Intel® Fortran Optimizing Applications

Document number: 304970-005US

Disclaimer and Legal Information

Table Of Contents

Optimizing Applications

Overview: Optimizing Applications

Optimizing Applications explains how to use the Intel® Fortran Compiler to help improve application performance.

How you use the information presented in this document depends on what you are trying to accomplish. You can start with the following topics:

- Optimizing with the Intel Compiler
- Optimizing for Performance
- Overview of Parallelism Methods
- Quick Reference Lists
- Other Resources
- Performance Analysis

Where applicable this document explains how compiler options and optimization methods differ on IA-32, Intel® 64, and IA-64 architectures on Linux* operating systems (OS), Intel®-based systems running Mac OS* X, and Windows* operating systems.

While the compiler supports Integrated Development Environments (IDE) on several different operating systems, the concepts and examples included here illustrate using the compiler from the command line.

In most cases, the compiler features and options supported for IA-32 or Intel® 64 architectures on Linux OS are also supported on Intel-based systems running Mac OS X. For more detailed information about support for specific operating systems, refer to the appropriate option in *Compiler Options*.

Optimizing with the Intel® Compiler

The Intel compiler supports a variety of options and features that allow optimization opportunities; however, in most cases you will benefit by applying optimization strategies in the order listed below.

Use Automatic Optimizations

Use the automatic optimization options, like `-O1`, `-O2`, `-O3`, or `-fast` (Linux* and Mac OS* X) and `/O1`, `/O2`, `/O3`, or `/fast` (Windows*) to determine what works best for your application. Use these options and measure the resulting performance after each compilation.

Note

The optimizer that integrates parallelization (IA-32, Intel® 64, and IA-64 architectures) and vectorization (IA-32 and Intel® 64 architectures) has been redesigned to provide performance improvements when specifying the `-O2` or `-O3` (Linux and Mac OS X) and `/O2` or `/O3` (Windows) options.

You might find specific options to work better on a particular architecture:

- IA-32 and Intel® 64 architectures: start with `-O2` (Linux and Mac OS X) or `/O2` (Windows).
- IA-64 architecture: start with `-O3` (Linux and Mac OS X) or `/O3` (Windows).

If you plan to run your application on specific architectures, experiment with combining the automatic optimizations with compiler options that specifically target processors.

You can combine the `-x` and `-ax` (Linux* and Mac OS* X) or `/Qx` and `/Qax` (Windows*) options to generate both code that is optimized for specific Intel processors and generic code that will run on most processors based on IA-32 and Intel® 64 architectures. See the following topics for more information about targeting processors:

- Targeting IA-32 and Intel® 64 Architecture Processors Automatically
- Targeting Multiple IA-32 and Intel 64 Architecture Processors Automatically for Run-time Performance
- Targeting Itanium® Processors Automatically

Attempt to combine the automatic optimizations with the processor-specific options before applying other optimizations techniques.

Use IPO and PGO
Experiment with Interprocedural Optimization (IPO) and Profile-guided Optimization (PGO). Measure performance after applying the optimizations to determine whether the application performance improved.

Use a top-down, iterative method for identifying and resolving performance-hindering code using performance monitoring tools, like the Intel® VTune™ Performance Analyzer or the compiler reports.

Use Parallelism
If you are planning to run the application on multi-core or multi-processor systems, start the parallelism process by using the parallelism options or OpenMP* options.

Compiling for Older Processors
Use automatic optimization options and other processor-independent compiler options to generate optimized code that do not take advantage of advances in processor design or extension support. Use the `-x` (Linux* and Mac OS* X) or `/Qx` (Windows*) option to generate processor dispatch for older processors.

Optimizing for Performance

The following table lists possible starting points for your optimization efforts.

If you are trying to...	Start with these topics or sections.
use performance analysis to begin the optimization process	• Optimizing with Intel® Compilers • Using a Performance Enhancement Methodology

	• Intel® Performance Analysis Tools and Libraries • Performance Enhancement Strategies
optimize for speed or a specific architecture	• Enabling Automatic Optimizations • Targeting IA-32 and Intel® 64 Architecture Processors Automatically • Targeting Multiple IA-32 and Intel® 64 Architecture Processors for Run-time Performance • Targeting IA-64 Architecture Processors Automatically
create parallel programs or parallelize existing programs	• Using Parallelism • Automatic Vectorization Overview • OpenMP* Support Overview • Auto-parallelization Overview
use Interprocedural Optimization	• Using IPO • IPO for Large Programs • Interprocedural Optimization (IPO) Quick Reference
create application profiles to help optimization	• Profile an Application • Profile-Guided Optimization (PGO) Quick Reference • profmerge and proforder Tools • PGO API and environment variables
generate reports on compiler optimizations	• Generating Reports • Compiler Reports Quick Reference
optimize loops, arrays, and data layout	• High-level Optimization (HLO) Overview
use programming strategies to improve performance	• Setting Data Type and Alignment • Using Arrays Efficiently • Improving I/O Performance • Improving Run-time Efficiency • Using Intrinsics for IA-64 architecture based Systems • Coding Guidelines for Intel Architectures

Overview of Parallelism Methods

The three major features of parallel programming supported by the Intel® compiler include:

- OpenMP*
- Auto-parallelization
- Auto-vectorization

Each of these features contributes to application performance depending on the number of processors, target architecture (IA-32, Intel® 64, and IA-64 architectures), and the nature of the application. These features of parallel programming can be combined to contribute to application performance.

Parallelism defined with the OpenMP* API is based on thread-level and task-level parallelism. Parallelism defined with auto-parallelization techniques is based on thread-level parallelism (TLP). Parallelism defined with auto-vectorization techniques is based on instruction-level parallelism (ILP).

Parallel programming can be *explicit*, that is, defined by a programmer using the OpenMP* API and associate options. Parallel programming can also be *implicit*, that is, detected automatically by the compiler. Implicit parallelism implements auto-parallelization of outer-most loops and auto-vectorization of innermost loops (or both).

To enhance the compilation of the code with auto-vectorization, users can also add vectorizer directives to their program.

Note

Software pipelining (SWP), a technique closely related to auto-vectorization, is available on systems based on IA-64 architecture.

The following table summarizes the different ways in which parallelism can be exploited with the Intel® Compiler.

Parallelism Method	Supported On
Implicit (parallelism generated by the compiler and by user-supplied hints)	
Auto-parallelization (Thread-Level Parallelism)	• IA-32 architecture, Intel® 64 architecture, IA-64 architecture based multi-processor systems, and multi-core processors • Hyper-Threading Technology-enabled systems
Auto-vectorization (Instruction-Level Parallelism)	• Pentium®, Pentium with MMX™ Technology, Pentium II, Pentium III, Pentium 4 processors, Intel® Core™ processor, Intel® Core™ 2 processor, and Intel® Atom™ processor.
Explicit (parallelism programmed by the user)	

OpenMP* (Thread-Level and Task-Level Parallelism)	• IA-32 architecture, Intel® 64 architecture, IA-64 architecture-based multiprocessor systems, and multi-core processors • Hyper-Threading Technology-enabled systems

Intel provides performance libraries that contain highly optimized, extensively threaded routines, including the Intel® Math Kernel Library (Intel® MKL).

In addition to these major features supported by the Intel compiler, certain operating systems support application program interface (API) function calls that provide explicit threading controls. For example, Windows* operating systems support API calls such as `CreateThread`, and multiple operating systems support POSIX* threading APIs.

Performance Analysis
For performance analysis of your parallel program, you can use the Intel® VTune™ Performance Analyzer and/or the Intel® Threading Tools to show performance information. You can obtain detailed information about which portions of the code require the largest amount of time to execute and where parallel performance problems are located.

Threading Resources
For general information about threading an existing serial application or design considerations for creating new threaded applications, see Other Resources and the web site http://go-parallel.com.

Quick Reference Lists

There are several quick reference guides in this document. Use these quick reference guides to quickly familiarize yourself with the compiler options or features available for specific optimizations.

- Enabling Automatic Optimizations
- Interprocedural Optimization (IPO) Quick Reference
- Profile-guided Optimization (PGO) Quick Reference
- Auto-Vectorization Options Quick Reference
- OpenMP* Options Quick Reference
- Auto-Parallelization Options Quick Reference
- Compiler Reports Quick Reference
- Data Alignment Options
- PGO Environment Variables
- OpenMP* Environment Variables

Other Resources

Understanding the capabilities of the specific processors and the underlying architecture on which your application will run is key to optimization. Intel distributes many hardware and software development resources that can help better understand how to optimize application source code for specific architectures.

Processor Information

You can find detailed information about processor numbers, capabilities, and technical specifications, along with documentation, from the following web sites:

- *Intel® Processor Spec Finder* (http://processorfinder.intel.com/)
- *Intel® Processor Numbers* (http://www.intel.com/products/processor_number/)
- *Intel® Processor Identification Utility* (http://www.intel.com/support/processors/tools/piu/)

Architecture Information

The architecture manuals provide specific details about the basic architecture, supported instruction sets, programming guidelines for specific operating systems, and performance monitoring.

The optimization manuals provide insight for developing high-performance applications for Intel® architectures.

- IA-32 and Intel® 64 architectures: http://www.intel.com/products/processor/manuals/index.htm
- IA-64 Architecture: http://www.intel.com/design/itanium/documentation.htm

Optimization Strategy Resources

For more information on advanced or specialized optimization strategies, refer to the Resource Centers for Software Developers, which can be accessed from www.intel.com. Refer to the articles, community forums, and links to additional resources in the listed areas of the following Developer Centers:

Tools and Technologies:

- *Threading/Multi-core*
- *Intel® Software Products*

Intel® Processors:

- *Intel® 64 and IA-32 Architectures Software Developer's Manuals*
- *Itanium® Processor Family*
- *Pentium® 4 Processor*
- *Intel® Xeon® Processor*

Environments:

- *High Performance Computing*

Evaluating Performance

Performance Analysis

The high-level information presented in this section discusses methods, tools, and compiler options used for analyzing runtime performance-related problems and increasing application performance.

The topics in this section discuss performance issues and methods from a general point of view, focusing primarily on general performance enhancements. The information in this section is separated in the following topics:

- Using a Performance Enhancement Methodology
- Performance Enhancement Strategies
- Using Intel Performance Analysis Tools

In most cases, other sections and topics in this document contain detailed information about the options, concepts, and strategies mentioned in this section.

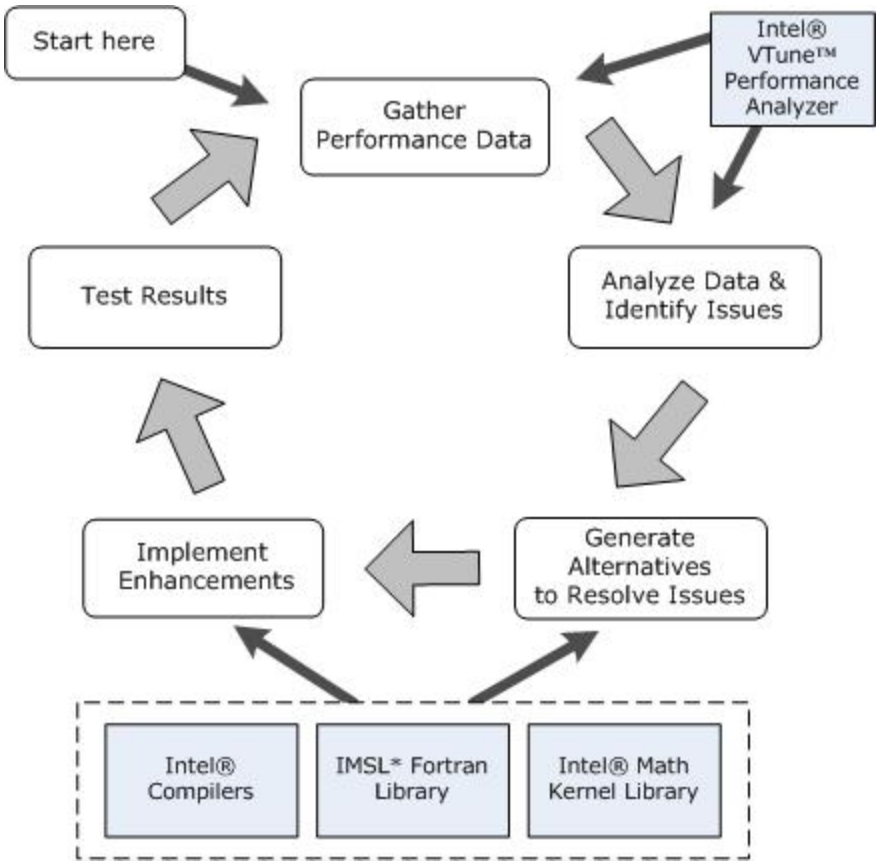
This section also contains information about using the compiler-supporting reports and diagnostics.

Using a Performance Enhancement Methodology

The recommended performance enhancement method for optimizing applications consists of several phases. When attempting to identify performance issues, move through the following general phases in the order presented:

- Gather performance data
- Analyze the data
- Generate alternatives
- Implement enhancements
- Test the results

The following figure shows the methodology phases and their relationships, along with some recommended tools to use in each appropriate phase.



The methodology can be summarized by the following statements:

- Make small changes and measure often.
- If you approach a point of diminishing return and can find no other performance issues, stop optimizing.

Gather performance data

Use tools to measure where performance bottlenecks occur; do not waste time guessing. Using the right tools for analysis provides an objective data set and baseline criteria to measure implementation changes and improvements introduced in the other stages. The VTune™ Performance Analyzer is one tool you can use to gather performance data and quickly identify areas where the code runs slowly, executes infrequently, or executes too frequently (hotspots) when measured as a percentage of time taken against the total code execution.

See Using Intel Performance Analysis Tool and Libraries for more information about some tools you can use to gather performance data.

Analyze the data

Determine if the data meet your expectations about the application performance. If not, choose one performance problem at a time for special interest. Limiting the scope of the corrections is critical in effective optimization.

In most cases, you will get the best results by resolving hotspots first. Since hotspots are often responsible for excessive activity or delay, concentrating on these areas tends to resolve or uncover other performance problems that would otherwise be undetectable.

Use the VTune™ Performance Analyzer, or some other performance tool, to discover where to concentrate your efforts to improve performance.

Generate alternatives

As in the analysis phase, limit the focus of the work. Concentrate on generating alternatives for the one problem area you are addressing. Identify and use tools and strategies to help resolve the issues. For example, you can use compiler optimizations, use Intel® Performance Library routines, or use some other optimization (like improved memory access patterns, reducing or eliminating division or other floating-point operations, rewriting the code to include intrinsics or assembly code, or other strategies).

See Performance Enhancement Strategies for suggestions.

While optimizing for the compiler and source levels, consider using the following strategies in the order presented:

1. Use available supported compiler options. This is the most portable, least intrusive optimization strategy.
2. Use compiler directives embedded in the source. This strategy is not overly intrusive since the method involves including a single line in code, which can be ignored (optionally) by the compiler.
3. Attempt manual optimizations.

Implement enhancements

As with the previous phases, limit the focus of the implementation. Make small, incremental changes. Trying to address too many issues at once can defeat the purpose and reduce your ability to test the effectiveness of your enhancements.

The easiest enhancements will probably involve enabling common compiler optimizations for easy gains. For applications that can benefit from the libraries, consider implementing Intel® Performance Library routines that may require some interface coding.

Test the results

If you have limited the scope of the analysis and implementation, you should see measurable differences in performance in this phase. Have a target performance level in mind so you know when you have reached an acceptable gain in performance.

Use a consistent, reliable test that reports a quantifiable item, like seconds elapsed, frames per second, and so forth, to determine if the implementation changes have actually helped performance.

If you think you can make significant improvement gains or you still have other performance issues to address, repeat the phases beginning with the first one: gather performance data.

Intel® Performance Analysis Tools and Libraries
Intel Corporation offers a variety of performance analysis tools and libraries that can help you optimize your application performance.

Performance Analysis Tools

These performance tools can help you analyze your application, find problem areas, and develop efficient programs. In some cases, these tools are critical to the optimization process.

Tool	Operating System	Description
Intel® VTune™ Performance Analyzer	Linux*, Windows* OS	This is a recommended tool for many optimizations. The VTune™ Performance A collects, analyzes, and provides Intel architecture-specific software performance d the system-wide view down to a specific module, function, and instruction in your The compiler, in conjunction with this tool, can use samples of events monitored b Performance Monitoring Unit (PMU) and create a hardware profiling data to furthe enhance optimizations for some programs. Start with the time-based and event-based sampling functionality of the VTune™ Performance Analyzer. Time-based sampling identifies the sections of code that u most processor time. Event-based sampling identifies microarchitecture bottlenec as cache misses and mispredicted branches. Clock ticks and instruction count are counters to use initially to identify specific functions of interest for tuning. Once you have identified specific functions through sampling, use call-graph anal provide thread-specific reports. Call-graph analysis returns the following informati the functions: • Number of times each function was called • Location from which each function was called • Total processor time spent executing each function You can also use the Counter monitor (which is equivalent to Microsoft* Perfmon* provide real-time performance data based on more than 200 possible operating s

		counters, or you can create custom counters created for specific environments ar
Intel® Threading Analysis Tools	Linux, Windows	Intel® Threading Analysis Tools consist of the Intel® Thread Checker and the Inte Thread Profiler. The Intel® Thread Checker can help identify shared and private variable conflicts isolate threading bugs to the source code line where the bug occurs. The Intel® Thread Profiler can show the critical path of an application as it moves thread to thread, and identify synchronization issues and excessive blocking time cause delays for Win32*, POSIX* threaded, and OpenMP* code.

Performance Libraries

These performance libraries can decrease development time and help to increase application performance.

Library	Operating System	Description
Intel® Math Kernel Library	Linux, Mac OS X, Windows	Intel® Math Kernel Library offers highly optimized, thread-safe math routines for science engineering, and financial applications that require maximum performance.

Performance Enhancement Strategies

Improving performance starts with identifying the characteristics of the application you are attempting to optimize. The following table lists some common application characteristics, indicates the overall potential performance impact you can expect, and provides suggested solutions to try. These strategies have been found to be helpful in many cases; experimentation is key with these strategies.

In the context of this discussion, view the potential impact categories as an indication of the possible performance increases that might be achieved when using the suggested strategy. It is possible that application or code design issues will prohibit achieving the indicated increases; however, the listed impacts are generally true. The impact categories are defined in terms of the following performance increases, when compared to the initially tested performance:

- Significant: more than 50%
- High: up to 50%
- Medium: up to 25%
- Low: up to 10%

The following table is ordered by application characteristics and then by strategy with the most significant potential impact.

Application Characteristics	Impact	Suggested Strategies
Technnical Applications		
Technical applications with loopy code	High	Technical applications are those programs that have some subset of functions that consume a majority of total CPU cycles in loop nests. Target loop nests using `-O3` (Linux* and Mac OS* X) or `/O3` (Windows*) to enable more aggressive loop transformations and prefetching. Use High-Level Optimization (HLO) reporting to determine which HLO optimizations the compiler elected to apply. See High-Level Optimization Report.
(same as above) IA-64 architecture only	High	For `-O2` and `-O3` (Linux) or `/O2` and `/O3` (Windows), use the swp report to determine if Software Pipelining occurred on key loops, and if not, why not. You might be able to change the code to allow software pipelining under the following conditions: • If recurrences are listed in the report that you suspect do not exist, eliminate aliasing problems, or use `IVDEP` directive on the loop. • If the loop is too large or runs out of registers, you might be able to distribute the loop into smaller segments; distribute the loop manually or by using the `distribute` directive. • If the compiler determines the Global Acyclic Scheduler can produce better results but you think the loop should still be pipelined, use the `SWP` directive on the loop.
(same as above) IA-32 and Intel® 64	High	See Vectorization Overview and the remaining topics in the Auto-Vectorization section for

architectures only		applicable options. See Vectorization Report for specific details about when you can change code.
(same as above)	Medium	Use PGO profile to guide other optimizations. See Profile-guided Optimizations Overview.
(same as above) Linux Only	Medium	Use the `-no-swap-pointers` option to specify that the compiler does not byte swap objects of type pointer. Using the option can decrease application size and improve performance.
Applications with many denormalized floating-point value operations	Significant	Experiment with `-fp-model fast=2` (Linux and Mac OS X) or `/fp:fast=2` or `-ftz` (Linux and Mac OS X) or `/Qftz` (Windows). The resulting performance increases can adversely affect floating-point calculation precision and reproducibility. See Floating-point Operations for more information about using the floating point options supported in the compiler.
Sparse matrix applications	Medium	See the suggested strategy for memory pointer disambiguation (below). Use `prefetch` directive or prefetch intrinsics. Experiment with different prefetching schemes on indirect arrays. See HLO Overview or Data Prefetching starting places for using prefetching.
Server application with branch-centric code and a fairly flat profile	Medium	Flat profile applications are those applications where no single module seems to consume CPU cycles inordinately. Use PGO to communicate typical hot paths and functions to the compiler, so the Intel® compiler can arrange code in the optimal manner. Use PGO on as much of the application as is feasible. See Profile-guided Optimizations Overview.

Other Application Types

Applications with many small functions that are called from multiple locations	Low	Use `-ip` (Linux and Mac OS X) or `/Qip` (Windows) to enable inter-procedural inlining within a single source module. Streamlines code execution for simple functions by duplicating the code within the code block that originally called the function. This will increase application size. As a general rule, do not inline large, complicated functions. See Interprocedural Optimizations Overview.
(same as above)	Low	Use `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows) to enable inter-procedural inlining both within and between multiple source modules. You might experience an additional increase over using `-ip` (Linux and Mac OS X) or `/Qip` (Windows). Using this option will increase link time due to the extended program flow analysis that occurs. Use Interprocedural Optimization (IPO) to attempt to perform whole program analysis, which can help memory pointer disambiguation.

Apart from application-specific suggestions listed above, there are many application-, OS/Library-, and hardware-specific recommendations that can improve performance as suggested in the following tables:

Application-specific Recommendations

Application Area	Impact	Suggested Strategies
Cache Blocking	High	Use `-O3` (Linux and Mac OS X) or `/O3` (Windows) to enable automatic cache blocking; use the HLO report to determine if the compiler enabled cache blocking automatically. If not consider manual cache blocking. See Cache Blocking.
Compiler directives for	Medium	Ignore vector dependencies. Use `IVDEP` and other

better alias analysis		directives to increase application speed. See Vectorization Support.
Memory pointer disambiguation compiler options	Medium	Experiment with the following compiler options: • `-fno-fnalias` (Linux and Mac OS X) • `-ansi-alias` (Linux and Mac OS X) or `/Qansi-alias` (Windows) • `-safe-cray-ptr` (Linux and Mac OS X) or `/Qsafe-cray-ptr` (Windows)
Math functions	Low	Call Math Kernel Library (MKL) instead of user code. Call F90 intrinsics instead of user code (to enable optimizations).

Library/OS Recommendations

Area	Impact	Description
Symbol preemption	Low	Linux has a less performance-friendly symbol preemption model than Windows. Linux uses full preemption, and Windows uses no preemption. Use `-fminshared -fvisibility=protected`. See Symbol Visibility Attribute Options.
Memory allocation	Low	Using third-party memory management libraries can help improve performance for applications that require extensive memory allocation.

Hardware/System Recommendations

Component	Impact	Description
Disk	Medium	Consider using more advanced hard drive storage strategies. For example, consider using SCSI instead of IDE. Consider using the appropriate RAID level. Consider increasing the number hard drives in your system.

Memory	Low	You can experience performance gains by distributing memory in a system. For example, if you have four open memory slots and only two slots are populated, populating the other two slots with memory will increase performance.
Processor		For many applications, performance scales is directly affected by processor speed, the number of processors, processor core type, and cache size.

Using Compiler Reports

Compiler Reports Overview

The Intel® compiler provides several reports that can help identify performance issues.

Some of these compiler reports are architecture-specific, most are more general. Start with the general reports that are common to all platforms (operating system/architecture), then use the reports unique to an architecture.

- Generating reports
- Interprocedural Optimizations (IPO) report
- Profile-Guided Optimizations (PGO) report
- High-level Optimizations (HLO) report
- Software-pipelining (SWP) report (IA-64 architecture only)
- Vectorization report
- Parallelism report
- OpenMP* report

Compiler Reports Quick Reference

The Intel® compiler provides the following options to generate and manage optimization reports:

Linux* and Mac OS* X	Windows*	Description
`-opt-report` or `-opt-report N`	`/Qopt-report` or `/Qopt-report:N`	Generates optimization report, with different levels of detail, directed to `stderr`. Valid values for *N* are 0 through 3. By default, when you specify this option without passing a value the compiler will generate a report with a medium level of detail.

`-opt-report-phase`	`/Qopt-report-phase`	Specifies the optimization phase to use when generating reports. If you do not specify a phase the compiler defaults to all, which can adversely affect compile times. See Generating Reports for more information about the supported phases.
`-opt-report-file`	`/Qopt-report-file`	Generates an optimization report and directs the report output to the specified file name. If the file is not in the local directory, supply the full path to the output file. This option overrides the `opt-report` option.
`-opt-report-routine`	`/Qopt-report-routine`	Generates reports from all routines with names containing a string as part of their name; pass the string as an argument to this option. If not specified, the compiler will generate reports on all routines.
`-diag-<type>`	`/Qdiag-<type>`	This option is not a compiler reports option; however, it controls the diagnostic message levels generated by the compiler, and it offers a means to provide useful information without needing to generate reports. `<type>` is a placeholder for several different values or lists.

This quick reference does not list the options for the vectorization, parallelism, or OpenMP* reports.

Refer to Quick Reference Lists for a complete listing of the quick reference topics.

If you use interprocedural optimization (IPO) options, you can request compiler reports when using the xi* tools , as described in Requesting Compiler Reports with the xi* Tools.

Generating Reports

Use the options listed in this topic to generate reports on the following optimizers.

- Interprocedural Optimization (IPO)
- Profile-guided Optimization (PGO)
- High Performance Optimizer (HPO)
- High-level Optimization (HLO)
- Intermediate Language Scalar Optimization (ILO)
- Software Pipelining (SWP)

Specify an optimizer phase by passing the *`phase`* argument to the `-opt-report-phase` (Linux* and Mac OS* X) or `/Qopt-report-phase` (Windows*) option.

Optimizer Phase	Supported Optimizer
pgo	Profile-guided Optimizer
ipo	Interprocedural Optimizer
ilo	Intermediate Language Scalar Optimizer
hpo	High Performance Optimizer
hlo	High-level Optimizer
ecg	Itanium® Compiler Code Generator Mac OS X: This phase is not supported.
ecg_swp	The software pipelining component of the Code Generator phase (Windows and Linux systems using IA-64 architecture only)
all	All optimizers supported on the architecture. This is not recommended; the resulting output can be too extensive to be useful. Experiment with targeted phase reports first.

Reports Available by Architecture and Option

IA-32, Intel® 64, and IA-64 architectures:

- `ilo` and `pgo`
- `hpo` and `hlo`: For IA-32 architecture, supported with `-x` (Linux and Mac OS X) or `/Qx` (Windows) option.
  For Intel® 64 architecture, supported with `-O2` (Linux and Mac OS X) or `/O2` (Windows) option. For IA-32 and Intel® 64 architectures, a subset of these optimizations are enabled at default optimization level (`O2`). For IA-64 architecture, supported with `-O3` (Linux and Mac OS X) or `/O3` (Windows) option.
- `ipo`: Interprocedural optimization is enabled for `-O2` (Linux and Mac OS X) or `/O2` (Windows) option or above.
- `all`: All of the above.

IA-64 architecture only:

- `ecg`

Running the Reports

Use syntax similar to the following to run the compiler reports.

Operating System	Sample Syntax

Linux and Mac OS X	`ifort -c -opt-report 2 -opt-report-phase=all sample.f90`
Windows	`ifort /c /Qopt-report:2 /Qopt-report-phase:all sample.f90`

The sample command instructs the compiler to generate a report and send the results to `stderr` and specifies the reports should include information about all available optimizers. In most cases, specifying `all` as the phase will generate too much information to be useful.

If you want to capture the report in an output file instead of sending it to `stderr`, specify `-opt-report-file` (Linux and Mac OS X) or `/Qopt-report-file` (Windows) and indicate an output file name. If you do not want the compiler to invoke the linker, specify `-c` (Linux and Mac OS X) or `/c` (Windows) as shown in the sample syntax above; by specifying the option you instruct the compiler to stop after generating object code and reporting the results.

See Compiler Reports Quick Reference for information about how to use the report related options.

When you specify a phase name, as shown above, the compiler generates all reports from that optimizer. The option can be used multiple times on the same command line to generate reports for multiple optimizers. For example, for if you specified `-opt-report-phase` *`ipo`* `-opt-report-phase` *`hlo`* (Linux and Mac OS X) or `/Qopt-report-phase` *`ipo`* `/Qopt-report-phase` *`hlo`* (Windows) the compiler generates reports from the interprocedural optimizer and the high-level optimizer code generator.

You do not need to fully specify an optimizer name in the command; in many cases, the first few characters should suffice to generate reports; however, all optimization reports that have a matching prefix are generated.

Each of the optimizer logical names supports many specific, targeted optimizations within them. Each of the targeted optimizations have the prefix of the optimizer name. Enter `-opt-report-help` (Linux and Mac OS X) or `/Qopt-report-help` (Windows) to list the names of optimizers that are supported. The following table lists some examples:

Optimizer	Description
`ipo_inl`	Interprocedural Optimizer, inline expansion of functions
`ipo_cp`	Interprocedural Optimizer, constant propagation
`hlo_unroll`	High-level Optimizer, loop unrolling
`hlo_prefetch`	High-level Optimizer, prefetching

Viewing the Optimization Reports Graphically (Linux)

In addition to the text-based optimization reports, the Intel compiler can now generate the necessary information to allow the OptReport feature in Intel® VTune™ Linux 8.x to display the optimization reports graphically. See the VTune™ Performance Analyzer documentation for details.

To generate the graphical report display, you must compile the application using the following optimization reports options, at a minimum: `-opt-report-phase` and `-opt-report-file`.

As with the text-based reports, the graphical report information can be generated on all architectures.

Interprocedural Optimizations (IPO) Report

The IPO report provides information on the functions that have been inlined and can help to identify the problem loops. The report can help to identify how and where the compiler applied IPO to the source files.

The following command examples demonstrate how to run the IPO reports with the minimum output.

Operating System	Syntax Examples
Linux* and Mac OS* X	`ifort -opt-report 1 -opt-report-phase=ipo a.f90 b.f90`
Windows*	`ifort /Qopt-report:1 /Qopt-report-phase:ipo a.f90 b.f90`

where `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows) invokes the report generator, and `-opt-report-phase=`*`ipo`* (Linux and Mac OS X) or `/Qopt-report-phase:`*`ipo`* (Windows) indicates the phase (`ipo`) to report.

You can use `-opt-report-file` (Linux and Mac OS X) or `/Qopt-report-file` (Windows) to specify an output file to capture the report results. Specifying a file to capture the results can help to reduce the time you spend analyzing the results and can provide a baseline for future testing.

