs the compiler to attempt both automatic loop parallelization and automatic loop vectorization in the same compilation. In most cases, the compiler will consider outermost loops for parallelization and innermost loops for vectorization. If deemed profitable, however, the compiler may even apply loop parallelization and vectorization to the same loop. See Guidelines for Effective Auto-parallelization Usage and Vectorization Key Programming Guidelines.

Note that in some rare cases successful loop parallelization (either automatically or by means of OpenMP* directives) may affect the messages reported by the compiler for a non-vectorizable loop in a non-intuitive way.

## Vectorization Key Programming Guidelines

The goal of vectorizing compilers is to exploit single-instruction multiple data (SIMD) processing automatically. Users can help however by supplying the compiler with additional information; for example, directives. Review these guidelines and restrictions, see code examples in further topics, and check them against your code to eliminate ambiguities that prevent the compiler from achieving optimal vectorization.

You will often need to make some changes to your loops. Guidelines for loop bodies follow.

**Use**:

- Straight-line code (a single basic block)
- Vector data only; that is, arrays and invariant expressions on the right hand side of assignments. Array references can appear on the left hand side of assignments.
- Only assignment statements

**Avoid**:

- Function calls
- Unvectorizable operations (other than mathematical)
- Mixing vectorizable types in the same loop
- Data-dependent loop exit conditions
- Loop unrolling (compiler does it)
- Decomposing one loop with several statements in the body into several single-statement loops.

There are a number of restrictions that you should be aware of. Vectorization depends on the two major factors:

- Hardware: The compiler is limited by restrictions imposed by the underlying hardware. In the case of Streaming SIMD Extensions, the vector memory operations are limited to `stride-1` accesses with a preference to 16-byte-aligned memory references. This means that if the compiler abstractly recognizes a loop as vectorizable, it still might not vectorize it for a distinct target architecture.
- Style: The style in which you write source code can inhibit optimization. For example, a common problem with global pointers is that they often prevent the compiler from being able to prove that two memory references refer to distinct locations. Consequently, this prevents certain reordering transformations.

Many stylistic issues that prevent automatic vectorization by compilers are found in loop structures. The ambiguity arises from the complexity of the keywords, operators, data references, and memory operations within the loop bodies.

However, by understanding these limitations and by knowing how to interpret diagnostic messages, you can modify your program to overcome the known limitations and enable effective vectorization. The following sections summarize the capabilities and restrictions of the vectorizer with respect to loop structures.

## Data Dependence

Data dependence relations represent the required ordering constraints on the operations in serial loops. Because vectorization rearranges the order in which operations are executed, any auto-vectorizer must have at its disposal some form of data dependence analysis.

An example where data dependencies prohibit vectorization is shown below. In this example, the value of each element of an array is dependent on the value of its neighbor that was computed in the previous iteration.

**Example of Data-dependent Loop:**

```
REAL DATA(0:N)
INTEGER I
DO I=1, N-1
DATA(I) =DATA(I-1)*0.25+DATA(I)*0.5+DATA(I+1)*0.25
END DO
```

The loop in the above example is not vectorizable because the WRITE to the current element `DATA(I)` is dependent on the use of the preceding element `DATA(I-1)`, which has already been written to and changed in the previous iteration. To see this, look at the access patterns of the array for the first two iterations as shown below.

**Example of Data Dependence Vectorization Patterns:**

```
I=1: READ DATA (0)
READ DATA (1)
READ DATA (2)
WRITE DATA (1)
I=2: READ DATA(1)
READ DATA (2)
READ DATA (3)
WRITE DATA (2)
```

In the normal sequential version of this loop, the value of `DATA(1)` read from during the second iteration was written to in the first iteration. For vectorization, it must be possible to do the iterations in parallel, without changing the semantics of the original loop.

**Data Dependence Analysis**

Data dependence analysis involves finding the conditions under which two memory accesses may overlap. Given two references in a program, the conditions are defined by:

- Whether the referenced variables may be aliases for the same (or overlapping) regions in memory, and, for array references
- The relationship between the subscripts

For IA-32, data dependence analyzer for array references is organized as a series of tests, which progressively increase in power as well as in time and space costs. First, a number of simple tests are performed in a dimension-by-dimension manner, since independence in any dimension will exclude any dependence relationship. Multidimensional arrays references that may cross their declared dimension boundaries can be converted to their linearized form before the tests are applied.

Some of the simple tests that can be used are the fast greatest common divisor (GCD) test and the extended bounds test. The GCD test proves independence if the GCD of the coefficients of loop indices cannot evenly divide the constant term. The extended bounds test checks for potential overlap of the extreme values in subscript expressions. If all simple tests fail to prove independence, we eventually resort to a powerful hierarchical dependence solver that uses Fourier-Motzkin elimination to solve the data dependence problem in all dimensions.

## Loop Constructs

Loops can be formed with the usual DO - END DO and DO WHILE, or by using IF/GOTO statements and a label. However, the loops must have a single entry and a single exit to be vectorized. Following are the examples of correct and incorrect usages of loop constructs.

**Example of Correct Usage:**

```
SUBROUTINE FOO (A, B, C)
DIMENSION A(100),B(100), C(100)
INTEGER I
I = 1
DO WHILE (I .LE. 100)
A(I) = B(I) * C(I)
IF (A(I) .LT. 0.0) A(I) = 0.0
I = I + 1
ENDDO
```

```
RETURN
END
```

**Example of Incorrect Usage:**

```
SUBROUTINE FOO (A, B, C)
DIMENSION A(100),B(100), C(100)
INTEGER I
I = 1
DO WHILE (I .LE. 100)
A(I) = B(I) * C(I)
C The next statement allows early
C exit from the loop and prevents
C vectorization of the loop.
IF (A(I) .LT. 0.0) GOTO 10
I = I + 1
ENDDO
10 CONTINUE
RETURN
END
```

## Loop Exit Conditions

Loop exit conditions determine the number of iterations that a loop executes. For example, fixed indexes for loops determine the iterations. The loop iterations must be countable; that is, the number of iterations must be expressed as one of the following:

- A constant
- A loop invariant term
- A linear function of outermost loop indices

Loops whose exit depends on computation are not countable. Examples below show countable and non-countable loop constructs.

**Correct Usage for Countable Loop, Example 1:**

```
SUBROUTINE FOO (A, B, C, N, LB)
DIMENSION A(N),B(N),C(N)
INTEGER N, LB, I, COUNT
! Number of iterations is "N - LB + 1"
COUNT = N
DO WHILE (COUNT .GE. LB)
A(I) = B(I) * C(I)
COUNT = COUNT - 1
I = I + 1
ENDDO  ! LB is not defined within loop
RETURN
END
```

**Correct Usage for Countable Loop, Example 2:**

```
! Number of iterations is (N-M+2) /2
SUBROUTINE FOO (A, B, C, M, N, LB)
DIMENSION A(N),B(N),C(N)
INTEGER I, L, M, N
I = 1;
DO L = M,N,2
A(I) = B(I) * C(I)
I = I + 1
ENDDO
RETURN
END
```

**Example of Incorrect Usage for Non-Countable Loop:**

```
! Number of iterations is dependent on A(I)
SUBROUTINE FOO (A, B, C)
DIMENSION A(100),B(100),C(100)
INTEGER I
I = 1
DO WHILE (A(I) .GT. 0.0)
A(I) = B(I) * C(I)
I = I + 1
ENDDO
RETURN
END
```

## Types of Loop Vectorized

For integer loops, the 64-bit MMX(TM) technology and 128-bit Streaming SIMD Extensions 2 (SSE2) provide SIMD instructions for most arithmetic and logical operators on 32-bit, 16-bit, and 8-bit integer data types. Vectorization may proceed if the final precision of integer wrap-around arithmetic will be preserved. A 32-bit shift-right operator, for instance, is not vectorized in 16-bit mode if the final stored value is a 16-bit integer. Because the MMX(TM) and SSE2 instruction sets are not fully orthogonal (shifts on byte operands, for instance, are not supported), not all integer operations can actually be vectorized.

For loops that operate on 32-bit single-precision and 64-bit double-precision floating-point numbers, SSE/SSE2 provides SIMD instructions for the arithmetic operators '+', '-', '*', and '/'. In addition, SSE/SSE2 provides SIMD instructions for the binary `MIN` and `MAX` and unary `SQRT` operators. SIMD versions of several other mathematical operators (like the trigonometric functions SIN, COS, TAN) are supported in software in a vector mathematical run-time library that is provided with the Intel® Fortran Compiler, of which the compiler takes advantage.

## Strip-mining and Cleanup

Strip-mining, also known as loop sectioning, is a loop transformation technique for enabling SIMD-encodings of loops, as well as providing a means of improving memory performance. By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure in two ways:

- It increases the temporal and spatial locality in the data cache if the data are reusable in different passes of an algorithm.
- It reduces the number of iterations of the loop by a factor of the length of each "vector," or number of operations being performed per SIMD operation. In the case of Streaming SIMD Extensions, this vector or strip-length is reduced by 4 times: four floating-point data items per single Streaming SIMD Extensions single-precision floating-point SIMD operation are processed.

First introduced for vectorizers, this technique consists of the generation of code when each vector operation is done for a size less than or equal to the maximum vector length on a given vector machine.

The compiler automatically strip-mines your loop and generates a cleanup loop.

**Example of Strip Mining and Cleaning Up Loops:**

```
! Before Vectorization

i = 1
do while (i<=n)
a(i) = b(i) + c(i) ! Original loop code
i = i + 1
end do

! After Vectorization

!The vectorizer generates the following two loops
i = 1
do while (i < (n - mod(n,4)))
! Vector strip-mined loop.
a(i:i+3) = b(i:i+3) + c(i:i+3)
i = i + 4
end do
do while (i <= n)
a(i) = b(i) + c(i)     !Scalar clean-up loop
i = i + 1
end do
```

**Loop Blocking**

It is possible to treat loop blocking as strip-mining in two or more dimensions. Loop blocking is a useful technique for memory performance optimization. The main purpose of loop blocking is to eliminate as many cache misses as possible. This technique transforms the memory domain into smaller chunks rather than sequentially traversing through the entire memory domain. Each chunk should be small enough to fit all the data for a given computation into the cache, thereby maximizing data reuse.

Consider the following example. The two-dimensional array A is referenced in the j (column) direction and then in the i (row) direction (column-major order); array B is referenced in the opposite manner (row-major order). Assume the memory layout is in column-major order; therefore, the access strides of array A and B for the code would be 1 and MAX, respectively.

In the B. example: BS = block_size; MAX must be evenly divisible by BS.

**Example of Loop Blocking of Arrays:**

```
A. Original loop
REAL A(MAX,MAX), B(MAX,MAX)
DO I =1, MAX
 DO J = 1, MAX
   A(I,J) = A(I,J) + B(J,I)
 ENDDO
ENDDO
```

```
B.Transformed Loop after blocking

REAL A(MAX,MAX), B(MAX,MAX)
DO I =1, MAX, BS
 DO J = 1, MAX, BS
   DO II = I, I+MAX, BS-1
     DO J = J, J+MAX, BS-1
       A(II,JJ) = A(II,JJ) + B(JJ,II)
     ENDDO
   ENDDO
 ENDDO
ENDDO
```

## Statements in the Loop Body

The vectorizable operations are different for floating point and integer data.

### Floating-point Array Operations

The statements within the loop body may be REAL operations (typically on arrays). Arithmetic operations supported are addition, subtraction, multiplication, division, negation, square root, MAX, MIN, and mathematical functions such as SIN and COS. Note that conversion to/from some types of floats is not valid. Operation on DOUBLE PRECISION types is not valid, unless optimizing for an Intel® Pentium® 4 and Intel® Xeon(TM) processors' system, and Intel® Pentium® M processor, using the `-xW` or `-axW` compiler option.

### Integer Array Operations

The statements within the loop body may be arithmetic or logical operations (again, typically for arrays). Arithmetic operations are limited to such operations as addition, subtraction, ABS, MIN, and MAX. Logical operations include bitwise `AND`, `OR` and `XOR` operators. You can mix data types only if the conversion can be done without a loss of precision. Some example operators where you can mix data types are multiplication, shift, or unary operators.

### Other Operations

No statements other than the preceding floating-point and integer operations are permitted. The loop body cannot contain any function calls other than the ones described above.

## Vectorization Examples

This section contains simple examples of some common issues in vector programming.

### Argument Aliasing: A Vector Copy

The loop in the example of a vector copy operation does not vectorize because the compiler cannot prove that `DEST(A(I))` and `DEST(B(I))` are distinct.

**Example of Unvectorizable Copy Due to Unproven Distinction:**

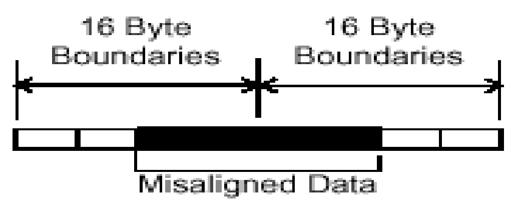
```
SUBROUTINE VEC_COPY(DEST,A,B,LEN)
DIMENSION DEST(*)
INTEGER A(*), B(*)
INTEGER LEN, I
DO I=1,LEN
DEST(A(I)) = DEST(B(I))
END DO
RETURN
END
```

### Data Alignment

A 16-byte or greater data structure or array should be aligned so that the beginning of each structure or array element is aligned in a way that its base address is a multiple of 16.

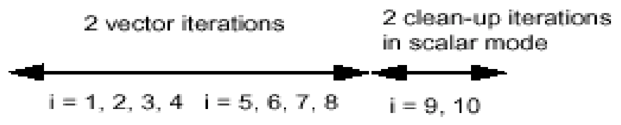
The Misaligned Data Crossing 16-Byte Boundary figure shows the effect of a data cache unit (DCU) split due to misaligned data. The code loads the misaligned data across a 16-byte boundary, which results in an additional memory access causing a six- to twelve-cycle stall. You can avoid the stalls if you know that the data is aligned and you specify to assume alignment

**Misaligned Data Crossing 16-Byte Boundary**



After vectorization, the loop is executed as shown in figure below.

**Vector and Scalar Clean-up Iterations**



Both the vector iterations `A(1:4) = B(1:4)`; and `A(5:8) = B(5:8)`; can be implemented with aligned moves if both the elements `A(1)` and `B(1)` are 16-byte aligned.

**Caution**
If you specify the vectorizer with incorrect alignment options, the compiler will generate code with unexpected behavior. Specifically, using aligned moves on unaligned data, will result in an illegal instruction exception!

### Alignment Strategy

The compiler has at its disposal several alignment strategies in case the alignment of data structures is not known at compile-time. A simple example is shown below (several other strategies are supported as well). If in the loop shown below the alignment of A is unknown, the compiler will generate a prelude loop that iterates until the array reference, that occurs the most, hits an aligned address. This makes the alignment properties of A known, and the vector loop is optimized accordingly. In this case, the vectorizer applies dynamic loop peeling, a specific Intel® Fortran feature.

**Example of Data Alignment:**

**Original loop:**

```
SUBROUTINE DOIT(A)
REAL A(100)      ! alignment of argument A is unknown
DO I = 1, 100
A(I) = A(I) + 1.0
ENDDO
END SUBROUTINE
```

**Aligning Data:**

```
! The vectorizer will apply dynamic loop peeling as follows:
SUBROUTINE DOIT(A)
REAL A(100)
! let P be (A%16)where A is address of A(1)
IF (P .NE. 0) THEN
P = (16 - P) /4    ! determine run-time peeling
           ! factor
DO I = 1, P
A(I) = A(I) + 1.0
ENDDO
ENDIF
! Now this loop starts at a 16-byte boundary,
! and will be vectorized accordingly
DO I = P + 1, 100
A(I) = A(I) + 1.0
ENDDO
END SUBROUTINE
```

## Loop Interchange and Subscripts: Matrix Multiply

Matrix multiplication is commonly written as shown in the following example.

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

The use of `B(K,J)`, is not a `stride-1` reference and therefore will not normally be vectorizable. If the loops are interchanged, however, all the references will become `stride-1` as in the Matrix Multiplication with Stride-1 example that follows.

**Note**
Interchanging is not always possible because of dependencies, which can lead to different results.

**Example of Matrix Multiplication with Stride-1:**

```
DO J=1,N
 DO K=1,N
   DO I=1,N
     C(I,J) = C(I,J) + A(I,K)*B(K,J)
   ENDDO
 ENDDO
ENDDO
```

For additional information, see publications on Compiler Optimizations.

## Auto-parallelization

### Auto-parallelization Overview

The auto-parallelization feature of the Intel® Fortran Compiler automatically translates serial portions of the input program into equivalent multithreaded code. The auto-parallelizer analyzes the dataflow of the program's loops and generates multithreaded code for those loops which can be safely and efficiently executed in parallel. This enables the potential exploitation of the parallel architecture found in symmetric multiprocessor (SMP) systems.