Reading the Output

The IPO report details information in two general sections: whole program analysis and inlining. By default, the report generates a medium level of detail. You can specify an output file to capture the report results. Running maximum IPO report results can be very extensive and technical; specifying a file to capture the results can help to reduce analysis time. The following sample report illustrates the general layout.

Sample IPO Report

```
<;-1:-1;IPO;;0>
WHOLE PROGRAM (SAFE) [EITHER METHOD]: TRUE
WHOLE PROGRAM (SEEN) [TABLE METHOD]: TRUE
```

```
WHOLE PROGRAM (READ) [OBJECT READER METHOD]: TRUE
INLINING OPTION VALUES:
  -inline-factor: 100
  -inline-min-size: 20
  -inline-max-size: 230
  -inline-max-total-size: 2000
  -inline-max-per-routine: disabled
  -inline-max-per-compile: disabled
<ipo sample main.f90;29:37;IPO INLINING;MAIN  ;0>
INLINING REPORT: (MAIN  ) [1/4=25.0%]
  -> for write seq lis(EXTERN)
  -> INLINE: mysum (5) (isz = 23) (sz = 30 (13+17))
...
<ipo sample init.f90;32:40;IPO CONSTANT PROPAGATION;init ;0>
  CONSTANT PROPAGATION:  VARS(1)
...
```

The following table summarizes the common report elements and provides a general description to help interpret the results.

Report Element	Description
`WHOLE PROGRAM (SAFE) [EITHER METHOD]:`	`TRUE` or `FALSE`. • TRUE: The compiler determined, using one or both of the whole program analysis models, that the whole program was present during compilation. • FALSE: The compiler determined the whole program was not present during compilation. The compiler could not apply whole program IPO. See Inteprocedural Optimizations (IPO) Overview for more information on the whole program analysis models.
`WHOLE PROGRAM (SEEN) [TABLE METHOD]:`	`TRUE` or `FALSE`. • TRUE: The compiler resolved all references, either within the application code or in the standard table for the functions within the compiler. • FALSE: The compiler could not resolve all references. One or more functions references could not be found either in the user code or standard functions table.
`WHOLE PROGRAM (READ) [OBJECT READER METHOD]:`	`TRUE` or `FALSE`. • TRUE: The compiler determined that all conditions were met for linking at a level equivalent to the `-O0` (Linux and Mac OS X) or `/Od` (Windows) option. • FALSE: The compiler could not resolve one or more references. The linking step failed.
`INLINING OPTION`	Displays the compilation values used for the following

`VALUES:`

developer-directed inline expansion options:

- inline-factor
- inline-min-size
- inline-max-size
- inline-max-total-size
- inline-max-per-routine
- inline-max-per-compile

If you specify the one or more of the appropriate options, the report lists the values you specified; if you do not specify an option and value the compiler uses the defaults values for the listed options, and the compiler will list the default values.

The values indicate the same intermediate language units listed in Compiler Options for each of these options. See Developer Directed Expansion of User Functions for more information about using these options.

`INLINING REPORT:`

Includes a string in the format of the following

`(<name>) [<current number>/<total number>=<percent complete>]`

where

- `<name>`: the name of the function being reported on.
- `<current number>` is the number of the function being reported on. Not every function can be inlined; gaps in the current number are common.
- `<total number>` is the total number of functions being evaluated.
- `<percent complete>` is a simple percentage of the functions being inlined.

`-> INLINE:`

If a function is inlined, the function line has the prefix "`-> INLINE: _`".

The option reports displays the names of the functions.

The report uses the following general syntax format:

```
-> INLINE: _<name>(#) (isz) (sz)
```

where

- `<name>` : Indicates the name of the function Static functions display the function name with numbered suffix.
- `#`: Indicates the unique integer specifying the function number.
- `sz`: Indicates the function size before optimization. This is a rough estimate loosely representative of the number of original instructions before optimization.
- `isz`: Indicates the function size after optimization. This value (isz) will always be less than or equal to the unoptimized size (sz).
- `exec_cnt`: Indicates that Profile-guided Optimization was specified during compilation. Indicates the number of times the function was called from this site.

Function calls that could not be inlined lack the `INLINE` prefix. Additionally, the compiler marks non-inlined functions using the following conventions:

- `EXTERN` indicates the function contained an external function call for which the compiler was not supplied the code.
- `ARGS_IN_REGS` indicates inlining did not occur. For IA-32 and Intel® 64 architectures, an alternative for functions that were not inlined is to allow the compiler to pass the function arguments in registers rather than using standard calling conventions; however, for IA-64 architecture this is the default behavior.

`DEAD STATIC FUNCTION ELIMINATION:` Indicates the reported function is a dead static. Code does not need to be created for these functions. This behavior allows the compiler to reduce the overall code size.

Profile-guided Optimization (PGO) Report

The PGO report can help to identify can help identify where and how the compiler used profile information to optimize the source code. The PGO report is most useful when combined with the PGO compilation steps outlined in Profile an Application. Without the

profiling data generated during the application profiling process the report will generally not provide useful information.

Combine the final PGO step with the reporting options by including `-prof-use` (Linux* and Mac OS* X) or `/Qprof-use` (Windows*). The following syntax examples demonstrate how to run the report using the combined options.

Operating System	Syntax Examples
Linux and Mac OS X	`ifort -prof-use -opt-report -opt-report-phase=pgo pgotools_sample.f90`
Windows	`ifort /Qprof-use /Qopt-report /Qopt-report-phase:pgo pgotools_sample.f90`

By default the PGO report generates a medium level of detail. You can use `-opt-report-file` (Linux and Mac OS X) or `/Qopt-report-file` (Windows) to specify an output file to capture the report results. Specifying a file to capture the results can help to reduce the time you spend analyzing the results and can provide a baseline for future testing.

Reading the Output

Running maximum PGO report results can produce long and detailed results. Depending on the sources being profiled, analyzing the report could be very time consuming. The following sample report illustrates typical results and element formatting for the default output.

Sample PGO Report

```
<pgotools sample.f90;-1:-1;PGO; DELEGATE.;0>
  DYN-VAL:  pgotools sample.f90   DELEGATE.
<pgotools sample.f90;-1:-1;PGO; ADDERMOD.;0>
  NO-DYN:   pgotools sample.f90   ADDERMOD.
<pgotools sample.f90;-1:-1;PGO; ADDERMOD mp MOD ADD;0>
  DYN-VAL:  pgotools sample.f90   ADDERMOD mp MOD ADD
<pgotools sample.f90;-1:-1;PGO; DELEGATE;0>
  DYN-VAL:  pgotools sample.f90   DELEGATE
<pgotools sample.f90;-1:-1;PGO; MAIN  ;0>
  DYN-VAL:  pgotools sample.f90   MAIN
<pgotools sample.f90;-1:-1;PGO; MAIN. MAIN ADD;0>
  DYN-VAL:  pgotools sample.f90   MAIN. MAIN ADD
<pgotools sample.f90;-1:-1;PGO;;0>
      5 FUNCTIONS HAD VALID DYNAMIC PROFILES
      1 FUNCTIONS HAD NO DYNAMIC PROFILES
  FILE CURRENT  QUALITY METRIC:  91.7%
  FILE POSSIBLE QUALITY METRIC:  91.7%
  FILE QUALITY  METRIC RATIO:   100.0%
```

The following table summarizes some of the common report elements and provides a general description to help interpret the results.

Report Element	Description
String listing information about the function being reported on. The string uses	The compact string contains the following information: • <source name>: Name of the source file being

the following format. `<source name>;<start line>;<end line>;<optimization>;<function name>;<element type>`	examined. • <start line>: Indicates the starting line number for the function being examined. A value of -1 means that the report applies to the entire function. • <end line>: Indicates the ending line number for the function being examined. • <optimization>: Indicates the optimization phase; for this report the indicated phase should be PGO. • <function name>: Indicates the name of the function. • <element type>: Indicates the type of the report element; 0 indicates the element is a comment.
`DYN-VAL`	Indicates that valid profiling data was generated for the function indicated; the source file containing the function is also listed.
`NO-DYN`	Indicates that no profiling data was generated for the function indicated; the source file containing the function is also listed.
`FUNCTIONS HAD VALID DYNAMIC PROFILES`	Indicates the number of functions that had valid profile information.
`FUNCTIONS HAD NO DYNAMIC PROFILES`	Indicated the number of functions that did not have valid profile information. This element could indicate that the function was not executed during the instrumented executable runs.
`FUNCTIONS HAD VALID STATIC PROFILES`	Indicates the number of functions for which static profiles were generated. The most likely cause for having a non-zero number is that dynamic profiling did not happen and static profiles were generated for all of the functions.
`IPO CURRENT QUALITY METRIC`	Indicates the general quality, represented as a percentage value between 50% and 100%t. A value of 50% means no functions had dynamic profiles, and a value of 100% means that all functions have dynamic profiles. The larger the number the greater the percentage of functions that

	had dynamic profiles.
`IPO POSSIBLE QUALITY METRIC`	Indicates the number of possible dynamic profiles. This number represent the best possible value, as a percentage, for `Current Quality`. This number is the highest value possible and represents the ideal quality for the given data set and the instrumented executable.
`IPO QUALITY METRIC RATIO`	Indicates the ratio of `Possible Quality` to `Current Quality`. A value of 100% indicates that all dynamic profiles were accepted. Any value less than 100% indicates rejected profiles.

High-level Optimization (HLO) Report

High-level Optimization (HLO) performs specific optimizations based on the usefulness and applicability of each optimization. The HLO report can provide information on all relevant areas plus structure splitting and loop-carried scalar replacement, and it can provide information about interchanges not performed for the following reasons:

- Function call are inside the loop
- Imperfect loop nesting
- Reliance on data dependencies; dependencies preventing interchange are also reported.
- Original order was proper but it might have been considered inefficient to perform the interchange.

For example, the report can provide clues to why the compiler was unable to apply loop interchange to a loop nest that might have been considered a candidate for optimization. If the reported problems (bottlenecks) can be removed by changing the source code, the report suggests the possible loop interchanges.

Depending on the operating system, you must specify the following options to enable HLO and generate the reports:

- Linux* and Mac OS* X: `-x`, `-O2` or `-O3`, `-opt-report 3`, `-opt-report-phase=hlo`
- Windows*: `/Qx`, `/O2` or `/O3`, `/Qopt-report:3`, `/Qopt-report-phase:hlo`

See High-level Optimization Overview for information about enabling HLO.

The following command examples illustrate the general command needed to create HLO report with combined options.

Operating System	Example Command

Linux and Mac OS X	`ifort -c -xSSE3 -O3 -opt-report 3 -opt-report-phase=hlo sample.f90`
Windows	`ifort /c /QxSSE3 /O3 /Qopt-report:3 /Qopt-report-phase:hlo sample.f90`

You can use `-opt-report-file` (Linux and Mac OS X) or `/Qopt-report-file` (Windows) to specify an output file to capture the report results. Specifying a file to capture the results can help to reduce the time you spend analyzing the results and can provide a baseline for future testing.

Reading the report results

The report provides information using a specific format. The report format for Windows* is different from the format on Linux* and Mac OS* X. While there are some common elements in the report output, the best way to understand what kinds of advice the report can provide is to show example code and the corresponding report output.

Example 1: This example illustrates the condition where a function call is inside a loop.

Example 1

```
subroutine foo (A, B, bound)
   integer i,j,n,bound
   integer A(bound), B(bound,bound)
   n = bound
   do j = 1, n
      do i = 1, n
        B(j,i) = B(j,i) + A(j)
        call bar(A,B)
      end do
   end do
   return
end subroutine foo
```

Regardless of the operating system, the reports list optimization results on specific functions by presenting a line above there reported action. The line format and description are included below.

The following table summarizes the common report elements and provides a general description to help interpret the results.

Report Element	Description
String listing information about the function being reported on. The string uses the following format. `<source name>;<start line>;<end line>;<optimization>;<function name>;<element type>` For example, the reports	The compact string contains the following information: • <source name>: Name of the source file being examined. • <start line>: Indicates the starting line number for the function being examined. A value of -1 means that the report applies to the entire function. • <end line>: Indicates the ending line number for the function being examined.

listed below report the following information:

Linux and Mac OS X:

```
<sample1.f90;-1:-
1;hlo;foo_;0>
```

Windows:

```
<sample1.f90;-1:-
1;hlo;_FOO;0>
```

- <optimization>: Indicates the optimization phase; for this report the indicated phase should be hlo.
- <function name>: Name of the function being examined.
- <element type>: Indicates the type of the report element; 0 indicates the element is a comment.

Several report elements grouped together.

```
QLOOPS 2/2      ENODE
LOOPS 2 unknown 0
multi exit do 0 do 2
linear do 2
LINEAR HLO
EXPRESSIONS:  17 / 18
```

Windows only: This section of the report lists the following information:

- `QLOOPS`: Indicates the number of well-formed loops found out of the loops discovered.
- `ENODE LOOPS`: Indicates number of preferred forms (canonical) of the loops generated by HLO. This indicates the number of loops generated by HLO.
- `unknown`: Indicates the number of loops that could not be counted.
- `multi_exit_do`: Indicates the countable loops containing multiple exits.
- `do`: Indicates the total number of loops with trip counts that can be counted.
- `linear_do`: Indicates the number of loops with bounds that can be represented in a linear form.
- `LINEAR HLO EXPRESSIONS`: Indicates the number of expressions (first number) in all of the intermediate forms (ENODE) of the expression (second number) that can be represented in a linear form.

The code sample list above will result in a report output similar to the following.

Operating System	Example 1 Report Output

Linux and Mac OS X	`<sample1.f90;-1:-1;hlo;foo ;0>`
	`High Level Optimizer Report (foo )`
	
	`Block, Unroll, Jam Report:`
	`(loop line numbers, unroll factors and type of transformation)`
	
	`<sample1.f90;7:7;hlo unroll;foo ;0>`
	`Loop at line 7 unrolled with remainder by 2`

Windows	`<sample1.f90;-1:-1;hlo; FOO;0>`
	`High Level Optimizer Report ( FOO)`
	`QLOOPS 2/2      ENODE LOOPS 2 unknown 0 multi_exit_do 0 do 2 linear_do`

```
2
LINEAR HLO EXPRESSIONS:  17 / 18
------------------------------------------------------------------------
-------

C:\samples\sample1.f90;6:6;hlo linear trans; FOO;0>
Loop Interchange not done due to: User Function Inside Loop Nest
Advice: Loop Interchange, if possible, might help Loopnest at lines: 6
7
      : Suggested Permutation: (1 2 ) --> ( 2 1 )
```

Example 2: This example illustrates the condition where the loop nesting prohibits interchange.

Example 2

```
subroutine foo (A, B, bound)
   integer i,j,n,bound
   integer A(bound), B(bound,bound)
   n = bound
   do j = 1, n
      A(j) = j + B(1,j)
      do i = 1, n
        B(j,i) = B(j,i) + A(j)
      end do
   end do
   return
end subroutine foo
```

The code sample listed above will result in a report output similar to the following.

Operating System	Example 2 Report Output

Linux and Mac OS X	`<sample2.f90;-1:-1;hlo;foo ;0>`
	`High Level Optimizer Report (foo )`
	
	`Block, Unroll, Jam Report:`
	`(loop line numbers, unroll factors and type of transformation)`
	
	`<sample2.f90;8:8;hlo unroll;foo ;0>`
	`Loop at line 8 unrolled with remainder by 2`

Windows	`<sample2.f90;-1:-1;hlo; FOO;0>`
	`High Level Optimizer Report ( FOO)`
	`QLOOPS 2/2  ENODE LOOPS 2 unknown 0 multi exit do 0 do 2 linear do 2`
	`LINEAR HLO EXPRESSIONS:  24 / 24`
	`------------------------------------------------------------------------`
	`-----`
	
	`C:\samples\sample2.f90;6:6;hlo linear trans; FOO;0>`
	`Loop Interchange not done due to: Imperfect Loop Nest (Either at Source`
	`or due t`
	`o other Compiler Transformations)`
	`Advice: Loop Interchange, if possible, might help Loopnest at lines: 6 8`
	`  : Suggested Permutation: (1 2 ) --> ( 2 1 )`

Example 3: This example illustrates the condition where data dependence prohibits loop interchange.

Example 3

```
subroutine foo (bound)
   integer i,j,n,bound
   integer A(100,100), B(100,100), C(100,100)
   equivalence (B(2),A)
   n = bound
   do j = 1, n
      do i = 1, n
        A(j,i) = C(j,i) * 2
```

```
        B(j,i) = B(j,i) + A(j,i) * C(j,i)
      end do
   end do
   return
end subroutine foo
```

The code sample listed above will result in a report output similar to the following.

Operating System	Example 3 Report Output

Linux and Mac OS X	`<sample3.f90;-1:-1;hlo;foo ;0>`
	`High Level Optimizer Report (foo )`
	
	`<sample3.f90;8:8;hlo scalar replacement;in foo ;0>`
	`#of Array Refs Scalar Replaced in foo  at line 8=2`
	
	`Block, Unroll, Jam Report:`
	`(loop line numbers, unroll factors and type of transformation)`
	
	`<sample3.f90;8:8;hlo unroll;foo ;0>`
	`Loop at line 8 unrolled with remainder by 2`

Windows	`<sample3.f90;-1:-1;hlo; FOO;0>`
	`High Level Optimizer Report ( FOO)`
	`QLOOPS 2/2  ENODE LOOPS 2 unknown 0 multi exit do 0 do 2 linear do 2`
	`LINEAR HLO EXPRESSIONS:  24 / 24`
	`----------`
	
	`C:\samples\sample3.f90;8:8;hlo scalar replacement;in  FOO ;0>`
	`#of Array Refs Scalar Replaced in  FOO at line 8=1`
	`C:\samples\3.f90;7:7;hlo linear trans; FOO;0>`
	`Loop Interchange not done due to: Data Dependencies`
	`  Dependencies found between following statements:`
	`  [From Line# -> (Dependency Type) To Line#]`
	`  [9 ->(Flow) 10] [9 ->(Output) 10] [10 ->(Anti) 10]`
	`  [10 ->(Anti) 9] [10 ->(Output) 9]`
	`Advice: Loop Interchange, if possible, might help Loopnest at lines: 7 8`
	`  : Suggested Permutation: (1 2 ) --> ( 2 1 )`

Example 4: This example illustrates the condition where the loop order was determined to be proper, but loop interchange might offer only marginal relative improvement.

Example 4

```
subroutine foo (A, B, bound, value)
   integer i,j,n,bound,value
   integer A(bound, bound), B(bound,bound)
   n = bound
   do j = 1, n
      do i = 1, n
        A(i,j) = A(i,j) + B(j,i)
      end do
   end do
   value = A(1,1)
   return
end subroutine foo
```

The code sample listed above will result in a report output similar to the following.

Operating System	Example 4 Report Output

Linux and Mac	`<sample4.f90;-1:-1;hlo;foo ;0>`
	`High Level Optimizer Report (foo )`
	
	`Block, Unroll, Jam Report:`

OS X

```
(loop line numbers, unroll factors and type of transformation)

<sample4.f90;7:7;hlo unroll;foo ;0>
Loop at line 7 unrolled with remainder by 2
```

Windows

```
<sample4.f90;-1:-1;hlo; FOO;0>
High Level Optimizer Report ( FOO)
QLOOPS 2/2      ENODE LOOPS 2 unknown 0 multi exit do 0 do 2
linear do 2
LINEAR HLO EXPRESSIONS:  18 / 18
```

Example 5: This example illustrates the conditions where the loop nesting was imperfect and the loop order was good, but loop interchange would offer marginal relative improvements.

Example

```
subroutine foo (A, B, C, bound, value)
   integer i,j,n,bound,value
   integer A(bound, bound), B(bound,bound), C(bound, bound)
   n = bound
   do j = 1, n
      value = value + A(1,1)
      do i = 1, n
        value = B(i,j) + C(j,i)
      end do
   end do
   return
end subroutine foo
```

The code sample listed above will result in a report output similar to the following.

Operating System	Example 5 Report Output

Linux and Mac OS X	`<sample5.f90;-1:-1;hlo;foo ;0>`
	`High Level Optimizer Report (foo )`
	
	`Loopnest Preprocessing Report:`
	
	`<sample5.f90;7:8;hlo;foo ;0>`
	`Preprocess Loopnests <foo_>: Moving Out Store @Line<8> in Loop @Line<7>`

Windows	`<sample5.f90;-1:-1;hlo; FOO;0>`
	`High Level Optimizer Report ( FOO)`
	`QLOOPS 2/2      ENODE LOOPS 2 unknown 0 multi exit do 0 do 2 linear do 2`
	`LINEAR HLO EXPRESSIONS:  20 / 25`
	`----------`
	`------`
	
	`Loopnest Preprocessing Report:`
	`C:\samples\sample5.f90;7:8;hlo; FOO;0>`
	`Preprocess Loopnests < FOO>: Moving Out Store @Line<8> in Loop @Line<7>`
	`C:\samples\sample5.f90;5:5;hlo linear trans; FOO;0>`
	`Loop Interchange not done due to: Imperfect Loop Nest (Either at Source or due t`
	`o other Compiler Transformations)`
	`Advice: Loop Interchange, if possible, might help Loopnest at lines: 5 7`
	`      : Suggested Permutation: (1 2 ) --> ( 2 1 )`

Changing Code Based on the Report Results

While the HLO report tells you what loop transformations the compiler performed and provides some advice, the omission of a given loop transformation might imply that there are transformations the compiler might attempt. The following list suggests some

transformations you might want to apply. (Manual optimization techniques, like manual cache blocking, should be avoided or used only as a last resort.)

- Loop Interchanging - Swap the execution order of two nested loops to gain a cache locality or unit-stride access performance advantage.
- Distributing - Distribute or split up one large loop into two smaller loops. This strategy might provide an advantage when too many registers are being consumed in a large loop.
- Fusing - Fuse two smaller loops with the same trip count together to improve data locality.
- Loop Blocking - Use cache blocking to arrange a loop so it will perform as many computations as possible on data already residing in cache. (The next block of data is not read into cache until computations using the first block are finished.)
- Unrolling - Unrolling is a way of partially disassembling a loop structure so that fewer numbers of iterations of the loop are required; however, each resulting loop iteration is larger. Unrolling can be used to hide instruction and data latencies, to take advantage of floating point loadpair instructions, and to increase the ratio of real work done per memory operation.
- Prefetching - Request the compiler to bring data in from relatively slow memory to a faster cache several loop iterations ahead of when the data is actually needed.
- Load Pairing - Use an instruction to bring two floating point data elements in from memory in a single step.

High Performance Optimizer (HPO) Report

The following command examples illustrate the general command needed to create HPO report with combined options.

Operating System	Example Command
Linux* and Mac OS* X	`ifort -opt-report 0 -opt-report-phasehpo sample.f90`
Windows*	`ifort /Qopt-report:0 /Qopt-report-phase:hpo sample.f90`

Use `-opt-report-help` (Linux and Mac OS X) or `/Qopt-report-help` (Windows) to list the names of HPO report categories.

You must specify different compiler options to use the specific HPO report categories.

- For OpenMP*, add `-openmp` (Linux and Mac OS X) or `/Qopenmp` (Windows) to the command line.
- For parallelism, add `-parallel` (Linux and Mac OS X) or `/Qparallel` (Windows) to the command line.

- For vectorization, add `-x` (Linux and Mac OS X) or `/Qx` (Windows) to the command line; valid processor values are `SSE4.1`, `SSSE3`, `SSE3`, and `SSE2`.

Parallelism Report

The `-par-report` (Linux* and Mac OS* X) or `/Qpar-report` (Windows*) option controls the diagnostic levels 0, 1, 2, or 3 of the auto-parallelizer. Specify a value of 3 to generate the maximum diagnostic details.

Run the diagnostics report by entering commands similar to the following:

Operating System	Commands
Linux and Mac OS X	`ifort -c -parallel -par-report 3 sample.f90`
Windows	`ifort /c /Qparallel /Qpar-report:3 sample.f90`

where `-c` (Linux and Mac OS X) or `/c` (Windows) instructs the compiler to compile the example without generating an executable.

Note

Linux and Mac OS X: The space between the option and the phase is optional.
Windows: The colon between the option and phase is optional.

For example, assume you want a full diagnostic report on the following example code:

Example

```
subroutine no par(a, MAX)
   integer :: i, a(MAX)
   do i = 1, MAX
     a(i) = mod((i * 2), i) * 1 + sqrt(3.0)
     a(i) = a(i-1) + i
   end do
end subroutine no_par
```

The following example output illustrates the diagnostic report generated by the compiler for the example code shown above. In most cases, the comment listed next to the line is self-explanatory.

Example Report Output

```
procedure: NO PAR
sample.f90(7):(4) remark #15049: loop was not parallelized: loop is not
a parallelization candidate
sample.f90(7):(4) remark #15050: loop was not parallelized: existence
of parallel dependence
sample.f90(13):(6) remark #15051: parallel dependence: proven FLOW
dependence between A line 13, and A line 13
```

Responding to the results

The `-par-threshold{n}` (Linux* and Mac OS* X) or `/Qpar-threshold[:n]` (Windows*) 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. You can use `-par-threshold0` (Linux and Mac OS X) or `/Qpar-threshold:0` (Windows) to auto-parallelize loops regardless of computational work.

Use `-ipo[value]` (Linux and Mac OS X) or `/Qipo` (Windows) to eliminate assumed side-effects done to function calls.

Use the `!DEC$ PARALLEL` directive to eliminate assumed data dependency.

Software Pipelining (SWP) Report (Linux* and Windows*)

The SWP report can provide details information about loops currently taking advantage of software pipelining available on IA-64 architecture based systems. The report can suggest reasons why the loops are not being pipelined.

The following command syntax examples demonstrates how to generate a SWP report for the Itanium® Compiler Code Generator (ECG) Software Pipeliner (SWP).

Operating System	Syntax Examples
Linux*	`ifort -c -opt-report -opt-report-phase=ecg swp sample.f90`
Windows*	`ifort /c /Qopt-report /Qopt-report-phase:ecg swp sample.f90`

where `-c` (Linux) or `/c` (Windows) tells the compiler to stop at generating the object code (no linking occurs), `-opt-report` (Linux) or `/Qopt-report` (Windows) invokes the report generator, and `-opt-report-phase=`*`ecg_swp`* (Linux) or `/Qopt-report-phase:`*`ecg_swp`* (Windows) indicates the phase (ecg) for which to generate the report.

You can use `-opt-report-file` (Linux) or `/Qopt-report-file` (Windows) to specify an output file to capture the report results. Specifying a file to capture the results can help to reduce the time you spend analyzing the results and can provide a baseline for future testing.

Typically, loops that software pipeline will have a line that indicates the compiler has scheduled the loop for SWP in the report. If the `-O3` (Linux) or `/O3` (Windows) option is specified, the SWP report merges the loop transformation summary performed by the loop optimizer.

Some loops will not software pipeline (SWP) and others will not vectorize if function calls are embedded inside your loops. One way to get these loops to SWP or to vectorize is to inline the functions using IPO.

You can compile this example code to generate a sample SWP report. The sample reports is also shown below.

Example
`!#define NUM 1024`

```
subroutine multiply d(a,b,c,NUM)
  implicit none
  integer :: i,j,k,NUM
  real :: a(NUM,NUM), b(NUM,NUM), c(NUM,NUM)
  NUM=1024
  do i=0,NUM
    do j=0,NUM
      do k=0,NUM
        c(j,i) = c(j,i) + a(j,k) * b(k,i)
      end do
    end do
  end do
end subroutine multiply_d
```

The following sample report shows the report phase that results from compiling the example code shown above (when using the `ecg_swp` phase).

Sample SWP Report

```
Swp report for loop at line 10 in  Z10multiply dPA1024 dS0 S0  in file SWP report.f90
 Resource II  = 2
 Recurrence II = 2
 Minimum II  = 2
 Scheduled II = 2
 Estimated GCS II   = 7
 Percent of Resource II needed by arithmetic ops  = 100%
 Percent of Resource II needed by memory ops  =  50%
 Percent of Resource II needed by floating point ops =  50%
 Number of stages in the software pipeline = 6
```

Reading the Reports

To understand the SWP report results, you must know something about the terminology used and the related concepts. The following table describes some of the terminology used in the SWP report.

Term	Definition
II	Initiation Interval (II). The number of cycles between the start of one iteration and the next in the SWP. The presence of the term II in any SWP report indicates that SWP succeeded for the loop in question. II can be used in a quick calculation to determine how many cycles your loop will take, if you also know the number of iterations. Total cycle time of the loop is approximately N * Scheduled II + number Stages (Where N is the number of iterations of the loop). This is an approximation because it does not take into account the ramp-up and ramp-down of the prolog and epilog of the SWP, and only considers the kernel of the SWP loop. As you modify your code, it is generally better to see scheduled II go down, though it is really N* (Scheduled II) + Number of stages in the software pipeline that is ultimately the figure of merit.
Resource II	Resource II implies what the Initiation Interval should be when considering the number of functional units available.
Recurrence	Recurrence II indicates what the Initiation Interval should be when there is

II	a recurrence relationship in the loop. A recurrence relationship is a particular kind of a data dependency called a flow dependency like `a[i] = a[i-1]` where `a[i]` cannot be computed until `a[i-1]` is known. If Recurrence II is non-zero and there is no flow dependency in the code, then this indicates either Non-Unit Stride Access or memory aliasing. See Helping the Compiler for more information.
Minimum II	Minimum II is the theoretical minimum Initiation Interval that could be achieved.
Scheduled II	Scheduled II is what the compiler actually scheduled for the SWP.
number of stages	Indicates the number of stages. For example, in the report results below, the line `Number of stages in the software pipeline = 3` indicates there were three stages of work, which will show, in assembly, to be a load, an FMA instruction and a store.
loop-carried memory dependence edges	The loop-carried memory dependence edges means the compiler avoided WAR (Write After Read) dependency. Loop-carried memory dependence edges can indicate problems with memory aliasing. See Helping the Compiler.

Using the Report to Resolve Issues

One fast way to determine if specific loops have been software pipelined is to look for ”r;`Number of stages in the software pipeline`” in the report; the phrase indicates that software pipelining for the associated loop was successfully applied.

Analyze the loops that did not SWP in order to determine how to enable SWP. If the compiler reports the `Loop was not SWP because...`, see the following table for suggestions about how to correct possible problems:

Message in Report	Suggested Action
`acyclic global scheduler can achieve a better schedule: => loop not pipelined`	Indicates that the most likely cause is memory aliasing issues. For memory alias problems see memory aliasing (restrict, `#pragma ivdep`). Might indicate the application is accessing memory in a non-Unit Stride fashion. Non-Unit Stride issues may be indicated by an artificially high recurrence II; If you know there is no recurrence relationship (`a[i] = a[i-1] + b[i]`) in the loop, then a high recurrence II (greater than 0) is a sign that you are

	accessing memory non-Unit Stride. Rearranging code, perhaps a loop interchange, might help mitigate this problem.
`Loop body has a function call`	Indicates inlining the function might help solve the problem.
`Not enough static registers`	Indicates you should distribute the loop by separating it into two or more loops.
`Not enough rotating registers`	Indicates the loop carried values use the rotating registers. Distribute the loop.
`Loop too large`	Indicates you should distribute the loop.
`Loop has a constant trip count < 4`	Indicates unrolling was insufficient. Attempt to fully unroll the loop. However, with small loops fully unrolling the loop is not likely to affect performance significantly.
`Too much flow control`	Indicates complex loop structure. Attempt to simplify the loop.

Index variable type used can greatly impact performance. In some cases, using loop index variables of type short or unsigned int can prevent software pipelining. If the report indicates performance problems in loops where the index variable is not int and if there are no other obvious causes, try changing the loop index variable to type int.

Vectorization Report

The vectorization report can provide information about loops that could take advantage of Intel® Streaming SIMD Extensions (Intel® SSE3, SSE2, and SSE) vectorization, and it is available on systems based on IA-32 and Intel® 64 architectures.

See Using Parallelism for information on other vectorization options.

The `-vec-report` (Linux* and Mac OS* X) or `/Qvec-report` (Windows*) option directs the compiler to generate the vectorization reports with different levels of information. Specify a value of 3 to generate the maximum diagnostic details.

Operating System	Command
Linux and Mac OS X	`ifort -c -xSSSE3 -vec-report3 sample.f90`
Windows	`ifort /c /QxSSSE3 /Qvec-report:3 sample.f90`

where `-c` (Linux and Mac OS X) or `/c` (Windows) instructs the compiler to compile the example without generating an executable.

Note

Linux and Mac OS X: The space between the option and the phase is optional.
Windows: The colon between the option and phase is optional.

The following example results illustrate the type of information generated by the vectorization report:

Example results

```
sample.f90(27) : (col. 9) remark: loop was not vectorized: not inner loop.
sample.f90(28) : (col. 11) remark: LOOP WAS VECTORIZED.
sample.f90(31) : (col. 9) remark: loop was not vectorized: not inner loop.
sample.f90(32) : (col. 11) remark: LOOP WAS VECTORIZED.
sample.f90(37) : (col. 10) remark: loop was not vectorized: not inner loop.
sample.f90(38) : (col. 12) remark: loop was not vectorized: not inner loop.
sample.f90(40) : (col. 14) remark: loop was not vectorized: vectorization possible but
sample.f90(46) : (col. 10) remark: loop was not vectorized: not inner loop.
sample.f90(47) : (col. 12) remark: loop was not vectorized: contains unvectorizable sta
```

If the compiler reports ”r;`Loop was not vectorized`” because of the existence of vector dependence, then you should analyze the loop for vector dependence. If you determine there is no legitimate vector dependence, then the message indicates that the compiler was assuming the pointers or arrays in the loop were dependent, which implies the pointers or arrays were aliased. Use memory disambiguation techniques to resolve these cases.

There are three major types of vector dependence: `FLOW`, `ANTI`, and `OUTPUT`.

There are a number of situations where the vectorization report may indicate vector dependencies. The following situations will sometimes be reported as vector dependencies, non-unit stride, low trip count, and complex subscript expression.

Non-Unit Stride

The report might indicate that a loop could not be vectorized when the memory is accessed in a non-Unit Stride manner. This means that nonconsecutive memory locations are being accessed in the loop. In such cases, see if loop interchange can help or if it is practical. If not then you can force vectorization sometimes through `vector always` directive; however, you should verify improvement.

See Understanding Runtime Performance for more information about non-unit stride conditions.

Usage with Other Options

The vectorization reports are generated during the final compilation phase, which is when the executable is generated; therefore, there are certain option combinations you

cannot use if you are attempting to generate a report. If you use the following option combinations, the compiler issues a warning and does not generate a report:

- `-c` or `-ipo` or `-x` with `-vec-report` (Linux* and Mac OS* X) and `/c` or `/Qipo` or `/Qx` with `/Qvec-report` (Windows*)
- `-c` or `-ax` with `-vec-report` (Linux and Mac OS X) and `/c` or `/Qax` with `/Qvec-report` (Windows)

The following example commands can generate vectorization reports:

Operating System	Command Examples
Linux and Mac OS X	`ifort -xSSSE3 -vec-report3 sample.f90` `ifort -xSSSE3 -ipo -vec-report3 sample.f90`
Windows	`ifort /QxSSSE3 /Qvec-report:3 sample.f90` `ifort /QxSSSE3 /Qipo /Qvec-report:3 sample.f90`

Responding to the Results

You might consider changing existing code to allow vectorization under the following conditions:

- The vectorization report indicates that the program `contains unvectorizable statement at line XXX`.
- The vectorization report states there is a `vector dependence: proven FLOW dependence between 'r;variable' line XXX, and 'r;variable' line XXX` or `loop was not vectorized: existence of vector dependence.` Generally, these conditions indicate true loop dependencies are stopping vectorization. In such cases, consider changing the loop algorithm.

For example, consider the two equivalent algorithms producing identical output below. "Foo" will not vectorize due to the FLOW dependence but "bar" does vectorize.

Example

```
subroutine foo(y)
  implicit none
  integer :: i
  real :: y(10)
  do i=2,10
    y (i) = y (i-1)+1
  end do
end subroutine foo
subroutine bar(y)
  implicit none
  integer :: i
  real :: y(10)
  do i=2,10
    y (i) = y (1)+i
  end do
end subroutine bar
```

Unsupported loop structures may prevent vectorization. An example of an unsupported loop structure is a loop index variable that requires complex computation. Change the

structure to remove function calls to loop limits and other excessive computation for loop limits.

Example

```
function func(n)
  implicit none
  integer :: func, n
  func = n*n-1
end function func
subroutine unsupported loop structure(y,n)
  implicit none
  integer :: i,n, func
  real :: y(n)
  do i=0,func(n)
    y(i) = y(i) * 2.0
  end do
end subroutine unsupported_loop_structure
```

Non-unit stride access might cause the report to state that `vectorization possible but seems inefficient`. Try to restructure the loop to access the data in a unit-stride manner (for example, apply loop interchange), or try directive `vector always`.

Using mixed data types in the body of a loop might prevent vectorization. In the case of mixed data types, the vectorization report might state something similar to `loop was not vectorized: condition too complex`.

The following example code demonstrates a loop that cannot vectorize due to mixed data types within the loop. For example, `withinborder` is an integer while all other data types in loop are not. Simply changing the `withinborder` data type will allow this loop to vectorize.

Example

```
subroutine howmany close(x,y,n)
  implicit none
  integer :: i,n,withinborder
  real :: x(n), y(n), dist
  withinborder=0
  do i=0,100
    dist=sqrt(x(i)*x(i) + y(i)*y(i))
    if (dist<5) withinborder= withinborder+1
  end do
end subroutine howmany_close
```

OpenMP* Report

The `-openmp-report` (Linux* and Mac OS* X) or `/Qopenmp-report` (Windows*) option controls the diagnostic levels for OpenMP* reporting. The OpenMP* report information is not generated unless you specify the option with a value of either 1 or 2. Specifying 2 provides the most useful information.

You must specify the `-openmp` (Linux and Mac OS X) or `/Qopenmp` (Windows) along with this option.

Linux OS and Mac OS X	Windows OS	Description
`-openmp-`	`/Qopenmp-`	Report results are same as when specifying 1 except

`report 2`	`report:2`	the results also include diagnostics indicating constructs, like directives, that were handled successfully. This is the recommend level.
`-openmp-report 1`	`/Qopenmp-report:1`	Reports on loops, regions, and sections that were parallelized successfully. This is the default level.
`-openmp-report 0`	`/Qopenmp-report:0`	No diagnostics report generated.

The following example commands demonstrate how to run the report using the combined commands.

Operating System	Syntax Examples
Linux and Mac OS X	`ifort -openmp -openmp-report 2 sample1.f90 sample2.f90`
Windows	`ifort /Qopenmp /Qopenmp-report:2 sample1.f90 sample2.f90`

Note

Linux and Mac OS X: The space between the option and the level is optional.
Windows: The colon between the option and level is optional.

The following example results illustrate the typical format of the generated information:

Example results

```
openmp sample.f90(77): (col. 7) remark : OpenMP DEFINED LOOP WAS PARALLELIZED.
openmp_sample.f90(68): (col. 7) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
```

See also:

- OpenMP* Options Quick Reference for information about these options.
- OpenMP* Support Overview for information on using OpenMP* in Intel® compilers.

Using Compiler Optimizations

Automatic Optimizations Overview

Intel® compilers allow you to compile applications for processors based on IA-32 architectures (32-bit applications), Intel® 64 architectures (32-bit and 64-bit applications), or IA-64 architectures (64-bit applications).

By default the compiler chooses a set of optimizations that balances compile-time and run-time performance for your application. Also, you can manually select optimizations based on the specific needs of the application.

The following table summarizes the common optimization options you can use for quick, effective results.

Linux* and Mac OS* X	Windows*	Description
-O3	/O3	Enables aggressive optimization for code speed. Recommended for code with loops that perform substantial calculations or process large data sets.
-O2 (or -O)	/O2	Affects code speed. This is the default option; the compiler uses this optimization level if you do not specify anything.
-O1	/O1	Affects code size and locality. Disables specific optimizations.
-fast	/fast	Enables a collection of common, recommended optimizations for run-time performance. Can introduce architecture dependency.
-O0	/Od	Disables optimization. Use this for rapid compilation while debugging an application.

The variety of automatic optimizations enable you to quickly enhance your application performance. In addition to automatic optimizations, the compiler invokes other optimization enabled with source code directives, optimized library routines, and performance-enhancing utilities.

The remaining topics in this section provide more details on the automatic optimizations supported in the Intel compilers. See Enabling Automatic Optimizations and Restricting Optimizations for more information.

Enabling Automatic Optimizations

This topic lists the most common code optimization options, describes the characteristics shared by IA-32, Intel® 64, and IA-64 architectures, and describes the general behavior for each architecture.

The architectural differences and compiler options enabled or disabled by these options are also listed in more specific detail in the associated Compiler Options topics; therefore, each option discussion listed below includes a link to the appropriate reference topic.

Linux* and Mac OS* X	Windows*	Description
-O1	/O1	Optimizes to favor smaller code size and code locality. In most cases, -O2 (Linux* OS and Mac OS* X) or /O2 (Windows* OS) is recommended over this option. This optimization disables some optimizations that normally increase code size. This level might improve performance for

applications with very large code size, many branches, and execution time not dominated by code within loops. In general, this optimization level does the following:

- Enables global optimization.
- Disables intrinsic recognition and inlining of intrinsics.

IA-64 architecture:

- The option disables software pipelining, loop unrolling, and global code scheduling.

`-O2` or `-O` `/O2` Optimizes for code speed. Since this is the default optimization, if you do not specify an optimization level the compiler will use this optimization level automatically. This is the generally recommended optimization level; however, specifying other compiler options can affect the optimization normally gained using this level.

In general, the resulting code size will be larger than the code size generated using `-O1` (Linux and Mac OS X) or `/O1` (Windows).

This option enables the following capabilities for performance gain: inlining intrinsic functions, 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 optimizations, and dead store elimination.

For IA-32 and Intel 64 architectures:

- Enables certain optimizations for speed, such as vectorization.

IA-64 architecture:

- Enables optimizations for speed, including global code scheduling, software pipelining, predication, speculation, and data prefetch.

`-O3` `/O3` Enables `-O2` (Linux and Mac OS X) or `/O2` (Windows) optimizations, as well as more aggressive optimizations, including prefetching, scalar replacement, cache blocking,

and loop and memory access transformations.

As compared to -O2 (Linux) or /O2 (Windows), the optimizations enabled by this option often result in faster program execution, but can slow down code execution in some cases. Using this option may result in longer compilation times.

This option is recommended for loop-intensive applications that perform substantial floating-point calculations or process large data sets.

`-fast` `/fast` Provides a single, simple optimization that enables a collection of optimizations that favor run-time performance.

This is a good, general option for increasing performance in many programs.

For IA-32 and Intel 64 architectures, the `-xSSSE3` (Linux and Mac OS X) or `/QxSSSE3` (Windows) option that is set by this option cannot be overridden by other command line options. If you specify this option along with a different processor-specific option, such as `-x`SSE2 (Linux) or `/Qx`SSE2 (Windows), the compiler will issue a warning stating the `-x`SSSE3 or `/Qx`SSSE3 option cannot be overridden; the best strategy for dealing with this restriction is to explicitly specify the options you want to set from the command line.

Caution

Programs compiled with the `-xSSSE3` (Linux and Mac OS X) or `/Qx`SSSE3 (Windows) option will detect non-compatible processors and generate an error message during execution.

While this option enables other options quickly, the specific options enabled by this option might change from one compiler release to the next. Be aware of this possible behavior change in the case where you use makefiles.

The following syntax examples demonstrate using the default option to compile an application:

Operating System	Example

Linux and Mac OS X	`ifort -O2 prog.f90`
Windows	`ifort /O2 prog.f90`

Refer to Quick Reference Lists for a complete listing of the quick reference topics.

Targeting IA-32 and Intel(R) 64 Architecture Processors Automatically

The `-x` (Linux* and Mac OS* X) or `/Qx` (Windows*) option can automatically optimize your application for specific Intel® processors based on IA-32 and Intel® 64 architectures.

The automatic optimizations allow you to take advantage of the architectural differences, new instruction sets, or advances in processor design; however, the resulting, optimized code might contain unconditional use of features that are not supported on other, earlier processors. Therefore, using these options effectively sets a minimum hardware requirement for your application.

The optimizations can include generating Intel® Streaming SIMD Extensions 4 (SSE4), Supplemental Streaming SIMD Extensions 3 (SSSE3), Streaming SIMD Extensions 3 (SSE3), Streaming SIMD Extensions 2 (SSE2), or Streaming SIMD Extensions (SSE) instructions.

If you intend to run your programs on multiple processors based on IA-32 or Intel® 64 architectures, do not use this option; instead, consider using the `-ax` (Linux and Mac OS X) or `/Qax` (Windows) option to achieve both processor-specific performance gains and portability among different processors.

Linux OS and Mac OS X	Windows OS	Description
`-xHost`	`/QxHost`	Can generate instructions for the highest instruction set and processor available on the compilation host.
`-xAVX`	`/QxAVX`	Optimizes for Intel processors that support Intel® Advanced Vector Extensions (Intel® AVX).
`-xSSE4.1`	`/QxSSE4.1`	Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel processors. Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture. This replaces value S, which is deprecated.
`-xSSE4.2`	`/QxSSE4.2`	Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core™ i7 processors. Can generate Intel® SSE4

		Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for the Intel® Core™ processor family.
-xSSE3_ATOM	/QxSSE3_ATOM	Can generate MOVBE instructions for Intel processors and it can optimize for the Intel® Atom™ processor and Intel® Centrino® Atom™ Processor Technology.
-xSSSE3	/QxSSSE3	Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for the Intel® Core™2 Duo processor family. This replaces value T, which is deprecated.
-xSSE3	/QxSSE3	Can generate Intel® SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for processors based on Intel® Core™ microarchitecture and Intel NetBurst® microarchitecture. This replaces value P, which is deprecated. Mac OS X: Supported on IA-32 architectures.
-xSSE2	/QxSSE2	Can generate Intel® SSE2 and SSE instructions for Intel processors, and it can optimize for Intel® Pentium® 4 processors, Intel® Pentium® M processors, and Intel® Xeon® processors with Intel® SSE2. Mac OS X: Not supported.

Certain keywords for compiler options -m and /arch produce binaries that should run on processors not made by Intel that implement the same capabilities as the corresponding Intel processors. For details, see Compiler Options.

To prevent illegal instruction and similar unexpected run-time errors during program execution, the compiler inserts code in the main routine of the program to check for proper processor usage. Using this option limits you to a minimum processor level. For example, if you target an application to run on Intel® Xeon® processors based on the Intel® Core™ microarchitecture, it is unlikely the resulting application will operate correctly on earlier Intel processors.

If you target more than one processor value, the resulting code will be generated for the highest-performing processor specified if the compiler determines there is an advantage in doing so. The highest- to lowest-performing processor values are as follows:

1. SSE4.1
2. SSSE3

3. SSE3
4. SSE2

Executing programs compiled with processor values of SSE4.1, SSSE3, SSE3, or SSE2 on unsupported processors will display a run-time error. For example, if you specify the SSSE3 processor value to compile an application but execute the application on an Intel® Pentium® 4 processor, the application generates an error similar to the following:

Run-time Error
`Fatal Error: This program was not built to run on the processor in your system.` `The allowed processors are: Intel(R) Core(TM) Duo processors and compatible Intel processors with supplemental Streaming SIMD Extensions 3 (SSSE3) instruction support.`

The following examples demonstrate compiling an application for Intel® Core™2 Duo processor and compatible processors. The resulting binary might not execute correctly on earlier processors or on IA-32 architecture processors not made by Intel Corporation.

Operating System	Example
Linux and Mac OS X	`ifort -xSSSE3 sample.f90`
Windows	`ifort /QxSSSE3 sample.f90`

Targeting Multiple IA-32 and Intel(R) 64 Architecture Processors for Run-time Performance

The `-ax` (Linux* and Mac OS* X) or `/Qax` (Windows*) option instructs the compiler to determine if opportunities exist to generate multiple, specialized code paths to take advantage of performance gains and features available on newer Intel® processors based on IA-32 and Intel® 64 architectures. This option also instructs the compiler to generate a more generic (baseline) code path that should allow the same application to run on a larger number of processors; however, the baseline code path is usually slower than the specialized code.

The compiler inserts run-time checking code to help determine which version of the code to execute. The size of the compiled binary increases because it contains both a processor-specific version of some of the code and a generic baseline version of all code. Application performance is affected slightly due to the run-time checks needed to determine which code to use. The code path executed depends strictly on the processor detected at run time.

Processor support for the baseline code path is determined by the processor family or instruction set specified in the `-m` or `-x` (Linux and Mac OS X) or `/arch` or `/Qx` (Windows) option, which has default values for each architecture.

This allows you to impose a more strict processor or instruction set requirement for the baseline code path; however, such generic baseline code will not operate correctly on

processors that are not compatible with the minimum processor or instruction set requirement. For the IA-32 architecture, you can specify a baseline code path that will work on all IA-32 compatible processors using the `-mia32` (Linux) or `/arch:IA32` (Windows) options. You should always specify the processor or instruction set requirements explicitly for the baseline code path, rather than depend on the defaults for the architecture.

Optimizations in the specialized code paths can include generating and using Intel® Streaming SIMD Extensions 4 (SSE4), Supplemental Streaming SIMD Extensions 3 (SSSE3), Streaming SIMD Extensions 3 (SSE3), or Streaming SIMD Extensions 2 (SSE2) instructions for supported Intel processors; however, such specialized code paths are executed only after checking verifies that the code is supported by the run-time host processor.

If not indicated otherwise, the following processor values are valid for IA-32 and Intel® 64 architectures.

Linux OS and Mac OS X	Windows OS	Description
`-axSSE4.2`	`/QaxSSSE4.2`	Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core™ i7 processors. Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for the Intel® Core™ processor family.
`-axSSE4.1`	`/QaxSSSE4.1`	Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel processors. Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture. This replaces value S, which is deprecated. Mac OS X: IA-32 and Intel® 64 architectures.
`-axSSSE3`	`/QaxSSSE3`	Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for the Intel® Core™2 Duo processor family. This replaces value T, which is deprecated. Mac OS X: IA-32 architecture.
`-axSSE3`	`/QaxSSE3`	Can generate Intel® SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for processors based on Intel® Core™

		microarchitecture and Intel NetBurst® microarchitecture. This replaces value P, which is deprecated. Mac OS X: IA-32 architecture.
`-axSSE2`	`/QaxSSE2`	Can generate Intel® SSE2 and SSE instructions for Intel processors, and it can optimize for Intel® Pentium® 4 processors, Intel® Pentium® M processors, and Intel® Xeon® processors with Intel® SSE2. Linux and Windows: IA-32 architecture.

Note

You can specify `-diag-disable cpu-dispatch` (Linux and Mac OS X) or `/Qdiag-disable:cpu-dispatch` (Windows) to disable the display of remarks about multiple code paths for CPU dispatch.

If your application for IA-32 or Intel® 64 architectures does not need to run on multiple processors , consider using the `-x` (Linux and Mac OS X) or `/Qx` (Windows) option instead of this option.

The following compilation examples demonstrate how to generate an IA-32 architecture executable that includes an optimized version for Intel® Core™2 Duo processors, as long as there is a performance gain, an optimized version for Intel® Core™ Duo processors, as long as there is a performance gain, and a generic baseline version that runs on any IA-32 architecture processor.

Note

If you combine the arguments, you must add a comma (",") separator between the individual arguments.

Operating System	Example
Linux	`ifort -axSSSE3,SSE3 -mia32 sample.f90`
Windows	`ifort /QaxSSSE3,SSE3 /arch:IA32 sample.f90`

Targeting IA-64 Architecture Processors Automatically

The Intel compiler supports options that optimize application performance for Intel® Itanium® processors based on the IA-64 architecture.

Linux* OS	Windows* OS	Optimizes applications for...

`-mtune=itanium2-p9000`	`/G2-p9000`	Default. Dual-Core Intel® Itanium® 2 processor (9000 series)
`-mtune=itanium2`	`/G2`	Intel® Itanium® 2 processors

Note

Mac OS* X: These options are not supported.

While the resulting executable is backward compatible, generated code is optimized for specific processors; therefore, code generated with `-mtune=itanium2-p9000` (Linux) or `/G2-p9000` (Windows) will run correctly on Itanium® 2 processors.

The following examples demonstrate using the default options to target an Itanium® 2 processor (9000 series). The same binary will also run on Intel® Itanium® 2 processors.

Operating System	Example
Linux	`ifort -mtune=itanium2-p9000 prog.f90`
Windows	`ifort /G2-p9000 prog.f90`

Restricting Optimizations

The following table lists options that restrict the ability of the Intel® compiler to optimize programs.

Linux* and Mac OS* X	Windows*	Effect
`-O0`	`/Od`	Disables all optimizations. Use this during development stages where fast compile times are desired. Linux* and Mac OS* X: • Sets option `-fomit-frame-pointer` and option `-fmath-errno`. Windows*: • Use `/Od` to disable all optimizations while specifying particular optimizations, such as: `/Od /Ob1` (disables all optimizations, but only enables inlining) For more information, see the following topic: • `-O0` compiler option
`-g`	`/Zi, /Z7`	Generates symbolic debugging information in object files for use by debuggers. This option enables or disables other compiler options

		depending on architecture and operating system; for more information about the behavior, see the following topic: • `-g` compiler option
`-fmath-errno`, `-fno-math-errno`	No equivalent	Instructs the compiler to assume that the program tests `errno` after calls to math library functions. • `-fmath-errno` compiler option

Diagnostic Options

Linux and Mac OS X	Windows	Effect
`-sox`	`/Qsox`	Instructs the compiler to save the compiler options and version number in the executable. During the linking process, the linker places information strings into the resulting executable. Slightly increases file size, but using this option can make identifying versions for regression issues much easier. For more information, see the following topic: • `-sox` compiler option

Using Parallelism: OpenMP* Support

OpenMP* Support Overview

The Intel® compiler supports the OpenMP* Version 3.0 API specification. For complete Fortran language support for OpenMP, see the *OpenMP Application Program Interface* Version 3.0 specification, which is available from the OpenMP web site (http://www.openmp.org/, click the Specifications link).

This version of the Intel compiler also introduces OpenMP API Version 3.0 API specification support, as described in the OpenMP web site (http://www.openmp.org/, click Specifications).

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 creation, scheduling, and synchronization.
- Provides the benefit of the performance available from shared memory multiprocessor and multi-core processor systems on IA-32, Intel® 64, and IA-64 architectures, including those processors with Hyper-Threading Technology.

The compiler performs transformations to generate multithreaded code based on a developer'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 and compiles parallel programs annotated with OpenMP directives.

The compiler provides Intel-specific extensions to the OpenMP Version 3.0 specification including run-time library routines and environment variables. However, these extensions are only supported by the Intel compilers. A summary of the compiler options that apply to OpenMP* appears in OpenMP* Options Quick Reference.

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 compiler processes the application and internally produces a multithreaded version of the code which is then compiled. The output is an executable with the parallelism implemented by threads that execute parallel regions or constructs. See Programming with OpenMP.

Using Other Compilers

The OpenMP specification does not define interoperability of multiple implementations; therefore, the OpenMP implementation supported by other compilers and OpenMP support in Intel compilers might not be interoperable. Even if you compile and build the entire application with one compiler, be aware that different compilers might not provide OpenMP source compatibility that would allow you to compile and link the same set of application sources with a different compiler and get the expected parallel execution results.

Intel compilers include two sets of OpenMP libraries, as described in OpenMP Source Compatibility and Interoperability with Other Compilers.

OpenMP* Options Quick Reference
These options are supported on IA-32, Intel® 64, and IA-64 architectures.

Linux* OS and Mac OS* X	Windows* OS	Description
`-openmp`	`/Qopenmp`	This option enables the parallelizer to generate multi-threaded code based on the OpenMP* directives. The code can be executed in parallel on both uniprocessor and multiprocessor

		systems. IA-64 architecture only: • Implies `-opt-mem-bandwith1` (Linux) or `/Qopt-mem-bandwidth1` (Windows).
`-openmp-report`	`/Qopenmp-report`	This option controls the OpenMP parallelizer's level of diagnostic messages. To use this option, you must also specify `-openmp` (Linux and Mac OS X) or `/Qopenmp` (Windows).
`-openmp-stubs`	`/Qopenmp-stubs`	This option enables compilation of OpenMP programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked.
`-openmp-profile`	`/Qopenmp-profile`	This option enables analysis of OpenMP* applications. To use this option, you must have previously installed Intel® Thread Profiler, which is one of the Intel® Threading Analysis Tools. This option can adversely affect performance because of the additional profiling and error checking invoked to enable compatibility with the threading tools. Do not use this option unless you plan to use the Intel® Thread Profiler.
`-openmp-lib`	`/Qopenmp-lib`	This option lets you specify an OpenMP* run-time library to use for linking. The legacy OpenMP run-time library is not compatible with object files created using OpenMP run-time libraries supported in other compilers. The compatibility OpenMP run-time library is compatible with object files created using the Microsoft* OpenMP run-time library (vcomp) and GNU OpenMP run-time library (libgomp).
`-openmp-link`	`/Qopenmp-link`	This option controls whether the compiler links to static or dynamic OpenMP run-time libraries. To link to the static OpenMP run-time library (RTL) and create a purely static executable, you must specify `-openmp-link` static (Linux and Mac OS X) or `/Qopenmp-link:static` (Windows).

		However, we strongly recommend you use the default setting, `-openmp-link dynamic` (Linux and Mac OS X) or `/Qopenmp-link:dynamic` (Windows).
`-openmp-threadprivate`	`/Qopenmp-threadprivate`	This option lets you specify an OpenMP* threadprivate implementation. The legacy OpenMP run-time library is not compatible with object files created using OpenMP run-time libraries supported in other compilers.

When both `-openmp` and `-parallel` (Linux OS and Mac OS X) or `/Qopenmp` and `/Qparallel` (Windows OS) are specified on the command line, the `parallel` option is only applied in loop nests that do not have OpenMP directives. For loop nests with OpenMP directives, only the `openmp` option is applied.
Refer to the following topics for information about OpenMP environment variable and run-time routines:

- OpenMP Environment Variables
- OpenMP Run-time Library Routines
- Intel Extension Routines to OpenMP

Refer to Quick Reference Lists for a complete listing of the quick reference topics.

OpenMP* Source Compatibility and Interoperability with Other Compilers
Intel compilers include two sets of OpenMP libraries:

- The Compatibility OpenMP libraries, which provide compatibility with OpenMP support provided by certain versions of the Microsoft Visual C++* compiler on Windows* OS, certain versions of the GNU* compilers on Linux* OS and Mac OS* X, as well as the Intel compiler version 10.*x* (and later).
- The Legacy OpenMP libraries, which provide compatibility with OpenMP support provided by Intel compilers, including Intel compilers prior to version 10.0.

To select the Compatibility (default) or Legacy OpenMP libraries, use the Intel compiler to link your application and specify the Intel compiler option `/Qopenmp-lib` (Windows OS) or `-openmp-lib` (Linux OS and Mac OS X) .

The term *object-level interoperability* refers to the ability to link object files and libraries generated by one compiler with object files and libraries generated by the second compiler, such that the resulting executable runs successfully. In contrast, *source compatibility* means that the entire application is compiled and linked by one compiler, and you do not need to modify the sources to get the resulting executable to run successfully.

Different compilers support different versions of the OpenMP specification. Based on the OpenMP features your application uses, determine what version of the OpenMP

specification your application requires. If your application uses an OpenMP specification level equal or less than the OpenMP specification level supported by all the compilers, your application should have source compatibility with all compilers, but you need to link all object files and libraries with the same compiler's OpenMP libraries.

OpenMP Compatibility Libraries Provided by Intel Compilers

The Compatibility libraries provide source compatibility and object-level interoperability with the OpenMP support provided by:

- On Windows* OS, certain versions of Microsoft Visual C++* that support OpenMP, starting with Microsoft Visual C++ 2005.
- On Linux* OS and Mac OS* X, certain versions of GNU* `gcc`* that support OpenMP, starting with GNU* `gcc`* version 4.2.
- Intel compilers versions 10.0 and later and their supplied OpenMP libraries.

For Fortran applications on Linux systems, it is not possible to link objects compiled by the Intel® Fortran Compiler (`ifort`) with objects compiled by the GNU* Fortran compiler (`gfortran`). Thus, for mixed-language C++ and Fortran applications, you can do one of the following:

- Combine objects created by `gfortran` and Intel® C++ objects, if you specify the Intel OpenMP Compatibility libraries during linking.
- Combine objects created by the Intel C++ compiler and the Intel Fortran Compiler, using Intel OpenMP Compatibility or Legacy libraries.

OpenMP Legacy Libraries Provided by Intel Compilers

The set of Legacy OpenMP libraries has been provided by Intel compilers for multiple releases and provide source compatibility and object-level interoperability with the current Legacy libraries and OpenMP libraries provided by previous Intel compiler versions, including those prior to version 10.0. The Legacy libraries are not compatible with OpenMP support from non-Intel compilers, such as Microsoft Visual C++*, GNU `gcc`*, or GNU Fortran.

You should only use the Legacy libraries if your application requires object-level interoperability with OpenMP library versions provided prior to Intel compilers version 10.0.

Guidelines for Using Different Intel Compiler Versions

To avoid possible linking or run-time problems, follow these guidelines:

- If you compile your application using only the Intel compilers, avoid mixing the Compatibility and Legacy OpenMP runtime libraries. That is, you must link the entire application with either the Compatibility or Legacy libraries.
- When using the Legacy libraries, use the most recent Intel compiler to link the entire application. However, be aware that the Legacy libraries are deprecated, so for a future release, you will need to link the entire application with the Compatibility libraries.
- Use dynamic instead of static OpenMP libraries to avoid linking multiple copies of the libraries into a single program. For details, see OpenMP Support Libraries.