Automatic parallelization relieves the user from:

- Having to deal with the details of finding loops that are good worksharing candidates
- Performing the dataflow analysis to verify correct parallel execution
- Partitioning the data for threaded code generation as is needed in programming with OpenMP* directives.

The parallel run-time support provides the same run-time features as found in OpenMP, such as handling the details of loop iteration modification, thread scheduling, and synchronization.

While OpenMP directives enable serial applications to transform into parallel applications quickly, the programmer must explicitly identify specific portions of the application code that contain parallelism and add the appropriate compiler directives. Auto-parallelization triggered by the `-parallel` option automatically identifies those loop structures, which contain parallelism. During compilation, the compiler automatically attempts to decompose the code sequences into separate threads for parallel processing. No other effort by the programmer is needed.

The following example illustrates how a loop's iteration space can be divided so that it can be executed concurrently on two threads:

```
Original Serial Code
do i=1,100
a(i) = a(i) + b(i) * c(i)
enddo

Transformed Parallel Code
Thread 1
do i=1,50
a(i) = a(i) + b(i) * c(i)
enddo
Thread 2
do i=51,100
a(i) = a(i) + b(i) * c(i)
enddo
```

## Programming with Auto-parallelization

Auto-parallelization feature implements some concepts of OpenMP, such as worksharing construct (with the PARALLEL DO directive). See Programming with OpenMP for worksharing construct. This section provides specifics of auto-parallelization.

### Guidelines for Effective Auto-parallelization Usage

A loop is parallelizable if:

- The loop is countable at compile time: this means that an expression representing how many times the loop will execute (also called "the loop trip count") can be generated just before entering the loop.
- There are no `FLOW` (READ after WRITE), `OUTPUT` (WRITE after WRITE) or `ANTI` (WRITE after READ) loop-carried data dependences. A loop-carried data dependence occurs when the same memory location is referenced in different iterations of the loop. At the compiler's discretion, a loop may be parallelized if any assumed inhibiting loop-carried dependencies can be resolved by run-time dependency testing.

The compiler may generate a run-time test for the profitability of executing in parallel for loop with loop parameters that are not compile-time constants.

#### Coding Guidelines

Enhance the power and effectiveness of the auto-parallelizer by following these coding guidelines:

- Expose the trip count of loops whenever possible; specifically use constants where the trip count is known and save loop parameters in local variables.
- Avoid placing structures inside loop bodies that the compiler may assume to carry dependent data, for example, procedure calls, ambiguous indirect references or global references.
- Insert the !DEC$ PARALLEL directive to disambiguate assumed data dependencies.
- Insert the !DEC$ NOPARALLEL directive before loops known to have insufficient work to justify the overhead of sharing among threads.

### Auto-parallelization Data Flow

For auto-parallelization processing, the compiler performs the following steps:

**Data flow analysis ---> Loop classification ---> Dependence analysis ---> High-level parallelization --> Data partitioning ---> Multi-threaded code generation**.

These steps include:

- Data flow analysis: compute the flow of data through the program
- Loop classification: determine loop candidates for parallelization based on correctness and efficiency as shown by threshold analysis
- Dependence analysis: compute the dependence analysis for references in each loop nest
- High-level parallelization:

  - analyze dependence graph to determine loops which can execute in parallel.

  - compute run-time dependency

- Data partitioning: examine data reference and partition based on the following types of access: SHARED, PRIVATE, and FIRSTPRIVATE
- Multi-threaded code generation:

  - modify loop parameters

  - generate entry/exit per threaded task

  - generate calls to parallel run-time routines for thread creation and synchronization

## Auto-parallelization: Enabling, Options, Directives, and Environment Variables

To enable the auto-parallelizer, use the `-parallel` option. The `-parallel` option detects parallel loops capable of being executed safely in parallel and automatically generates multithreaded code for these loops. An example of the command using auto-parallelization is as follows:

```
ifort -c -parallel myprog.f
```

**Auto-parallelization Options**

The `-parallel` option enables the auto-parallelizer if the `-O2` (or `-O3`) optimization option is also on (the default is `-O2`). The `-parallel` option detects parallel loops capable of being executed safely in parallel and automatically generates multithreaded code for these loops.

| | |
|---|---|
| `-parallel` | Enables the auto-parallelizer |
| `-par_threshold{0-100}` | Controls the work threshold needed for auto-parallelization. |
| `-par_report{1\|2\|3}` | Controls the diagnostic messages from the auto-parallelizer, see later subsection. |

**Auto-parallelization Directives**

Auto-parallelization uses two specific directives, !DEC$ PARALLEL and !DEC$ NOPARALLEL.

The format of an Intel Fortran auto-parallelization compiler directive is:

```
<prefix> <directive>
```

where the brackets above mean:

- `<xxx>:` the prefix and directive are required

For fixed form source input, the prefix is !DEC$ or CDEC$

For free form source input, the prefix is !DEC$ only.

The prefix is followed by the directive name; for example:

!DEC$ PARALLEL

Since auto-parallelization directives begin with an exclamation point, the directives take the form of comments if you omit the `-parallel` option.

### Examples

The !DEC$ PARALLEL directive instructs the compiler to ignore dependencies which it assumes may exist and which would prevent correct parallelization in the immediately following loop. However, if dependencies are proven, they are not ignored.

The !DEC$ NOPARALLEL directive disables auto-parallelization for the immediately following loop.

```
program main
parameter (n=100)
integer x(n),a(n)

!DEC$ NOPARALLEL
do i=1,n
x(i) = i
enddo

!DEC$ PARALLEL
do i=1,n
a( x(i) ) = i
enddo
end
```

### Auto-parallelization Environment Variables

| Option | Description | Default |
|---|---|---|
| `OMP_NUM_THREADS` | Controls the number of threads used. | Number of processors currently installed in the system while generating the executable |
| `OMP_SCHEDULE` | Specifies the type of run-time scheduling. | `static` |

## Auto-parallelization Threshold Control and Diagnostics

### Threshold Control

The `-par_threshold{`*n*`}` option sets a threshold for auto-parallelization of loops based on the probability of profitable execution of the loop in parallel. The value of *n* can be from 0 to 100. The default value is 100. The `-par_threshold{`*n*`}` option should be used when the computation work in loops cannot be determined at compile-time.

The meaning for various values of n is as follows:

- *n* = 100. Parallelization will only proceed when performance gains are predicted based on the compiler analysis data. This is the default. This value is used when `-par_threshold{n}` is not specified on the command line or is used without specifying a value of n.
- *n* = 0, `-par_threshold0` is specified. The loops get auto-parallelized regardless of computation work volume, that is, parallelize always.
- The intermediate 1 to 99 values represent the percentage probability for profitable speed-up. For example, *n*=50 would mean: parallelize only if there is a 50% probability of the code speeding up if executed in parallel.

The compiler applies a heuristic that tries to balance the overhead of creating multiple threads versus the amount of work available to be shared amongst the threads.

**Diagnostics**

The `-par_report{0|1|2|3}` option controls the auto-parallelizer's diagnostic levels 0, 1, 2, or 3 as follows:

`-par_report0` = no diagnostic information is displayed.

`-par_report1` = indicates loops successfully auto-parallelized (default). Issues a "`LOOP AUTO-PARALLELIZED`" message for parallel loops.

`-par_report2` = indicates successfully auto-parallelized loops as well as unsuccessful loops.

`-par_report3` = same as 2 plus additional information about any proven or assumed dependences inhibiting auto-parallelization (reasons for not parallelizing).

The following example shows an output generated by `-par_report3` as a result from the command:

**`ifort -c -parallel -par_report3`** ***`myprog.f90`***

where the program `myprog.f90` is as follows:

```
 program myprog
      integer a(10000), q
C Assumed side effects
      do i=1,10000
         a(i) = foo(i)
      enddo
C Actual dependence
      do i=1,10000
         a(i) = a(i-1) + i
      enddo
      end
```

**Example of `-par_report` Output:**

```
program myprog
procedure: myprog
serial loop: line 5: not a parallel candidate
due to statement at line 6
serial loop: line 9
  flow data dependence from line 10 to line
10, due to "a"
12 Lines Compiled
```

**Troubleshooting Tips**

- Use `-par_threshold0` to see if the compiler assumed there was not enough computational work
- Use `-par_report3` to view diagnostics
- Use the !DIR$ PARALLEL directive to eliminate assumed data dependencies
- Use `-ipo` to eliminate assumed side-effects done to function calls.

## Parallelization with OpenMP*

### Parallelization with OpenMP* Overview

The Intel® Fortran Compiler supports the OpenMP* Fortran version 2.0 API specification, except for the WORKSHARE directive. OpenMP provides symmetric multiprocessing (SMP) with the following major features:

- Relieves the user from having to deal with the low-level details of iteration space partitioning, data sharing, and thread scheduling and synchronization.
- Provides the benefit of the performance available from shared memory, multiprocessor systems; and, for IA-32 systems, from Hyper-Threading Technology-enabled systems (for Hyper-Threading Technology, refer to the IA-32 Intel® Architecture Optimization Reference Manual).

The Intel Fortran Compiler performs transformations to generate multithreaded code based on the user's placement of OpenMP directives in the source program making it easy to add threading to existing software. The Intel compiler supports all of the current industry-standard OpenMP directives, except WORKSHARE , and compiles parallel programs annotated with OpenMP directives.

In addition, the Intel Fortran Compiler provides Intel-specific extensions to the OpenMP Fortran version 2.0 specification including run-time library routines and environment variables.

See parallelization options summary for all options of the OpenMP feature in the Intel Fortran Compiler. For complete information on the OpenMP standard, visit the www.openmp.org web site. For complete Fortran language specifications, see the OpenMP Fortran version 2.0 specifications.

### Parallel Processing with OpenMP

To compile with OpenMP, you need to prepare your program by annotating the code with OpenMP directives in the form of the Fortran program comments. The Intel Fortran Compiler first processes the application and produces a multithreaded version of the code which is then compiled. The output is a Fortran executable with the parallelism implemented by threads that execute parallel regions or constructs. See Programming with OpenMP.

### Performance Analysis

For performance analysis of your program, you can use the VTune(TM) analyzer and/or the Intel® Threading Tools to show performance information. You can obtain detailed information about which portions of the code that require the largest amount of time to execute and where parallel performance problems are located.

## Programming with OpenMP

The Intel® Fortran Compiler accepts a Fortran program containing OpenMP directives as input and produces a multithreaded version of the code. When the parallel program begins execution, a single thread exists. This thread is called the master thread. The master thread will continue to process serially until it encounters a parallel region.

### Parallel Region

A parallel region is a block of code that must be executed by a team of threads in parallel. In the OpenMP Fortran API, a parallel construct is defined by placing OpenMP directives PARALLEL at the beginning and END PARALLEL at the end of the code segment. Code segments thus bounded can be executed in parallel.

A structured block of code is a collection of one or more executable statements with a single point of entry at the top and a single point of exit at the bottom.

The Intel Fortran Compiler supports worksharing and synchronization constructs. Each of these constructs consists of one or two specific OpenMP directives and sometimes the enclosed or following structured block of code. For complete definitions of constructs, see the OpenMP Fortran version 2.0 specifications.

At the end of the parallel region, threads wait until all team members have arrived. The team is logically disbanded (but may be reused in the next parallel region), and the master thread continues serial execution until it encounters the next parallel region.

**Worksharing Construct**

A worksharing construct divides the execution of the enclosed code region among the members of the team created on entering the enclosing parallel region. When the master thread enters a parallel region, a team of threads is formed. Starting from the beginning of the parallel region, code is replicated (executed by all team members) until a worksharing construct is encountered. A worksharing construct divides the execution of the enclosed code among the members of the team that encounter it.

The OpenMP SECTIONS or DO constructs are defined as worksharing constructs because they distribute the enclosed work among the threads of the current team. A worksharing construct is only distributed if it is encountered during dynamic execution of a parallel region. If the worksharing construct occurs lexically inside of the parallel region, then it is always executed by distributing the work among the team members. If the worksharing construct is not lexically (explicitly) enclosed by a parallel region (that is, it is `orphaned`), then the worksharing construct will be distributed among the team members of the closest dynamically-enclosing parallel region, if one exists. Otherwise, it will be executed serially.

When a thread reaches the end of a worksharing construct, it may wait until all team members within that construct have completed their work. When all of the work defined by the worksharing construct is finished, the team exits the worksharing construct and continues executing the code that follows.

A combined parallel/worksharing construct denotes a parallel region that contains only one worksharing construct.

**Parallel Processing Directive Groups**

The parallel processing directives include the following groups:

**Parallel Region**

- PARALLEL and END PARALLEL

**Worksharing Construct**

- The DO and END DO directives specify parallel execution of loop iterations.

- The SECTIONS and END SECTIONS directives specify parallel execution for arbitrary blocks of sequential code. Each SECTION is executed once by a thread in the team.

- The SINGLE and END SINGLE directives define a section of code where exactly one thread is allowed to execute the code; threads not chosen to execute this section ignore the code.

**Combined Parallel/Worksharing Constructs**

The combined parallel/worksharing constructs provide an abbreviated way to specify a parallel region that contains a single worksharing construct. The combined parallel/worksharing constructs are:

- PARALLEL DO and END PARALLEL DO
- PARALLEL SECTIONS and END PARALLEL SECTIONS

**Synchronization and MASTER**

Synchronization is the interthread communication that ensures the consistency of shared data and coordinates parallel execution among threads. Shared data is consistent within a team of threads when all threads obtain the identical value when the data is accessed. A synchronization construct is used to insure this consistency of the shared data.

- The OpenMP synchronization directives are CRITICAL, ORDERED, ATOMIC, FLUSH, and BARRIER.
  - Within a parallel region or a worksharing construct only one thread at a time is allowed to execute the code within a CRITICAL construct.
  - The ORDERED directive is used in conjunction with a DO or SECTIONS construct to impose a serial order on the execution of a section of code.
  - The ATOMIC directive is used to update a memory location in an uninterruptable fashion.
  - The FLUSH directive is used to insure that all threads in a team have a consistent view of memory.
  - A BARRIER directive forces all team members to gather at a particular point in code. Each team member that executes a BARRIER waits at the BARRIER until all of the team members have arrived. A BARRIER cannot be used within worksharing or other synchronization constructs due to the potential for deadlock.
- The MASTER directive is used to force execution by the master thread.

See the list of OpenMP Directives and Clauses.

## Data Sharing

Data sharing is specified at the start of a parallel region or worksharing construct by using the SHARED and PRIVATE clauses. All variables in the SHARED clause are shared among the members of a team. It is the application's responsibility to:

- Synchronize access to these variables. All variables in the PRIVATE clause are private to each team member. For the entire parallel region, assuming `t` team members, there are `t+1` copies of all the variables in the PRIVATE clause: one global copy that is active outside parallel regions and a PRIVATE copy for each team member.
- Initialize PRIVATE variables at the start of a parallel region, unless the FIRSTPRIVATE clause is specified. In this case, the PRIVATE copy is initialized from the global copy at the start of the construct at which the FIRSTPRIVATE clause is specified.
- Update the global copy of a PRIVATE variable at the end of a parallel region. However, the LASTPRIVATE clause of a DO directive enables updating the global copy from the team member that executed serially the last iteration of the loop.

In addition to `shared` and PRIVATE variables, individual variables and entire COMMON blocks can be privatized using the THREADPRIVATE directive.

## Orphaned Directives

OpenMP contains a feature called orphaning which dramatically increases the expressiveness of parallel directives. Orphaning is a situation when directives related to a parallel region are not required to occur lexically within a single program unit. Directives such as CRITICAL, BARRIER, SECTIONS, SINGLE, MASTER and DO, can occur by themselves in a program unit, dynamically "binding" to the enclosing parallel region at run time.

Orphaned directives enable parallelism to be inserted into existing code with a minimum of code restructuring. Orphaning can also improve performance by enabling a single parallel region to bind with multiple do directives located within called subroutines. Consider the following code segment:

```
...
!$omp parallel
call phase1
call phase2
!$omp end parallel
...

subroutine phase1
!$omp do private(i) shared(n)
do i = 1, n
```

```
call some_work(i)
end do
!$omp end do
end

subroutine phase2
!$omp do private(j) shared(n)
do j = 1, n
call more_work(j)
end do
!$omp end do
end
```

The following orphaned directives usage rules apply

- An orphaned worksharing construct (SECTIONS, SINGLE, DO) is executed by a team consisting of one thread, that is, serially.
- Any collective operation (worksharing construct or BARRIER) executed inside of a worksharing construct is illegal.
- It is illegal to execute a collective operation (worksharing construct or BARRIER) from within a synchronization region (CRITICAL/ORDERED).
- The opening and closing directives of a directive pair (for example, DO and END DO) must occur in a single block of the program.
- Private scoping of a variable can be specified at a worksharing construct. Shared scoping must be specified at the parallel region. For complete details, see the OpenMP Fortran version 2.0 specifications.