Guidelines for Using Intel and Non-Intel Compilers

To avoid possible linking or run-time problems, follow these guidelines:

- Always link the entire application using the Intel compiler OpenMP Compatibility libraries. This avoids linking multiple copies of the OpenMP runtime libraries from different compilers. It is easiest if you use the Intel compiler command (driver) to link the application, but it is possible to link with the Intel compiler OpenMP Compatibility libraries when linking the application using the GNU* or Visual C++ compiler (or linker) commands.
- If possible, compile all the OpenMP sources with the same compiler. If you compile (not link) using multiple compilers such as the Microsoft Visual C++* or GNU compilers that provide object-level interoperability with the Compatibility libraries, see the instructions in Using the OpenMP Compatibility Libraries.
- Use dynamic instead of static OpenMP libraries to avoid linking multiple copies of the libraries into a single program. For details, see OpenMP Support Libraries.

Limitations When Using OpenMP Compatibility Libraries with Other Compilers

Limitations of threadprivate objects on object-level interoperability:

- On Windows OS systems, the Microsoft Visual C++* compiler uses a different mechanism than the Intel compilers to reference threadprivate data. If you declare a variable as threadprivate in your code and you compile the code with both Intel compilers and Visual C++ compilers, the code compiled by the Intel compiler and the code compiled by the Visual C++* compiler will reference different locations for the variable even when referenced by the same thread. Thus, use the same compiler to compile all source modules that use the same threadprivate objects.
- On Linux OS systems, the GNU* compilers use a different mechanism than the Intel compilers to reference threadprivate data. If you declare a variable as threadprivate in your code and you compile the code with both Intel compilers and GNU compilers, the code compiled by the Intel compiler and the code

compiled by the GNU compiler will reference different locations for the variable even when referenced by the same thread. Thus, use the same compiler to compile all source modules that use the same threadprivate objects.

- On Mac OS* X systems, the operating system does not currently support the mechanism used by the GNU* compiler to support threadprivate data. Threadprivate data objects will only be accessible by name from object files compiled by the Intel compilers.

Using OpenMP*

Using OpenMP* in your application requires several steps. To use OpenMP, you must do the following:

1. Add OpenMP directives to your application source code.
2. Compile the application with `-openmp` (Linux* and Mac OS* X) or `/Qopenmp` (Windows*) option.
3. For applications with large local or temporary arrays, you may need to increase the stack space available at run-time. In addition, you may need to increase the stack allocated to individual threads by using the KMP_STACKSIZE environment variable or by setting the corresponding library routines.

You can set other environment variables for the multi-threaded code execution.

Add OpenMP Support to the Application

Add the OpenMP API routine declarations to your application by adding a statement similar to the following in your code:

Example

```
include 'omp_lib.h'
```

or (depending on the Fortran language level)

```
use omp_lib
```

OpenMP Directive Syntax

OpenMP directives use a specific format and syntax. Intel Extension Routines to OpenMP* describes the OpenMP extensions to the specification that have been added to the Intel® compiler.

The following syntax illustrates using the directives in your source.

Syntax

```
<prefix> <directive> [<clause>[[,]<clause>...]]
```

where:

- *<prefix>* - Required for all OpenMP directives. For free form source input, the prefix is `!$OMP` only; for fixed form source input, the prefix is `!$OMP` or `C$OMP`.

- `<directive>` - A valid OpenMP directive. Must immediately follow the prefix; for example: `!$OMP PARALLEL`.
- `[<clause>]` - Optional. Clauses can be in any order and repeated as necessary, unless otherwise restricted.
- `[<newline>]` - A required component of directive syntax. It precedes the structured block which is enclosed by this directive.
- `[,]`: Optional. Commas between more than one `<clause>` are optional.

The directives are interpreted as comments if you omit the `-openmp` (Linux and Mac OS X) or `/Qopenmp` (Windows*) option.

The OpenMP constructs defining a parallel region have one of the following syntax forms:

Example

```
!$OMP <directive>
  <structured block of code>
!$OMP END <directive>
```

or

```
!$OMP <directive>
  <structured block of code>
```

or

```
!$OMP <directive>
```

The following example demonstrates one way of using an OpenMP directive to parallelize a loop.

Example

```
subroutine simple_omp(a, N)
  use omp_lib
  integer :: N, a(N)
!$OMP PARALLEL DO
  do i = 1, N
    a(i) = i*2
  end do
end subroutine simple_omp
```

See OpenMP* Examples for more examples on using directives in specific circumstances.

Compile the Application

The `-openmp` (Linux* and Mac OS* X) or `/Qopenmp` (Windows*) option enables the parallelizer to generate multi-threaded code based on the OpenMP directives in the source. The code can be executed in parallel on single processor, multi-processor, or multi-core processor systems.

Note

IA-64 Architecture: Specifying this option implies `-opt-mem-bandwith1` (Linux) or `/Qopt-mem-bandwidth1` (Windows).

The `openmp` option works with both `-O0` (Linux and Mac OS X) and `/Od` (Windows) and with any optimization level of `-O1`, `-O2` and `-O3` (Linux and Mac OS X) or `/O1`, `/O2` and `/O3` (Windows).

Specifying `-O0` (Linux and Mac OS X) or `/Od` (Windows) with the OpenMP option helps to debug OpenMP applications.

Compile your application using commands similar to those shown below:

Operating System	Description
Linux and Mac OS X	`ifort -openmp source_file`
Windows	`ifort /Qopenmp source_file`

Assume that you compile the sample above, using the commands similar to the following, where `-c` (Linux and Mac OS X) or `/c` (Windows) instructs the compiler to compile the code without generating an executable:

Operating System	Example
Linux and Mac OS X	`ifort -openmp -c parallel.f90`
Windows	`ifort /Qopenmp /c parallel.f90`

The compiler might return a message similar to the following:

```
parallel.f90(20) : (col. 6) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
```

Configure the OpenMP Environment

Before you run the multi-threaded code, you can set the number of desired threads using the OpenMP environment variable, `OMP_NUM_THREADS`. See the OpenMP Environment Variables.

Parallel Processing Model

A program containing OpenMP* API compiler directives begins execution as a single thread, called the initial thread of execution. The initial thread executes sequentially until the first parallel construct is encountered.

In the OpenMP API, the `PARALLEL` and `END PARALLEL` directives define the extent of the parallel construct. When the initial thread encounters a parallel construct, it creates a team of threads, with the initial thread becoming the master of the team. All program statements enclosed by the parallel construct are executed in parallel by each thread in the team, including all 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 all statements encountered during the execution of a construct by a thread, including all called routines.

When a thread encounters the end of a structured block enclosed by a parallel construct, the thread waits until all threads in the team have arrived. When that happens the team is dissolved, and only the master thread continues execution of the code following the parallel construct. The other threads in the team enter a wait state until they are needed to form another team. 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.

The following example illustrates, from a high level, the execution model for the OpenMP constructs. The comments in the code explain the structure of each construct or section.

Example

```
PROGRAM MAIN              ! Begin serial execution.
  ...                     ! Only the initial thread executes.
 !$OMP PARALLEL           ! Begin a Parallel construct, form a team.
   ...                    ! This code is executed by each team member.
  !$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               !
  !$OMP END DO NOWAIT     ! End of worksharing construct, NOWAIT
                          ! is specified (threads need not wait).
                          ! This code is executed by each team member.
  !$OMP CRITICAL          ! Begin critical construct.
     ...                  ! One thread executes at a time.
  !$OMP END CRITICAL      ! End the critical construct.
   ...                    ! This code is executed by each team member.
  !$OMP BARRIER           ! Wait for all team members to arrive.
   ...                    ! This code is executed by each team member.
 !$OMP END PARALLEL       ! End of parallel onstruct, disband team.
                          ! and continue with serial execution.
  ...                     ! Possibly more parallel constructs.
END PROGRAM MAIN          ! End serial execution.
```

Using Orphaned Directives

In routines called from within parallel constructs, you can also use directives. Directives that are not in the static extent of the parallel construct, but are in the dynamic extent, are called orphaned directives. Orphaned directives allow you to execute 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:

Example

```
subroutine F
...
!$OMP PARALLEL...
   call G
...
subroutine G
!$OMP DO... ! This is an orphaned directive.
...
```

This is an orphaned `DO` loop directive since the parallel region is not lexically present in `subroutine G`.

Data Environment Controls

You can control the data environment within parallel and worksharing constructs. Using directives and data environment clauses on directives, you can privatize named global-lifetime objects by using `THREADPRIVATE` directive, or control data scope attributes by using the data environment clauses for directives that support them.

The data scope attribute clauses are:

- `DEFAULT`
- `PRIVATE`
- `FIRSTPRIVATE`
- `LASTPRIVATE`
- `REDUCTION`
- `SHARED`

The data copying clauses are:

- `COPYIN`
- `COPYPRIVATE`

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; however, if you do not specify a data scope attribute clause on a directive, the behavior for the variable is determined by the default scoping rules, which are described in the OpenMP API specification, for the variables affected by the directive.

Determining How Many Threads to Use

For applications where the workload depends on application input that can vary widely, delay the decision about the number of threads to employ until runtime when the input sizes can be examined. Examples of workload input parameters that affect the thread count include things like matrix size, database size, image/video size and resolution, depth/breadth/bushiness of tree based structures, and size of list based structures. Similarly, for applications designed to run on systems where the processor count can vary widely, defer the number of threads to employ until application run-time when the machine size can be examined.

For applications where the amount of work is unpredictable from the input data, consider using a calibration step to understand the workload and system characteristics to aid in choosing an appropriate number of threads. If the calibration step is expensive, the calibration results can be made persistent by storing the results in a permanent place like the file system.

Avoid simultaneously using more threads than the number of processing units on the system. This situation causes the operating system to multiplex the processors and typically yields sub-optimal performance.

When developing a library as opposed to an entire application, provide a mechanism whereby the user of the library can conveniently select the number of threads used by the library, because it is possible that the user has higher-level parallelism that renders the parallelism in the library unnecessary or even disruptive.

Use the `NUM_THREADS` clause on parallel regions to control the number of threads employed and use the if clause on parallel regions to decide whether to employ multiple threads at all. The `OMP_SET_NUM_THREADS` routine can also be used, but it also affects parallel regions created by the calling thread. The `NUM_THREADS` clause is local in its effect, so it does not impact other parallel regions.

The `OMP_THREADS_LIMIT` environment variable limits the number of OpenMP threads to use for the whole OpenMP program.

OpenMP* Directives

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 variable in any clause other than the following:

- `COPYIN`
- `COPYPRIVATE`
- `SCHEDULE`
- `NUM_THREADS`
- `IF`

In the following example common blocks `BLK1` and `FIELDS` are specified as thread private:

Example

```
COMMON /BLK1/ SCRATCH
COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
!$OMP THREADPRIVATE(/BLK1/,/FIELDS/)
```

For more details on this directive, see OpenMP* Fortran Compiler Directives.

OpenMP* Advanced Issues

This topic discusses how to use the OpenMP* library functions and environment variables and discusses some guidelines for enhancing performance with OpenMP*.

OpenMP* provides specific function calls, and environment variables. See the following topics to refresh you memory about the primary functions and environment variable used in this topic:

- OpenMP* Run-time Library Routines
- OpenMP* Environment Variables

To use the function calls, include the `omp_lib.h` header file or specify `use omp_lib` to use the module file, which are installed in the `INCLUDE` directory during the compiler installation, and compile the application using the `-openmp` (Linux* and Mac OS* X) or `/Qopenmp` (Windows*) option.

The following example, which demonstrates how to use the OpenMP* functions to print the alphabet, also illustrates several important concepts.

First, when using functions instead of directives, your code must be rewritten; rewrites can mean extra debugging, testing, and maintenance efforts.

Second, it becomes difficult to compile without OpenMP support.

Third, it is very easy to introduce simple bugs, as in the loop (below) that fails to print all the letters of the alphabet when the number of threads is not a multiple of 26.

Fourth, you lose the ability to adjust loop scheduling without creating your own work-queue algorithm, which is a lot of extra effort. You are limited by your own scheduling, which is mostly likely static scheduling as shown in the example.

Example

```
include "omp lib.h"
integer i
integer LettersPerThread, ThisThreadNum, StartLetter, EndLetter
call omp set num threads(4)
  !$OMP PARALLEL PRIVATE(i)
    ! OMP NUM THREADS is not a multiple of 26,
    ! which can be considered a bug in this code.
    LettersPerThread = 26 / omp get num threads()
    ThisThreadNum = omp get thread num()
    StartLetter = 'a'+ThisThreadNum*LettersPerThread
    EndLetter = 'a'+ThisThreadNum*LettersPerThread+LettersPerThread
    DO i = StartLetter, EndLetter - 1
       write( *,FMT='(A)',ADVANCE='NO') char(i)
    END DO
  !$OMP END PARALLEL
write(*,*)
end
```

Debugging threaded applications is a complex process, because debuggers change the run-time performance, which can mask race conditions. Even `print` statements can mask issues, because they use synchronization and operating system functions. OpenMP* itself also adds some complications, because it introduces additional structure by distinguishing private variables and shared variables, and inserts additional code. A specialized debugger that supports OpenMP, such as the Intel® Debugger, can help you to examine variables and step through threaded code. You can also use the Intel® Thread Checker to detect many hard-to-find threading errors analytically. Sometimes, a process of elimination can help identify problems without resorting to sophisticated debugging tools.

Remember that most mistakes are race conditions. Most race conditions are caused by shared variables that really should have been declared private. Start by looking at the variables inside the parallel regions and make sure that the variables are declared private when necessary. Next, check functions called within parallel constructs.

The `DEFAULT(NONE)` clause, shown below, can be used to help find those hard-to-spot variables. If you specify `DEFAULT(NONE)`, then every variable must be declared with a data-sharing attribute clause.

Example

```
!$OMP PARALLEL DO DEFAULT(NONE) PRIVATE(x,y) SHARED(a,b)
```

Another common mistake is using uninitialized variables. Remember that private variables do not have initial values upon entering a parallel construct. Use the `FIRSTPRIVATE` and `LASTPRIVATE` clauses to initialize them only when necessary, because doing so adds extra overhead.

If you still can't find the bug, then consider the possibility of reducing the scope. Try a binary-hunt. Another method is to force large chunks of a parallel region to be critical sections. Pick a region of the code that you think contains the bug and place it within a critical section. Try to find the section of code that suddenly works when it is within a critical section and fails when it is not. Now look at the variables, and see if the bug is apparent. If that still doesn't work, try setting the entire program to run in serial by setting the compiler-specific environment variable `KMP_LIBRARY=serial`.

If the code is still not working, compile it without the `-openmp` (Linux and Mac OS X) or `/Qopenmp` (Windows) option to make sure the serial version works.

Performance

OpenMP threaded application performance is largely dependent upon the following things:

- The underlying performance of the single-threaded code.
- CPU utilization, idle threads, and load balancing.
- The percentage of the application that is executed in parallel by multiple threads.

- The amount of synchronization and communication among the threads.
- The overhead needed to create, manage, destroy, and synchronize the threads, made worse by the number of single-to-parallel or parallel-to-single transitions called fork-join transitions.
- Performance limitations of shared resources such as memory, bus bandwidth, and CPU execution units.
- Memory conflicts caused by shared memory or falsely shared memory.

Performance always begins with a properly constructed parallel algorithm or application. For example, parallelizing a bubble-sort, even one written in hand-optimized assembly language, is not a good place to start. Keep scalability in mind; creating a program that runs well on two CPUs is not as efficient as creating one that runs well on `n` CPUs. With OpenMP, the number of threads is chosen by the compiler, so programs that work well regardless of the number of threads are highly desirable. Producer/consumer architectures are rarely efficient, because they are made specifically for two threads.

Once the algorithm is in place, make sure that the code runs efficiently on the targeted Intel® architecture; a single-threaded version can be a big help. Turn off the `-openmp` (Linux and Mac OS X) or `/Qopenmp` (Windows) option to generate a single-threaded version, , or build with `–openmp-stubs` (Linux and Mac OS X) or `/Qopenmp-stubs` (Windows), and run the single-threaded version through the usual set of optimizations.

Once you have gotten the single-threaded performance, it is time to generate the multi-threaded version and start doing some analysis.

Start by looking at the amount of time spent in the idle loop of the operating system. The VTune™ Performance Analyzer and the Intel® Threading Analysis Tools are good tools to help with the investigation. Idle time can indicate unbalanced loads, lots of blocked synchronization, and serial regions. Fix those issues, and then use the VTune™ Analyzer again to look for excessive cache misses and memory issues like false-sharing. Solve these basic problems, and you should have a well-optimized parallel program that will run well on dual- and multi-core processors, processors that support Hyper-Threading Technology, and multiple physical processors.

Optimizations are really a combination of patience, experimentation, and practice. Make little test programs that mimic the way your application uses the computer resources to get a feel for what things are faster than others. Be sure to try the different scheduling clauses for the parallel sections of code.

Cluster OpenMP* Support (Linux*)

Cluster OpenMP* is an add-on to the Intel® compilers that allows you to run an OpenMP program on a set of distributed symmetric multi-processors.

All OpenMP directives are also supported as one additional directive: `SHARABLE`. The `SHARABLE` directive declares that listed variables are kept consistent between the processes.

Syntax

```
!DIR OMP SHARABLE (<list>)
```

Cluster OpenMP is supported on Linux* systems running on IA-64 or Intel® 64 architectures.

For more information about using Cluster OpenMP, the supported compiler options, and associated tools, refer to the *Cluster OpenMP* User Manual*, which is included with the compiler documentation in the `<install-dir>/doc/cluster_omp_docs/` directory.

OpenMP* Examples
The following examples show how to use several OpenMP* features.

A Simple Difference Operator

This example shows a simple parallel loop where the amount of work in each iteration is different. Dynamic scheduling is used to improve load balancing.

The `END DO` has a `NOWAIT` because there is an implicit barrier at the end of the parallel region.

Example

```
subroutine do 1(a,b,n)
  real a(n,n), b(n,n)
  !$OMP PARALLEL SHARED(A,B,N)
     !$OMP DO SCHEDULE(DYNAMIC,1) PRIVATE(I,J)
       do i = 2, n
          do j = 1, i
            b(j,i) = ( a(j,i) + a(j,i-1) ) / 2.0
          end do
       end do
  !$OMP END DO NOWAIT
!$OMP END PARALLEL
end
```

Two Difference Operators: DO Loop Version

The example uses two parallel loops fused to reduce fork/join overhead. The first `END DO` directive has a `NOWAIT` clause because all the data used in the second loop is different than all the data used in the first loop.

Example

```
subroutine do 2(a,b,c,d,m,n)
  real a(n,n), b(n,n), c(m,m), d(m,m)
  !$OMP PARALLEL SHARED(A,B,C,D,M,N) PRIVATE(I,J)
    !$OMP DO SCHEDULE(DYNAMIC,1)
      do i = 2, n
        do j = 1, i
          b(j,i) = ( a(j,i) + a(j,i-1) ) / 2.0
        end do
      end do
    !$OMP END DO NOWAIT
    !$OMP DO SCHEDULE(DYNAMIC,1)
      do i = 2, m
        do j = 1, i
          d(j,i) = ( c(j,i) + c(j,i-1) ) / 2.0
        end do
```

```
      end do
    !$OMP END DO NOWAIT
  !$OMP END PARALLEL
end
```

Two Difference Operators: SECTIONS Version

The 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 the example above there are `n-1 + m-1` units of work.

Example

```
subroutine sections 1(a,b,c,d,m,n)
  real a(n,n), b(n,n), c(m,m), d(m,m)
  !$OMP PARALLEL SHARED(A,B,C,D,M,N) PRIVATE(I,J)
    !$OMP SECTIONS
       !$OMP SECTION
          do i = 2, n
             do j = 1, i
                b(j,i)=( a(j,i) + a(j,i-1) ) / 2.0
             end do
          end do
       !$OMP SECTION
          do i = 2, m
             do j = 1, i
                d(j,i)=( c(j,i) + c(j,i-1) ) / 2.0
             end do
          end do
    !$OMP END SECTIONS NOWAIT
  !$OMP END PARALLEL
end
```

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.

Example

```
subroutine sp 1a(a,b,n)
  real a(n), b(n)
  !$OMP PARALLEL SHARED(A,B,N) PRIVATE(I)
    !$OMP DO
       do i = 1, n
         a(i) = 1.0 / a(i)
       end do
    !$OMP SINGLE
      a(1) = min( a(1), 1.0 )
    !$OMP END SINGLE
    !$OMP DO
      do i = 1, n
        b(i) = b(i) / a(i)
      end do
    !$OMP END DO NOWAIT
  !$OMP END PARALLEL
end
```

OpenMP* Environment Variables

The Intel® Compiler supports OpenMP* environment variables (with the `OMP_` prefix) and extensions in the form of Intel-specific environment variables (with the `KMP_` prefix).

OpenMP Environment Variables

The syntax examples assume bash on Linux* and Mac OS* X. Use the `set` command for Windows*.

Variable Name	Default	Description and Syntax
`OMP_NUM_THREADS`	Number of processors visible to the operating system.	Sets the maximum number of threads to use for OpenMP* parallel regions if no other value is specified in the application. This environment variable applies to both `-openmp` and `-parallel` (Linux and Mac OS X) or `/Qopenmp` and `/Qparallel` (Windows). Example syntax: `export OMP_NUM_THREADS=value`
`OMP_SCHEDULE`	`STATIC`, no chunk size specified	Sets the run-time schedule type and an optional chunk size. Example syntax: `export OMP_SCHEDULE="kind[,chunk_size]"`
`OMP_DYNAMIC`	`.FALSE.`	Enables (`.TRUE.`) or disables (`.FALSE.`) the dynamic adjustment of the number of threads. Example syntax: `export OMP_DYNAMIC=value`
`OMP_NESTED`	`.FALSE.`	Enables (`.TRUE.`) or disables (`.FALSE.`)nested parallelism. Example syntax: `export OMP_NESTED=value`
`OMP_STACKSIZE`	IA-32 architecture: 2M Intel® 64 and IA-32 Architecures: 4M	Sets the number of bytes to allocate for each OpenMP thread to use as the private stack for the thread. Recommended size is 16M. Use the optional suffixes: B (bytes), K

		(Kilobytes), M (Megabytes), G (Gigabytes), or T (Terabytes) to specify the units. If only value is specified and B, K, M, G, or T is not specified, then size is assumed to be K (Kilobytes). This variable does not affect the native operating system threads created by the user program nor the thread executing the sequential part of an OpenMP program or parallel programs created using `-parallel` (Linux and Mac OS X) or `/Qparallel` (Windows). `kmp_{set,get}_stacksize_s()` routines set/retrieve the value. `kmp_set_stacksize_s()` routine must be called from sequential part, before first parallel region is created. Otherwise, calling `kmp_set_stacksize_s()` has no effect. Related env variables: `KMP_STACKSIZE`. `KMP_STACKSIZE` overrides `OMP_STACKSIZE`. Example syntax: `export OMP_STACKSIZE=value`
`OMP_MAX_ACTIVE_LEVELS`	No enforced limit	Limits the number of simultaneously executing threads in an OpenMP program. If this limit is reached and another native operating system thread encounters OpenMP API calls or constructs, the program can abort with an error message. If this limit is reached when an OpenMP parallel region begins, a one-time warning message might be generated indicating that the number of threads in the team was reduced, but the program will

		continue. This environment variable is only used for programs compiled with the following options: `-openmp` or `-openmp-profile` or `-parallel` (Linux and Mac OS X) and `/Qopenmp` or `/Qopenmp-profile` or `/Qparallel` (Windows). `omp_get_thread_limit()` routine returns the value of the limit. Related env variable: `KMP_ALL_THREADS`. `OMP_THREAD_LIMIT` overrides `KMP_ALL_THREADS`. Example syntax: `export OMP_THREAD_LIMIT=value`
`OMP_THREAD_LIMIT`	No enforced limit	Limits the number of simultaneously executing threads in an OpenMP* program. If this limit is reached and another native operating system thread encounters OpenMP* API calls or constructs, the program can abort with an error message. If this limit is reached when an OpenMP parallel region begins, a one-time warning message might be generated indicating that the number of threads in the team was reduced, but the program will continue. This environment variable is only used for programs compiled with the following options: -openmp or -openmp-profile or -parallel (Linux and Mac OS X) and /Qopenmp or /Qopenmp-profile or /Qparallel (Windows). omp_get_thread_limit() routine returns

	the value of the limit. Related environment variable: KMP_ALL_THREADS. Its value overrides OMP_THREAD_LIMIT. Example syntax: `export OMP_THREAD_LIMIT=value`

Intel Environment Variables Extensions

Variable Name	Default	Description
`KMP_ALL_THREADS`	No enforced limit	Limits the number of simultaneously executing threads in an OpenMP* program. If this limit is reached and another native operating system thread encounters OpenMP* API calls or constructs, the program can abort with an error message. If this limit is reached when an OpenMP parallel region begins, a one-time warning message might be generated indicating that the number of threads in the team was reduced, but the program will continue. This environment variable is only used for programs compiled with the following options: `-openmp` or `-openmp-profile` (Linux and Mac OS X) and `/Qopenmp` or `/Qopenmp-profile` (Windows).
`KMP_BLOCKTIME`	`200` milliseconds	Sets the time, in milliseconds, that a thread should wait, after completing the execution of a parallel region, before sleeping. Use the optional character suffixes: `s` (seconds), `m` (minutes), `h` (hours), or `d` (days)

		to specify the units. Specify `infinite` for an unlimited wait time. See also the throughput execution mode and the `KMP_LIBRARY` environment variable.
`KMP_LIBRARY`	`throughput`	Selects the OpenMP run-time library execution mode. The options for the variable value are `throughput`, `turnaround`, and `serial`.
`KMP_SETTINGS`	0	Enables (1) or disables (0) the printing OpenMP run-time library environment variables during program execution. Two lists of variables are printed: user-defined environment variables settings and effective values of variables used by OpenMP run-time library.
`KMP_STACKSIZE`	IA-32 architecture: `2m` Intel® 64 and IA-64 architectures: `4m`	Sets the number of bytes to allocate for each OpenMP* thread to use as the private stack for the thread. Recommended size is 16`m`. Use the optional suffixes: `b` (bytes), `k` (kilobytes), `m` (megabytes), `g` (gigabytes), or `t` (terabytes) to specify the units. This variable does not affect the native operating system threads created by the user program nor the thread executing the sequential part of an OpenMP* program or parallel programs created using `-parallel` (Linux and Mac OS X) or `/Qparallel`

		(Windows).
KMP_MONITOR_STACKSIZE	max (32k, system minimum thread stack size)	Sets the number of bytes to allocate for the monitor thread, which is used for book-keeping during program execution. Use the optional suffixes: b (bytes), k (kilobytes), m (megabytes), g (gigabytes), or t (terabytes) to specify the units.
KMP_VERSION	.FALSE.	Enables (.TRUE.) or disables (.FALSE.) the printing of OpenMP run-time library version information during program execution.
KMP_AFFINITY	noverbose, respect, granularity=core	Enables run-time library to bind threads to physical processing units. See Thread Affinity Interface for more information on the default and the affect this environment variable has on the parallel environment.
KMP_CPUINFO_FILE	none	Specifies an alternate file name for file containing machine topology description. The file must be in the same format as /proc/cpuinfo.

GNU Environment Variables Extensions

These environment variables are GNU extensions. They are recognized by the Intel OpenMP compatibility library.

Variable Name	Default	Description
GOMP_STACKSIZE	See OMP_STACKSIZE description	OMP_STACKSIZE overrides GOMP_STACKSIZE. KMP_STACKSIZE overrides OMP_STACKSIZE and GOMP_STACKSIZE

`GOMP_CPU_AFFINITY` TBD

OpenMP* Directives and Clauses Summary

This is a summary of the OpenMP* directives and clauses supported in the Intel® Compiler. For detailed information about the OpenMP API, see the *OpenMP Application Program Interface* Version 2.5 specification, which is available from the OpenMP web site (http://www.openmp.org/).

OpenMP Directives

Directive	Description
`PARALLEL` `END PARALLEL`	Defines a parallel region.
`TASK` `END TASK`	Defines a task region.
`DO` `END DO`	Identifies an iterative worksharing construct in which the iterations of the associated loop should be divided among threads in a team.
`SECTIONS` `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` `END SINGLE`	Identifies a construct that specifies that the associated structured block is executed by only one thread in the team.
`PARALLEL DO` `END PARALLEL DO`	A shortcut for a `PARALLEL` region that contains a single `DO` directive. **Note** The OpenMP `PARALLEL DO` or `DO` 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 compiler issues a syntax error.
`PARALLEL SECTIONS` `END PARALLEL SECTIONS`	Provides a shortcut form for specifying a parallel region containing a single `SECTIONS` construct.
`MASTER` `END MASTER`	Identifies a construct that specifies a structured block that is executed by only the master thread of the team.
`CRITICAL[`*`name`*`]` `END CRITICAL[`*`name`*`]`	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 *`name`* argument.
`TASKWAIT`	Indicates a wait on the completion of child tasks generated since the beginning of the current task.
`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` `END ORDERED`	The enclosed structured block 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 ({operator|intrinsic}:*list*)	Performs a reduction on variables 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 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 (*expression*)	The enclosed parallel region is executed in parallel only if the *expression* evaluates to .TRUE., otherwise the parallel region is serialized. The expression must be scalar logical.
NUM_THREADS (*expression*)	Requests the number of threads specified by *expression* for the parallel region. The expressions must be scalar integers.
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.
COLLAPSE (n)	Specifies how many loops are associated with the OpenMP loop construct for collapsing.
COPYIN (*list*)	Provide a mechanism to specify

	that the master thread data values be copied to the `THREADPRIVATE` copy of the common blocks or variables specified in *`list`* at the beginning of the parallel region.
`UNTIED`	Indicates that a resumed task does not have to be executed by same thread executing it before it was suspended.

Directives and Clauses Cross-reference

See Data Scope Attribute Clauses Overview.

Directive	Use these Clauses
`PARALLEL` `END PARALLEL`	• `IF` • `NUM_THREADS` • `DEFAULT` • `PRIVATE` • `FIRSTPRIVATE` • `SHARED` • `COPYIN` • `REDUCTION`
`DO` `END DO`	• `SCHEDULE` • `PRIVATE` • `FIRSTPRIVATE` • `LASTPRIVATE` • `REDUCTION` • `ORDERED` • `NOWAIT` • `COLLAPSE`
`SECTIONS` `END SECTIONS`	• `PRIVATE` • `FIRSTPRIVATE`

	• LASTPRIVATE • REDUCTION • NOWAIT
SECTION	None
SINGLE END SINGLE	• PRIVATE • FIRSTPRIVATE • COPYPRIVATE • NOTWAIT
PARALLEL DO END PARALLEL DO	• IF • NUM_THREADS • SCHEDULE • DEFAULT • PRIVATE • FIRSTPRIVATE • LASTPRIVATE • SHARED • COPYIN • REDUCTION • ORDERED • COLLAPSE
PARALLEL SECTIONS END PARALLEL SECTIONS	• IF • NUM_THREADS • DEFAULT • PRIVATE • FIRSTPRIVATE • LASTPRIVATE • SHARED • COPYIN • REDUCTION
All others	None

OpenMP* Library Support

OpenMP* Run-time Library Routines

OpenMP* provides several run-time library routines to help you manage 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 let you 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.

This topic provides a summary of the OpenMP run-time library routines. See OpenMP* Support Overview for additional resources; refer to the OpenMP API Version 3.0 specification for detailed information about using these routines.

Include the appropriate declarations of the routines by adding a statement similar to the following in your source code:

Example

```
include "omp_lib.h"
```

The header files are provided in the `../include` (Linux* and Mac OS* X) or `..\include` (Windows*) directory of your compiler installation.

The following tables specify the interfaces to these routines. (The names for the routines are in user name space.)

Execution Environment Routines

Use these routines to monitor and influence threads and the parallel environment.

Function	Description
`SUBROUTINE OMP_SET_NUM_THREADS(num_threads)` `INTEGER num_threads`	Sets the number of threads to use for subsequent parallel regions created by the calling thread.
`INTEGER FUNCTION OMP_GET_NUM_THREADS()`	Returns the number of threads that are being used in the current parallel region. This function does not return the value inherited by the calling thread from the `OMP_SET_NUM_THREADS()` function.

`INTEGER FUNCTION OMP_GET_MAX_THREADS()`	Returns the number of threads available to subsequent parallel regions created by the calling thread. This function returns the value inherited by the calling thread from the `OMP_SET_NUM_THREADS()` function.
`INTEGER FUNCTION OMP_GET_THREAD_NUM()`	Returns the thread number of the calling thread, within the context of the current parallel region..
`INTEGER FUNCTION OMP_GET_NUM_PROCS()`	Returns 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)` `LOGICAL dynamic_threads`	Enables or disables dynamic adjustment of the number of threads used to execute a parallel region. If *`dynamic_threads`* is `.TRUE.`, dynamic threads are enabled. If *`dynamic_threads`* is `.FALSE.`, dynamic threads are disabled. Dynamic 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)` `LOGICAL 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.`.

`SUBROUTINE`
`OMP_SET_SCHEDULE(kind,modifier)`
`INTEGER(KIND=omp_sched_t) kind`
`INTEGER modifier`
	Determines the schedule of a worksharing loop that is applied when 'runtime' is used as schedule kind.

`SUBROUTINE`
`OMP_GET_SCHEDULE(kind,modifier)`
`INTEGER(KIND=omp_sched_t) kind`
`INTEGER modifier`
	Returns the schedule of a worksharing loop that is applied when the 'runtime' schedule is used.

`INTEGER FUNCTION` `OMP_GET_THREAD_LIMIT()`	Returns the maximum number of simultaneously executing threads in an OpenMP* program.

`SUBROUTINE`
`OMP_SET_MAX_ACTIVE_LEVELS(max_active_levels)`
`INTEGER max_active_levels`
	Limits the number of nested active parallel regions. The call is ignored if negative max_active_levels specified.

`INTEGER FUNCTION` `OMP_GET_MAX_ACTIVE_LEVELS()`	Returns the maximum number of nested active parallel regions.
`INTEGER FUNCTION` `OMP_GET_ACTIVE_LEVEL()`	Returns the number of nested, active parallel regions enclosing the task that contains the call.

`INTEGER FUNCTION OMP_GET_LEVEL()`	Returns the number of nested parallel regions (whether active or inactive) enclosing the task that contains the call, not including the implicit parallel region.
`INTEGER FUNCTION OMP_GET_ANCESTOR_THREAD_NUM(level) INTEGER level`	Returns the thread number of the ancestor at a given nest level of the current thread.
`INTEGER FUNCTION OMP_GET_TEAM_SIZE(level) INTEGER level`	Returns the size of the thread team to which the ancestor belongs.

Lock Routines

Use these routines to affect OpenMP locks.

Function	Description
`SUBROUTINE OMP_INIT_LOCK(`*`lock`*`) INTEGER (KIND=OMP_LOCK_KIND)::`*`lock`*	Initializes the lock associated with *`lock`* for use in subsequent calls.
`SUBROUTINE OMP_DESTROY_LOCK(`*`lock`*`) INTEGER(KIND=OMP_LOCK_KIND)::`*`lock`*	Causes the lock specified by *`lock`* to become undefined or uninitialized. The lock must be initialized and not locked.
`SUBROUTINE OMP_SET_LOCK(`*`lock`*`) INTEGER(KIND=OMP_LOCK_KIND)::`*`lock`*	Forces the executing thread to wait until the lock associated with *`lock`* is available. The thread is granted ownership of the lock when it becomes available. The lock must be initialized.
`SUBROUTINE OMP_UNSET_LOCK(`*`lock`*`) 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)` `INTEGER(KIND=OMP_LOCK_KIND)::lock`	Attempts to set the lock associated with `lock`. If successful, returns `.TRUE.`, otherwise returns `.FALSE.`. The lock must be initialized.
`SUBROUTINE OMP_INIT_NEST_LOCK(lock)` `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)` `INTEGER(KIND=OMP_NEST_LOCK_KIND)::lock`	Causes the nested lock associated with `lock` to become undefined or uninitialized. The lock must be initialized and not locked.
`SUBROUTINE OMP_SET_NEST_LOCK(lock)` `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. The lock must be initialized.
`SUBROUTINE OMP_UNSET_NEST_LOCK(lock)` `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)` `INTEGER(KIND=OMP_NEST_LOCK_KIND)::lock`	Attempts to set the nested lock specified by `lock`. If successful, returns the nesting count, otherwise returns zero.

Timing Routines

Function	Description
`DOUBLE PRECISION FUNCTION OMP_GET_WTIME()`	Returns a double precision value equal to the elapsed wall clock 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 to OpenMP*

The Intel® Compiler implements the following group of routines as an extensions to the OpenMP* run-time library:

- Getting and setting the execution environment
- Getting and setting stack size for parallel threads
- Memory allocation
- Getting and setting thread sleep time for the throughput execution mode

The Intel extension routines described in this section can be used for low-level tuning to verify that the library code and application are functioning as intended. These routines are generally not recognized by other OpenMP-compliant compilers, which may cause the link stage to fail in other compiler. These OpenMP routines require that you use the `-openmp-stubs` (Linux* and Mac OS* X) or `/Qopenmp-stubs` (Windows*) command-line option to execute.

See OpenMP* Run-time Library Routines for details about including support for these declarations in your source, and see OpenMP* Support Libraries for detailed information about execution environment (mode).

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 `OMP_STACKSIZE` environment variable rather than the `KMP_SET_STACKSIZE_S()` library routine.

Note

A run-time call to an Intel extension routine takes precedence over the corresponding environment variable setting.

Execution Environment Routines

Function	Description
`SUBROUTINE KMP_SET_DEFAULTS(STRING) CHARACTER*(*) STRING`	Sets OpenMP environment variables defined as a list of variables separated by "|" in the argument.
`SUBROUTINE`	Sets execution mode to throughput, which

`KMP_SET_LIBRARY_THROUGHPUT()`	is the default. Allows the application to determine the runtime environment. Use in multi-user environments.
`SUBROUTINE KMP_SET_LIBRARY_TURNAROUND()`	Sets execution mode to turnaround. Use in dedicated parallel (single user) environments.
`SUBROUTINE KMP_SET_LIBRARY_SERIAL()`	Sets execution mode to serial.
`SUBROUTINE KMP_SET_LIBRARY(LIBNUM) INTEGER (KIND=OMP_INTEGER_KIND) LIBNUM`	Sets execution mode indicated by the value passed to the function. Valid values are: • 1 - serial mode • 2 - turnaround mode • 3 - throughput mode Call this routine before the first parallel region is executed.
`FUNCTION KMP_GET_LIBRARY() INTEGER (KIND=OMP_INTEGER_KIND) KMP_GET_LIBRARY`	Returns a value corresponding to the current execution mode: 1 (serial), 2 (turnaround), or 3 (throughput).

Stack Size

For IA-64 architecture it is recommended to always use `KMP_SET_STACKSIZE_S()` and `KMP_GET_STACKSIZE_S()`. The `_S()` variants must be used if you need to set a stack size ≥ 2**31 bytes (2 gigabytes).

Function	Description
`FUNCTION KMP_GET_STACKSIZE_S() 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 with `KMP_SET_STACKSIZE_S()` routine, prior to the first parallel region or via the `KMP_STACKSIZE` environment variable.
`FUNCTION KMP_GET_STACKSIZE() INTEGER KMP_GET_STACKSIZE`	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) 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_S(size) INTEGER size`	Provided for backward compatibility only. Use `KMP_SET_STACKSIZE_S(size)` for compatibility across different families of Intel processors.

Memory Allocation

The Intel® 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 freed by a different thread, this mode of operation has a slight performance penalty.

Working with the local heap might lead to improved application performance since synchronization is not required.

Function	Description
`FUNCTION KMP_MALLOC(size) INTEGER(KIND=KMP_POINTER_KIND)KMP_MALLOC INTEGER(KIND=KMP_SIZE_T_KIND)size`	Allocate memory block of *size* bytes from thread-local heap.
`FUNCTION KMP_CALLOC(nelem,elsize) INTEGER(KIND=KMP_POINTER_KIND)KMP_CALLOC INTEGER(KIND=KMP_SIZE_T_KIND)nelem INTEGER(KIND=KMP_SIZE_T_KIND)elsize`	Allocate array of *nelem* elements of size *elsize* from thread-local heap.
`FUNCTION KMP_REALLOC(ptr, size) INTEGER(KIND=KMP_POINTER_KIND)KMP_REALLOC INTEGER(KIND=KMP_POINTER_KIND)ptr`	Reallocate memory block at address *ptr* and *size* bytes

`INTEGER(KIND=KMP_SIZE_T_KIND)`*`size`*	from thread-local heap.
`SUBROUTINE KMP_FREE(`*`ptr`*`)` `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()`.

Thread Sleep Time

In the throughput execution mode, threads wait for new parallel work at the ends of parallel regions, and then sleep, after a specified period of time. This time interval can be set by the `KMP_BLOCKTIME` environment variable or by the `KMP_SET_BLOCKTIME()` function.

Function	Description
`FUNCTION KMP_GET_BLOCKTIME(` `INTEGER KMP_GET_BLOCKTIME`	Returns the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping, as set either by the `KMP_BLOCKTIME` environment variable or by `KMP_SET_BLOCKTIME()`.
`FUNCTION KMP_SET_BLOCKTIME(`*`msec`*`)` `INTEGER` *`msec`*	Sets the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping. This routine affects the block time setting for the calling thread and any OpenMP team threads formed by the calling thread. The routine does not affect the block time for any other threads.

OpenMP* Support Libraries

The Intel® Compiler provides support libraries for OpenMP*. There are several kinds of libraries:

- Performance: supports parallel OpenMP execution.

- Profile: supports parallel OpenMP execution and allows use of Intel® Thread Profiler.
- Stubs: supports serial execution of OpenMP applications.

Each kind of library is available for both dynamic and static linking.

Note

The use of static OpenMP libraries is not recommended, because they might cause multiple libraries to be linked in an application. The condition is not supported and could lead to unpredictable results.

This section describes the compatibility libraries and legacy libraries provided with the Intel compiler, as well as the selection of run-time execution modes.

Compatibility Libraries

To use the Compatibility OpenMP libraries, specify the (default) `/Qopenmp-lib:compat` (Windows OS) or `-openmp-lib compat` (Linux OS and Mac OS X) compiler option during linking.

On Linux and Mac OS X systems, to use dynamically linked libraries during linking, specify `-openmp-link=dynamic` option; to use static linking, specify the `-openmp-link=static` option.

On Windows systems, to use dynamically linked libraries during linking, specify the `/MD` and `/Qopenmp-link:dynamic` options; to use static linking, specify the `/MT` and `/Qopenmp-link:static` options.

To provide run-time support for dynamically linked applications, the supplied DLL (Windows OS) or shared library (Linux OS and Mac OS X) must be available to the application at run time.

Performance Libraries

To use these libraries, specify the `-openmp` (Linux* and Mac OS* X) or `/Qopenmp` (Windows*) compiler option.

Operating System	Dynamic Link	Static Link
Linux	`libiomp5.so`	`libiomp5.a`
Mac OS X	`libiomp5.dylib`	`libiomp5.a`
Windows	`libiomp5md.lib` `libiomp5md.dll`	`libiomp5mt.lib`

Profile Libraries

To use these libraries, specify `-openmp-profile` (Linux* and Mac OS* X) or `/Qopenmp-profile` (Windows*) compiler option. These allow you to use Intel® Thread Profiler to analyze OpenMP applications.

Operating System	Dynamic Link	Static Link
Linux	`libiompprof5.so`	`libiompprof5.a`
Mac OS X	`libiompprof5.dylib`	`libiompprof5.a`
Windows	`libiompprof5md.lib` `libiompprof5md.dll`	`libiompprof5mt.lib`

Stubs Libraries

To use these libraries, specify `-openmp-stubs` (Linux* and Mac OS* X) or `/Qopenmp-stubs` (Windows*) compiler option. These allow you to compile OpenMP applications in serial mode and provide stubs for OpenMP routines and extended Intel-specific routines.

Operating System	Dynamic Link	Static Link
Linux	`libiompstubs5.so`	`libiompstubs5.a`
Mac OS X	`libiompstubs5.dylib`	`libiompstubs5.a`
Windows	`libiompstubs5md.lib` `libiompstubs5md.dll`	`libiompstubs5mt.lib`

Legacy Libraries

To use the Legacy OpenMP libraries, specify the `/Qopenmp-lib:legacy` (Windows OS) or `-openmp-lib legacy` (Linux OS and Mac OS X) compiler options during linking. Legacy libraries are deprecated.

On Linux and Mac OS X systems, to use dynamically linked libraries during linking, specify the `-openmp-link:dynamic` option; to use static linking, specify the `-openmp-link:static` option.

On Windows systems, to use dynamically linked libraries during linking, specify the `/MD` and `/Qopenmp-link=dynamic` options; to use static linking, specify the `/MT` and `/Qopenmp-link=static` options.

To provide run-time support for dynamically linked applications, the supplied DLL (Windows OS) or shared library (Linux OS and Mac OS X) must be available to the application at run time.

Performance Libraries

To use these libraries, specify `-openmp` (Linux* and Mac OS* X) or `/Qopenmp` (Windows*) compiler option.

Operating System	Dynamic Link	Static Link
Linux	`libguide.so`	`libguide.a`
Mac OS X	`libguide.dylib`	`libguide.a`
Windows	`libguide40.lib` `libguide40.dll`	`libguide.lib`

Profile Libraries

To use these libraries, specify `-openmp-profile` (Linux* and Mac OS* X) or `/Qopenmp-profile` (Windows*) compiler option. These allow you to use Intel® Thread Profiler to analyze OpenMP applications.

Operating System	Dynamic Link	Static Link
Linux	`libguide_stats.so`	`libguide_stats.a`
Mac OS X	`libguide_stats.dylib`	`libguide_stats.a`
Windows	`libguide40_stats.lib` `libguide40_stats.dll`	`libguide_stats.lib`

Stubs Libraries

To use these libraries, specify `-openmp-stubs` (Linux* and Mac OS* X) or `/Qopenmp-stubs` (Windows*) compiler option. These allow you to compile OpenMP applications in serial mode and provide stubs for OpenMP routines and extended Intel-specific routines.

Operating System	Dynamic Link	Static Link
Linux	`libompstub.so`	`libompstub.a`
Mac OS X	`libompstub.dylib`	`libompstub.a`
Windows	`libompstub40.lib` `libompstub40.dll`	`libompstub.lib`

Execution modes

The Intel compiler enables you to run an application under different execution modes specified at run time; the libraries support the turnaround, throughput, and serial modes. Use the `KMP_LIBRARY` environment variable to select the modes at run time.

Mode	Description
throughput (default)	The throughput mode allows the program to detect its environment conditions (system load) and adjust resource usage to produce efficient execution in a dynamic environment. 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 yield to other threads while waiting for more parallel work. 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. Until more parallel work becomes available, sleeping allows processor and resources to be used for other work 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 blocktime value may offer better overall performance if your application contains non-OpenMP threaded code that executes between parallel regions. A larger 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.
turnaround	The turnaround mode is designed to keep active all processors involved in the parallel computation, which minimizes execution time of a single job. In this mode, the worker threads actively wait for more parallel work, without yielding to other threads. 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 use all processors all of the time.  Note Avoid over-allocating system resources. The condition can occur 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.
serial	The serial mode forces parallel applications to run as a single thread.

Using the OpenMP Compatibility Libraries

This section describes the steps needed to set up and use the OpenMP Compatibility Libraries from the command line. On Windows* systems, you can also build applications compiled with the OpenMP Compatibility libraries in the Microsoft Visual Studio* development environment.

For a summary of the support provided by the Compatibility and Legacy libraries provided with Intel compilers, see OpenMP* Source Compatibility and Interoperability with Other Compilers.

For a list of the options and libraries used by the OpenMP libraries, see OpenMP Support Libraries.

During C/C++ compilation, ensure the version of `omp.h` used when compiling is the version provided by that compiler. For example, on Linux systems when compiling with the GNU C/C++ compiler, use the `omp.h` provided with the GNU C/C++ compiler. Similarly, during Fortran compilation, ensure that the version of `omp_lib.h` or `omp_lib.mod` used when compiling is the version provided by that compiler.

The following table lists the commands used by the various command-line compilers for both C and C++ source files.:

Operating System	C Source Module	C++ Source Module
Linux*	GNU: `gcc` Intel: `icc`	GNU: `g++` Intel: `icpc`
Mac OS* X	GNU: `gcc` Intel: `icc`	GNU: `g++` Intel: `icpc`
Windows*	Visual C++: `cl` Intel: `icl`	Visual C++: `cl` Intel: `icl`

The command for the Intel® Fortran compiler is `ifort` on Linux, Mac OS X, and Windows operating systems.

For information on the OpenMP libraries and options used by the Intel compiler, see OpenMP Support Libraries.

Command-Line Examples, Windows OS

To compile and link (build) the entire application with one command using the Compatibility libraries, specify the following Intel compiler command:

Type of File	Commands

C source, dynamic link	`icl /MD /Qopenmp /Qopenmp-lib:compat hello.c`
C++ source, dynamic link	`icl /MD /Qopenmp /Qopenmp-lib:compat hello.cpp`
Fortran source, dynamic link	`ifort /MD /Qopenmp /Qopenmp-lib:compat hello.f90`

By default, the Intel compilers perform a dynamic link of the OpenMP libraries. To perform a static link (not recommended), use the `/MT` option in place of the `/MD` option above and add the option `/Qopenmp-link:static`. The Intel compiler option `/Qopenmp-link` controls whether the linker uses static or dynamic OpenMP libraries on Windows OS systems (default is `/Qopenmp-link:dynamic`).

When using Microsoft Visual C++ compiler, you should link with the Intel OpenMP compatibility library. You need to avoid linking the Microsoft OpenMP run-time library (vcomp) and explicitly pass the name of the Intel OpenMP compatibility library as linker options (following `/link`):

Type of File	Commands
C source, dynamic link	`cl /MD /openmp hello.c /link /nodefaultlib:vcomp libiomp5md.lib`
C++ source, dynamic link	`cl /MD /openmp hello.cpp /link /nodefaultlib:vcomp libiomp5md.lib`

Performing a static link is not recommended, but would require use of the `/MT` option in place of the `/MD` option above.

You can also use both Intel C++ and Visual C++ compilers to compile parts of the application and create object files (object-level interoperability). In this example, the Intel compiler links the entire application:

Type of File	Commands

C source, dynamic link	`cl /MD /openmp hello.cpp /c f1.c f2.c`
	`icl /MD /Qopenmp /Qopenmp-lib:compat /c f3.c f4.c`
	`icl /MD /Qopenmp /Qopenmp-lib:compat f1.obj f2.obj f3.obj f4.obj /Feapp /link /nodefaultlib:vcomp`

The first command produces two object files compiled by Visual C++ compiler, and the second command produces two more object files compiled by Intel C++ Compiler. The final command links all four object files into an application.

Alternatively, the third line below uses the Visual C++ linker to link the application and specifies the Compatibility library `libiomp5md.lib` at the end of the third command:

Type of File	Commands

C source, dynamic link	`cl /MD /openmp hello.cpp /c f1.c f2.c`
	`icl /MD /Qopenmp /Qopenmp-lib:compat /c f3.c f4.c`
	`link f1.obj f2.obj f3.obj f4.obj /out:app.exe`
	`/nodefaultlib:vcomp libiomp5md.lib`

The following example shows the use of interprocedural optimization by the Intel compiler on several files, the Visual C++ compiler compiles several files, and the Visual C++ linker links the object files to create the executable:

Type of File	Commands

C source, dynamic link	`icl /MD /Qopenmp /Qopenmp-lib:compat /O3 /Qipo /Qipo-c f1.c f2.c f3.c`
	`cl /MD /openmp /O2 /c f4.c f5.c`
	`cl /MD /openmp /O2 ipo out.obj f4.obj f5.obj /Feapp /link /nodefaultlib:vcomp libiomp5md.lib`

The first command uses the Intel® C++ Compiler to produce an optimized multi-file object file named `ipo_out.obj` by default (the `/Fe` option is not required; see Using IPO). The second command uses the Visual C++ compiler to produce two more object files. The third command uses the Visual C++ `cl` command to link all three object files using the Intel compiler OpenMP Compatibility library. Performing a static link (not recommended) requires use of the `/MT` option in place of the `/MD` option in each line above.

Command-Line Examples, Linux OS and Mac OS X

To compile and link (build) the entire application with one command using the Compatibility libraries, specify the following Intel compiler command:

Type of File	Commands
C source	`icc -openmp -openmp-lib=compat hello.c`
C++ source	`icpc -openmp -openmp-lib=compat hello.cpp`
Fortran source	`ifort -openmp -openmp-lib=compat hello.f90`

By default, the Intel compilers perform a dynamic link of the OpenMP libraries. To perform a static link (not recommended), add the option `-openmp-link static`. The Intel compiler option `-openmp-link` controls whether the linker uses static or dynamic

OpenMP libraries on Linux OS and Mac OS X systems (default is `-openmp-link dynamic`).

You can also use both Intel C++ `icc/icpc` and GNU `gcc/g++` compilers to compile parts of the application and create object files (object-level interoperability). In this example, the GNU compiler compiles the C file `foo.c` (the `gcc` option `-fopenmp` enables OpenMP support), and the Intel compiler links the application using the OpenMP Compatibility library:

Type of File	Commands
C source	`gcc -fopenmp -c foo.c` `icc -openmp -openmp-lib=compat foo.o`
C++ source	`g++ -fopenmp -c foo.cpp` `icpc -openmp -openmp-lib=compat foo.o`

When using GNU `gcc` or `g++` compiler to link the application with the Intel compiler OpenMP compatibility library, you need to explicitly pass the Intel OpenMP compatibility library name using the `-l` option, the GNU `pthread` library using the `-l` option, and path to the Intel libraries where the Intel C++ compiler is installed using the `-L` option:

Type of File	Commands
C source	`gcc -fopenmp -c foo.c bar.c` `gcc foo.o bar.o -liomp5 -lpthread -L<icc_dir>/lib`

You can mix object files, but it is easier to use the Intel compiler to link the application so you do not need to specify the `gcc -l` option, `-L` option, and the `-lpthread` option:

Type of File	Commands

C source	`gcc -fopenmp -c foo.c`
	`icc -openmp -c bar.c`
	`icc -openmp -openmp=compat foo.o bar.o`

You can mix OpenMP object files compiled with the GNU gcc compiler, the Intel® C++ Compiler, and the Intel® Fortran Compiler. This example uses use the Intel Fortran Compiler to link all the objects:

Type of File	Commands

Mixed C and Fortran sources	`ifort -openmp -c foo.f`
	`icc -openmp -c ibar.c`
	`gcc -fopenmp -c gbar.c`
	`ifort -openmp -openmp-lib=compat foo.o ibar.o gbar.o`

When using the Intel Fortran compiler, if the main program does not exist in a Fortran object file that is compiled by the Intel Fortran Compiler `ifort`, specify the `-nofor-main` option on the `ifort` command line during linking.

Note

Do not mix objects created by the Intel Fortran Compiler (`ifort`) with the GNU Fortran Compiler (`gfortran`); instead, recompile all Fortran sources with the same Fortran compiler. The GNU Fortran Compiler is only available on Linux operating systems.

Similarly, you can mix object files compiled with the Intel® C++ Compiler, the GNU C/C++ compiler, and the GNU Fortran Compiler (gfortran), if you link with the GNU Fortran Compiler (`gfortran`). When using GNU `gfortran` compiler to link the application with the Intel compiler OpenMP compatibility library, you need to explicitly pass the Intel OpenMP compatibility library name and the Intel `irc` libraries using the `-l` options, the GNU `pthread` library using the `-l` option, and path to the Intel libraries where the Intel C++ compiler is installed using the `-L`. option. You do not need to specify the `-fopenmp` option on the link line:

Type of File	Commands

Mixed C and GNU Fortran sources	`gfortran -fopenmp -c foo.f`
	`icc -openmp -c ibar.c`
	`gcc -fopenmp -c gbar.c`
	`gfortran foo.o ibar.o gbar.o -lirc -liomp5 -lpthread -lc -L<icc_dir>/lib`

Alternatively, you could use the Intel compiler to link the application, but need to pass multiple `gfortran` libraries using the `-l` options on the link line:

Type of File	Commands

Mixed C and Fortran sources	`gfortran -fopenmp -c foo.f`
	`icc -openmp -c ibar.c`
	`icc -openmp -openmp-lib=compat foo.o bar.o -lgfortranbegin -lgfortran`

Thread Affinity Interface (Linux* and Windows*)

The Intel® compiler OpenMP* runtime library has the ability to bind OpenMP threads to physical processing units. The interface is controlled using the `KMP_AFFINITY` environment variable. Depending on the system (machine) topology, application, and operating system, thread affinity can have a dramatic effect on the application speed.

Thread affinity restricts execution of certain threads (virtual execution units) to a subset of the physical processing units in a multiprocessor computer. Depending upon the topology of the machine, thread affinity can have a dramatic effect on the execution speed of a program.

Thread affinity is supported on Windows OS systems and versions of Linux OS systems that have kernel support for thread affinity, but is not supported by Mac OS* X. The thread affinity interface is supported only for Intel® processors.

The Intel compiler's OpenMP runtime library has the ability to bind OpenMP threads to physical processing units. There are three types of interfaces you can use to specify this binding, which are collectively referred to as the Intel OpenMP Thread Affinity Interface:

- The high-level affinity interface uses an environment variable to determine the machine topology and assigns OpenMP threads to the processors based upon their physical location in the machine. This interface is controlled entirely by the KMP_AFFINITY environment variable.
- The mid-level affinity interface uses an environment variable to explicitly specifies which processors (labeled with integer IDs) are bound to OpenMP threads. This interface provides compatibility with the GNU gcc* GOMP_CPU_AFFINITY environment variable, but you can also invoke it by using the KMP_AFFINITY environment variable. The GOMP_CPU_AFFINITY environment variable is supported on Linux systems only, but users on Windows or Linux systems can use the similar functionality provided by the KMP_AFFINITY environment variable.
- The low-level affinity interface uses APIs to enable OpenMP threads to make calls into the OpenMP runtime library to explicitly specify the set of processors on which they are to be run. This interface is similar in nature to `sched_setaffinity` and related functions on Linux* systems or to `SetThreadAffinityMask` and related functions on Windows* systems. In addition, you can specify certain options of the KMP_AFFINITY environment variable to affect the behavior of the low-level API interface. For example, you can set the affinity type KMP_AFFINITY to disabled, which disables the low-level affinity interface, or you could use the KMP_AFFINITY or GOMP_CPU_AFFINITY environment variables to set the initial affinity mask, and then retrieve the mask with the low-level API interface.

The following terms are used in this section

- The total number of processing elements on the machine is referred to as the number of *OS thread contexts.*
- Each processing element is referred to as an Operating System processor, or *OS proc*.
- Each OS processor has a unique integer identifier associated with it, called an *OS proc ID*.
- The term *package* refers to a single or multi-core processor chip.
- The term *OpenMP Global Thread ID* (GTID) refers to an integer which uniquely identifies all threads known to the Intel OpenMP runtime library. The thread that

first initializes the library is given GTID 0. In the normal case where all other threads are created by the library and when there is no nested parallelism, then *n-threads-var* – 1 new threads are created with GTIDs ranging from 1 to *ntheads-var* – 1, and each thread's GTID is equal to the OpenMP thread number returned by function `omp_get_thread_num`(). The high-level and mid-level interfaces rely heavily on this concept. Hence, their usefulness is limited in programs containing nested parallelism. The low-level interface does not make use of the concept of a GTID, and can be used by programs containing arbitrarily many levels of parallelism.

Note

Linux only. For Cluster OpenMP*, this environment variable applies individually to each process created by the run-time system, and global thread IDs are contiguous within each process and across all processes. An individual machine topology map and message listing is created for each process.

The `KMP_AFFINITY` Environment Variable

The `KMP_AFFINITY` environment variable uses the following general syntax:

Syntax

`KMP_AFFINITY=[<modifier>,...]<type>[,<permute>][,<offset>]`

For example, to list a machine topology map, specify `KMP_AFFINITY=verbose,none` to use a *modifier* of `verbose` and a *type* of `none`.

The following table describes the supported specific arguments.

Argument	Default	Description

`modifier`	`noverbose`
	`respect`
	`granularity=core`
		Optional. String consisting of keyword and specifier.
		• `granularity=<specifier>` takes the following specifiers: `fine`, `thread`, and `core`
		• `norespect`
		• `noverbose`
		• `nowarnings`
		• `proclist={<proc-list>}`
		• `respect`
		• `verbose`
		• `warnings`

		The syntax for <*proc-list*> is explained in mid-level affinity interface.
type	none	Required string. Indicates the thread affinity to use. • compact • disabled • explicit • none • scatter • logical (deprecated; instead use compact, but omit any permute value) • physical (deprecated; instead use scatter, possibly with an offset value) The logical and physical types are deprecated but supported for backward compatibility.
permute	0	Optional. Positive integer value. Not valid with type values of explicit, none, or disabled.
offset	0	Optional. Positive integer value. Not valid with type values of explicit, none, or disabled.

Affinity Types

Type is the only required argument.

type = none (default)

Does not bind OpenMP threads to particular thread contexts; however, if the operating system supports affinity, the compiler still uses the OpenMP thread affinity interface to determine machine topology. Specify KMP_AFFINITY=verbose,none to list a machine topology map.

type = compact

Specifying compact assigns the OpenMP thread <*n*>+1 to a free thread context as close as possible to the thread context where the <*n*> OpenMP thread was placed. For example, in a topology map, the nearer a node is to the root, the more significance the node has when sorting the threads.

type = disabled

Specifying `disabled` completely disables the thread affinity interfaces. This forces the OpenMP run-time library to behave as if the affinity interface was not supported by the operating system. This includes the low-level API interfaces such as `kmp_set_affinity` and `kmp_get_affinity`, which have no effect and will return a nonzero error code.

type = explicit

Specifying `explicit` assigns OpenMP threads to a list of OS proc IDs that have been explicitly specified by using the proclist= modifier, which is required for this affinity type. See Explicitly Specifying OS Proc IDs (GOMP_CPU_AFFINITY).

type = scatter

Specifying `scatter` distributes the threads as evenly as possible across the entire system. `scatter` is the opposite of `compact`; so the leaves of the node are most significant when sorting through the machine topology map.