**Preparing Code for OpenMP Processing**

The following are the major stages and steps of preparing your code for using OpenMP. Typically, the first two stages can be done on uniprocessor or multiprocessor systems; later stages are typically done only on multiprocessor systems.

**Before Inserting OpenMP Directives**

Before inserting any OpenMP parallel directives, verify that your code is safe for parallel execution by doing the following:

- Place local variables on the stack. This is the default behavior of the Intel Fortran Compiler when `-openmp` is used.
- Use `-automatic` (or `-auto_scalar`) to make the locals automatic. This is the default behavior of the Intel Fortran Compiler when `-openmp` is used. Avoid using the `-save` option, which inhibits stack allocation of local variables. By default, automatic local scalar variables become shared across threads, so you may need to add synchronization code to ensure proper access by threads.

### Analyze

Analysis includes the following major actions:

- Profile the program to find out where it spends most of its time. This is the part of the program that benefits most from parallelization efforts. This stage can be accomplished using VTune(TM) analyzer or basic PGO options.
- Wherever the program contains nested loops, choose the outer-most loop, which has very few cross-iteration dependencies.

### Restructure

To restructure your program for successful OpenMP implementation, you can perform some or all of the following actions:

1. If a chosen loop is able to execute iterations in parallel, introduce a PARALLEL DO construct around this loop.
2. Try to remove any cross-iteration dependencies by rewriting the algorithm.
3. Synchronize the remaining cross-iteration dependencies by placing CRITICAL constructs around the uses and assignments to variables involved in the dependencies.
4. List the variables that are present in the loop within appropriate SHARED, PRIVATE, LASTPRIVATE, FIRSTPRIVATE, or REDUCTION clauses.

5. List the DO index of the parallel loop as PRIVATE. This step is optional.
6. COMMON block elements must not be placed on the PRIVATE list if their global scope is to be preserved. The THREADPRIVATE directive can be used to privatize to each thread the `common` block containing those variables with global scope. THREADPRIVATE creates a copy of the COMMON block for each of the threads in the team.
7. Any I/O in the parallel region should be synchronized.
8. Identify more parallel loops and restructure them.
9. If possible, merge adjacent PARALLEL DO constructs into a single parallel region containing multiple DO directives to reduce execution overhead.

### Tune

The tuning process should include minimizing the sequential code in critical sections and load balancing by using the SCHEDULE clause or the `omp_schedule` environment variable.

**Note**
This step is typically performed on a multiprocessor system.

## Parallel Processing Thread Model

This topic explains the processing of the parallelized program and adds more definitions of the terms used in the parallel programming.

### The Execution Flow

A program containing OpenMP Fortran API compiler directives begins execution as a single process, called the *master* thread of execution. The master thread executes sequentially until the first *parallel construct* is encountered.

In OpenMP Fortran API, the PARALLEL and END PARALLEL directives define the parallel construct. When the master thread encounters a parallel construct, it creates a *team* of threads, with the master thread becoming the master of the team. The program statements enclosed by the parallel construct are executed in parallel by each thread in the team. These statements include routines called from within the enclosed statements.

The statements enclosed lexically within a construct define the *static extent* of the construct. The *dynamic extent* includes the static extent as well as the routines called from within the construct. When the END PARALLEL directive is encountered, the threads in the team synchronize at that point, the team is dissolved, and only the master thread continues execution. The other threads in the team enter a wait state.

You can specify any number of parallel constructs in a single program. As a result, thread teams can be created and dissolved many times during program execution.

#### Using Orphaned Directives

In routines called from within parallel constructs, you can also use directives. Directives that are not in the lexical extent of the parallel construct, but are in the dynamic extent, are called orphaned directives. Orphaned directives allow you to execute major portions of your program in parallel with only minimal changes to the sequential version of the program. Using this functionality, you can code parallel constructs at the top levels of your program call tree and use directives to control execution in any of the called routines. For example:

```
subroutine F
...
```

```
!$OMP parallel...
...
call G
...
subroutine G
...
!$OMP DO...
...
```

The !$OMP DO is an orphaned directive because the parallel region it will execute in is not lexically present in G.

#### Data Environment Directive

A data environment directive controls the data environment during the execution of parallel constructs.

You can control the data environment within parallel and worksharing constructs. Using directives and data environment clauses on directives, you can:

- Privatize named common blocks by using THREADPRIVATE directive
- Control data scope attributes by using the THREADPRIVATE directive's clauses.

  The data scope attribute clauses are:

  - COPYIN
  - DEFAULT
  - PRIVATE
  - FIRSTPRIVATE
  - LASTPRIVATE
  - REDUCTION
  - SHARED

You can use several directive clauses to control the data scope attributes of variables for the duration of the construct in which you specify them. If you do not specify a data scope attribute clause on a directive, the default is SHARED for those variables affected by the directive.

For detailed descriptions of the clauses, see the OpenMP Fortran version 2.0 specifications.

### Pseudo Code of the Parallel Processing Model

A sample program using some of the more common OpenMP directives is shown in the code example that follows. This example also indicates the difference between serial regions and parallel regions.

```
PROGRAM MAIN               ! Begin serial execution
...                        ! Only the master thread executes
!$OMP PARALLEL             ! Begin a Parallel construct, form a
team
...                        ! This is Replicated Code where each
team
                           !  member executes the same code
!$OMP SECTIONS             ! Begin a Worksharing construct
!$OMP SECTION              ! One unit of work
...                        !
!$OMP SECTION              ! Another unit of work
...                        !
!$OMP END SECTIONS         ! Wait until both units of work
complete
...                        ! More Replicated Code
  !$OMP DO                 ! Begin a Worksharing construct,
      DO                   !  each iteration is a unit of work
      ...                  ! Work is distributed among the team
      END DO               !
!$OMP END DO NOWAIT        ! End of Worksharing construct,
NOWAIT
                           !  is specified (threads need not
wait
                           !  until all work is completed before
                           !  proceeding)
...                        ! More Replicated Code
!$OMP END PARALLEL         ! End of PARALLEL construct, disband
team
                           !  and continue with serial execution
...                        ! Possibly more PARALLEL Constructs
END PROGRAM MAIN           ! End serial execution
```

## Compiling with OpenMP, Directive Format, and Diagnostics

To run the Intel® Fortran Compiler in OpenMP* mode, you need to invoke the Intel compiler with the
-openmp option:

**`ifort -openmp`** ***`input_file(s)`***

Before you run the multithreaded code, you can set the number of desired threads to the OpenMP environment variable, OMP_NUM_THREADS. See the OpenMP Environment Variables section for further information. The Intel Extensjon Routines topic describes the OpenMP extensions to the specification that have been added by Intel in the Intel® Fortran Compiler.

### -openmp Option

The `-openmp` option enables the parallelizer to generate multithreaded code based on the OpenMP directives. The code can be executed in parallel on both uniprocessor and multiprocessor systems.

The `-openmp` option works with both `-O0` (no optimization) and any optimization level of `-O1`,
`-O2` (default) and `-O3`. Specifying `-O0` with `-openmp` helps to debug OpenMP applications.

When you use the `-openmp` option, the compiler sets the `-auto` option (causes all variables to be allocated on the stack, rather than in local static storage.) for the compiler unless you specified it on the command line.

## OpenMP Directive Format and Syntax

The OpenMP directives use the following format:

```
<prefix> <directive> [<clause> [[,] <clause> . . .]]
```

where the brackets above mean:

- `<xxx>:` the prefix and directive are required
- `[<xxx>]:` if a directive  uses one clause or more, the clause(s) is required
- `[,]:` commas between the `<clause>`s are optional.

For fixed form source input, the prefix is `!$omp` or `c$omp`

For free form source input, the prefix is `!$omp` only.

The prefix is followed by the directive name; for example:

```
!$omp parallel
```

Since OpenMP directives begin with an exclamation point, the directives take the form of comments if you omit the `-openmp` option.

### Syntax for Parallel Regions in the Source Code

The OpenMP constructs defining a parallel region have one of the following syntax forms:

```
!$omp <directive>
<structured block of code>
!$omp end <directive>
```

or

```
!$omp <directive>
<structured block of code>
```

or

```
!$omp <directive>
```

where `<directive>` is the name of a particular OpenMP directive.

### OpenMP Diagnostic Reports

The `-openmp_report{0|1|2}` option controls the OpenMP parallelizer's diagnostic levels `0`, `1`, or `2` as follows:

`-openmp_report0` = no diagnostic information is displayed.

`-openmp_report1` = display diagnostics indicating loops, regions, and sections successfully parallelized.

`-openmp_report2` = same as `-openmp_report1` plus diagnostics indicating MASTER constructs, SINGLE constructs, CRITICAL constructs, ORDERED constructs, ATOMIC directives, etc. successfully handled.

The default is `-openmp_report1`.

## OpenMP Directives and Clauses Summary

This topic provides a summary of the OpenMP directives and clauses. For detailed descriptions, see the OpenMP Fortran version 2.0 specifications.

### OpenMP Directives

| Directive | Description |
|---|---|
| PARALLEL<br>END PARALLEL | Defines a parallel region. |
| DO<br>END DO | Identifies an iterative worksharing construct in which the iterations of the associated loop should be executed in parallel. |
| SECTIONS<br>END SECTIONS | Identifies a non-iterative worksharing construct that specifies a set of structured blocks that are to be divided among threads in a team. |
| SECTION | Indicates that the associated structured block should be executed in parallel as part of the enclosing sections construct. |

| | |
|---|---|
| SINGLE<br>END SINGLE | Identifies a construct that specifies that the associated structured block is executed by only one thread in the team. |
| PARALLEL DO<br>END PARALLEL DO | A shortcut for a parallel region that contains a single DO directive.<br><br>**Note**<br>The PARALLEL DO or DO OpenMP directive must be immediately followed by a DO statement (`do-stmt` as defined by R818 of the ANSI Fortran standard). If you place another statement or an OpenMP directive between the PARALLEL DO or DO directive and the DO statement, the Intel Fortran Compiler issues a syntax error. |
| PARALLEL SECTIONS<br>END PARALLEL SECTIONS | Provides a shortcut form for specifying a parallel region containing a single SECTIONS construct. |
| MASTER<br>END MASTER | Identifies a construct that specifies a structured block that is executed by only the MASTER thread of the team. |
| CRITICAL[*lock*]<br>END CRITICAL[*lock*] | Identifies a construct that restricts execution of the associated structured block to a single thread at a time. Each thread waits at the beginning of the critical construct until no other thread is executing a critical construct with the same *lock* argument. |
| BARRIER | Synchronizes all the threads in a team. Each thread waits until all of the other threads in that team have reached this point. |
| ATOMIC | Ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneously writing threads. |
| FLUSH [(*list*)] | Specifies a "cross-thread" sequence point at which the implementation is required to ensure that all the threads in a team have a consistent view of certain objects in memory. The optional *list* argument consists of a comma-separated list of variables to be flushed. |
| ORDERED<br>END ORDERED | The structured block following an ORDERED directive is executed in the order in which iterations would be executed in a sequential loop. |

| | |
|---|---|
| THREADPRIVATE (*list*) | Makes the named COMMON blocks or variables private to a thread. The *list* argument consists of a comma-separated list of COMMON blocks or variables. |

**OpenMP Clauses**

| Clause | Description |
|---|---|
| PRIVATE (*list*) | Declares variables in *list* to be PRIVATE to each thread in a team. |
| FIRSTPRIVATE (*list*) | Same as PRIVATE, but the copy of each variable in the *list* is initialized using the value of the original variable existing before the construct. |
| LASTPRIVATE (*list*) | Same as PRIVATE, but the original variables in *list* are updated using the values assigned to the corresponding PRIVATE variables in the last iteration in the DO construct loop or the last SECTION construct. |
| COPYPRIVATE (*list*) | Uses private variables in *list* to broadcast values, or pointers to shared objects, from one member of a team to the other members at the end of a single construct. |
| NOWAIT | Specifies that threads need not wait at the end of worksharing constructs until they have completed execution. The threads may proceed past the end of the worksharing constructs as soon as there is no more work available for them to execute. |
| SHARED (*list*) | Shares variables in *list* among all the threads in a team. |
| DEFAULT (*mode*) | Determines the default data-scope attributes of variables not explicitly specified by another clause. Possible values for *mode* are PRIVATE, SHARED, or NONE. |
| REDUCTION | Performs a reduction on variables |

| | |
|---|---|
| ({*operator*\|*intrinsic*}:*list*) | that appear in *list* with the operator `operator` or the intrinsic procedure name `intrinsic`; `operator` is one of the following: +, *, `.and.`, `.or.`, `.eqv.`, `.neqv.`; `intrinsic` refers to one of the following: MAX, MIN, IAND, IOR, or IEOR. |
| ORDERED<br>END ORDERED | Used in conjunction with a DO or SECTIONS construct to impose a serial order on the execution of a section of code. If ORDERED constructs are contained in the dynamic extent of the DO construct, the ordered clause must be present on the DO directive. |
| IF (*scalar_logical_expression*) | The enclosed parallel region is executed in parallel only if the *scalar_logical_expression* evaluates to `.TRUE.`; otherwise the parallel region is serialized. |
| NUM_THREADS (*scalar_integer_expression*) | Requests the number of threads specified by *scalar_integer_expression* for the parallel region. |
| SCHEDULE (*type*[,*chunk*]) | Specifies how iterations of the DO construct are divided among the threads of the team. Possible values for the *type* argument are STATIC, DYNAMIC, GUIDED, and RUNTIME. The optional *chunk* argument must be a positive scalar integer expression. |
| COPYIN (*list*) | Specifies that the master thread's data values be copied to the THREADPRIVATE's copies of the common blocks or variables specified in *list* at the beginning of the parallel region. |

### Directives and Clauses Cross-reference

| **Directive** | **Uses These Clauses** |
|---|---|
| PARALLEL | COPYIN, DEFAULT, PRIVATE, |

| | |
|---|---|
| END PARALLEL | FIRSTPRIVATE, REDUCTION, SHARED |
| DO<br>END DO | PRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION, SCHEDULE |
| SECTIONS<br>END SECTIONS | PRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION |
| SECTION | PRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION |
| SINGLE<br>END SINGLE | PRIVATE, FIRSTPRIVATE |
| PARALLEL DO<br>END PARALLEL DO | COPYIN, DEFAULT, PRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION, SHARED, SCHEDULE |
| PARALLEL SECTIONS<br>END PARALLEL SECTIONS | COPYIN, DEFAULT, PRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION, SHARED |
| MASTER<br>END MASTER | None |
| CRITICAL[*lock*]<br>END CRITICAL[*lock*] | None |
| BARRIER | None |
| ATOMIC | None |
| FLUSH[(*list*)] | None |
| ORDERED<br>END ORDERED | None |
| THREADPRIVATE (*list*) | None |

## OpenMP Directive Descriptions

### Parallel Region Directives

The PARALLEL and END PARALLEL directives define a parallel region as follows:

```
!$OMP PARALLEL
! parallel region
!$OMP END PARALLEL
```

When a thread encounters a parallel region, it creates a team of threads and becomes the master of the team. You can control the number of threads in a team by the use of an environment variable or a run-time library call, or both.

The PARALLEL directive takes an optional comma-separated list of clauses. Clauses include:

- IF: whether the statements in the parallel region are executed in parallel by a team of threads or serially by a single thread.
- PRIVATE, FIRSTPRIVATE, SHARED, or REDUCTION: variable types
- DEFAULT: variable data scope attribute
- COPYIN: master thread common block values are copied to THREADPRIVATE copies of the common block

**Changing the Number of Threads**

Once created, the number of threads in the team remains constant for the duration of that parallel region. To explicitly change the number of threads used in the next parallel region, call the `OMP_SET_NUM_THREADS` run-time library routine from a serial portion of the program. This routine overrides any value you may have set using the `OMP_NUM_THREADS` environment variable.

Assuming you have used the `OMP_NUM_THREADS` environment variable to set the number of threads to 6, you can change the number of threads between parallel regions as follows:

```
   CALL OMP_SET_NUM_THREADS(3)
!$OMP PARALLEL
.
.
.
!$OMP END PARALLEL
CALL OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL DO
.
.
.
!$OMP END PARALLEL DO
```

**Setting Units of Work**

Use the worksharing directives such as DO, SECTIONS, and SINGLE to divide the statements in the parallel region into units of work and to distribute those units so that each unit is executed by one thread.

In the following example, the !$OMP DO and !$OMP END DO directives and all the statements enclosed by them comprise the static extent of the parallel region:

```
!$OMP PARALLEL
!$OMP DO
DO I=1,N
B(I) = (A(I) + A(I-1))/ 2.0
END DO
```

```
!$OMP END DO
!$OMP END PARALLEL
```

In the following example, the !$OMP DO and !$OMP END DO directives and all the statements enclosed by them, including all statements contained in the WORK subroutine, comprise the dynamic extent of the parallel region:

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
DO I=1,N
CALL WORK(I,N)
END DO
!$OMP END DO
!$OMP END PARALLEL
```

**Setting Conditional Parallel Region Execution**

When an IF clause is present on the PARALLEL directive, the enclosed code region is executed in parallel only if the scalar logical expression evaluates to `.TRUE.`. Otherwise, the parallel region is serialized. When there is no IF clause, the region is executed in parallel by default.

In the following example, the statements enclosed within the !$OMP DO and !$OMP END DO directives are executed in parallel only if there are more than three processors available. Otherwise the statements are executed serially:

```
!$OMP PARALLEL IF (OMP_GET_NUM_PROCS() .GT. 3)
!$OMP DO
DO I=1,N
Y(I) = SQRT(Z(I))
END DO
!$OMP END DO
!$OMP END PARALLEL
```

If a thread executing a parallel region encounters another parallel region, it creates a new team and becomes the master of that new team. By default, nested parallel regions are always executed by a team of one thread.

**Note**
To achieve better performance than sequential execution, a parallel region must contain one or more worksharing constructs so that the team of threads can execute work in parallel. It is the contained worksharing constructs that lead to the performance enhancements offered by parallel processing.

**Worksharing Construct Directives**

A worksharing construct must be enclosed dynamically within a parallel region if the worksharing directive is to execute in parallel. No new threads are launched and there is no implied barrier on entry to a worksharing construct.

The worksharing constructs are:

- DO and END DO directives
- SECTIONS, SECTION, and END SECTIONS directives
- SINGLE and END SINGLE directives

#### DO and END DO

The DO directive specifies that the iterations of the immediately following DO loop must be dispatched across the team of threads so that each iteration is executed by a single thread. The loop that follows a DO directive cannot be a DO WHILE or a DO loop that does not have loop control. The iterations of the DO loop are dispatched among the existing team of threads.

The DO directive optionally lets you:

- Control data scope attributes (see Controlling Data Scope Attributes)
- Use the SCHEDULE clause to specify schedule type and chunk size (see Specifying Schedule Type and Chunk Size)

#### Clauses Used

The clauses for DO directive specify:

- Whether variables PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION
- How loop iterations are SCHEDULEd onto threads
- In addition, the ORDERED clause must be specified if the ORDERED directive appears in the dynamic extent of the DO directive.
- If you do not specify the optional NOWAIT clause on the END DO directive, threads synchronize at the END DO directive. If you specify NOWAIT, threads do not synchronize, and threads that finish early proceed directly to the instructions following the END DO directive.

#### Usage Rules

- You cannot use a GOTO statement, or any other statement, to transfer control onto or out of the DO construct.
- If you specify the optional END DO directive, it must appear immediately after the end of the DO loop. If you do not specify the END DO directive, an END DO directive is assumed at the end of the DO loop, and threads synchronize at that point.

- The loop iteration variable is private by default, so it is not necessary to declare it explicitly.

**SECTIONS, SECTION and END SECTIONS**

Use the noniterative worksharing SECTIONS directive to divide the enclosed sections of code among the team. Each section is executed just one time by one thread.

Each section should be preceded with a SECTION directive, except for the first section, in which the SECTION directive is optional. The SECTION directive must appear within the lexical extent of the SECTIONS and END SECTIONS directives.

The last section ends at the END SECTIONS directive. When a thread completes its section and there are no undispatched sections, it waits at the END SECTIONS directive unless you specify NOWAIT.

The SECTIONS directive takes an optional comma-separated list of clauses that specifies which variables are PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION.

The following example shows how to use the SECTIONS and SECTION directives to execute subroutines `X_AXIS`, `Y_AXIS`, and `Z_AXIS` in parallel. The first SECTION directive is optional:

```
!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION
CALL X_AXIS
!$OMP SECTION
CALL Y_AXIS
!$OMP SECTION
CALL Z_AXIS
!$OMP END SECTIONS
!$OMP END PARALLEL
```

**SINGLE and END SINGLE**

Use the SINGLE directive when you want just one thread of the team to execute the enclosed block of code.

Threads that are not executing the SINGLE directive wait at the END SINGLE directive unless you specify NOWAIT.

The SINGLE directive takes an optional comma-separated list of clauses that specifies which variables are PRIVATE or FIRSTPRIVATE.

When the END SINGLE directive is encountered, an implicit barrier is erected and threads wait until all threads have finished. This can be overridden by using the NOWAIT option.

In the following example, the first thread that encounters the SINGLE directive executes subroutines `OUTPUT` and `INPUT`:

```
!$OMP PARALLEL DEFAULT(SHARED)
CALL WORK(X)
!$OMP BARRIER
!$OMP SINGLE
CALL OUTPUT(X)
CALL INPUT(Y)
!$OMP END SINGLE
CALL WORK(Y)
!$OMP END PARALLEL
```

## Combined Parallel/Worksharing Constructs

The combined parallel/worksharing constructs provide an abbreviated way to specify a parallel region that contains a single worksharing construct. The combined parallel/worksharing constructs are:

- PARALLEL DO
- PARALLEL SECTIONS

### PARALLEL DO and END PARALLEL DO

Use the PARALLEL DO directive to specify a parallel region that implicitly contains a single DO directive.

You can specify one or more of the clauses for the PARALLEL and the DO directives.

The following example shows how to parallelize a simple loop. The loop iteration variable is private by default, so it is not necessary to declare it explicitly. The END PARALLEL DO directive is optional:

```
!$OMP PARALLEL DO
 DO I=1,N
   B(I) = (A(I) + A(I-1)) / 2.0
 END DO
!$OMP END PARALLEL DO
```

### PARALLEL SECTIONS and END PARALLEL SECTIONS

Use the PARALLEL SECTIONS directive to specify a parallel region that implicitly contains a single SECTIONS directive.

You can specify one or more of the clauses for the PARALLEL and the SECTIONS directives.

The last section ends at the END PARALLEL SECTIONS directive.

In the following example, subroutines `X_AXIS`, `Y_AXIS`, and `Z_AXIS` can be executed concurrently. The first SECTION directive is optional. Note that all SECTION directives must appear in the lexical extent of the PARALLEL SECTIONS/END PARALLEL SECTIONS construct:

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
 CALL X_AXIS
!$OMP SECTION
 CALL Y_AXIS
!$OMP SECTION
 CALL Z_AXIS
!$OMP END PARALLEL SECTIONS
```

## Synchronization Constructs

Synchronization constructs are used to ensure the consistency of shared data and to coordinate parallel execution among threads.

The synchronization constructs are:

- ATOMIC directive
- BARRIER directive
- CRITICAL directive
- FLUSH directive
- MASTER directive
- ORDERED directive

### ATOMIC Directive

Use the ATOMIC directive to ensure that a specific memory location is updated atomically instead of exposing the location to the possibility of multiple, simultaneously writing threads.

This directive applies only to the immediately following statement, which must have one of the following forms:

`x =` *`x operator expr`*

`x =` *`expr operator x`*

`x =` *`intrinsic (x, expr)`*

```
x = intrinsic (expr, x)
```

In the preceding statements:

- *x* is a scalar variable of intrinsic type
- `expr` is a scalar expression that does not reference x
- `intrinsic` is either MAX, MIN, IAND, IOR, or IEOR
- `operator` is either +, *, -, /, .AND., .OR., .EQV., or .NEQV.

This directive permits optimization beyond that of a critical section around the assignment. An implementation can replace all ATOMIC directives by enclosing the statement in a critical section. All of these critical sections must use the same unique name.

Only the load and store of *x* are atomic; the evaluation of `expr` is not atomic. To avoid race conditions, all updates of the location in parallel must be protected by using the ATOMIC directive, except those that are known to be free of race conditions. The function `intrinsic`, the operator `operator`, and the assignment must be the intrinsic function, operator, and assignment.

This restriction applies to the ATOMIC directive: All references to storage location *x* must have the same type parameters.

In the following example, the collection of Y locations is updated atomically:

```
!$OMP ATOMIC
Y = Y + B(I)
```

**BARRIER Directive**

To synchronize all threads within a parallel region, use the BARRIER directive. You can use this directive only within a parallel region defined by using the PARALLEL directive. You cannot use the BARRIER directive within the DO, PARALLEL DO, SECTIONS, PARALLEL SECTIONS, and SINGLE directives.

When encountered, each thread waits at the BARRIER directive until all threads have reached the directive.

In the following example, the BARRIER directive ensures that all threads have executed the first loop and that it is safe to execute the second loop:

```
c$OMP PARALLEL
c$OMP DO PRIVATE(i)
DO i = 1, 100
b(i) = i
END DO
c$OMP BARRIER
c$OMP DO PRIVATE(i)
```

```
DO i = 1, 100
a(i) = b(101-i)
END DO
c$OMP END PARALLEL
```

### CRITICAL and END CRITICAL

Use the CRITICAL and END CRITICAL directives to restrict access to a block of code, referred to as a critical section, to one thread at a time.

A thread waits at the beginning of a critical section until no other thread in the team is executing a critical section having the same name.

When a thread enters the critical section, a latch variable is set to closed and all other threads are locked out. When the thread exits the critical section at the END CRITICAL directive, the latch variable is set to open, allowing another thread access to the critical section.

If you specify a critical section name in the CRITICAL directive, you must specify the same name in the END CRITICAL directive. If you do not specify a name for the CRITICAL directive, you cannot specify a name for the END CRITICAL directive.

All unnamed CRITICAL directives map to the same name. Critical section names are global to the program.

The following example includes several CRITICAL directives, and illustrates a queuing model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the same task, the dequeuing operation must be in a critical section. Because there are two independent queues in this example, each queue is protected by CRITICAL directives having different names, X_AXIS and Y_AXIS, respectively:

```
!$OMP PARALLEL DEFAULT(PRIVATE,SHARED(X,Y)
!$OMP CRITICAL(X_AXIS)
CALL DEQUEUE(IX_NEXT, X)
!$OMP END CRITICAL(X_AXIS)
CALL WORK(IX_NEXT, X)
!$OMP CRITICAL(Y_AXIS)
CALL DEQUEUE(IY_NEXT,Y)
!$OMP END CRITICAL(Y_AXIS)
CALL WORK(IY_NEXT, Y)
!$OMP END PARALLEL
```

Unnamed critical sections use the global lock from the Pthread package. This allows you to synchronize with other code by using the same lock. Named locks are created and maintained by the compiler and can be significantly more efficient.

**FLUSH Directive**

Use the FLUSH directive to identify a synchronization point at which a consistent view of memory is provided. Thread-visible variables are written back to memory at this point.

To avoid flushing all thread-visible variables at this point, include a list of comma-separated named variables to be flushed.

The following example uses the FLUSH directive for point-to-point synchronization between thread 0 and thread 1 for the variable `ISYNC`:

```
!$OMP PARALLEL DEFAULT(PRIVATE),SHARED(ISYNC)
IAM = OMP_GET_THREAD_NUM()
ISYNC(IAM) = 0
!$OMP BARRIER
CALL WORK()
! I Am Done With My Work, Synchronize With My Neighbor
ISYNC(IAM) = 1
!$OMP FLUSH(ISYNC)
! Wait Till Neighbor Is Done
DO WHILE (ISYNC(NEIGH) .EQ. 0)
!$OMP FLUSH(ISYNC)
END DO
!$OMP END PARALLEL
```

**MASTER and END MASTER**

Use the MASTER and END MASTER directives to identify a block of code that is executed only by the master thread.

The other threads of the team skip the code and continue execution. There is no implied barrier at the END MASTER directive.

In the following example, only the master thread executes the routines `OUTPUT` and `INPUT`:

```
!$OMP PARALLEL DEFAULT(SHARED)
CALL WORK(X)
!$OMP MASTER
CALL OUTPUT(X)
CALL INPUT(Y)
!$OMP END MASTER
CALL WORK(Y)
!$OMP END PARALLEL
```

**ORDERED and END ORDERED**

Use the ORDERED and END ORDERED directives within a DO construct to allow work within an ordered section to execute sequentially while allowing work outside the section to execute in parallel.

When you use the ORDERED directive, you must also specify the ORDERED clause on the DO directive.

Only one thread at a time is allowed to enter the ordered section, and then only in the order of loop iterations.

In the following example, the code prints out the indexes in sequential order:

```
!$OMP DO ORDERED,SCHEDULE(DYNAMIC)
DO I=LB,UB,ST
CALL WORK(I)
END DO
SUBROUTINE WORK(K)
!$OMP ORDERED
WRITE(*,*) K
!$OMP END ORDERED
```

**THREADPRIVATE Directive**

You can make named common blocks private to a thread, but global within the thread, by using the THREADPRIVATE directive.

Each thread gets its own copy of the common block with the result that data written to the common block by one thread is not directly visible to other threads. During serial portions and MASTER sections of the program, accesses are to the master thread copy of the common block.

You cannot use a thread private common block or its constituent variables in any clause other than the COPYIN clause.

In the following example, common blocks `BLK1` and `FIELDS` are specified as thread private:

```
COMMON /BLK1/ SCRATCH
 COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
!$OMP THREADPRIVATE(/BLK1/,/FIELDS/)
```

## OpenMP Clause Descriptions

### Controlling Data Scope

#### Data Scope Attribute Clauses Overview

You can use several directive clauses to control the data scope attributes of variables for the duration of the construct in which you specify them. If you do not specify a data scope attribute clause on a directive, the default is SHARED for those variables affected by the directive.

Each of the data scope attribute clauses accepts a list, which is a comma-separated list of named variables or named common blocks that are accessible in the scoping unit. When you specify named common blocks, they must appear between slashes ( /*name*/ ).

Not all of the clauses are allowed on all directives, but the directives to which each clause applies are listed in the clause descriptions.

The data scope attribute clauses are:

- COPYIN
- DEFAULT
- PRIVATE
- FIRSTPRIVATE
- LASTPRIVATE
- REDUCTION
- SHARED

**COPYIN Clause**

Use the COPYIN clause on the PARALLEL, PARALLEL DO, and PARALLEL SECTIONS directives to copy the data in the master thread common block to the thread private copies of the common block. The copy occurs at the beginning of the parallel region. The COPYIN clause applies only to common blocks that have been declared THREADPRIVATE.

You do not have to specify a whole common block to be copied in; you can specify named variables that appear in the THREADPRIVATE common block. In the following example, the common blocks `BLK1` and `FIELDS` are specified as thread private, but only one of the variables in common block `FIELDS` is specified to be copied in:

```
  COMMON /BLK1/ SCRATCH
 COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
!$OMP THREADPRIVATE(/BLK1/, /FIELDS/)
!$OMP PARALLEL DEFAULT(PRIVATE),COPYIN(/BLK1/,ZFIELD)
```

**DEFAULT Clause**

Use the DEFAULT clause on the PARALLEL, PARALLEL DO, and PARALLEL SECTIONS directives to specify a default data scope attribute for all variables within the lexical extent of a parallel region. Variables in THREADPRIVATE common blocks are not affected by this clause. You can specify only one DEFAULT clause on a directive. The default data scope attribute can be one of the following:

- PRIVATE

  Makes all named objects in the lexical extent of the parallel region private to a thread. The objects include common block variables, but exclude THREADPRIVATE variables.

- SHARED

  Makes all named objects in the lexical extent of the parallel region shared among all the threads in the team.

- NONE

  Declares that there is no implicit default as to whether variables are PRIVATE or SHARED. You must explicitly specify the scope attribute for each variable in the lexical extent of the parallel region.

If you do not specify the DEFAULT clause, the default is DEFAULT(SHARED). However, loop control variables are always PRIVATE by default.