Deprecated Types: logical and physical

Types `logical` and `physical` are deprecated and may become unsupported in a future release. Both are supported for backward compatibility.

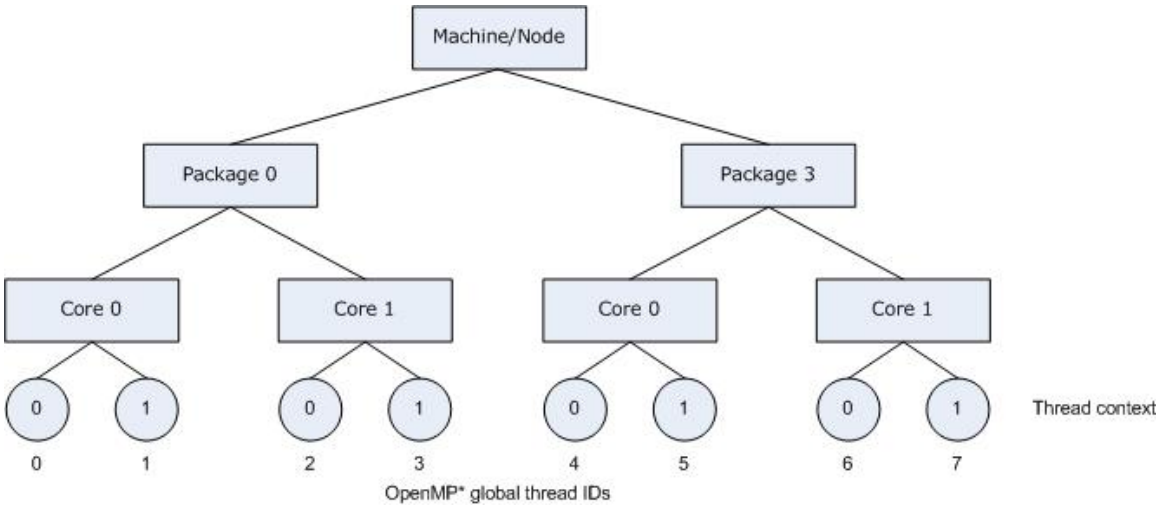
For `logical` and `physical` affinity types, a single trailing integer is interpreted as an `offset` specifier instead of a `permute` specifier. In contrast, with `compact` and `scatter` types, a single trailing integer is interpreted as a `permute` specifier.

- Specifying `logical` assigns OpenMP threads to consecutive logical processors, which are also called hardware thread contexts. The type is equivalent to `compact`, except that the `permute` specifier is not allowed. Thus, `KMP_AFFINITY=logical,`*`n`* is equivalent to `KMP_AFFINITY=compact,0,`*`n`* (this equivalence is true regardless of the whether or not a `granularity=fine` modifier is present).
- Specifying `physical` assigns threads to consecutive physical processors (cores). For systems where there is only a single thread context per core, the type is equivalent to `logical`. For systems where multiple thread contexts exist per core, `physical` is equivalent to `compact` with a `permute` specifier of 1; that is, `KMP_AFFINITY=physical,`*`n`* is equivalent to `KMP_AFFINITY=compact,1,`*`n`* (regardless of the whether or not a `granularity=fine` modifier is present). This equivalence means that when the compiler sorts the map it should permute the innermost level of the machine topology map to the outermost, presumably the thread context level. This type does not support the `permute` specifier.

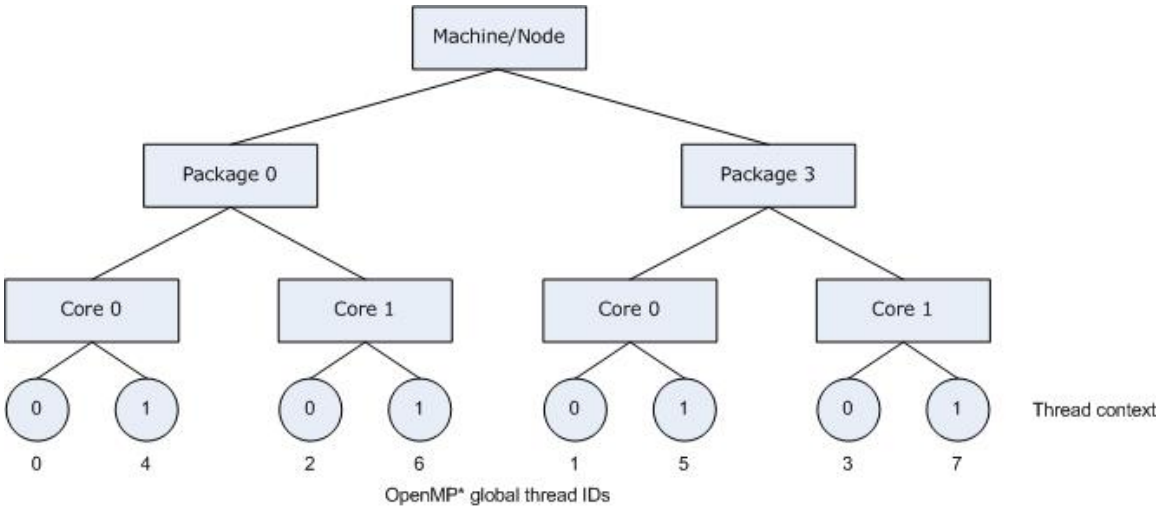
Examples of Types `compact` and `scatter`

The following figure illustrates the topology for a machine with two processors, and each processor has two cores; further, each core has Hyper-Threading Technology (HT Technology) enabled.

The following figure also illustrates the binding of OpenMP thread to hardware thread contexts when specifying `KMP_AFFINITY=granularity=fine,compact`.



Specifying `scatter` on the same system as shown in the figure above, the OpenMP threads would be assigned the thread contexts as shown in the following figure, which shows the result of specifying `KMP_AFFINITY=granularity=fine,scatter`.



permute and offset combinations

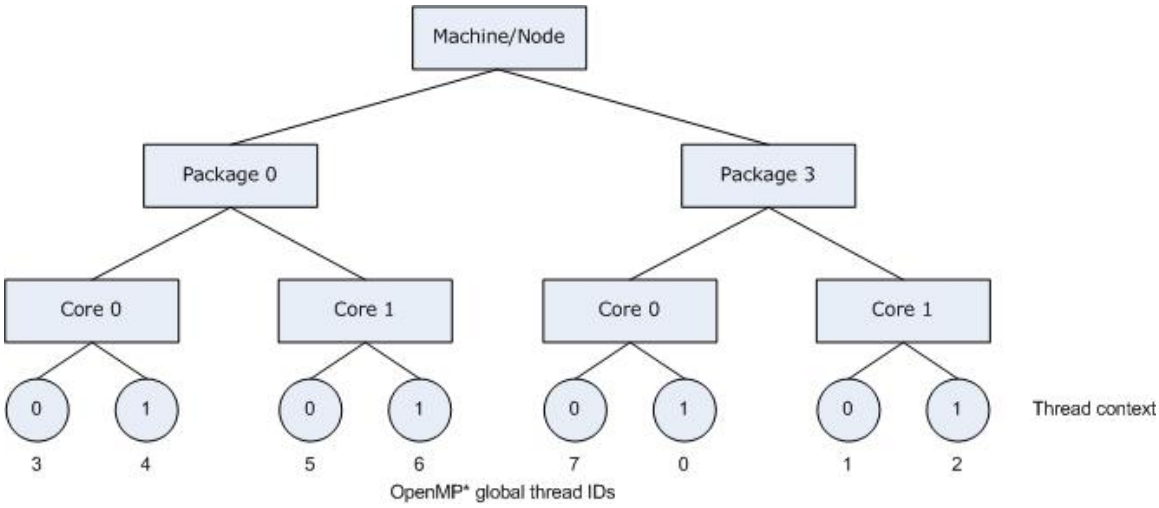
For both `compact` and `scatter`, `permute` and `offset` are allowed; however, if you specify only one integer, the compiler interprets the value as a permute specifier. Both `permute` and `offset` default to 0.

The `permute` specifier controls which levels are most significant when sorting the machine topology map. A value for permute forces the mappings to make the specified

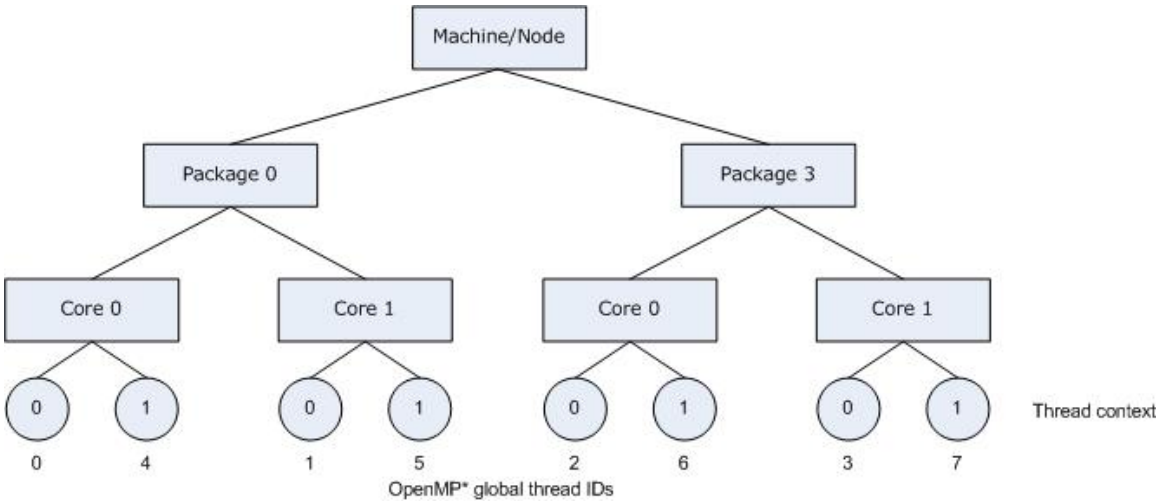
number of most significant levels of the sort the least significant, and it inverts the order of significance. The root node of the tree is not considered a separate level for the sort operations.

The `offset` specifier indicates the starting position for thread assignment.

The following figure illustrates the result of specifying `KMP_AFFINITY=granularity=fine,compact,0,3`.



Consider the hardware configuration from the previous example, running an OpenMP application which exhibits data sharing between consecutive iterations of loops. We would therefore like consecutive threads to be bound close together, as is done with `KMP_AFFINITY=compact`, so that communication overhead, cache line invalidation overhead, and page thrashing are minimized. Now, suppose the application also had a number of parallel regions which did not utilize all of the available OpenMP threads. It is desirable to avoid binding multiple threads to the same core and leaving other cores not utilized, since a thread normally executes faster on a core where it is not competing for resources with another active thread on the same core. Since a thread normally executes faster on a core where it is not competing for resources with another active thread on the same core, you might want to avoid binding multiple threads to the same core while leaving other cores unused. The following figure illustrates this strategy of using `KMP_AFFINITY=granularity=fine,compact,1,0` as a setting.



The OpenMP thread *n*+1 is bound to a thread context as close as possible to OpenMP thread *n*, but on a different core. Once each core has been assigned one OpenMP thread, the subsequent OpenMP threads are assigned to the available cores in the same order, but they are assigned on different thread contexts.

Modifier Values for Affinity Types

Modifiers are optional arguments that precede type. If you do not specify a modifier, the `noverbose`, `respect`, and `granularity=core` modifiers are used automatically.

Modifiers are interpreted in order from left to right, and can negate each other. For example, specifying `KMP_AFFINITY=verbose,noverbose,scatter` is therefore equivalent to setting `KMP_AFFINITY=noverbose,scatter`, or just `KMP_AFFINITY=scatter`.

modifier = noverbose (default)

Does not print verbose messages.

modifier = verbose

Prints messages concerning the supported affinity. The messages include information about the number of packages, number of cores in each package, number of thread contexts for each core, and OpenMP thread bindings to physical thread contexts.

Information about binding OpenMP threads to physical thread contexts is indirectly shown in the form of the mappings between hardware thread contexts and the operating system (OS) processor (proc) IDs. The affinity mask for each OpenMP thread is printed as a set of OS processor IDs.

For example, specifying `KMP_AFFINITY=verbose,scatter` on a dual core system with two processors, with Hyper-Threading Technology (HT Technology) disabled, results in a message listing similar to the following when then program is executed:

Verbose, scatter message

```
...
KMP AFFINITY: Affinity capable, using global cpuid info
KMP AFFINITY: Initial OS proc set respected:
{0,1,2,3}
KMP AFFINITY: 4 available OS procs - Uniform topology of
KMP AFFINITY: 2 packages x 2 cores/pkg x 1 threads/core (4 total cores)
KMP AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP AFFINITY: OS proc 0 maps to package 0 core 0 [thread 0]
KMP AFFINITY: OS proc 2 maps to package 0 core 1 [thread 0]
KMP AFFINITY: OS proc 1 maps to package 3 core 0 [thread 0]
KMP AFFINITY: OS proc 3 maps to package 3 core 1 [thread 0]
KMP AFFINITY: Internal thread 0 bound to OS proc set {0}
KMP AFFINITY: Internal thread 2 bound to OS proc set {2}
KMP AFFINITY: Internal thread 3 bound to OS proc set {3}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {1}
```

The verbose modifier generates several standard, general messages. The following table summarizes how to read the messages.

Message String	Description
"affinity capable"	Indicates that all components (compiler, operating system, and hardware) support affinity, so thread binding is possible.
"using global cpuid info"	Indicates that the machine topology was discovered by binding a thread to each operating system processor and decoding the output of the `cpuid` instruction.
"using local cpuid info"	Indicates that compiler is decoding the output of the `cpuid` instruction, issued by only the initial thread, and is assuming a machine topology using the number of operating system processors.
"using /proc/cpuinfo"	Linux* only. Indicates that `cpuinfo` is being used to determine machine topology.
"using flat"	Operating system processor ID is assumed to be equivalent to physical package ID. This method of determining machine topology is used if none of the other methods will work, but may not accurately detect the actual machine topology.
"uniform topology of"	The machine topology map is a full tree with no missing leaves at any level.

The mapping from the operating system processors to thread context ID is printed next. The binding of OpenMP thread context ID is printed next unless the affinity type is `none`. The thread level is contained in brackets (in the listing shown above). This implies that

there is no representation of the thread context level in the machine topology map. For more information, see Determining Machine Topology.

modifier = granularity

Binding OpenMP threads to particular packages and cores will often result in a performance gain on systems with Intel processors with Hyper-Threading Technology (HT Technology) enabled; however, it is usually not beneficial to bind each OpenMP thread to a particular thread context on a specific core. Granularity describes the lowest levels that OpenMP threads are allowed to float within a topology map.

This modifier supports the following additional specifiers.

Specifier	Description
`core`	Default. Broadest granularity level supported. Allows all the OpenMP threads bound to a core to float between the different thread contexts.
`fine` or `thread`	The finest granularity level. Causes each OpenMP thread to be bound to a single thread context. The two specifiers are functionally equivalent.

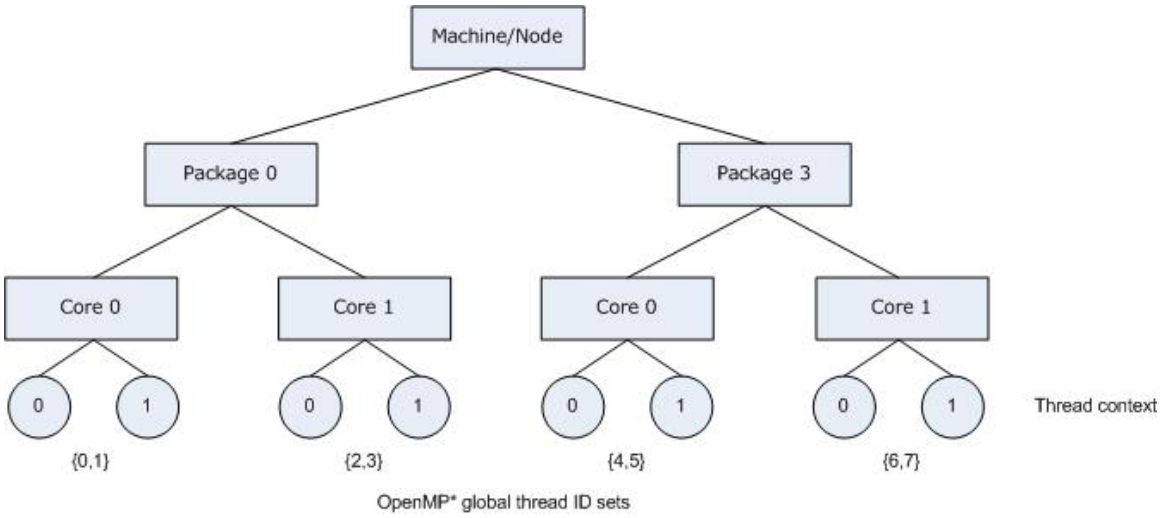
Specifying `KMP_AFFINITY=verbose,granularity=core,compact` on the same dual core system with two processors as in the previous section, but with HT Technology enabled, results in a message listing similar to the following when the program is executed:

Verbose, granularity=core,compact message

```
KMP AFFINITY: Affinity capable, using global cpuid info
KMP AFFINITY: Initial OS proc set respected:
{0,1,2,3,4,5,6,7}
KMP AFFINITY: 8 available OS procs - Uniform topology of
KMP AFFINITY: 2 packages x 2 cores/pkg x 2 threads/core (4 total cores)
KMP AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
KMP AFFINITY: OS proc 4 maps to package 0 core 0 thread 1
KMP AFFINITY: OS proc 2 maps to package 0 core 1 thread 0
KMP AFFINITY: OS proc 6 maps to package 0 core 1 thread 1
KMP AFFINITY: OS proc 1 maps to package 3 core 0 thread 0
KMP AFFINITY: OS proc 5 maps to package 3 core 0 thread 1
KMP AFFINITY: OS proc 3 maps to package 3 core 1 thread 0
KMP AFFINITY: OS proc 7 maps to package 3 core 1 thread 1
KMP AFFINITY: Internal thread 0 bound to OS proc set {0,4}
KMP AFFINITY: Internal thread 1 bound to OS proc set {0,4}
KMP AFFINITY: Internal thread 2 bound to OS proc set {2,6}
KMP AFFINITY: Internal thread 3 bound to OS proc set {2,6}
KMP AFFINITY: Internal thread 4 bound to OS proc set {1,5}
KMP AFFINITY: Internal thread 5 bound to OS proc set {1,5}
KMP AFFINITY: Internal thread 6 bound to OS proc set {3,7}
KMP_AFFINITY: Internal thread 7 bound to OS proc set {3,7}
```

The affinity mask for each OpenMP thread is shown in the listing (above) as the set of operating system processor to which the OpenMP thread is bound.

The following figure illustrates the machine topology map, for the above listing, with OpenMP thread bindings.



In contrast, specifying `KMP_AFFINITY=verbose,granularity=fine,compact` or `KMP_AFFINITY=verbose,granularity=thread,compact` binds each OpenMP thread to a single hardware thread context when the program is executed:

Verbose, granularity=fine,compact message

```
KMP AFFINITY: Affinity capable, using global cpuid info
KMP AFFINITY: Initial OS proc set respected:
{0,1,2,3,4,5,6,7}
KMP AFFINITY: 8 available OS procs - Uniform topology of
KMP AFFINITY: 2 packages x 2 cores/pkg x 2 threads/core (4 total cores)
KMP AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
KMP AFFINITY: OS proc 4 maps to package 0 core 0 thread 1
KMP AFFINITY: OS proc 2 maps to package 0 core 1 thread 0
KMP AFFINITY: OS proc 6 maps to package 0 core 1 thread 1
KMP AFFINITY: OS proc 1 maps to package 3 core 0 thread 0
KMP AFFINITY: OS proc 5 maps to package 3 core 0 thread 1
KMP AFFINITY: OS proc 3 maps to package 3 core 1 thread 0
KMP AFFINITY: OS proc 7 maps to package 3 core 1 thread 1
KMP AFFINITY: Internal thread 0 bound to OS proc set {0}
KMP AFFINITY: Internal thread 1 bound to OS proc set {4}
KMP AFFINITY: Internal thread 2 bound to OS proc set {2}
KMP AFFINITY: Internal thread 3 bound to OS proc set {6}
KMP AFFINITY: Internal thread 4 bound to OS proc set {1}
KMP AFFINITY: Internal thread 5 bound to OS proc set {5}
KMP AFFINITY: Internal thread 6 bound to OS proc set {3}
KMP_AFFINITY: Internal thread 7 bound to OS proc set {7}
```

The OpenMP to hardware context binding for this example was illustrated in the first example.

Specifying `granularity=fine` will always cause each OpenMP thread to be bound to a single OS processor. This is equivalent to `granularity=thread`, currently the finest granularity level.

modifier = respect (default)

Respect the process' original affinity mask, or more specifically, the affinity mask in place for the thread that initializes the OpenMP run-time library. The behavior differs between Linux and Windows OS:

- On Windows: Respect original affinity mask for the process.
- On Linux: Respect the affinity mask for the thread that initializes the OpenMP run-time library.

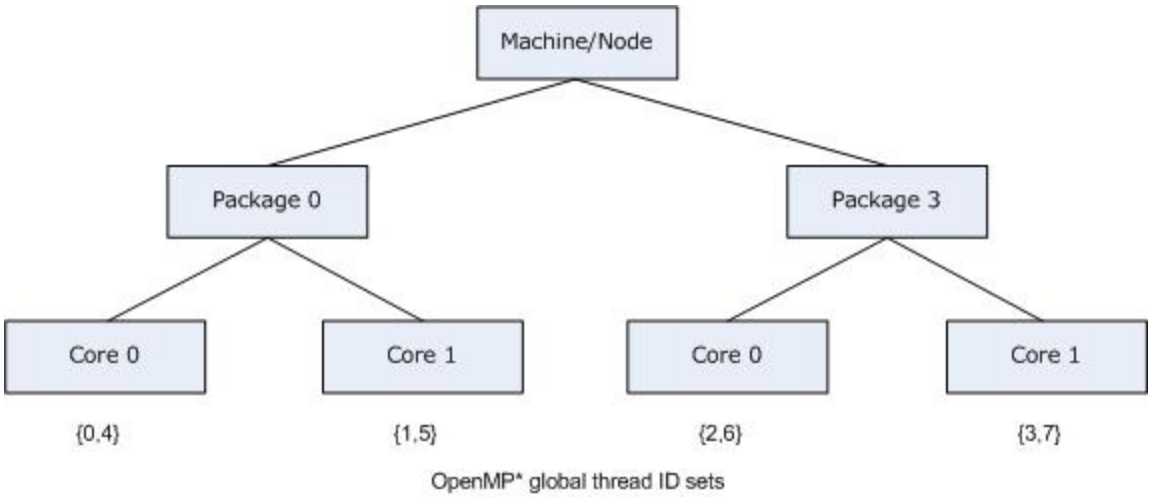
Specifying `KMP_AFFINITY=verbose,compact` for the same system used in the previous example, with HT Technology enabled, and invoking the library with an initial affinity mask of {4,5,6,7} (thread context 1 on every core) causes the compiler to model the machine as a dual core, two-processor system with HT Technology disabled.

Verbose,compact message

```
KMP AFFINITY: Affinity capable, using global cpuid info
KMP AFFINITY: Initial OS proc set respected:
{4,5,6,7}
KMP AFFINITY: 4 available OS procs - Uniform topology of
KMP AFFINITY: 2 packages x 2 cores/pkg x 1 threads/core (4 total cores)
KMP AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP AFFINITY: OS proc 4 maps to package 0 core 0 [thread 1]
KMP AFFINITY: OS proc 6 maps to package 0 core 1 [thread 1]
KMP AFFINITY: OS proc 5 maps to package 3 core 0 [thread 1]
KMP AFFINITY: OS proc 7 maps to package 3 core 1 [thread 1]
KMP AFFINITY: Internal thread 0 bound to OS proc set {4}
KMP AFFINITY: Internal thread 1 bound to OS proc set {6}
KMP AFFINITY: Internal thread 2 bound to OS proc set {5}
KMP AFFINITY: Internal thread 3 bound to OS proc set {7}
KMP AFFINITY: Internal thread 4 bound to OS proc set {4}
KMP AFFINITY: Internal thread 5 bound to OS proc set {6}
KMP AFFINITY: Internal thread 6 bound to OS proc set {5}
KMP_AFFINITY: Internal thread 7 bound to OS proc set {7}
```

Because there are eight thread contexts on the machine, by default the compiler created eight threads for an OpenMP `parallel` construct.

The brackets around thread 1 indicate that the thread context level is ignored, and is not present in the topology map. The following figure illustrates the corresponding machine topology map.



When using the local `cpuid` information to determine the machine topology, it is not always possible to distinguish between a machine that does not support Hyper-Threading Technology (HT Technology) and a machine that supports it, but has it disabled. Therefore, the compiler does not include a level in the map if the elements (nodes) at that level had no siblings, with the exception that the package level is always modeled. As mentioned earlier, the package level will always appear in the topology map, even if there only a single package in the machine.

modifier = norespect

Do not respect original affinity mask for the process. Binds OpenMP threads to all operating system processors.

In early versions of the OpenMP run-time library that supported only the `physical` and `logical` affinity types, `norespect` was the default and was not recognized as a modifier.

The default was changed to `respect` when types `compact` and `scatter` were added; therefore, thread bindings for the `logical` and `physical` affinity types may have changed with the newer compilers in situations where the application specified a partial initial thread affinity mask.

modifier = nowarnings

Do not print warning messages from the affinity interface.

modifier = warnings (default)

Print warning messages from the affinity interface (default).

Determining Machine Topology

On IA-32 and Intel® 64 architecture systems, if the package has an APIC (Advanced Programmable Interrupt Controller), the compiler will use the `cpuid` instruction to obtain

the `package id`, `core id`, and `thread context id`. Under normal conditions, each thread context on the system is assigned a unique APIC ID at boot time. The compiler obtains other pieces of information obtained by using the *cpuid* instruction, which together with the number of OS thread contexts (total number of processing elements on the machine), determine how to break the APIC ID down into the `package ID`, `core ID`, and `thread context ID`.

Normally, all `core ids` on a package and all `thread context ids` on a core are contiguous; however, numbering assignment gaps are common for `package ids`, as shown in the figure above.

On IA-64 architecture systems on Linux* operating systems, the compiler obtains this information from `/proc/cpuinfo`. The `package id`, `core id`, and `thread context id` are obtained from the `physical id`, `core id`, and `thread id` fields from `/proc/cpuinfo`. The `core id` and `thread context id` default to 0, but the `physical id` field must be present in order to determine the machine topology, which is not always the case. If the information contained in `/proc/cpuinfo` is insufficient or erroneous, you may create an alternate specification file and pass it to the OpenMP runtime library by using the KMP_CPUINFO_FILE environment variable, as described in KMP_CPUINFO and /proc/cpuinfo.

If the compiler cannot determine the machine topology using either method, but the operating system supports affinity, a warning message is printed, and the topology is assumed to be `flat`. For example, a flat topology assumes the operating system process *N* maps to package *N*, and there exists only one thread context per core and only one core for each package. (This assumption is always the case for processors based on IA-64 architecture running Windows.)

If the machine topology cannot be accurately determined as described above, the user can manually copy `/proc/cpuinfo` to a temporary file, correct any errors, and specify the machine topology to the OpenMP runtime library via the environment variable `KMP_CPUINFO_FILE=<temp_filename>`, as described in the section KMP_CPUINFO_FILE and /proc/cpuinfo.

Regardless of the method used in determining the machine topology, if there is only one thread context per core for every core on the machine, the thread context level will not appear in the topology map. If there is only one core per package for every package in the machine, the core level will not appear in the machine topology map. The topology map need not be a full tree, because different packages may contain a different number of cores, and different cores may support a different number of thread contexts.

The package level will always appear in the topology map, even if there only a single package in the machine.

KMP_CPUINFO and /proc/cpuinfo

One of the methods the Intel compiler OpenMP runtime library can use to detect the machine topology on Linux* systems is to parse the contents of `/proc/cpuinfo`. If the contents of this file (or a device mapped into the Linux file system) are insufficient or erroneous, you can consider copying its contents to a writable temporary file <*temp_file*>,correct it or extend it with the necessary information, and set KMP_CPUINFO_FILE=<*temp_file*>.

If you do this, the OpenMP runtime library will read the <*temp_file*> location pointed to by KMP_CPUINFO_FILE instead of the information contained in `/proc/cpuinfo` or attempting to detect the machine topology by decoding the APIC IDs. That is, the information contained in the <*temp_file*> overrides these other methods. You can use the KMP_CPUINFO_FILE interface on Windows* systems, where `/proc/cpuinfo` does not exist.

The content of `/proc/cpuinfo` or <*temp_file*> should contain a list of entries for each processing element on the machine. Each processor element contains a list of entries (descriptive name and value on each line). A blank line separates the entries for each processor element. Only the following fields are used to determine the machine topology from each entry, either in <*temp_file*> or `/proc/cpuinfo`:

Field	Description
processor :	Specifies the OS ID for the processing element. The OS ID must be unique. The `processor` and `physical id` fields are the only ones that are required to use the interface.
physical id :	Specifies the package ID, which is a physical chip ID. Each package may contain multiple cores. The package level always exists in the Intel compiler OpenMP run-time library's model of the machine topology.
core id :	Specifies the core ID. If it does not exist, it defaults to 0. If every package on the machine contains only a single core, the core level will not exist in the machine topology map (even if some of the core ID fields are non-zero).
thread id :	Specifies the thread ID. If it does not exist, it defaults to 0. If every core on the machine contains only a single thread, the thread level will not exist in the machine topology map (even if some thread ID fields are non-zero).
node_*n* id :	This is a extension to the normal contents of `/proc/cpuinfo` that can be used to specify the nodes at different levels of the memory interconnect on Non-Uniform Memory Access (NUMA) systems. Arbitrarily many levels *n* are supported. The node_0 level is

closest to the package level; multiple packages comprise a node at level 0. Multiple nodes at level 0 comprise a node at level 1, and so on.

Each entry must be spelled exactly as shown, in lowercase, followed by optional whitespace, a colon (:), more optional whitespace, then the integer ID. Fields other than those listed are simply ignored.

Note

It is common for the `thread id` field to be missing from `/proc/cpuinfo` on many Linux variants, and for a field labeled `siblings` to specify the number of threads per node or number of nodes per package. However, the Intel compiler OpenMP runtime library ignores fields labeled `siblings` so it can distinguish between the `thread id` and `siblings` fields. When this situation arises, the warning message `Physical node/pkg/core/thread ids not unique` appears (unless the `type specified is nowarnings`).

The following is a sample entry for an IA-64 architecture system that has been extended to model the different levels of the memory interconnect:

Sample `/proc/cpuinfo` or <*temp-file*>

```
processor : 23
vendor : GenuineIntel
arch : IA-64
family : 32
model : 0
revision : 7
archrev : 0
features : branchlong, 16-byte atomic ops
cpu number : 0
cpu regs : 4
cpu MHz : 1594.000007
itc MHz : 399.000000
BogoMIPS : 3186.68
siblings : 2
node 3 id : 0
node 2 id : 1
node 1 id : 0
node 0 id : 1
physical id : 2563
core id: 1
thread id: 0
```

This example includes the fields from `/proc/cpuinfo` that affect the functionality of the Intel compiler OpenMP Affinity Interface: `processor`, `physical id`, `core id`, and `thread id`. Other fields (`vendor`, `arch`, ..., `siblings`) from `/proc/cpuinfo` are ignored. The four fields `node_`*n* are extensions.