You can exempt variables from the default data scope attribute by using other scope attribute clauses on the parallel region as shown in the following example:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE),
FIRSTPRIVATE(I),SHARED(X),
!$OMP& SHARED(R) LASTPRIVATE(I)
```

**PRIVATE, FIRSTPRIVATE, and LASTPRIVATE Clauses**

**PRIVATE**

Use the PRIVATE clause on the PARALLEL, DO, SECTIONS, SINGLE, PARALLEL DO, and PARALLEL SECTIONS directives to declare variables to be private to each thread in the team.

The behavior of variables declared PRIVATE is as follows:

- A new object of the same type and size is declared once for each thread in the team, and the new object is no longer storage associated with the original object.
- All references to the original object in the lexical extent of the directive construct are replaced with references to the private object.

- Variables defined as PRIVATE are undefined for each thread on entering the construct, and the corresponding shared variable is undefined on exit from a parallel construct.
- Contents, allocation state, and association status of variables defined as PRIVATE are undefined when they are referenced outside the lexical extent, but inside the dynamic extent, of the construct unless they are passed as actual arguments to called routines.

In the following example, the values of I and J are undefined on exit from the parallel region:

```
INTEGER I,J
 I =1
 J =2
!$OMP PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
 I =3
 J =J+ 2
!$OMP END PARALLEL
 PRINT *, I, J
```

#### FIRSTPRIVATE

Use the FIRSTPRIVATE clause on the PARALLEL , DO, SECTIONS, SINGLE, PARALLEL DO, and PARALLEL SECTIONS directives to provide a superset of the PRIVATE clause functionality.

In addition to the PRIVATE clause functionality, private copies of the variables are initialized from the original object existing before the parallel construct.

#### LASTPRIVATE

Use the LASTPRIVATE clause on the DO, SECTIONS, PARALLEL DO, and PARALLEL SECTIONS directives to provide a superset of the PRIVATE clause functionality.

When the LASTPRIVATE clause appears on a DO or PARALLEL DO directive, the thread that executes the sequentially last iteration updates the version of the object it had before the construct.

When the LASTPRIVATE clause appears on a SECTIONS or PARALLEL SECTIONS directive, the thread that executes the lexically last section updates the version of the object it had before the construct.

Subobjects that are not assigned a value by the last iteration of the DO loop or the lexically last SECTION directive are undefined after the construct.

Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. You must list all such variables as arguments to a

LASTPRIVATE clause so that the values of the variables are the same as when the loop is executed sequentially. As shown in the following example, the value of I at the end of the parallel region is equal to `N+1`, as it would be with sequential execution.

```
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(I)
DO I=1,N
  A(I) = B(I) + C(I)
END DO
!$OMP END PARALLEL
CALL REVERSE(I)
```

**REDUCTION Clause**

Use the REDUCTION clause on the PARALLEL, DO, SECTIONS, PARALLEL DO, and PARALLEL SECTIONS directives to perform a reduction on the specified variables by using an operator or intrinsic as shown:

```
    REDUCTION (

    operator

      or

    intrinsic

    :list )
```

*`Operator`* can be one of the following: `+`, `*`, `-`, `.AND.`, `.OR.`, `.EQV.`, or `.NEQV.`.

*`Intrinsic`* can be one of the following: MAX, MIN, IAND, IOR, or IEOR.

The specified variables must be named scalar variables of intrinsic type and must be SHARED in the enclosing context. A private copy of each specified variable is created for each thread as if you had used the PRIVATE clause. The private copy is initialized to a value that depends on the operator or intrinsic as shown in the following table. The actual initialization value is consistent with the data type of the reduction variable.

**Operators/Intrinsics and Initialization Values for Reduction Variables**

| Operator/Intrinsic | Initialization Value |
|---|---|
| + | 0 |
| * | 1 |

| | |
|---|---|
| `-` | 0 |
| `.AND.` | `.TRUE.` |
| `.OR.` | `.FALSE.` |
| `.EQV.` | `.TRUE.` |
| `.NEQV.` | `.FALSE.` |
| MAX | Largest representable number |
| MIN | Smallest representable number |
| IAND | All bits on |
| IOR | 0 |
| IEOR | 0 |

At the end of the construct to which the reduction applies, the shared variable is updated to reflect the result of combining the original value of the SHARED reduction variable with the final value of each of the private copies using the specified operator.

Except for subtraction, all of the reduction operators are associative and the compiler can freely reassociate the computation of the final value. The partial results of a subtraction reduction are added to form the final value.

The value of the shared variable becomes undefined when the first thread reaches the clause containing the reduction, and it remains undefined until the reduction computation is complete. Normally, the computation is complete at the end of the REDUCTION construct. However, if you use the REDUCTION clause on a construct to which NOWAIT is also applied, the shared variable remains undefined until a barrier synchronization has been performed. This ensures that all of the threads have completed the REDUCTION clause.

The REDUCTION clause is intended to be used on a region or worksharing construct in which the reduction variable is used only in reduction statements having one of the following forms:

```
x = x operator expr
x = expr operator x (except for subtraction)
x = intrinsic (x,expr)
x = intrinsic (expr, x)
```

Some reductions can be expressed in other forms. For instance, a MAX reduction might be expressed as follows:

```
IF (x .LT. expr) x = expr
```

Alternatively, the reduction might be hidden inside a subroutine call. Be careful that the operator specified in the REDUCTION clause matches the reduction operation.

Any number of reduction clauses can be specified on the directive, but a variable can appear only once in a REDUCTION clause for that directive as shown in the following example:

```
!$OMP DO REDUCTION(+: A, Y),REDUCTION(.OR.: AM)
```

The following example shows how to use the REDUCTION clause:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE),SHARED(A,B,REDUCTION(+:
A,B)
DO I=1,N
CALL WORK(ALOCAL,BLOCAL)
A = A + ALOCAL
B = B + BLOCAL
END DO
!$OMP END PARALLEL DO
```

#### SHARED Clause

Use the SHARED clause on the PARALLEL, PARALLEL DO, and PARALLEL SECTIONS directives to make variables shared among all the threads in a team.

In the following example, the variables `X` and `NPOINTS` are shared among all the threads in the team:

```
!$OMP PARALLEL DEFAULT(PRIVATE),SHARED(X,NPOINTS)
IAM = OMP_GET_THREAD_NUM()
NP = OMP_GET_NUM_THREADS()
IPOINTS = NPOINTS/NP
CALL SUBDOMAIN(X,IAM,IPOINTS)
!$OMP END PARALLEL
```

### Specifying Schedule Type and Chunk Size

The SCHEDULE clause of the DO or PARALLEL DO directive specifies a scheduling algorithm that determines how iterations of the `DO` loop are divided among and dispatched to the threads of the team. The SCHEDULE clause applies only to the current DO or PARALLEL DO directive.

Within the SCHEDULE clause, you must specify a *schedule type* and, optionally, a *chunk size*. A *chunk* is a contiguous group of iterations dispatched to a thread. Chunk size must be a scalar integer expression.

The following list describes the schedule types and how the chunk size affects scheduling:

- STATIC

  The iterations are divided into pieces having a size specified by chunk. The pieces are statically dispatched to threads in the team in a round-robin manner in the order of thread number.

  When chunk is not specified, the iterations are first divided into contiguous pieces by dividing the number of iterations by the number of threads in the team. Each piece is then dispatched to a thread before loop execution begins.

- DYNAMIC

  The iterations are divided into pieces having a size specified by chunk. As each thread finishes its currently dispatched piece of the iteration space, the next piece is dynamically dispatched to the thread.

  When no chunk is specified, the default is 1.

- GUIDED

  The chunk size is decreased exponentially with each succeeding dispatch. Chunk specifies the minimum number of iterations to dispatch each time. If there are less than chunk number of iterations remaining, the rest are dispatched.

  When no chunk is specified, the default is 1.

- RUNTIME

  The decision regarding scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by using the `OMP_SCHEDULE` environment variable.

  When you specify RUNTIME, you cannot specify a chunk size.

The following list shows which schedule type is used, in priority order:

1. The schedule type specified in the SCHEDULE clause of the current DO or PARALLEL DO directive
2. If the schedule type for the current DO or PARALLEL DO directive is RUNTIME, the default value specified in the `OMP_SCHEDULE` environment variable

3. The compiler default schedule type of STATIC

The following list shows which chunk size is used, in priority order:

1. The chunk size specified in the SCHEDULE clause of the current DO or PARALLEL DO directive
2. For RUNTIME schedule type, the value specified in the `OMP_SCHEDULE` environment variable
3. For DYNAMIC and GUIDED schedule types, the default value 1
4. If the schedule type for the current DO or PARALLEL DO directive is STATIC, the loop iteration space divided by the number of threads in the team.

## OpenMP Support Libraries

The Intel Fortran Compiler with OpenMP* support provides a production support library, `libguide.a`. This library enables you to run an application under different execution modes. It is used for normal or performance-critical runs on applications that have already been tuned.

### Execution modes

The compiler with OpenMP enables you to run an application under different execution modes that can be specified at run time. The libraries support the serial, turnaround, and throughput modes. These modes are selected by using the `kmp_library` environment variable at run time.

#### Turnaround

In a multi-user environment where the load on the parallel machine is not constant or where the job stream is not predictable, it may be better to design and tune for throughput. This minimizes the total time to run multiple jobs simultaneously. In this mode, the worker threads will yield to other threads while waiting for more parallel work.

The throughput mode is designed to make the program aware of its environment (that is, the system load) and to adjust its resource usage to produce efficient execution in a dynamic environment. This mode is the default.

After completing the execution of a parallel region, threads wait for new parallel work to become available. After a certain period of time has elapsed, they stop waiting and sleep. Sleeping allows the threads to be used, until more parallel work becomes available, by non-OpenMP threaded code that may execute between parallel regions, or by other applications. The amount of time to wait before sleeping is set either by the `KMP_BLOCKTIME` environment variable or by the `kmp_set_blocktime()` function. A small `KMP_BLOCKTIME` value may offer better overall performance if your application contains non-OpenMP threaded code that executes between parallel regions. A larger KMP_BLOCKTIME value may be more appropriate if threads are to be reserved solely for use for OpenMP execution, but may penalize other concurrently-running OpenMP or threaded applications.

**Throughput**

In a dedicated (batch or single user) parallel environment where all processors are exclusively allocated to the program for its entire run, it is most important to effectively utilize all of the processors all of the time. The turnaround mode is designed to keep active all of the processors involved in the parallel computation in order to minimize the execution time of a single job. In this mode, the worker threads actively wait for more parallel work, without yielding to other threads.

**Note**
Avoid over-allocating system resources. This occurs if either too many threads have been specified, or if too few processors are available at run time. If system resources are over-allocated, this mode will cause poor performance. The throughput mode should be used instead if this occurs.

## OpenMP Environment Variables

This topic describes the standard OpenMP* environment variables (with the OMP_ prefix) and Intel-specific environment variables (with the KMP_ prefix) that are Intel extensions to the standard Fortran Compiler .

### Standard Environment Variables

| Variable | Description | Default |
|---|---|---|
| OMP_SCHEDULE | Sets the run-time schedule type and chunk size. | STATIC, no chunk size specified |
| OMP_NUM_THREADS | Sets the number of threads to use during execution. | Number of processors |
| OMP_DYNAMIC | Enables (true) or disables (false) the dynamic adjustment of the number of threads. | false |
| OMP_NESTED | Enables (true) or disables (false)nested parallelism. | false |

### Intel Extension Environment Variables

| **Environment Variable** | **Description** | **Default** |
|---|---|---|
| `KMP_ALL_THREADS` | Sets the maximum number of threads that can be used by any parallel region. | `max(32, 4 * OMP_NUM_THREADS, 4 * number of processors)` |
| `KMP_BLOCKTIME` | Sets the time, in milliseconds, that a thread should wait, after completing the execution of a parallel region, before sleeping.<br><br>See also the throughput execution mode and the KMP_LIBRARY environment variable. Use the optional character suffix s, m, h, or d, to specify seconds, minutes, hours, or days. | 200 milliseconds |
| `KMP_LIBRARY` | Selects the OpenMP run-time library throughput. The options for the variable value are: `serial`, `turnaround`, or `throughput` indicating the execution mode. The default value of `throughput` is used if this variable is not | `throughput` (execution mode) |

| | | |
|---|---|---|
| | specified. | |
| `KMP_MONITOR_STACKSIZE` | Sets the number of bytes to allocate for the monitor thread, which is used for book-keeping during program execution. Use the optional suffix b, k, `m`, g, or `t`, to specify bytes, kilobytes, megabytes, gigabytes, or terabytes. | `max(32k, system minimum thread stack size)` |
| `KMP_STACKSIZE` | Sets the number of bytes to allocate for each parallel thread to use as its private stack. Use the optional suffix b, `k`, `m`, g, or `t`, to specify bytes, kilobytes, megabytes, gigabytes, or terabytes. | IA-32: `2m`<br>Itanium compiler: `4m` |
| `KMP_VERSION` | Enables (set) or disables (unset) the printing of OpenMP run-time library version information during program execution. | Disabled |

## OpenMP Run-time Library Routines

OpenMP provides several run-time library routines to assist you in managing your program in parallel mode. Many of these run-time library routines have corresponding environment variables that can be set as defaults. The run-time library routines enable you to dynamically change these factors to assist in controlling your program. In all cases, a call to a run-time library routine overrides any corresponding environment variable.

The following table specifies the interface to these routines. The names for the routines are in user name space. The `omp_lib.f`, `omp_lib.h` and `omp_lib.mod` header files are provided in the INCLUDE directory of your compiler installation. The `omp_lib.h` header file is provided in the INCLUDE directory of your compiler installation for use with the Fortran INCLUDE statement. The `omp_lib.mod` file is provided in the INCLUDE directory for use with the Fortran USE statement.

There are definitions for two different locks, `omp_lock_t` and `omp_nest_lock_t`, which are used by the functions in the table that follows.

This topic provides a summary of the OpenMP run-time library routines. For detailed descriptions, see the OpenMP Fortran version 2.0 specifications.

| Function | Description |
|---|---|
| **Execution Environment Routines** | |
| `subroutine omp_set_num_threads(`*`num_threads`*`)` `integer` *`num_threads`* | Sets the number of threads to use for subsequent parallel regions. |
| `integer function omp_get_num_threads()` | Returns the number of threads that are being used in the current parallel region. |
| `integer function omp_get_max_threads()` | Returns the maximum number of threads that are available for parallel execution. |
| `integer function omp_get_thread_num()` | Determines the unique thread number of the thread currently executing this section of code. |
| `integer function omp_get_num_procs()` | Determines the number of processors available to the program. |

| | |
|---|---|
| `logical function omp_in_parallel()` | Returns `.TRUE.` if called within the dynamic extent of a parallel region executing in parallel; otherwise returns `.FALSE.`. |
| `subroutine omp_set_dynamic(`*`dynamic_threads`*`)`<br>`logical` *`dynamic_threads`* | Enables or disables dynamic adjustment of the number of threads used to execute a parallel region. If d*`ynamic_threads`* is `.TRUE.`, dynamic threads are enabled. If *`dynamic_threads`* is `.FALSE.`, dynamic threads are disabled. Dynamics threads are disabled by default. |
| `logical function omp_get_dynamic()` | Returns `.TRUE.` if dynamic thread adjustment is enabled, otherwise returns `.FALSE.`. |
| `subroutine omp_set_nested(`*`nested`*`)`<br>`integer` *`nested`* | Enables or disables nested parallelism. If *`nested`* is `.TRUE.`, nested parallelism is enabled. If *`nested`* is `.FALSE.`, nested parallelism is disabled. Nested parallelism is disabled by default. |
| `logical function omp_get_nested()` | Returns `.TRUE.` if nested parallelism is enabled, otherwise returns `.FALSE.`. |
| **Lock Routines** | |
| `subroutine omp_init_lock(`*`lock`*`)`<br>`integer (kind=omp_lock_kind)::`*`lock`* | Initializes the lock associated with *`lock`* for use in subsequent calls. |
| `subroutine omp_destroy_lock(`*`lock`*`)`<br>`integer (kind=omp_lock_kind)::`*`lock`* | Causes the lock associated with *`lock`* to become undefined. |
| `subroutine omp_set_lock(`*`lock`*`)` | Forces the executing |

| | |
|---|---|
| `integer (kind=omp_lock_kind)::`*`lock`* | thread to wait until the lock associated with *`lock`* is available. The thread is granted ownership of the lock when it becomes available. |
| `subroutine omp_unset_lock(`*`lock`*`)`<br>`integer (kind=omp_lock_kind)::`*`lock`* | Releases the executing thread from ownership of the lock associated with *`lock`*. The behavior is undefined if the executing thread does not own the lock associated with *`lock`*. |
| `logical omp_test_lock(`*`lock`*`)`<br>`integer (kind=omp_lock_kind)::`*`lock`* | Attempts to set the lock associated with *`lock`*. If successful, returns `.TRUE.`, otherwise returns `.FALSE.`. |
| `subroutine omp_init_nest_lock(lock)`<br>`integer(kind=omp_nest_lock_kind)::`*`lock`* | Initializes the nested lock associated with *`lock`* for use in the subsequent calls. |
| `subroutine omp_destroy_nest_lock(`*`lock`*`)`<br>`integer(kind=omp_nest_lock_kind)::`*`lock`* | Causes the nested lock associated with *`lock`* to become undefined. |
| `subroutine omp_set_nest_lock(lock)`<br>`integer(kind=omp_nest_lock_kind)::`*`lock`* | Forces the executing thread to wait until the nested lock associated with *`lock`* is available. The thread is granted ownership of the nested lock when it becomes available. |
| `subroutine omp_unset_nest_lock(lock)`<br>`integer(kind=omp_nest_lock_kind)::`*`lock`* | Releases the executing thread from ownership of the nested lock associated with *`lock`* if the nesting count is zero. Behavior is undefined if the executing thread does not own the nested lock associated with *`lock`*. |

| | |
|---|---|
| `integer omp_test_nest_lock(`*`lock`*`)`<br>`integer(kind=omp_nest_lock_kind)::`*`lock`* | Attempts to set the nested lock associated with *`lock`*. If successful, returns the nesting count, otherwise returns zero. |
| **Timing Routines** | |
| `double-precision function omp_get_wtime()` | Returns a double-precision value equal to the elapsed wallclock time (in seconds) relative to an arbitrary reference time. The reference time does not change during program execution. |
| `double-precision function omp_get_wtick()` | Returns a double-precision value equal to the number of seconds between successive clock ticks. |

## Intel Extension Routines

The Intel® Fortran Compiler implements the following group of routines as an extension to the OpenMP run-time library: getting and setting stack size for parallel threads and memory allocation.