Explicitly Specifying OS Processor IDs (GOMP_CPU_AFFINITY)

Instead of allowing the library to detect the hardware topology and automatically assign OpenMP threads to processing elements, the user may explicitly specify the assignment

by using a list of operating system (OS) processor (proc) IDs. However, this requires knowledge of which processing elements the OS proc IDs represent.

This list may either be specified by using the proclist modifier along with the explicit affinity type in the KMP_AFFINITY environment variable, or by using the GOMP_CPU_AFFINITY environment variable (for compatibility with `gcc`) when using the Intel OpenMP compatibility libraries.

On Linux systems when using the Intel OpenMP compatibility libraries enabled by the compiler option `-openmp-lib compat`, you can use the GOMP_CPU_AFFINITY environment variable to specify a list of OS processor IDs. Its syntax is identical to that accepted by `libgomp` (assume that <*proc_list*> produces the entire GOMP_CPU_AFFINITY environment string):

<*proc_list*> := <entry> | <*elem*> , <list> | <elem> <whitespace> <list>

<*elem*> := <*proc_spec*> | <*range*>

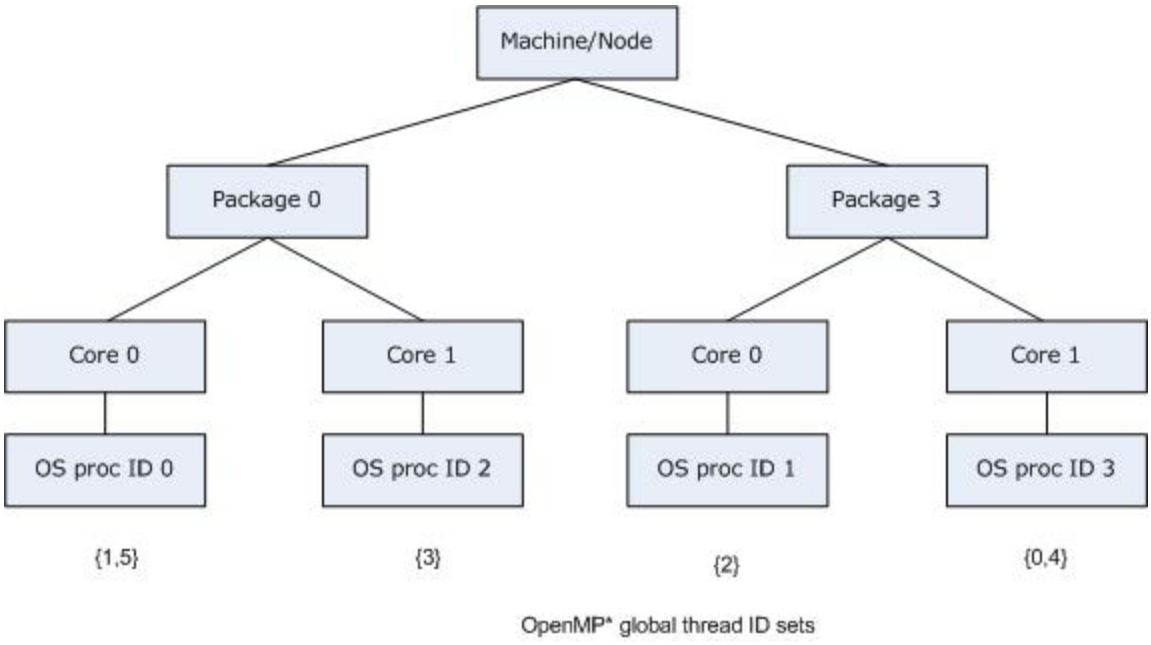
<*proc_spec*> := <*proc_id*>

<*range*> := <*proc_id*> – <*proc_id*> | <*proc_id*> – <*proc_id*> : <int>

<*proc_id*> := <positive_int>

OS processors specified in this list are then assigned to OpenMP threads, in order of OpenMP Global Thread IDs. If more OpenMP threads are created than there are elements in the list, then the assignment occurs modulo the size of the list. That is, OpenMP Global Thread ID *n* is bound to list element *n* mod <*list_size*>.

Consider the machine previously mentioned: a dual core, dual-package machine without Hyper-Threading Technology (HT Technology) enabled, where the OS proc IDs are assigned in the same manner as the example in a previous figure. Suppose that the application creates 6 OpenMP threads instead of 4 (the default), oversubscribing the machine. If `GOMP_CPU_AFFINITY=3,0-2,` then OpenMP threads are bound as shown in the figure below, just as should happen when compiling with `gcc` and linking with `libgomp`:



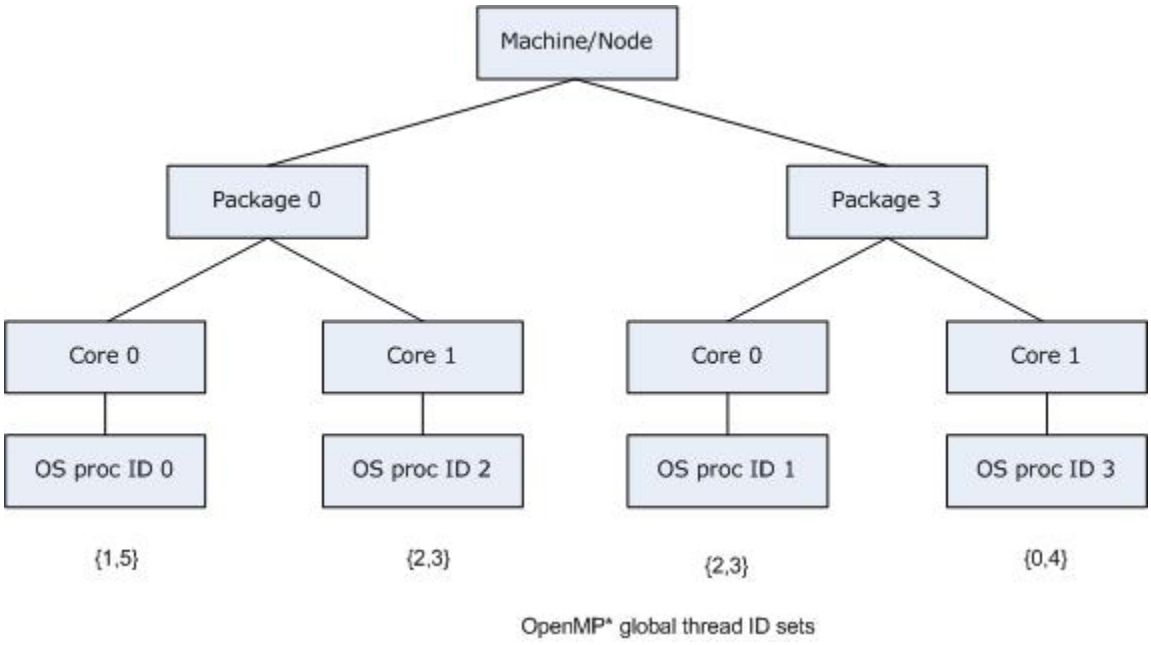
The same syntax can be used to specify the OS proc ID list in the proclist=[<proc_list>] modifier in the KMP_AFFINITY environment variable string. There is a slight difference: in order to have strictly the same semantics as in the `gcc` OpenMP runtime library `libgomp`: the GOMP_CPU_AFFINITY environment variable implies `granularity=fine`. If you specify the OS proc list in the KMP_AFFINITY environment variable without a `granularity=` specifier, then the default `granularity` is not changed. That is, OpenMP threads are allowed to float between the different thread contexts on a single core. Thus GOMP_CPU_AFFINITY=`<proc_list>` is an alias for KMP_AFFINITY=`granularity=fine,proclist=[<proc_list>],explicit`

In the KMP_AFFINITY environment variable string, the syntax is extended to handle operating system processor ID sets. The user may specify a set of operating system processor IDs among which an OpenMP thread may execute (“float”) enclosed in brackets:

<*proc_list*> := <proc_id> | { <float_list> }

<*float_list*> := <proc_id> | <proc_id> , <float_list>

This allows functionality similarity to the `granularity= specifier`, but it is more flexible. The OS processors on which an OpenMP thread executes may exclude other OS processors nearby in the machine topology, but include other distant OS processors. Building upon the previous example, we may allow OpenMP threads 2 and 3 to “float” between OS processor 1 and OS processor 2 by using KMP_AFFINITY=”`granularity=fine,proclist=[3,0,{1,2},{1,2}],explicit`”, as shown in the figure below:



OpenMP* global thread ID sets

If `verbose` were also specified, the output when the application is executed would include:

```
KMP_AFFINITY="granularity=verbose,fine,proclist=[3,0,{1,2},{1,2}],explicit"
```

```
KMP AFFINITY: Affinity capable, using global cpuid info
KMP AFFINITY: Initial OS proc set respected: {0,1,2,3}
KMP AFFINITY: 4 available OS procs - Uniform topology of
KMP AFFINITY: 2 packages x 2 cores/pkg x 1 threads/core (4 total cores)
KMP AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP AFFINITY: OS proc 0 maps to package 0 core 0 [thread 0]
KMP AFFINITY: OS proc 2 maps to package 0 core 1 [thread 0]
KMP AFFINITY: OS proc 1 maps to package 3 core 0 [thread 0]
KMP AFFINITY: OS proc 3 maps to package 3 core 1 [thread 0]
KMP AFFINITY: Internal thread 0 bound to OS proc set {3}
KMP AFFINITY: Internal thread 1 bound to OS proc set {0}
KMP AFFINITY: Internal thread 2 bound to OS proc set {1,2}
KMP AFFINITY: Internal thread 3 bound to OS proc set {1,2}
KMP AFFINITY: Internal thread 4 bound to OS proc set {3}
KMP_AFFINITY: Internal thread 5 bound to OS proc set {0}}
```

Low Level Affinity API

Instead of relying on the user to specify the OpenMP thread to OS proc binding by setting an environment variable before program execution starts (or by using the `kmp_settings` interface before the first parallel region is reached), each OpenMP thread may determine the desired set of OS procs on which it is to execute and bind to them with the `kmp_set_affinity` API call.

The Fortran API interfaces follow, where the type name `kmp_affinity_mask_t` is defined in `omp.h` or `omp.mod`:

Syntax	Description
`integer function kmp_set_affinity(mask) integer (kind=kmp_affinity_mask_kind) mask`	Sets the affinity mask for the current OpenMP thread to `mask`, where `mask` is a set of OS proc IDs that has been created using the API calls listed below, and the thread will only execute on OS procs in the set. Returns either a zero (0) upon success or a nonzero error code.
`integer kmp_get_affinity(mask) integer (kind=kmp_affinity_mask_kind) mask`	Retrieves the affinity mask for the current OpenMP thread, and stores it in `mask`, which must have previously been initialized with a call to `kmp_create_affinity_mask()`. Returns either a zero (0) upon success or a nonzero error code.
`integer function kmp_get_affinity_max_proc()`	Returns the maximum OS proc ID that is on the machine, plus 1. All OS proc IDs are guaranteed to be between 0 (inclusive) and `kmp_get_affinity_max_proc()` (exclusive).
`subroutine kmp_create_affinity_mask(mask) integer (kind=kmp_affinity_mask_kind) mask`	Allocates a new OpenMP thread affinity mask, and initializes `mask` to the empty set of OS procs. The implementation is free to use an object of `kmp_affinity_mask_t` either as the set itself, a pointer to the actual set, or an index into a table describing the set. Do not make any assumption as to what the actual representation is.
`subroutine kmp_destroy_affinity_mask(mask) integer (kind=kmp_affinity_mask_kind) mask`	Deallocates the OpenMP thread affinity mask. For each call to `kmp_create_affinity_mask()`, there should be a corresponding call to `kmp_destroy_affinity_mask()`.

`integer function kmp_set_affinity_mask_proc(proc, mask)`
`integer proc`
`integer (kind=kmp_affinity_mask_kind) mask`
	Adds the OS proc ID proc to the set `mask`, if it is not already. Returns either a zero (0) upon success or a nonzero error code.

`integer function kmp_unset_affinity_mask_proc(proc, mask)`
`integer proc`
`integer (kind=kmp_affinity_mask_kind) mask`
	If the OS proc ID proc is in the set `mask`, it removes it. Returns either a zero (0) upon success or a nonzero error code.

`integer function kmp_get_affinity_mask_proc(proc, mask)`
`integer proc`
`integer (kind=kmp_affinity_mask_kind) mask`
	Returns 1 if the OS proc ID proc is in the set `mask`; if not, it returns 0.

Once an OpenMP thread has set its own affinity mask via a successful call to `kmp_affinity_set_mask()`, then that thread remains bound to the corresponding OS proc set until at least the end of the parallel region, unless reset via a subsequent call to `kmp_affinity_set_mask()`.

Between parallel regions, the affinity mask (and the corresponding OpenMP thread to OS proc bindings) can be considered thread private data objects, and have the same persistence as described in the OpenMP Application Program Interface. For more information, see the OpenMP API specification (http://www.openmp.org), some relevant parts of which are provided below:

> "In order for the affinity mask and thread binding to persist between two consecutive active parallel regions, all three of the following conditions must hold:
>
> - Neither parallel region is nested inside another explicit parallel region.
> - The number of threads used to execute both parallel regions is the same.
> - The value of the dyn-var internal control variable in the enclosing task region is false at entry to both parallel regions."

Therefore, by creating a parallel region at the start of the program whose sole purpose is to set the affinity mask for each thread, the user can mimic the behavior of the KMP_AFFINITY environment variable with low-level affinity API calls, if program execution obeys the three aforementioned rules from the OpenMP specification. Consider again the example presented in the previous figure. To mimic

KMP_AFFINITY=compact, in each OpenMP thread with global thread ID *n*, we need to create an affinity mask containing OS proc IDs *n* modulo *c*, *n* modulo *c* + *c*, and so on, where *c* is the number of cores. This can be accomplished by inserting the following C code fragment into the application that gets executed at program startup time:

Example

```
int main() {
#pragma omp parallel
    {
        int tmax = omp get max threads();
        int tnum = omp get thread num();
        int nproc = omp get num procs();
        int ncores = nproc / 2;
        int i;
        kmp affinity mask t mask;
        kmp create affinity mask(&mask);
        for (i = tnum % ncores; i < tmax; i += ncores) {
            kmp set affinity mask proc(i, &mask);
        }
        if (kmp set affinity(&mask) != 0) <error>;
    }
```

This program fragment was written with knowledge about the mapping of the OS proc IDs to the physical processing elements of the target machine. On another machine, or on the same machine with a different OS installed, the program would still run, but the OpenMP thread to physical processing element bindings could differ.

Using Parallelism: Automatic Parallelization

Auto-parallelization Overview

The auto-parallelization feature of the Intel® compiler automatically translates serial portions of the input program into 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 needed in programming with OpenMP* directives. The OpenMP and auto-parallelization applications provide the performance gains from shared memory on multiprocessor and dual core systems.

The auto-parallelization feature of the Intel® compiler automatically translates serial portions of the input program into equivalent multithreaded code. The auto-parallelizer analyzes the dataflow of the loops in the application source code and generates multithreaded code for those loops which can safely and efficiently be executed in parallel.

This behavior enables the potential exploitation of the parallel architecture found in symmetric multiprocessor (SMP) systems.

Automatic parallelization frees developers from having to:

- find loops that are good worksharing candidates
- perform the dataflow analysis to verify correct parallel execution

- partition 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, a programmer must explicitly identify specific portions of the application code that contain parallelism and add the appropriate compiler directives.

Auto-parallelization, which is triggered by the `-parallel` (Linux* OS and Mac OS* X) or `/Qparallel` (Windows* OS) option, automatically identifies those loop structures that contain parallelism. During compilation, the compiler automatically attempts to deconstruct the code sequences into separate threads for parallel processing. No other effort by the programmer is needed.

Note

IA-64 architecture only: Specifying these options implies `-opt-mem-bandwith1` (Linux) or `/Qopt-mem-bandwidth1` (Windows).

Serial code can be divided so that the code can execute concurrently on multiple threads. For example, consider the following serial code example.

Example 1: Original Serial Code

```
subroutine ser(a, b, c)
  integer, dimension(100) :: a, b, c
  do i=1,100
    a(i) = a(i) + b(i) * c(i)
  enddo
end subroutine ser
```

The following example illustrates one method showing how the loop iteration space, shown in the previous example, might be divided to execute on two threads.

Example 2: Transformed Parallel Code

```
subroutine par(a, b, c)
  integer, dimension(100) :: a, b, c
  ! 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
end subroutine par
```

Auto-Vectorization and Parallelization

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, thread-level parallelism can be exploited in the outermost loop, while instruction-level parallelism can be exploited in the innermost loop.

Example

```
DO I = 1, 100     ! Execute groups of iterations in different hreads (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 systems based on Pentium®, Pentium with MMX™ technology, Pentium II, Pentium III, and Pentium 4 processors.

With the right choice of options, you can:

- Increase the performance of your application with minimum effort
- Use compiler features to develop multithreaded programs faster

Additionally, with the relatively small effort of adding OpenMP directives to existing code you can transform a sequential program into a parallel program. The following example shows OpenMP directives within the code.

Example

```
!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 other effect
ENDDO
```

See examples of the auto-parallelization and auto-vectorization directives in the following topics.

Auto-Parallelization Options Quick Reference

These options are supported on IA-32, Intel® 64, and IA-64 architectures.

Linux* and Mac OS* X	Windows*	Description
`-parallel`	`/Qparallel`	Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel. IA-64 architecture only: • Implies `-opt-mem-bandwith1` (Linux) or `/Qopt-mem-bandwidth1` (Windows). Depending on the program and level of

		parallelization desired, you might need to set the `KMP_STACKSIZE` environment variable to an appropriately large size.
`-par-threshold{n}`	`/Qpar-threshold[:n]`	Sets a threshold for the auto of loops based on the probability of profitable execution of the loop in parallel; valid values of *n* can be 0 to 100.
`-par-schedule-keyword`	`/Qpar-schedule-keyword`	Specifies the scheduling algorithm or a tuning method for loop iterations. It specifies how iterations are to be divided among the threads of the team.
`-par-report`	`/Qpar-report`	Controls the diagnostic levels in the auto-parallelizer optimizer.

Refer to Quick Reference Lists for a complete listing of the quick reference topics.

Auto-parallelization: Enabling, Options, Directives, and Environment Variables

To enable the auto-parallelizer, use the `-parallel` (Linux* and Mac OS* X) or `/Qparallel` (Windows*) option. This option detects parallel loops capable of being executed safely in parallel and automatically generates multi-threaded code for these loops.

Note

You might need to set the `KMP_STACKSIZE` environment variable to an appropriately large size to enable parallelization with this option..

An example of the command using auto-parallelization is as follows:

Operating System	Description
Linux and Mac OS X	`ifort -c -parallel myprog.f`
Windows	`ifort -c /Qparallel myprog.f`

Auto-parallelization uses two specific directives, `!DEC$ PARALLEL` and `!DEC$ NO PARALLEL`.

Auto-parallelization Directives Format and Syntax

The format of an auto-parallelization compiler directive is:

Syntax
`<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:

Syntax

```
!DEC$ PARALLEL
```

Since auto-parallelization directives begin with an exclamation point, the directives take the form of comments if you omit the `-parallel` (Linux) or `/Qparallel` (Windows) option.

The `!DEC$ PARALLEL` directive instructs the compiler to ignore dependencies that 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 following loop:

Example

```
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

Auto-parallelization uses the following OpenMP* environment variables.

- `OMP_NUM_THREADS`
- `OMP_SCHEDULE`
- `KMP_STACKSIZE`

See OpenMP* Environment Variables for more information about the default settings and how to use these variables.

Programming with Auto-parallelization

The auto-parallelization feature implements some concepts of OpenMP*, such as the worksharing construct (with the `PARALLEL DO` directive). See Programming with OpenMP for worksharing construct. This section provides details on auto-parallelization.

Guidelines for Effective Auto-parallelization Usage

A loop can be parallelized if it meets the following criteria:

- 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 dependencies. A loop-carried data dependency 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:

1. Data flow analysis: Computing the flow of data through the program.
2. Loop classification: Determining loop candidates for parallelization based on correctness and efficiency, as shown by threshold analysis.
3. Dependency analysis: Computing the dependency analysis for references in each loop nest.
4. High-level parallelization: Analyzing dependency graph to determine loops which can execute in parallel, and computing run-time dependency.
5. Data partitioning: Examining data reference and partition based on the following types of access: `SHARED`, `PRIVATE`, and `FIRSTPRIVATE`.

6. Multithreaded code generation: Modifying loop parameters, generating entry/exit per threaded task, and generating calls to parallel run-time routines for thread creation and synchronization.

Programming for Multithread Platform Consistency

For applications where most of the computation is carried out in simple loops, Intel compilers may be able to generate a multithreaded version automatically. This information applies to applications built for deployment on symmetric multiprocessors (SMP), systems with Hyper-Threading Technology (HT Technology) enabled, and dual core processor systems.

The compiler can analyze dataflow in loops to determine which loops can be safely and efficiently executed in parallel. Automatic parallelization can sometimes result in shorter execution times. Compiler enabled auto-parallelization can help reduce the time spent performing several common tasks:

- searching for loops that are good candidates for parallel execution
- performing dataflow analysis to verify correct parallel execution
- adding parallel compiler directives manually

Parallelization is subject to certain conditions, which are described in the next section. If `-openmp` and `-parallel` (Linux* and Mac OS* X) or `/Qopenmp` and `/Qparallel` (Windows*) are both specified on the same command line, the compiler will only attempt to parallelize those functions that do not contain OpenMP* directives.

The following program contains a loop with a high iteration count:

Example

```
subroutine no dep
  parameter (n=100000000)
  real a, c(n)
  do i = 1, n
    a = 2 * i - 1
    c(i) = sqrt(a)
  enddo
  print*, n, c(1), c(n)
end subroutine no_dep
```

Dataflow analysis confirms that the loop does not contain data dependencies. The compiler will generate code that divides the iterations as evenly as possible among the threads at runtime. The number of threads defaults to the number of processors but can be set independently using the OMP_NUM_THREADS environment variable. The increase in parallel speed for a given loop depends on the amount of work, the load balance among threads, the overhead of thread creation and synchronization, etc., but generally will be less than the number of threads. For a whole program, speed increases depend on the ratio of parallel to serial computation.

For builds with separate compiling and linking steps, be sure to link the OpenMP* runtime library when using automatic parallelization. The easiest way to do this is to use the Intel® compiler driver for linking.

Parallelizing Loops

Three requirements must be met for the compiler to parallelize a loop.

1. The number of iterations must be known before entry into a loop so that the work can be divided in advance. A while-loop, for example, usually cannot be made parallel.
2. There can be no jumps into or out of the loop.
3. The loop iterations must be independent.

In other words, correct results must not logically depend on the order in which the iterations are executed. There may, however, be slight variations in the accumulated rounding error, as, for example, when the same quantities are added in a different order. In some cases, such as summing an array or other uses of temporary scalars, the compiler may be able to remove an apparent dependency by a simple transformation.

Potential aliasing of pointers or array references is another common impediment to safe parallelization. Two pointers are aliased if both point to the same memory location. The compiler may not be able to determine whether two pointers or array references point to the same memory location. For example, if they depend on function arguments, run-time data, or the results of complex calculations. If the compiler cannot prove that pointers or array references are safe and that iterations are independent, the compiler will not parallelize the loop, except in limited cases when it is deemed worthwhile to generate alternative code paths to test explicitly for aliasing at run-time. If you know parallelizing a particular loop is safe and that potential aliases can be ignored, you can instruct the compiler to parallelize the loop using the `!DIR$ PARALLEL` directive.

The compiler can only effectively analyze loops with a relatively simple structure. For example, the compiler cannot determine the thread safety of a loop containing external function calls because it does not know whether the function call might have side effects that introduce dependences. Fortran 90 programmers can use the `PURE` attribute to assert that subroutines and functions contain no side effects. You can invoke interprocedural optimization with the `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows) compiler option. Using this option gives the compiler the opportunity to analyze the called function for side effects.

When the compiler is unable to parallelize automatically loops you know to be parallel use OpenMP*. OpenMP* is the preferred solution because you, as the developer, understand the code better than the compiler and can express parallelism at a coarser granularity. On the other hand, automatic parallelization can be effective for nested loops, such as those in a matrix multiply. Moderately coarse-grained parallelism results from threading of the outer loop, allowing the inner loops to be optimized for fine-grained parallelism using vectorization or software pipelining.

If a loop can be parallelized, it's not always the case that it should be parallelized. The compiler uses a threshold parameter to decide whether to parallelize a loop. The `-par-`

`threshold` (Linux and Mac OS X) or `/Qpar-threshold` (Windows) compiler option adjusts this behavior. The threshold ranges from 0 to 100, where 0 instructs the compiler to always parallelize a safe loop and 100 instructs the compiler to only parallelize those loops for which a performance gain is highly probable. Use the `-par-report` (Linux and Mac OS X) or `/Qpar-report` (Windows) compiler option to determine which loops were parallelized. The compiler will also report which loops could not be parallelized indicate a probably reason why it could not be parallelized. See Auto-parallelization: Threshold Control and Diagnostics for more information on the using these compiler options.

Because the compiler does not know the value of `k`, the compiler assumes the iterations depend on each other, for example if k equals -1, even if the actual case is otherwise. You can override the compiler inserting `!DEC$ parallel`:

Example

```
subroutine add(k, a, b)
  integer :: k
  real :: a(10000), b(10000)
  !$DEC parallel
  do i = 1, 10000
    a(i) = a(i+k) + b(i)
  end do
end subroutine add
```

As the developer, it's your responsibility to not call this function with a value of `k` that is less than 10000; passing a value less than 10000 could to incorrect results.

Thread Pooling

Thread pools offer an effective approach to managing threads. A thread pool is a group of threads waiting for work assignments. In this approach, threads are created once during an initialization step and terminated during a finalization step. This simplifies the control logic for checking for failures in thread creation midway through the application and amortizes the cost of thread creation over the entire application. Once created, the threads in the thread pool wait for work to become available. Other threads in the application assign tasks to the thread pool. Typically, this is a single thread called the thread manager or dispatcher. After completing the task, each thread returns to the thread pool to await further work. Depending upon the work assignment and thread pooling policies employed, it is possible to add new threads to the thread pool if the amount of work grows. This approach has the following benefits:

- Possible runtime failures midway through application execution due to inability to create threads can be avoided with simple control logic.
- Thread management costs from thread creation are minimized. This in turn leads to better response times for processing workloads and allows for multithreading of finer-grained workloads.

A typical usage scenario for thread pools is in server applications, which often launch a thread for every new request. A better strategy is to queue service requests for

processing by an existing thread pool. A thread from the pool grabs a service request from the queue, processes it, and returns to the queue to get more work.

Thread pools can also be used to perform overlapping asynchronous I/O. The I/O completion ports provided with the Win32* API allow a pool of threads to wait on an I/O completion port and process packets from overlapped I/O operations.

OpenMP* is strictly a fork/join threading model. In some OpenMP implementations, threads are created at the start of a parallel region and destroyed at the end of the parallel region. OpenMP applications typically have several parallel regions with intervening serial regions. Creating and destroying threads for each parallel region can result in significant system overhead, especially if a parallel region is inside a loop; therefore, the Intel OpenMP implementation uses thread pools. A pool of worker threads is created at the first parallel region. These threads exist for the duration of program execution. More threads may be added automatically if requested by the program. The threads are not destroyed until the last parallel region is executed.

Thread pools can be created on Windows and Linux using the thread creation API.

The function `CheckPoolQueue` executed by each thread in the pool is designed to enter a wait state until work is available on the queue. The thread manager can keep track of pending jobs in the queue and dynamically increase the number of threads in the pool based on the demand.

Using Parallelism: Automatic Vectorization

Automatic Vectorization Overview

The automatic vectorizer (also called the auto-vectorizer) is a component of the Intel® compiler that automatically uses SIMD instructions in the MMX™, Intel® Streaming SIMD Extensions (Intel® SSE, SSE2, SSE3 and SSE4 Vectorizing Compiler and Media Accelerators) and Supplemental Streaming SIMD Extensions (SSSE3) 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.

Automatic vectorization is supported on IA-32 and Intel® 64 architectures.

The section discusses the following topics, among others:

- High-level discussion of compiler options used to control or influence vectorization
- Vectorization Key Programming Guidelines
- Loop parallelization and vectorization
- Discussion and general guidelines on vectorization levels:
    - automatic vectorization
    - vectorization with user intervention
- Examples demonstrating typical vectorization issues and resolutions

The compiler supports a variety of directives that can help the compiler to generate effective vector instructions.

See Vectorization Support.

See The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance, A.J.C. Bik. Intel Press, June, 2004, for a detailed discussion of how to vectorize code using the Intel® compiler. Additionally, see the Related Publications topic in this document for other resources.

Automatic Vectorization Options Quick Reference
These options are supported on IA-32 and Intel® 64 architectures.

Linux* OS and Mac OS* X	Windows* OS	Description
`-x`	`/Qx`	Generates specialized code to run exclusively on processors with the extensions specified as the *`processor`* value. See Targeting IA-32 and Intel® 64 Architecture Processors Automatically for more information about using the option.
`-ax`	`/Qax`	Generates, in a single binary, code specialized to the extensions specified as the *`processor`* value and also generic IA-32 architecture code. The generic code is usually slower. See Targeting Multiple IA-32 and Intel® 64 Architecture Processors for Run-time Performance for more information about using the option.
`-vec`	`/Qvec`	Enables or disables vectorization and transformations enabled for vectorization. The default is that vectorization is enabled. Supported for IA-32 and Intel® 64 architectures only.
`-vec-report`	`/Qvec-report`	Controls the diagnostic messages from the vectorizer. See Vectorization Report.

Vectorization within the Intel® compiler depends upon ability of the compiler to disambiguate memory references. Certain options may enable the compiler to do better vectorization.

Refer to Quick Reference Lists for a complete listing of the quick reference topics.

Programming Guidelines for Vectorization
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, by using directives.

Guidelines

You will often need to make some changes to your loops. Follow these guidelines for loop bodies.

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

To make your code vectorizable, you will often need to make some changes to your loops. However, you should make only the changes needed to enable vectorization and no others. In particular, you should avoid these common changes:

- loop unrolling; the compiler does it automatically.
- decomposing one loop with several statements in the body into several single-statement loops.

Restrictions

There are a number of restrictions that you should be consider. Vectorization depends on two major factors: hardware and style of source code.

Factor	Description
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 of source code	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.

Vectorization and Loops
Combine the `-parallel` (Linux* and Mac OS* X) or `/Qparallel` (Windows*) and `-x` (Linux) or `/Qx` (Windows) options to 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 Programming Guidelines for Vectorization.