The Intel extension routines described in this section can be used for low-level debugging to verify that the library code and application are functioning as intended. It is recommended to use these routines with caution because using them requires the use of the `-openmp_stubs` command-line option to execute the program sequentially. These routines are also generally not recognized by other vendor's OpenMP-compliant compilers, which may cause the link stage to fail for these other compilers.

### Stack Size

In most cases, environment variables can be used in place of the extension library routines. For example, the stack size of the parallel threads may be set using the `KMP_STACKSIZE` environment variable rather than the `kmp_set_stacksize()` library routine.

**Note**
A run-time call to an Intel extension routine takes precedence over the corresponding environment variable setting.

The routines `kmp_set_stacksize()` and `kmp_get_stacksize()` take a 32-bit argument only. The routines `kmp_set_stacksize_s()` and `kmp_get_stacksize_s()` take a `size_t` argument, which can hold 64-bit integers.

On Itanium-based systems, it is recommended to always use `kmp_set_stacksize()` and `kmp_get_stacksize()`. These `_s()` variants must be used if you need to set a stack size ≥ 2**32 bytes (4 gigabytes).

See the definitions of stack size routines in the table that follows.

**Memory Allocation**

The Intel® Fortran Compiler implements a group of memory allocation routines as an extension to the OpenMP* run-time library to enable threads to allocate memory from a heap local to each thread. These routines are: `kmp_malloc`, `kmp_calloc`, and `kmp_realloc`.

The memory allocated by these routines must also be freed by the `kmp_free` routine. While it is legal for the memory to be allocated by one thread and `kmp_free`'d by a different thread, this mode of operation has a slight performance penalty.

See the definitions of these routines in the table that follows.

| **Function/Routine** | **Description** |
|---|---|
| **Stack Size** | |
| `function kmp_get_stacksize_s()`<br>`integer(kind=kmp_size_t_kind)kmp_get_stacksize_s` | Returns the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can be changed via the `kmp_get_stacksize_s` routine, prior to the first parallel region or via the `KMP_STACKSIZE` environment variable. |
| `function kmp_get_stacksize()`<br>`integer kmp_get_stacksize` | This routine is provided for backwards compatibility only; use `kmp_get_stacksize_s` routine for compatibility across different families of Intel processors. |
| `subroutine kmp_set_stacksize_s(`*`size`*`)`<br>`integer (kind=kmp_size_t_kind)` *`size`* | Sets to *`size`* the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can also |

| | |
|---|---|
| | be set via the `KMP_STACKSIZE` environment variable. In order for `kmp_set_stacksize_s` to have an effect, it must be called before the beginning of the first (dynamically executed) parallel region in the program. |
| `subroutine kmp_set_stacksize(`*`size`*`)`<br>`integer` *`size`* | This routine is provided for backward compatibility only; use `kmp_set_stacksize_s(`*`size`*`)` for compatibility across different families of Intel processors. |
| **Memory Allocation** | |
| `function kmp_malloc(`*`size`*`)`<br>`integer(kind=kmp_pointer_kind)kmp_malloc`<br>`integer(kind=kmp_size_t_kind)`*`size`* | Allocate memory block of *`size`* bytes from thread-local heap. |
| `function kmp_calloc(`*`nelem`*`,`*`elsize`*`)`<br>`integer(kind=kmp_pointer_kind)kmp_calloc`<br>`integer(kind=kmp_size_t_kind)`*`nelem`*<br>`integer(kind=kmp_size_t_kind)`*`elsize`* | Allocate array of *`nelem`* elements of size *`elsize`* from thread-local heap. |
| `function kmp_realloc(`*`ptr`*`, `*`size`*`)`<br>`integer(kind=kmp_pointer_kind)kmp_realloc`<br>`integer(kind=kmp_pointer_kind)`*`ptr`*<br>`integer(kind=kmp_size_t_kind)`*`size`* | Reallocate memory block at address *`ptr`* and *`size`* bytes from thread-local heap. |
| `subroutine kmp_free(`*`ptr`*`)`<br>`integer (kind=kmp_pointer_kind)` *`ptr`* | Free memory block at address *`ptr`* from thread-local heap. Memory must have been previously allocated with `kmp_malloc`, `kmp_calloc`, or `kmp_realloc`. |

## Examples of OpenMP Usage

The following examples show how to use the OpenMP feature. See more examples in the OpenMP Fortran version 2.0 specifications.

### DO: A Simple Difference Operator

This example shows a simple parallel loop where each iteration contains a different number of instructions. To get good load balancing, dynamic scheduling is used. The END DO has a NOWAIT because there is an implicit BARRIER at the end of the parallel region.

```
       subroutine do_1 (a,b,n)
real a(n,n), b(n,n)
```

```
c$omp parallel
c$omp&   shared(a,b,n)
c$omp&   private(i,j)
c$omp do schedule(dynamic,1)
do i = 2, n
do j = 1, i
b(j,i) = ( a(j,i) + a(j,i-1) ) / 2
enddo
enddo
c$omp end do nowait
c$omp end parallel
end
```

**DO: Two Difference Operators**

This example shows two parallel regions fused to reduce `fork/join` overhead. The first END DO has a NOWAIT because all the data used in the second loop is different than all the data used in the first loop.

```
       subroutine do_2 (a,b,c,d,m,n)
real a(n,n), b(n,n), c(m,m), d(m,m)
c$omp parallel
c$omp&   shared(a,b,c,d,m,n)
c$omp&   private(i,j)
c$omp do schedule(dynamic,1)
do i = 2, n
do j = 1, i
b(j,i) = ( a(j,i) + a(j,i-1) ) / 2
enddo
enddo
c$omp end do nowait
c$omp do schedule(dynamic,1)
do i = 2, m
do j = 1, i
d(j,i) = ( c(j,i) + c(j,i-1) ) / 2
enddo
enddo
c$omp end do nowait
c$omp end parallel
end
```

**SECTIONS: Two Difference Operators**

This example demonstrates the use of the SECTIONS directive. The logic is identical to the preceding DO example, but uses SECTIONS instead of DO. Here the speedup is limited to 2 because there are only two units of work whereas in DO: Two Difference Operators above there are `n-1 + m-1` units of work.

```
  subroutine sections_1 (a,b,c,d,m,n)
real a(n,n), b(n,n), c(m,m), d(m,m)
!$omp parallel
```

```
!$omp& shared(a,b,c,d,m,n)
!$omp& private(i,j)
!$omp sections
!$omp section
do i = 2, n
do j = 1, i
b(j,i)=( a(j,i) + a(j,i-1) ) / 2
enddo
enddo

!$omp section
do i = 2, m
do j = 1, i
d(j,i)=( c(j,i) + c(j,i-1) ) / 2
enddo
enddo
!$omp end sections nowait
!$omp end parallel
end
```

**SINGLE: Updating a Shared Scalar**

This example demonstrates how to use a SINGLE construct to update an element of the shared array a. The optional NOWAIT after the first loop is omitted because it is necessary to wait at the end of the loop before proceeding into the SINGLE construct.

```
  subroutine sp_1a (a,b,n)
real a(n), b(n)
!$omp parallel
!$omp& shared(a,b,n)
!$omp& private(i)
!$omp do
do i = 1, n
a(i) = 1.0 / a(i)
enddo
!$omp single
a(1) = min( a(1), 1.0 )
!$omp end single
!$omp do
do i = 1, n
b(i) = b(i) / a(i)
enddo
!$omp end do nowait
!$omp end parallel
end
```

# Debugging Multithreaded Programs

## Debugging Multithread Programs Overview

The debugging of multithreaded program discussed in this section applies to both the OpenMP* Fortran API and the Intel® Fortran parallel compiler directives. When a program uses parallel decomposition directives, you must take into consideration that the bug might be caused either by an incorrect program statement or it might be caused by an incorrect parallel decomposition directive. In either case, the program to be debugged can be executed by multiple threads simultaneously.

To debug the multithreaded programs, you can use:

- The Intel® Debugger for IA-32 and the Intel Debugger for Itanium®-based applications (idb)
- The Intel Fortran Compiler debugging options and methods.
- The Intel parallelization extension routines for low-level debugging.
- The VTune(TM) Performance Analyzer to define the problematic areas.

Other best known debugging methods and tips include:

- Correct the program in a single-threaded, uni-processor environment
- Statically analyze locks
- Use a trace statement (such as the PRINT statement)
- Think in parallel, make very few assumptions
- Step through your code
- Make sense of threads and callstack information
- Identify the primary thread
- Know what thread you are debugging
- Single stepping in one thread does not mean single stepping in others
- Watch out for context switch

**Debugger Limitations for Multithread Programs**

Debuggers such as Intel Debugger (IDB) for IA-32 and Intel Debugger (IDB) for Itanium-based applications support the debugging of programs that are executed by multiple threads. However, the currently available versions of such debuggers do not directly support the debugging of parallel decomposition directives, and therefore, there are limitations on the debugging features.

Some of the new features used in OpenMP are not yet fully supported by the debuggers, so it is important to understand how these features work to know how to debug them. The two problem areas are:

- Multiple entry points
- Shared variables

The Intel Debugger (IDB) is not aware of and currently does not handle unique OpenMP features that relate to multi-threading.

## Debugging Parallel Regions

The compiler implements a parallel region by enabling the code in the region and putting it into a separate, compiler-created entry point. Although this is different from outlining – the technique employed by other compilers, that is, creating a subroutine, – the same debugging technique can be applied.

### Constructing an Entry-point Name

The compiler-generated parallel region entry point name is constructed with a concatenation of the following strings:

- "__" character
- entry point name for the original routine (for example, `_parallel`)
- "_" character
- line number of the parallel region
- `__par_region` for OpenMP parallel regions (`!$OMP PARALLEL`) `__par_loop` for OpenMP parallel loops (`!$OMP PARALLEL DO`), `__par_section` for OpenMP parallel sections (`!$OMP PARALLEL SECTIONS`)
- sequence number of the parallel region (for each source file, sequence number starts from zero.)

When you use routine names (for example, `padd`) and entry names (for example, `_PADD`, `___PADD_6__par_loop0`), the following occurs. The Fortran Compiler, by default, first changes lower/mixed case routine names to upper case. For example, `pAdD()` becomes `PADD()`, and this becomes entry name by adding one underscore. The secondary entry name change happens after that. That's why the "`__par_loop`" part of the entry name stays as lower case. For some reason, the debugger doesn't accept the upper case routine name "`PADD`" to set the breakpoint. Instead, it accepts the lower case routine name "`padd`".

Example 1 shows the debugging of the code with a parallel region. Example 1 is produced by this command:

**`ifort -openmp -g -O0 -S`** ***`file.f90`***

Let us consider the code of subroutine `parallel` in Example 1.

### Subroutine PARALLEL() source listing

```
1    subroutine parallel
2    integer id,OMP_GET_THREAD_NUM
3 !$OMP PARALLEL PRIVATE(id)
4    id = OMP_GET_THREAD_NUM()
```

```
5 !$OMP END PARALLEL
6    end
```

The parallel region is at line 3. The compiler created two entry points: `parallel_` and `___parallel_3__par_region0`. The first entry point corresponds to the subroutine `parallel()`, while the second entry point corresponds to the OpenMP parallel region at line 3.

**Example 1 Debugging Code with Parallel Region**

```
Machine Code Listing of the Subroutine parallel()

        .globl parallel_
parallel_:
..B1.1:                   # Preds ..B1.0
..LN1:
pushl    %ebp                                   #1.0
movl     %esp, %ebp                             #1.0
subl     $44, %esp                              #1.0
pushl    %edi                                   #1.0
... ... ... ... ... ... ... ... ... ... ... ... ...
..B1.13:                   # Preds ..B1.9
addl     $-12, %esp                            #6.0
movl     $.2.1_2_kmpc_loc_struct_pack.2, (%esp) #6.0
movl     $0, 4(%esp)                           #6.0
movl     $_parallel__6__par_region1, 8(%esp)    #6.0
call     __kmpc_fork_call                      #6.0
       # LOE
..B1.31:                   # Preds ..B1.13
addl     $12, %esp                             #6.0
       # LOE
..B1.14:                   # Preds ..B1.31 ..B1.30
..LN4:
leave                                          #9.0
ret                                            #9.0
       # LOE

.type parallel_,@function
.size parallel_,.-parallel_
.globl _parallel__3__par_region0
_parallel__3__par_region0:
# parameter 1: 8 + %ebp
# parameter 2: 12 + %ebp
..B1.15:                   # Preds ..B1.0
pushl    %ebp                                  #9.0
movl     %esp, %ebp                            #9.0
subl     $44, %esp                             #9.0
..LN5:
call     omp_get_thread_num_                   #4.0
       # LOE eax
..B1.32:                   # Preds ..B1.15
movl     %eax, -32(%ebp)                       #4.0
```

```
       # LOE
..B1.16:                    # Preds ..B1.32
movl      -32(%ebp), %eax                        #4.0
movl      %eax, -20(%ebp)                        #4.0
..LN6:
leave                                            #9.0
ret                                              #9.0
       # LOE
.type _parallel__3__par_region0,@function
.size _parallel__3__par_region0,._parallel__3__par_region0
.globl _parallel__6__par_region1
_parallel__6__par_region1:
# parameter 1: 8 + %ebp
# parameter 2: 12 + %ebp
..B1.17:                     # Preds ..B1.0

pushl     %ebp                                   #9.0
movl      %esp, %ebp                             #9.0
subl      $44, %esp                              #9.0
..LN7:
call      omp_get_thread_num_                    #7.0
       # LOE eax
..B1.33:                    # Preds ..B1.17
movl      %eax, -28(%ebp)                        #7.0
       # LOE
..B1.18:                    # Preds ..B1.33
movl      -28(%ebp), %eax                        #7.0
movl      %eax, -16(%ebp)                        #7.0
..LN8:
leave                                            #9.0
ret                                              #9.0
.align    4,0x90
# mark_end;
```

Debugging the program at this level is just like debugging a program that uses POSIX threads directly. Breakpoints can be set in the threaded code just like any other routine. With GNU debugger, breakpoints can be set to source-level routine names (such as parallel). Breakpoints can also be set to entry point names (such as parallel_ and _parallel__3__par_region0). Note that the Intel Fortran Compiler for Linux converted the upper case Fortran subroutine name to the lower case one.

## Debugging Multiple Threads

When in a debugger, you can switch from one thread to another. Each thread has its own program counter so each thread can be in a different place in the code. Example 2 shows a Fortran subroutine `PADD()`. A breakpoint can be set at the entry point of OpenMP parallel region.

**`Source listing of the Subroutine PADD()`**

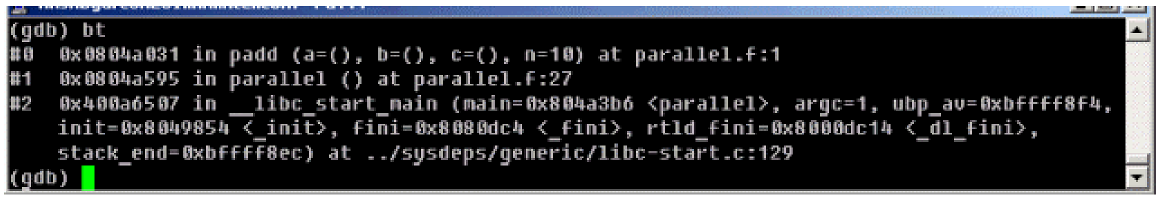
```
12.       SUBROUTINE PADD(A, B, C, N)
13.       INTEGER N
14.       INTEGER A(N), B(N), C(N)
15.       INTEGER I, ID, OMP_GET_THREAD_NUM
16. !$OMP PARALLEL DO SHARED (A, B, C, N) PRIVATE(ID)
17.       DO I = 1, N
18.         ID = OMP_GET_THREAD_NUM()
19.         C(I) = A(I) + B(I) + ID
20.       ENDDO
21. !$OMP END PARALLEL DO
22.       END
```

#### The Call Stack Dumps

The first call stack below is obtained by breaking at the entry to subroutine PADD using GNU debugger. At this point, the program has not executed any OpenMP regions, and therefore has only one thread. The call stack shows a system run-time `__libc_start_main` function calling the Fortran main program `parallel()`, and `parallel()` calls subroutine `padd()`. When the program is executed by more than one thread, you can switch from one thread to another. The second and the third call stacks are obtained by breaking at the entry to the parallel region. The call stack of master contains the complete call sequence. At the top of the call stack is `_padd__6__par_loop0()`. Invocation of a threaded entry point involves a layer of Intel OpenMP library function calls (that is, functions with `__kmp` prefix). The call stack of the worker thread contains a partial call sequence that begins with a layer of Intel OpenMP library function calls.

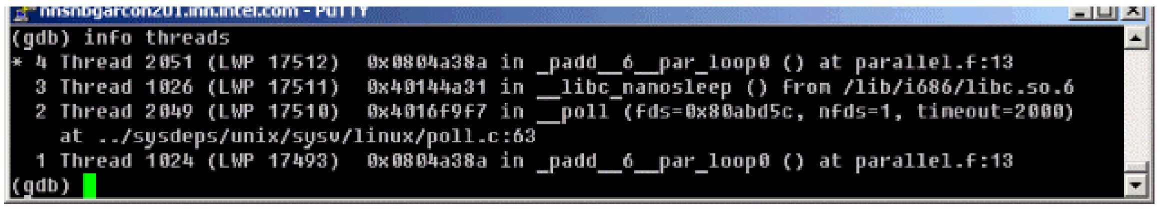
ERRATA: The GNU debugger sometimes fails to properly unwind the call stack of the immediate caller of the Intel OpenMP library function `__kmpc_fork_call()`.

### Call Stack Dump of Master Thread upon Entry to Subroutine PADD



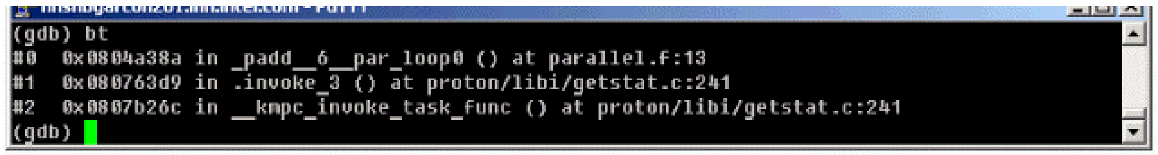
```
(gdb) bt
#0  0x0804a031 in padd (a=(), b=(), c=(), n=10) at parallel.f:1
#1  0x0804a595 in parallel () at parallel.f:27
#2  0x400a6507 in __libc_start_main (main=0x804a3b6 <parallel>, argc=1, ubp_av=0xbffff8f4,
    init=0x8049854 <_init>, fini=0x8080dc4 <_fini>, rtld_fini=0x8000dc14 <_dl_fini>,
    stack_end=0xbffff8ec) at ../sysdeps/generic/libc-start.c:129
(gdb)
```

### Switching from One Thread to Another



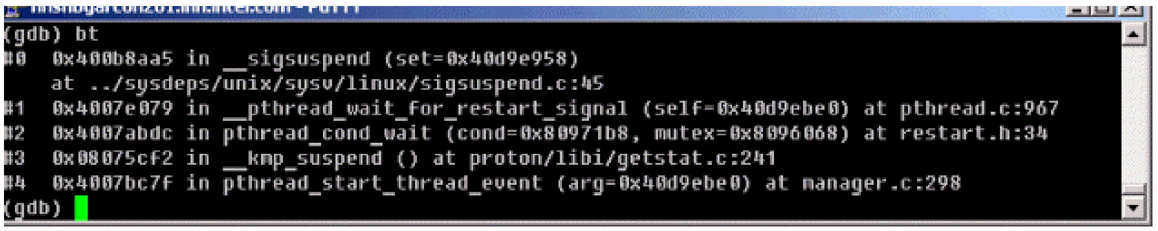
```
(gdb) info threads
* 4 Thread 2051 (LWP 17512)  0x0804a38a in _padd__6__par_loop0 () at parallel.f:13
  3 Thread 1026 (LWP 17511)  0x40144a31 in __libc_nanosleep () from /lib/i686/libc.so.6
  2 Thread 2049 (LWP 17510)  0x4016f9f7 in __poll (fds=0x80abd5c, nfds=1, timeout=2000)
    at ../sysdeps/unix/sysv/linux/poll.c:63
  1 Thread 1024 (LWP 17493)  0x0804a38a in _padd__6__par_loop0 () at parallel.f:13
(gdb)
```

**Call Stack Dump of Master Thread upon Entry to Parallel Region**



```
(gdb) bt
#0  0x0804a38a in _padd__6__par_loop0 () at parallel.f:13
#1  0x080763d9 in .invoke_3 () at proton/libi/getstat.c:241
#2  0x0807b26c in __kmpc_invoke_task_func () at proton/libi/getstat.c:241
(gdb)
```

**Call Stack Dump of Worker Thread upon Entry to Parallel Region**



```
(gdb) bt
#0  0x400b8aa5 in __sigsuspend (set=0x40d9e958)
    at ../sysdeps/unix/sysv/linux/sigsuspend.c:45
#1  0x4007e079 in __pthread_wait_for_restart_signal (self=0x40d9ebe0) at pthread.c:967
#2  0x4007abdc in pthread_cond_wait (cond=0x80971b8, mutex=0x8096068) at restart.h:34
#3  0x08075cf2 in __kmp_suspend () at proton/libi/getstat.c:241
#4  0x4007bc7f in pthread_start_thread_event (arg=0x40d9ebe0) at manager.c:298
(gdb)
```

**Example 2  Debugging Code Using Multiple Threads with Shared Variables**

```
Subroutine PADD() Machine Code Listing
     .globl padd_
padd_:
# parameter 1: 8 + %ebp
# parameter 2: 12 + %ebp
# parameter 3: 16 + %ebp
# parameter 4(n): 20 + %ebp
..B1.1:                      # Preds ..B1.0
..LN1:
pushl    %ebp                                     #1.0
... ... ... ... ... ... ... ... ... ... ... ... ...

..B1.19:                    # Preds ..B1.15
addl     $-28, %esp                               #6.0
movl     $.2.1_2_kmpc_loc_struct_pack.1, (%esp) #6.0
movl     $4, 4(%esp)                              #6.0
movl     $_padd__6__par_loop0, 8(%esp)            #6.0
movl     -196(%ebp), %eax                         #6.0
movl     %eax, 12(%esp)                           #6.0
movl     -152(%ebp), %eax                         #6.0
movl     %eax, 16(%esp)                           #6.0
movl     -112(%ebp), %eax                         #6.0
movl     %eax, 20(%esp)                           #6.0
lea      20(%ebp), %eax                           #6.0
movl     %eax, 24(%esp)                           #6.0
call     __kmpc_fork_call                         #6.0
              # LOE
..B1.39:                    # Preds ..B1.19
addl     $28, %esp                                #6.0
jmp      ..B1.31       # Prob 100%                #6.0
              # LOE
..B1.20:                    # Preds ..B1.30
... ... ... ... ... ... ... ... ... ... ... ... ...
```

```
call      __kmpc_for_static_init_4                #6.0
               # LOE
..B1.40:                  # Preds ..B1.20
addl      $36, %esp                               #6.0
               # LOE
... ... ... ... ... ... ... ... ... ... ... ... ...

..B1.26:                  # Preds ..B1.28 ..B1.21
addl      $-8, %esp                               #6.0
movl      $.2.1_2_kmpc_loc_struct_pack.1, (%esp) #6.0
movl      -8(%ebp), %eax                          #6.0
movl      %eax, 4(%esp)                           #6.0
call      __kmpc_for_static_fini                  #6.0
               # LOE
..B1.41:                  # Preds ..B1.26
addl      $8, %esp                                #6.0
jmp       ..B1.31       # Prob 100%               #6.0
               # LOE
..B1.27:                  # Preds ..B1.28 ..B1.25
..LN7:
call      omp_get_thread_num_                     #8.0
                # LOE eax
..B1.42:                   # Preds ..B1.27
... ... ... ... ... ... ... ... ... ... ... ... ...

cmpl      %edx, %eax                              #10.0
jle       ..B1.27      # Prob 50%                 #10.0
jmp       ..B1.26      # Prob 100%                #10.0
              # LOE
.type padd_,@function
.size padd_,.-padd_
.globl _padd__6__par_loop0
_padd__6__par_loop0:
# parameter 1: 8 + %ebp
# parameter 2: 12 + %ebp
# parameter 3: 16 + %ebp
# parameter 4: 20 + %ebp
# parameter 5: 24 + %ebp
# parameter 6: 28 + %ebp
..B1.30:                   # Preds ..B1.0
..LN16:
pushl     %ebp                                    #13.0
movl      %esp, %ebp                              #13.0
subl      $208, %esp                              #13.0
movl      %ebx, -4(%ebp)                          #13.0
..LN17:
movl      8(%ebp), %eax                           #6.0
movl      (%eax), %eax                            #6.0
movl      %eax, -8(%ebp)                          #6.0
movl      28(%ebp), %eax                          #6.0
..LN18:
movl      (%eax), %eax                            #7.0
movl      (%eax), %eax                            #7.0
movl      %eax, -80(%ebp)                              #7.0
movl      $1, -76(%ebp)                                #7.0
movl      -80(%ebp), %eax                              #7.0
testl     %eax, %eax                                   #7.0
jg        ..B1.20        # Prob 50%                    #7.0
              # LOE
..B1.31:                    # Preds ..B1.41 ..B1.39 ..B1.38
..B1.30
..LN19:
movl      -4(%ebp), %ebx                               #13.0
leave                                                  #13.0
ret                                                    #13.0
.align    4,0x90
# mark_end;
```

## Debugging Shared Variables

When a variable appears in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION clause on some block, the variable is made private to the parallel region by redeclaring it in the block. SHARED data, however, is not declared in the threaded code. Instead, it gets its declaration at the routine level. At the machine code level, these shared variables become incoming subroutine call arguments to the threaded entry points (such as `___PADD_6__par_loop0`).

In Example 2, the entry point `___PADD_6_par_loop0` has six incoming parameters. The corresponding OpenMP parallel region has four shared variables. First two parameters (parameters 1 and 2) are reserved for the compiler's use, and each of the remaining four parameters corresponds to one shared variable. These four parameters exactly match the last four parameters to `__kmpc_fork_call()` in the machine code of `PADD`.

**Note**

The FIRSTPRIVATE, LASTPRIVATE, and REDUCTION variables also require shared variables to get the values into or out of the parallel region.

Due to the lack of support in debuggers, the correspondence between the shared variables (in their original names) and their contents cannot be seen in the debugger at the threaded entry point level. However, you can still move to the call stack of one of the subroutines and examine the contents of the variables at that level. This technique can be used to examine the contents of shared variables. In Example 2, contents of the shared variables `A`, `B`, `C`, and `N` can be examined if you move to the call stack of `PARALLEL()`.

# Optimization Support Features

## Optimization Support Features Overview

This section describes the Intel® Fortran features such as directives, intrinsics, run-time library routines and various utilities which enhance your application performance in support of compiler optimizations. These features are Intel Fortran language extensions that enable you to optimize your source code directly. This section includes examples of optimizations supported by Intel extended directives and intrinsics or library routines that enhance and/or help analyze performance.

For complete details of the Intel® Fortran Compiler directives and examples of their use, see Chapter 14, "Directive Enhanced Compilation," in the *Intel® Fortran Language Reference*. For intrinsic procedures, see Chapter 9, "Intrinsic Procedures," in the *Intel® Fortran Language Reference*.

A final topic describes options that enable you to generate optimization reports for major compiler phases and major optimizations. The optimization report capability is used for Itanium®-based applications only.

## Compiler Directives

### Compiler Directives Overview

This section discusses the Intel® Fortran language extended directives that enhance optimizations of application code, such as software pipelining, loop unrolling, prefetching and vectorization. For complete list, descriptions and code examples of the Intel® Fortran Compiler directives, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

### Pipelining for Itanium®-based Applications

The SWP and NOSWP directives indicate preference for a loop to get software-pipelined or not. The SWP directive does not help data dependence, but overrides heuristics based on profile counts or lop-sided control flow.

The software pipelining optimization triggered by the SWP directive applies instruction scheduling to certain innermost loops, allowing instructions within a loop to be split into different stages, allowing increased instruction level parallelism. This can reduce the impact of long-latency operations, resulting in faster loop execution. Loops chosen for software pipelining are always innermost loops that do not contain procedure calls that are not inlined. Because the optimizer no longer considers fully unrolled loops as innermost loops, fully unrolling loops can allow an additional loop to become the innermost loop (see `-unroll[n]`]). You can request and view the optimization report to see whether software pipelining was applied (see Optimizer Report Generation).

```
!DEC$ SWP
do i = 1, m
if (a(i) .eq. 0)  then
b(i) = a(i) + 1
else
b(i) = a(i)/c(i)
endif
enddo
```

For more details on these directives, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

## Loop Count and Loop Distribution

### LOOP COUNT (N) Directive

The LOOP COUNT (n) directive indicates the loop count is likely to be n, where n is an integer constant.

The value of loop count affects heuristics used in software pipelining, vectorization and loop-transformations.

```
!DEC$ LOOP COUNT (10000)
do i =1,m
b(i) = a(i) +1 ! This is likely to enable
! the loop to get software-
! pipelined
enddo
```

For more details on this directive, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

### Loop Distribution Directive

The DISTRIBUTE POINT directive indicates a preference of performing loop distribution.

Loop distribution may cause large loops be distributed into smaller ones. This may enable more loops to get software-pipelined. If the directive is placed inside a loop, the distribution is performed after the directive and any loop-carried dependency is ignored. If the directive is placed before a loop, the compiler will determine where to distribute and data dependency is observed. Currently only one distribute directive is supported if it is placed inside the loop.

```
!DEC$ DISTRIBUTE POINT
do i =1, m
b(i) = a(i) +1
....
c(i) = a(i) + b(i) ! Compiler will decide where
 ! to distribute.
 ! Data dependency is observed
....
d(i) = c(i) + 1
enddo

do i =1, m
b(i) = a(i) +1
....
!DEC$  DISTRIBUTE POINT
call  sub(a, n)    ! Distribution will start here,
 ! ignoring all loop-carried
 ! dependency
c(i) = a(i) + b(i)
....
d(i) = c(i) + 1
enddo
```

For more details on this directive, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

## Loop Unrolling Support

The UNROLL[*n*] directive tells the compiler how many times to unroll a counted loop.

The *n* is an integer constant from 0 through 255.

The UNROLL directive must precede the DO statement for each DO loop it affects.

If *n* is specified, the optimizer unrolls the loop *n* times. If *n* is omitted or if it is outside the allowed range, the optimizer assigns the number of times to unroll the loop.

The UNROLL directive overrides any setting of loop unrolling from the command line.

Currently, the directive can be applied only for the innermost loop nest. If applied to the outer loop nests, it is ignored. The compiler generates correct code by comparing *n* and the loop count.

```
!DEC$ UNROLL(4)
do i = 1, m
```

```
b(i) = a(i) + 1
d(i) = c(i) + 1
enddo
```

For more details on these directives, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

## Prefetching Support

The PREFETCH and NOPREFETCH directives assert that the data prefetches be generated or not generated for some memory references. This affects the heuristics used in the compiler.

If loop includes expression `a(j)`, placing `PREFETCH a` in front of the loop, instructs the compiler to insert prefetches for `a(j + d)` within the loop. `d` is determined by the compiler. This directive is supported when option `-O3` is on.

```
CDEC$ NOPREFETCH c
CDEC$ PREFETCH a
do i = 1, m
b(i) = a(c(i)) + 1
enddo
```

For more details on these directives, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

## Vectorization Support

The directives discussed in this topic support vectorization.

### IVDEP Directive

The IVDEP directive instructs the compiler to ignore assumed vector dependences. To ensure correct code, the compiler treats an assumed dependence as a proven dependence, which prevents vectorization. This directive overrides that decision. Use IVDEP only when you know that the assumed loop dependences are safe to ignore.

For example, if the expression `j >= 0` is always true in the code fragment bellow, the IVDEP directive can communicate this information to the compiler. This directive informs the compiler that the conservatively assumed loop-carried flow dependences for values `j < 0` can be safely ignored:

```
!DEC$ IVDEP
do i = 1, 100
a(i) = a(i+j)
enddo
```

**Note**
The proven dependences that prevent vectorization are not ignored, only assumed dependences are ignored.

The usage of the directive differs depending on the loop form.

**Loop 1**
```
Do i
= a(*) + 1
a(*) =
enddo
```
**Loop 2**
```
Do i
a(*) =
= a(*) + 1
enddo
```

For loops of the form 1, use old values of a, and assume that there is no loop-carried flow dependencies from DEF to USE.

For loops of the form 2, use new values of a, and assume that there is no loop-carried anti-dependencies from USE to DEF.

In both cases, it is valid to distribute the loop, and there is no loop-carried output dependency.
**Example 1**
```
CDEC$ IVDEP
do  j=1,n
a(j) = a(j+m) + 1
enddo
```
**Example 2**
```
CDEC$ IVDEP
do  j=1,n
a(j) = b(j) +1
b(j) = a(j+m) + 1
enddo
```

Example 1 ignores the possible backward dependencies and enables the loop to get software pipelined.

Example 2 shows possible forward and backward dependencies involving array a in this loop and creating a dependency cycle. With IVDEP , the backward dependencies are ignored.

IVDEP has options: IVDEP:LOOP and IVDEP:BACK. The IVDEP:LOOP option implies no loop-carried dependencies. The IVDEP:BACK option implies no backward dependencies.

The IVDEP directive is also used with the `-ivdep_parallel` option for Itanium®-based applications.

For more details on these directives, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

### Overriding Vectorizer's Efficiency Heuristics

In addition to IVDEP directive, there are more directives that can be used to override the efficiency heuristics of the vectorizer:

VECTOR ALWAYS
NOVECTOR
VECTOR ALIGNED
VECTOR UNALIGNED
VECTOR NONTEMPORAL

The VECTOR directives control the vectorization of the subsequent loop in the program, but the compiler does not apply them to nested loops. Each nested loop needs its own directive preceding it. You must place the vector directive before the loop control statement.

For more details on these directives, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

#### The VECTOR ALWAYS and NOVECTOR Directives

The VECTOR ALWAYS directive overrides the efficiency heuristics of the vectorizer, but it only works if the loop can actually be vectorized, that is: use IVDEP to ignore assumed dependences.

The VECTOR ALWAYS directive can be used to override the default behavior of the compiler in the following situation. Vectorization of non-unit stride references usually does not exhibit any speedup, so the compiler defaults to not vectorizing loops that have a large number of non-unit stride references (compared to the number of unit stride references). The following loop has two references with `stride 2`. Vectorization would be disabled by default, but the directive overrides this behavior.

```
!DEC$ VECTOR ALWAYS
do i = 1, 100, 2
a(i) = b(i)
enddo
```

If, on the other hand, avoiding vectorization of a loop is desirable (if vectorization results in a performance regression rather than improvement), the NOVECTOR directive can be used in the source text to disable vectorization of a loop. For instance, the Intel® Compiler vectorizes the following example loop by default. If this behavior is not appropriate, the NOVECTOR directive can be used, as shown below.

```
!DEC$ NOVECTOR
do i = 1, 100
a(i) = b(i) + c(i)
enddo
```

For more details on these directives, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

#### The VECTOR ALIGNED and UNALIGNED Directives

Like VECTOR ALWAYS, these directives also override the efficiency heuristics. The difference is that the qualifiers UNALIGNED and ALIGNED instruct the compiler to use, respectively, unaligned and aligned data movement instructions for all array references. This disables all the advanced alignment optimizations of the compiler, such as determining alignment properties from the program context or using dynamic loop peeling to make references aligned.

**Note**
The directives VECTOR [ALWAYS, UNALIGNED, ALIGNED] should be used with care. Overriding the efficiency heuristics of the compiler should only be done if the programmer is absolutely sure the vectorization will improve performance. Furthermore, instructing the compiler to implement all array references with aligned data movement instructions will cause a run-time exception in case some of the access patterns are actually unaligned.

For more details on these directives, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

#### The VECTOR NONTEMPORAL Directive

The VECTOR NONTEMPORAL directive results in streaming stores on Pentium® 4 based systems. A floating-point type loop together with the generated assembly are shown in the example below. For large n, significant performance improvements result on a Pentium 4 systems over a non-streaming implementation.

The following example illustrates the use of the VECTOR NONTEMPORAL directive:

```
       subroutine set(a,n)
 integer i,n
 real a(n)
!DEC$ VECTOR NONTEMPORAL
!DEC$ VECTOR ALIGNED
 do i = 1, n
   a(i) = 1
 enddo
 end
 program setit
 parameter(n=1024*1204)
 real a(n)
 integer i
 do i = 1, n
   a(i) = 0
 enddo
 call set(a,n)
 do i = 1, n
    if (a(i).ne.1) then
      print *, 'failed nontemp.f', a(i), i
      stop
    endif
 enddo
 print *, 'passed nontemp.f'
 end
```

For more details on these directives, see "Directive Enhanced Compilation", section "General Directives", in the *Intel® Fortran Language Reference*.

## Optimizations and Debugging

This topic describes the command-line options that you can use to debug your compilation and to display and check compilation errors.

The options that enable you to get debug information while optimizing are as follows:

| | |
|---|---|
| `-O0` | Disables optimizations. Enables `-fp` option. |
| `-g` | Generates symbolic debugging information and line numbers in the object code for use by the source-level debuggers. Turns off `-O2` and makes `-O0` the default unless `-O2` (or `-O1` or `-O3`) is explicitly specified in the command line together with `-g`. |
| `-debug` *keyword* | Specifies settings that enhance debugging. To use this option, you must also specify the `-g` option. The only choice for *`keyword`* is |

| | |
|---|---|
| | `variable_locations`, which produces enhanced debug information useful in finding scalar local variables. |
| `-fp`<br>IA-32 only | Disables the use of the ebp register in optimizations. Directs to use the ebp-based stack frame for all functions. |

## Support for Symbolic Debugging, -g

Use the `-g` option to direct the compiler to generate code to provide symbolic debugging information and line numbers in the object code that will be used by your source-level debugger. For example:

**`ifort -g`** ***`prog1.f`***

Turns off `-O2` and makes `-O0` the default unless `-O2` (or `-O1` or `-O3`) is explicitly specified in the command line together with `-g`.

## The Use of ebp Register

### `-fp` (IA-32 only)

Most debuggers use the ebp register as a stack frame pointer to produce a stack backtrace. The `-fp` option disables the use of the ebp register in optimizations and directs the compiler to generate code that maintains and uses ebp as a stack frame pointer for all functions so that a debugger can still produce a stack backtrace without turning off `-O1, -O2,` or `-O3` optimizations.

Note that using this option reduces the number of available general-purpose registers by one, and results in slightly less efficient code.

### `-fp` Summary

| Default | OFF |
|---|---|
| `-O1` , -O2, or -O3 | Disables `-fp` |
| `-O0` | Enables `-fp` |

### The `-traceback` Option

The `-traceback` option also forces the compiler to use ebp as the stack frame pointer. In addition, the `-traceback` option causes the compiler to generate extra information into the object file, which allows a symbolic stack traceback to be produced if a run-time failure occurs.

## Combining Optimization and Debugging

The `-O0` option turns off all optimizations so you can debug your program before any optimization is attempted. To get the debug information, use the `-g` option.

The compiler lets you generate code to support symbolic debugging while one of the `-O1`, `-O2`, or `-O3` optimization options is specified on the command line along with `-g`, which produces symbolic debug information in the object file.

Note that if you specify an `-O1`, `-O2`, or `-O3` option with the `-g` option, some of the debugging information returned may be inaccurate as a side-effect of optimization.

It is best to make your optimization and/or debugging choices explicit:

- If you need to debug your program excluding any optimization effect, use the `-O0` option, which turns off all the optimizations.
- If you need to debug your program with optimization enabled, then you can specify the `-O1`, `-O2`, or `-O3` option on the command line along with `-g`.

**Note**
The `-g` option slows down the program when no optimization level (`-O`*n*) is specified. In this case `-g` turns on `-O0`, which is what slows the program down. However, if, for example, both `-O2` and `-g` are specified, the code should run very nearly at the same speed as if `-g` were not specified.

Refer to the table below for the summary of the effects of using the `-g` option with the optimization options.

| These options | Produce these results |
|---|---|
| `-g` | Debugging information produced, `-O0` enabled (optimizations disabled), `-fp` enabled for IA-32-targeted compilations |
| `-g -O1` | Debugging information produced, `-O1` optimizations enabled. |
| `-g -O2` | Debugging information produced, `-O2` optimizations enabled. |
| `-g -O3 -fp` | Debugging information produced, `-O3` optimizations enabled, `-fp` enabled for IA-32-targeted compilations. |

## Debugging and Assembling

The assembly listing file is generated without debugging information, but if you produce an object file, it will contain debugging information. If you link the object file and then use the GDB debugger on it, you will get full symbolic representation.

## Optimizer Report Generation

The Intel® Fortran Compiler provides options to generate and manage optimization reports.

- `-opt_report` generates optimizations report and places it in a file specified in `-opt_report_file`*`filename`*. If `-opt_report_file` is not specified, `-opt_report` directs the report to `stderr`. The default is OFF: no reports are generated.
- `-opt_report_file`*`filename`* generates optimizations report and directs it to a file specified in *`filename`*.
- `-opt_report_level{`*`min`*`|`*`med`*`|`*`max`*`}` specifies the detail level of the optimizations report. The *`min`* argument provides the minimal summary and the *`max`* the full report. The default is `-opt_report_level`*`min`*.
- `-opt_report_routine [`*`substring`*`]` generates reports from all routines with names containing the *`substring`* as part of their name. If `[`*`substring`* `]` is not specified, reports from all routines are generated. The default is to generate reports for all routines being compiled.

### Specifying Optimizations to Generate Reports

The compiler can generate reports for an optimizer you specify in the *`phase`* argument of the `-opt_report_phase`*`phase`* option.

The option can be used multiple times on the same command line to generate reports for multiple optimizers.

Currently, the reports for the following optimizers are supported:

| **Optimizer Logical Name** | **Optimizer Full Name** |
|---|---|
| `ipo` | Interprocedural Optimizer |
| `hlo` | High-level Language Optimizer |
| `ilo` | Intermediate Language Scalar Optimizer |
| `ecg` | Itanium®-based Compiler Code Generator |
| `all` | All optimizers |

When one of the above logical names for optimizers are specified all reports from that optimizer will be generated. For example, `-opt_report_phase`*`ipo`* and `-opt_report_phase`*`ecg`* generate reports from the interprocedural optimizer and the code generator.

Each of the optimizers can potentially have specific optimizations within them. Each of these optimizations are prefixed with the optimizer's logical name. For example:

| **Optimizer_optimization** | **Full Name** |
|---|---|
| `ipo_inl` | Interprocedural Optimizer, inline expansion of functions |
| `ipo_cp` | Interprocedural Optimizer, copy propagation |
| `hlo_unroll` | High-level Language Optimizer, loop unrolling |
| `hlo_prefetch` | High-level Language Optimizer, prefetching |
| `ilo_copy_propagation` | Intermediate Language Scalar Optimizer, copy propagation |
| `ecg_swp` | Itanium®-based Compiler Code Generator, software pipelining |

The following command generates a report for the Itanium®-based Compiler Code Generator (ecg):

```
ifort -c -opt_report -opt_report_phase ecg myfile.f
```

where:

- `-c` tells the compiler to stop at generating the object code, not linking
- `-opt_report` invokes the report generator
- `-opt_report_phase`*`ecg`* indicates the phase (ecg) for which to generate the report; the space between the option and the phase is optional.

The entire name for a particular optimization within an optimizer need not be specified in full, just a few characters is sufficient. All optimization reports that have a matching prefix with the specified optimizer are generated. For example, if `-opt_report_phase` *`ilo_co`* is specified, a report from both the constant propagation and the copy propagation are generated.

**The Availability of Report Generation**

The `-opt_report_help` option lists the logical names of optimizers and optimizations that are currently available for report generation.

For IA-32 systems, the reports can be generated for:

- `ilo`
- `hlo` if `-O3` is on
- `ipo` if interprocedural optimizer is invoked with `-ip` or `-ipo`
- `all` the above optimizers if `-O3` and `-ip` or `-ipo` options are on

For Itanium-based systems, the reports can be generated for:

- `ilo`
- `ecg`
- `hlo` if `-O3` is on
- `ipo` if interprocedural optimizer is invoked with `-ip` or `-ipo`
- `all` the above optimizers if `-O3` and `-ip` or `-ipo` options are on

**Note**
If `hlo` or `ipo` report is requested, but the controlling option (`-O3` or `-ip--ipo`, respectively) is not on, the compiler generates an empty report.

# Glossary

## Glossary

| | |
|---|---|
| alignment constraint | The proper boundary of the stack where data must be stored. |
| alternate loop transformation | An optimization in which the compiler generates a copy of a loop and executes the new loop depending on the boundary size. |
| branch count profiler | A tool that counts the number of times a program executes each branch statement. The utility also generates a database that shows how the program executed. |
| branch probability database | The database generated by the branch count profiler. The database contains the number of times each branch is executed. |
| cache hit | The situation when the information the processor wants is in the cache. |
| call site | A call site consists of the instructions immediately preceding a call instruction and the call instruction itself. |
| common subexpression elimination | An optimization in which the compiler detects and combines redundant computations. |
| conditionals | Any operation that takes place depending on whether or not a certain condition is true. |
| constant argument propagation | An optimization in which the compiler replaces the formal arguments of a routine with actual constant values. The compiler then propagates constant variables used as actual arguments. |
| constant branches | Conditionals that always take the same branch. |
| constant folding | An optimization in which the compiler, instead of storing the numbers and operators for computation when the program executes, evaluates the constant expression and uses the result. |
| copy propagation | An optimization in which the compiler eliminates unnecessary assignments by using the value assigned to a variable instead of using the variable itself. |
| dataflow | The movement of data through a system, from entry to destination. |

| | |
|---|---|
| dead-code elimination | An optimization in which the compiler eliminates any code that generates unused values or any code that will never be executed in the program. |
| dynamic linking | The process in which a shared object is mapped into the virtual address space of your program at run time. |
| empty declaration | A semicolon and nothing before it. |
| frame pointer | A pointer that holds a base address for the current stack and is used to access the stack frame. |
| in-line function expansion | An optimization in which the compiler replaces each function call with the function body expanded in place. |
| induction variable simplification | An optimization in which the compiler reduces the complexity of an array index calculation by using only additions. |
| instruction scheduling | An optimization in which the compiler reorders the generated machine instructions so that more than one can execute in parallel. |
| instruction sequencing | An optimization in which the compiler eliminates less efficient instructions and replaces them with instruction sequences that take advantage of a particular processor's features. |
| interprocedural optimization | An optimization that applies to the entire program except for library routines. |
| loop blocking | An optimization in which the compiler reorders the execution sequence of instructions so that the compiler can execute iterations from outer loops before completing all the iterations of the inner loop. |
| loop unrolling | An optimization in which the compiler duplicates the executed statements inside a loop to reduce the number of loop iterations. |
| loop-invariant code movement | An optimization in which the compiler detects multiple instances of a computation that does not change within a loop. |
| padding | The addition of bytes or words at the end of each data type in order to meet size and alignment constraints. |
| preloading | An optimization in which the compiler loads the vectors, one cache at a time, so that during the loop computation the number of external bus turnarounds is reduced. |
| profiling | A process in which detailed information is produced about the program's execution. |

| | |
|---|---|
| register variable detection | An optimization in which the compiler detects the variables that never need to be stored in memory and places them in register variables. |
| side effects | Results of the optimization process that might increase the code size and/or processing time. |
| static linking | The process in which a copy of the object file that contains a function used in your program is incorporated in your executable file at link time. |
| strength reduction | An optimization in which the compiler reduces the complexity of an array index calculation by using only additions. |
| strip mining | An optimization in which the compiler creates an additional level of nesting to enable inner loop computations on vectors that can be held in the cache. This optimization reduces the size of inner loops so that the amount of data required for the inner loop can fit the cache size. |
| token pasting | The process in which the compiler treats two tokens separated by a comment as one (for example, a/**/b become ab). |
| transformation | A rearrangement of code. In contrast, an optimization is a rearrangement of code where improved run-time performance is guaranteed. |
| unreachable code | Instructions that are never executed by the compiler. |
| unused code | Instructions that produce results that are not used in the program. |
| variable renaming | An optimization in which the compiler renames instances of a variable that refer to distinct entities. |

# Index