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; for example, in the cases where `-vec-report2` (Linux and Mac OS X) or `/Qvec-report2` (Windows) option indicating loops were not successfully vectorized. (See Vectorization Report.)

Types of Vectorized Loops

For integer loops, the 64-bit MMX™ technology and 128-bit Intel® Streaming SIMD Extensions (Intel® SSE) 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. Also, note that because the MMX™ and Intel® SSE 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, Intel® SSE provides SIMD instructions for the following arithmetic operators: addition (+), subtraction (-), multiplication (*), and division (/).

Additionally, the Streaming SIMD Extensions provide 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`, and `TAN`) are supported in software

in a vector mathematical run-time library that is provided with the Intel® compiler of which the compiler takes advantage.

Statements in the Loop Body

The vectorizable operations are different for floating-point and integer data.

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 valid. The loop body cannot contain any function calls other than the ones described above.

Data Dependency

Data dependency 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 dependency 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 1: Data-dependent Loop

```
subroutine dep(data, n)
  real :: data(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
end subroutine dep
```

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 2: Data-dependency 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 dependency Analysis

Data dependency 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
- for array references, the relationship between the subscripts

For IA-32 architecture, data dependency 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 independency in any dimension will exclude any dependency 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 independency 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 independency, the compiler will eventually resort to a powerful hierarchical dependency solver that uses Fourier-Motzkin elimination to solve the data dependency problem in all dimensions.

Loop Constructs

Loops can be formed with the usual `DO-END DO` and `DO WHILE`, or by using an `IF`/`GOTO` and a label. The loops must have a single entry and a single exit to be vectorized. The following examples illustrate loop constructs that can and cannot be vectorized.

Example: Vectorizable structure

```
subroutine vec(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
end subroutine vec
```

The following example shows a loop that cannot be vectorized because of the inherent potential for an early exit from the loop.

Example: Non-vectorizable structure

```
subroutine no vec(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)
! The next statement allows early
! exit from the loop and prevents
! vectorization of the loop.
    if (a(i) .lt. 0.0) go to 10
    i = i + 1
  enddo
  10 continue
end subroutine no vecN
END
```

Loop Exit Conditions

Loop exit conditions determine the number of iterations a loop executes. For example, fixed indexes for loops determine the iterations. The loop iterations must be countable; in other words, 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

In the case where a loops exit depends on computation, the loops are not countable. The examples below show loop constructs that are countable and non-countable.

Example: Countable Loop

```
subroutine cnt1 (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
end
```

The following example demonstrates a different countable loop construct.

Example: Countable Loop

```
! Number of iterations is (n-m+2)/2
subroutine cnt2 (a, b, c, m, n)
  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
end
```

The following examples demonstrates a loop construct that is non-countable due to dependency loop variant count value.

Example: 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
end
```

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 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. For example, assume the compiler attempts to strip-mine the following loop:

Example1: Before Vectorization

```
i = 1
do while (i<=n)
  a(i) = b(i) + c(i) ! Original loop code
  i = i + 1
end do
```

The compiler might handle the strip mining and loop cleaning by restructuring the loop in the following manner:

Example 2: 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 example 2: `BS` = block_size; MAX must be evenly divisible by `BS`.

Consider the following loop example code:

Example: 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
```

The arrays could be blocked into smaller chunks so that the total combined size of the two blocked chunks is smaller than the cache size, which can improve data reuse. One possible way of doing this is demonstrated below:

Example: 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
```

Loop Interchange and Subscripts: Matrix Multiply

Loop interchange need unit-stride constructs to be vectorized. Matrix multiplication is commonly written as shown in the following example:

Example: Typical Matrix Multiplication

```
subroutine matmul_slow(a, b, c)
  integer :: i, j, k
  real :: a(100,100), b(100,100), c(100,100)
  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
end subroutine matmul_slow
```

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 shown in the following example.

Example: Matrix Multiplication with Stride-1

```
subroutine matmul_fast(a, b, c)
  integer :: i, j, k
  real :: a(100,100), b(100,100), c(100,100)
  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
end subroutine matmul_fast
```

Interchanging is not always possible because of dependencies, which can lead to different results.

Using Interprocedural Optimization (IPO)

Interprocedural Optimization (IPO) Overview

Interprocedural Optimization (IPO) allows the compiler to analyze your code to determine where you can benefit from specific optimizations. In many cases, the optimizations that can be applied are related to the specific architectures.

The compiler might apply the following optimizations for the listed architectures:

Architecture	Optimization
IA-32, Intel® 64, and IA-64 architectures	• inlining • constant propagation • mod/ref analysis • alias analysis • forward substitution • routine key-attribute propagation

	• address-taken analysis • partial dead call elimination • symbol table data promotion • common block variable coalescing • dead function elimination • unreferenced variable removal • whole program analysis • array dimension padding • common block splitting • stack frame alignment • structure splitting and field reordering • formal parameter alignment analysis • indirect call conversion • specialization
IA-32 and Intel® 64 architectures	• Passing arguments in registers to optimize calls and register usage
IA-64 architecture only	• removing redundant EXTEND instructions • short data section allocation • prefetch analysis

IPO is an automatic, multi-step process: compilation and linking; however, IPO supports two compilation models: single-file compilation and multi-file compilation.

Single-file compilation, which uses the `-ip` (Linux* OS and Mac OS* X) or `/Qip` (Windows* OS) option, results in one, real object file for each source file being compiled. During single-file compilation the compiler performs inline function expansion for calls to procedures defined within the current source file.

The compiler performs some single-file interprocedural optimization at the default optimization level: `-O2` (Linux* and Mac OS* X) or `/O2` (Windows*); additionally some the compiler performs some inlining for the `-O1` (Linux* and Mac OS* X) or `/O1` (Windows*) optimization level, like inlining functions marked with inlining directives.

Multi-file compilation, which uses the `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows) option, results in one or more mock object files rather than normal object files. (See the *Compilation* section below for information about mock object files.) Additionally, the compiler collects information from the individual source files that make up the program.

Using this information, the compiler performs optimizations across functions and procedures in different source files. Inlining is the most powerful optimization supported by IPO. See Inline Function Expansion.

Note

Inlining and other optimizations are improved by profile information. For a description of how to use IPO with profile information for further optimization, see Profile an Application.

Mac OS* X: Intel®-based systems running Mac OS X do not support a multiple object compilation model.

Compilation

As each source file is compiled with IPO, the compiler stores an intermediate representation (IR) of the source code in a mock object file, which includes summary information used for optimization. The mock object files contain the IR, instead of the normal object code. Mock object files can be ten times larger, and in some cases more, than the size of normal object files.

During the IPO compilation phase only the mock object files are visible. The Intel compiler does not expose the real object files during IPO unless you also specify the `-ipo-c` (Linux and Mac OS X) or `/Qipo-c` (Windows) option.

Linkage

When you link with the `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows) option the compiler is invoked a final time. The compiler performs IPO across all object files that have an IR equivalent. The mock objects must be linked with the Intel compiler or by using the Intel linking tools. The compiler calls the linkers indirectly by using aliases (or wrappers) for the native linkers, so you must modify make files to account for the different linking tool names. For information on using the linking tools, see Using IPO; see the Linking Tools and Options topic for detailed information.

Caution

Linking the mock object files with ld (Linux and Mac OS X) or link.exe (Windows) will cause linkage errors. You must use the Intel linking tools to link mock object files.

During the compilation process, the compiler first analyzes the summary information and then produces mock object files for source files of the application.

Whole program analysis

The compiler supports a large number of IPO optimizations that either can be applied or have the effectiveness greatly increased when the whole program condition is satisfied.

Whole program analysis, when it can be done, enables many interprocedural optimizations. During the analysis process, the compiler reads all Intermediate Representation (IR) in the mock file, object files, and library files to determine if all references are resolved and whether or not a given symbol is defined in a mock object file. Symbols that are included in the IR in a mock object file for both data and functions are candidates for manipulation based on the results of whole program analysis.

There are two types of whole program analysis: object reader method and table method. Most optimizations can be applied if either type of whole program analysis determine that the whole program conditions exists; however, some optimizations require the results of the object reader method, and some optimizations require the results of table method.

Note

The IPO report provides details about whether whole program analysis was satisfied and indicate the method used during IPO compilation.

In the first type of whole program analysis, the object reader method, the object reader emulates the behavior of the native linker and attempts to resolve the symbols in the application. If all symbols are resolved correctly, the whole program condition is satisfied. This type of whole program analysis is more likely to detect the whole program condition.

Often the object files and libraries accessed by the compiler do not represent the whole program; there are many dependencies to well-known libraries. IPO linking, whole program analysis, determines whether or not the whole program can be detected using the available compiler resources.

The second type of whole program analysis, the table method, is where the compiler analyzes the mock object files and generates a call-graph.

The compiler contains detailed tables about all of the functions for all important language-specific libraries, like the Fortran runtime libraries. In this second method, the compiler constructs a call-graph for the application. The compiler then compares the function table and application call-graph. For each unresolved function in the call-graph, the compiler attempts to resolve the calls. If the compiler can resolve the functions call, the whole program condition exists.

Interprocedural Optimization (IPO) Quick Reference
IPO is a two step process: compile and link. See Using IPO.

Linux* and Mac OS* X	Windows*	Description

`-ipo` or `-ipoN`	`/Qipo` or `/QipoN`	Enables interprocedural optimization for multi-file compilations. Normally, multi-file compilations result in a single object file only. Passing an integer value for *N* allows you to specify number of true object files to generate; the default value is 0, which means the compiler determines the appropriate number of object files to generate. (See IPO for Large Programs.)
`-ipo-separate`	`/Qipo-separate`	Instructs the compiler to generate a separate, real object file for each mock object file. Using this option overrides any integer value passed for `ipoN`. (See IPO for Large Programs for specifics.)
`-ip`	`/Qip`	Enables interprocedural optimizations for single file compilations. Instructs the compiler to generate a separate, real object file for each source file.

Additionally, the compiler supports options that provide support for compiler-directed or developer-directed inline function expansion.

Refer to Quick Reference Lists for a complete listing of the quick reference topics.

Using IPO
This topic discusses how to use IPO from a command line. For specific information on using IPO from within an Integrated Development Environment (IDE), refer to the appropriate section in Building Applications.

Compiling and Linking Using IPO

The steps to enable IPO for compilations targeted for IA-32, Intel® 64, and IA-64 architectures are the same: compile and link.

First, compile your source files with `-ipo` (Linux* and Mac OS* X) or `/Qipo` (Windows*) as demonstrated below:

Operating System	Example Command
Linux and Mac OS X	`ifort -ipo -c a.f90 b.f90 c.f90`
Windows*	`ifort /Qipo /c a.f90 b.f90 c.f90`

The output of the above example command differs according to operating system:

- Linux and Mac OS X: The commands produce `a.o`, `b.o`, and `c.o` object files.
- Windows: The commands produce `a.obj`, `b.obj`, and `c.obj` object files.

Use `-c` (Linux and Mac OS X) or `/c` (Windows) to stop compilation after generating `.o` or `.obj` files. The output files contain compiler intermediate representation (IR) corresponding to the compiled source files. (See the section below on capturing the intermediate IPO output.)

Second, link the resulting files. The following example command will produce an executable named `app`:

Operating System	Example Command
Linux and Mac OS X	`ifort -o app a.o b.o c.o`
Windows	`ifort /exe:app a.obj b.obj c.obj`

The command invoke the compiler on the objects containing IR and creates a new list of objects to be linked. Alternately, you can use the `xild` (Linux and Mac OS X) or `xilink` (Windows) tool, with the appropriate linking options.

Combining the Steps

The separate compile and link commands demonstrated above can be combined into a single command, as shown in the following examples:

Operating System	Example Command
Linux and Mac OS X	`ifort -ipo -o app a.f90 b.f90 c.f90`
Windows	`ifort /Qipo /exe:app a.f90 b.f90 c.f90`

The `ifort` command, shown in the examples above, calls `GCC ld` (Linux and Mac OS X) or Microsoft* `link.exe` (Windows only) to link the specified object files and produce the executable application, which is specified by the `-o` (Linux and Mac OS X) or `/exe` (Windows) option.
The Intel linking tools emulate the behavior of compiling at `-O0` (Linux and Mac OS X) and `/Od` (Windows).

Multi-file IPO is applied only to the source files that have intermediate representation (IR); otherwise, the object file passes to link stage.

Capturing Intermediate IPO Output

The `-ipo-c` (Linux and Mac OS X) or `/Qipo-c` (Windows*) and `-ipo-S` (Linux and Mac OS X) or `/Qipo-S` (Windows) options are useful for analyzing the effects of multi-file IPO, or when experimenting with multi-file IPO between modules that do not make up a complete program.

- Use the `ipo-c` option to optimize across files and produce an object file. The option performs optimizations as described for the `ipo` option but stops prior to the final link stage, leaving an optimized object file. The default name for this file is `ipo_out.s` (Linux and Mac OS X) or `ipo_out.obj` (Windows).
- Use the `ipo-S` option to optimize across files and produce an assembly file. The 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` (Linux) or `ipo_out.asm` (Windows).

For both options, you can use the `-o` (Linux and Mac OS X) or `/exe` (Windows) option to specify a different name.

These 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` (Linux and Mac OS X) or `/exe` (Windows) option.

The names of subsequent files are derived from the first file with an appended numeric value to the file name. For example, if the first object file is named `foo.o` (Linux and Mac OS X) or `foo.obj` (Windows), the second object file will be named `foo1.o` or `foo1.obj`.

You can use the object file generated with the `-ipo-c` (Linux and Mac OS X) or `/Qipo-c` (Windows) option, but you will not get the same benefits from the optimizations applied that would otherwise if the whole program analysis condition had been satisfied.

The object file created using the `ipo-c` option is a real object file, in contrast to the mock file normally generated using IPO; however, the generated object file is significantly different than the mock object file. It does not contain the IR information needed to fully optimize the application using IPO.

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

IPO-Related Performance Issues

There are some general optimization guidelines for IPO that you should keep in mind:

- Large IPO compilations might trigger internal limits of other compiler optimization phases.
- Applications where the compiler does not have sufficient intermediate representation (IR) coverage to do whole program analysis might not perform as well as those where IR information is complete.

In addition to the general guidelines, there are some practices to avoid while using IPO. The following list summarizes the activities to avoid:

- Do not use the link phase of an IPO compilation using mock object files produced for your application by a different compiler. The Intel® Compiler cannot inspect mock object files generated by other compilers for optimization opportunities.

- Do not link mock files with the `-prof-use` (Linux* and Mac OS* X) or `/Qprof-use` (Windows*) option unless the mock files were also compiled with the `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows) option.
- Update make files to call the appropriate Intel linkers when using IPO from scripts. For Linux and Mac OS X, replaces all instances of `ld` with `xlid`; for Windows, replace all instances of `link` with `xilink`.
- Update make file to call the appropriate Intel archiver. Replace all instances of `ar` with `xiar`.

IPO for Large Programs

In most cases, IPO generates a single object file for the link-time compilation. This behavior is not optimal for very large programs, perhaps even making it impossible to use `-ipo` (Linux* and Mac OS* X) or `/Qipo` (Windows*) on the application.

The compiler provides two methods to avoid this problem. The first method is an automatic size-based heuristic, which causes the compiler to generate multiple object files for large link-time compilations. The second method is to manually instruct the compiler to perform multi-object IPO.

- Use the `-ipo`*`N`* (Linux and Mac OS X) or `/Qipo`*`N`* (Windows) option and pass an integer value in the place of *`N`*.
- Use the `-ipo-separate` (Linux and Mac OS X) or `/Qipo-separate` (Windows) option.
  The number of true object files generated by the link-time compilation is invisible to the user unless either the `-ipo-c` or `-ipo-S` (Linux and Mac OS X) or `/Qipo-c` or `/Qipo-S` (Windows) option is used.

Regardless of the method used, it is best to use the compiler defaults first and examine the results. If the defaults do not provide the expected results then experiment with generating more object files.

You can use the `-ipo-jobs` (Linux and Mac OS X) or `/Qipo-jobs` (Windows) option to control the number of commands, or jobs, executed during parallel builds.

Using -ipo*N* or /Qipo*N* to Create Multiple Object Files

If you specify `-ipo0` (Linux and Mac OS X) or `/Qipo0` (Windows), which is the same as not specifying a value, the compiler uses heuristics to determine whether to create one or more object files based on the expected size of the application. The compiler generates one object file for small applications, and two or more object files for large applications. If you specify any value greater than `0`, the compiler generates that number of object files, unless the value you pass a value that exceeds the number of source files. In that case, the compiler creates one object file for each source file then stops generating object files.

The following example commands demonstrate how to use `-ipo2` (Linux and Mac OS X) or `/Qipo2` (Windows) to compile large programs.

Operating System	Example Command
Linux and Mac OS X	`ifort -ipo2 -c a.f90 b.f90`
Windows	`ifort /Qipo2 /c a.f90 b.f90`

Because the example command shown above, the compiler generates object files using an OS-dependent naming convention. On Linux and Mac OS X, the example command results in object files named `ipo_out.o`, `ipo_out1.o`, `ipo_out2.o`, and `ipo_out3.o`. On Windows, the file names follow the same convention; however, the file extensions will be `.obj`.

Link the resulting object files as shown in Using IPO or Linking Tools and Options.

Creating the Maximum Number of Object Files

Using `-ipo-separate` (Linux and Mac OS X) or `/Qipo-separate` (Windows) allows you to force the compiler to generate the maximum number of true object files that the compiler will support during multiple object compilation.

For example, if you passed example commands similar to the following:

Operating System	Example Command
Linux and Mac OS X	`ifort a.o b.o c.o -ipo-separate -ipo-c`
Windows	`ifort a.obj b.obj c.obj /Qipo-separate /Qipo-c`

The compiler will generate multiple object file, which use the same naming convention discussed above. The first object file contains global variables. The other object files contain code for the functions or routines used the source files.

Link the resulting object files as shown in Using IPO or Linking Tools and Options.

Considerations for Large Program Compilation

For many large programs, compiling with IPO can result in a single, large object file. Compiling to produce large objects can create problems for efficient compilation. During compilation, the compiler attempts to swap the memory usage during compiles; a large object file might result in poor swap usage, which could result in out-of-memory message or long compilation times. Using multiple, relatively small object files during compilation causes the system to use resources more efficiently.

Understanding Code Layout and Multi-Object IPO

One of the optimizations performed during an IPO compilation is code layout. The analysis performed by the compiler during multi-file IPO determines a layout order for all

of the routines for which it has intermediate representation (IR) information. For a multi-object IPO compilation, the compiler must tell the linker about the desired order.

If you are generating an executable in the link step, the compiler does all of this automatically. However, if you are generating object files instead of an executable, the compiler generates a layout script, which contains the correct information needed to optimally link the executable when you are ready to create it.

This linking tool script must be taken into account if you use either `-ipo-c` or `-ipo-S` (Linux*) or `/Qipo-c` or `/Qipo-S` (Windows*). With these options, the IPO compilation and actual linking are done by different invocations of the compiler. When this occurs, the compiler issues a message indicating that it is generating an explicit linker script, `ipo_layout.script`.

The Windows linker (link.exe) automatically collates these sections lexigraphically in the desired order.

The compiler first puts each routine in a named text section that varies depending on the operating system:

Windows:

- The first routine is placed in `.text$00001`, the second is placed in `.text$00002`, and so on.

Linux:

- The first routine is placed in `.text00001`, the second is placed in `.text00002`, and so on. It then generates a linker script that tells the linker to first link contributions from `.text00001`, then `.text00002`.

When `ipo_layout.script` is generated, you should modify your link command if you want to use the script to optimize code layout:

Example

```
--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:

Example output

```
.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. The addition is harmless, since the ”r;*(…)” 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.

Creating a Library from IPO Objects

Linux* and Mac OS* X

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

Example

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

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

If the objects have been created using `-ipo -c` then the archive will not only contain a valid object, but the archive will also contain intermediate representation (IR) for that object file. For example, the following example will produce `a.o` and `b.o` that may be archived to produce a library containing both object code and IR for each source file.

Example

```
ifort -ipo -c a.f90 b.f90
```

This program will invoke the compiler on the IR saved in the object file and generate a valid object that can be inserted into a library.

Using `xiar` is the same as specifying `xild -lib`.

Mac OS X Only

When using `xilibtool`, specify `-static` to generate static libraries, or specify `-dynamic` to create dynamic libraries. For example, the following example command will create a static library named `mylib.a` that includes the `a.o`, `b.o`, and `c.o` objects.

Example

```
xilibtool -static -o mylib.a a.o b.o c.o
```

Alternately, the following example command will create a dynamic library named `mylib.dylib` that includes the `a.o`, `b.o`, and `c.o` objects.

Example

```
xilibtool -dynamic -o mylib.dylib a.o b.o c.o
```

Specifying `xilibtool` is the same as specifying `xild -libtool`.

Windows* Only

Create libraries using `xilib` or `xilink -lib` to create libraries of IPO mock object files and link them on the command line.

For example, assume that you create three mock object files by using a command similar to the following:

Example

```
ifort /c /Qipo a.obj b.obj c.obj
```

Further assume `a.obj` contains the main subprogram. You can enter commands similar to the following to link the objects.

Example

```
xilib -out:main.lib b.obj c.obj
```

or

```
xilink -lib -out:main.lib b.obj c.obj
```

You can link the library and the main program object file by entering a command similar to the following:

Example

```
xilink -out:result.exe a.obj main.lib
```

Requesting Compiler Reports with the xi* Tools

The compiler option `-opt-report` (Linux* and Mac OS* X) or `/Qopt-report` (Windows*) generates optimization reports with different levels of detail. Related compiler options described in Compiler Reports Quick Reference allow you to specify the phase, direct output to a file (instead of stderr), and request reports from all routines with names containing a string as part of their name.

The xi* tools are used with interprocedural optimization (IPO) during the final stage of IPO compilation. You can request compiler reports to be generated during the final IPO compilation by using certain options. The supported xi* tools are:

- Linker tools: `xilink` (Windows) and `xild` (Linux and Mac OS X)
- Library tools: `xilib` (Windows), `xiar` (Linux and Mac OS X), `xilibtool` (Mac OS X)

The following tables lists the compiler report options for the xi* tools. These options are equivalent to the corresponding compiler options, but occur during the final IPO compilation.

Tool Option	Description
`-qopt-report[=n]`	Enables optimization report generation with different levels of

	detail, directed to stderr. Valid values for *n* are 0 through 3. By default, when you specify this option without passing a value the compiler will generate a report with a medium level of detail.
`-qopt-report-file=`*`file`*	Generates an optimization report and directs the report output to the specified *file* name. If you omit the path, the file is created in the current directory. To create the file in a different directory, specify the full path to the output file and its file name.
`-qopt-report-phase=`*`name`*	Specifies the optimization phase *name* to use when generating reports. If you do not specify a phase the compiler defaults to all. You can request a list of all available phase by using the -qopt-report-help option.
`-qopt-report-routine=`*`string`*	Generates reports from all routines with names containing a *string* as part of their name. If not specified, the compiler will generate reports on all routines.
`-qopt-report-help`	Displays the optimization phases available.

To understand the compiler reports, use the links provided in Compiler Reports Overview.

Inline Expansion of Functions

Inline Function Expansion

Because inline function expansion does not require that the applications meet the criteria for whole program analysis normally require by IPO, this optimization is one of the primary optimizations used in Interprocedural Optimization (IPO). For function calls that the compiler believes are frequently executed, the Intel® compiler often decides to replace the instructions of the call with code for the function itself.

In the compiler, inline function expansion typically favors relatively small user functions over functions that are relatively large. This optimization improves application performance by performing the following:

- Removing the need to set up parameters for a function call
- Eliminating the function call branch
- Propagating constants

Function inlining can improve execution time by removing the runtime overhead of function calls; however, function inlining can increase code size, code complexity, and compile times. In general, when you instruct the compiler to perform function inlining, the

compiler can examine the source code in a much larger context, and the compiler can find more opportunities to apply optimizations.

Specifying `-ip` (Linux* and Mac OS* X) or `/Qip` (Windows*), single-file IP, causes the compiler to perform inline function expansion for calls to procedures defined within the current source file; in contrast, specifying `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows), multi-file IPO, causes the compiler to perform inline function expansion for calls to procedures defined in other files.

 **Caution**

Using the `-ip` and `-ipo` (Linux and Mac OS X) or `/Qip` and `/Qipo` (Windows) options can in some cases significantly increase compile time and code size.

Selecting Routines for Inlining

The compiler attempts to select the routines whose inline expansions will provide the greatest benefit to program performance. The selection is done using the default heuristics. The inlining heuristics used by the compiler differ based on whether or not you use Profile-Guided Optimizations (PGO): `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows).

When you use PGO with `-ip` or `-ipo` (Linux and Mac OS X) or `/Qip` or `/Qipo` (Windows), the compiler uses the following guidelines for applying heuristics:

- The default heuristic focuses on the most frequently executed call sites, based on the profile information gathered for the program.
- The default heuristic always inlines very small functions that meet the minimum inline criteria.

PGO (Windows)

Combining IPO and PGO produces better results than using IPO alone. PGO produces dynamic profiling information that can usually provide better optimization opportunities than the static profiling information used in IPO.

The compiler uses characteristics of the source code to estimate which function calls are executed most frequently. It applies these estimates to the PGO-based guidelines described above. The estimation of frequency, based on static characteristics of the source, is not always accurate.

Avoid using static profile information when combining PGO and IPO; with static profile information, the compiler can only estimate the application performance for the source files being used. Using dynamically generated profile information allows the compiler to accurately determine the real performance characteristics of the application.

Compiler Directed Inline Expansion of User Functions

Without directions from the user, the compiler attempts to estimate what functions should be inlined to optimize application performance. See Criteria for Inline Function Expansion for more information.

The following options are useful in situations where an application can benefit from user function inlining but does not need specific direction about inlining limits. Except where noted, these options are supported on IA-32, Intel® 64, and IA-64 architectures.

Linux* and Mac OS* X	Windows*	Effect
`-inline-level`	`/Ob`	Specifies the level of inline function expansion. Depending on the value specified, the option can disable or enable inlining. By default, the option enables inlining of any function if the compiler believes the function can be inlined. For more information, see the following topic: • `-inline-level` compiler option
`-ip-no-inlining`	`/Qip-no-inlining`	Disables only inlining normally enabled by the following options: • Linux and Mac OS X: `-ip` or `-ipo` • Windows: `/Qip`, `/Qipo`, or `/Ob2` No other IPO optimization are disabled. For more information, see the following topic: • `-ip-no-inlining` compiler option
`-ip-no-pinlining`	`/Qip-no-pinlining`	Disables partial inlining normally enabled by the following options: • Linux and Mac OS X: `-ip` or `-ipo` • Windows: `/Qip` or `/Qipo` No other IPO optimization are disabled. For more information, see the following topic: • `-ip-no-pinlining` compiler option
`-inline-debug-info`	`/Qinline-debug-info`	Keeps source information for inlined functions. The additional source code can be used by the Intel® Debugger track the user-defined call stack while using inlining. To use this option you must also specify an additional option to enable debugging: • Linux: `-g`

- Mac OS X: This option is not supported.
- Windows: `/debug`

For more information, see the following topic:

- `-inline-debug-info` compiler option

Developer Directed Inline Expansion of User Functions

In addition to the options that support compiler directed inline expansion of user functions, the compiler also provides compiler options that allow you to more precisely direct when and if inline function expansion occurs.

The compiler measures the relative size of a routine in an abstract value of intermediate language units, which is approximately equivalent to the number of instructions that will be generated. The compiler uses the intermediate language unit estimates to classify routines and functions as relatively small, medium, or large functions. The compiler then uses the estimates to determine when to inline a function; if the minimum criteria for inlining is met and all other things are equal, the compiler has an affinity for inlining relatively small functions and not inlining relative large functions.

The following developer directed inlining options provide the ability to change the boundaries used by the inlining optimizer to distinguish between small and large functions. These options are supported on IA-32, Intel® 64, and IA-64 architectures.

In general, you should use the `-inline-factor` (Linux* and Mac OS* X) and `/Qinline-factor` (Windows*) option before using the individual inlining options listed below; this single option effectively controls several other upper-limit options.

Linux* and Mac OS* X	Windows*	Effect
`-inline-factor`	`/Qinline-factor`	Controls the multiplier applied to all inlining options that define upper limits: `inline-max-size`, `inline-max-total-size`, `inline-max-per-routine`, and `inline-max-per-compile`. While you can specify an individual increase in any of the upper-limit options, this single option provides an efficient means of controlling all of the upper-limit options with a single command. By default, this option uses a multiplier of 100, which corresponds to a factor of 1. Specifying 200 implies a factor of 2, and so on. Experiment with the multiplier carefully. You could increase the upper limits to allow too much inlining, which might result in your system

		running out of memory. For more information, see the following topic: • `-inline-factor` compiler option
`-inline-forceinline`	`/Qinline-forceinline`	Instructs the compiler to force inlining of functions suggested for inlining whenever the compiler is capable doing so. Typically, the compiler targets functions that have been marked for inlining based on the presence of directives, like `DEC$ ATTRIBUTES FORCEINLINE`, in the source code; however, all such directives in the source code are treated only as suggestions for inlining. The option instructs the compiler to view the inlining suggestion as mandatory and inline the marked function if it can be done legally. For more information, see the following topic: • `-inline-forceinline` compiler option
`-inline-min-size`	`/Qinline-min-size`	Redefines the maximum size of small routines; routines that are equal to or smaller than the value specified are more likely to be inlined. For more information, see the following topic: • `-inline-min-size` compiler option
`-inline-max-size`	`/Qinline-max-size`	Redefines the minimum size of large routines; routines that are equal to or larger than the value specified are less likely to be inlined. For more information, see the following topic: • `-inline-max-size` compiler option
`-inline-max-total-size`	`/Qinline-max-total-size`	Limits the expanded size of inlined functions. For more information, see the following topic: • `-inline-max-total-size` compiler option
`-inline-max-per-routine`	`/Qinline-max-per-routine`	Limits the number of times inlining can be applied within a routine. For more information, see the following topic: • `-inline-max-per-routine` compiler option
`-inline-`	`/Qinline-`	Limits the number of times inlining can be applied

`max-per-compile`	`max-per-compile`	within a compilation unit. The compilation unit limit depends on the whether or not you specify the `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows) option. If you enable IPO, all source files that are part of the compilation are considered one compilation unit. For compilations not involving IPO each source file is considered an individual compilation unit. For more information, see the following topic: • `-inline-max-per-compile` compiler option

Using Profile-Guided Optimization (PGO)

Profile-Guided Optimizations Overview

Profile-guided Optimization (PGO) improves application performance by reorganizing code layout to reduce instruction-cache problems, shrinking code size, and reducing branch mispredictions. PGO provides information to the compiler about areas of an application that are most frequently executed. By knowing these areas, the compiler is able to be more selective and specific in optimizing the application.

PGO consists of three phases or steps.

1. Step one is to instrument the program. In this phase, the compiler creates and links an instrumented program from your source code and special code from the compiler.
2. Step two is to run the instrumented executable. Each time you execute the instrumented code, the instrumented program generates a dynamic information file, which is used in the final compilation.
3. Step three is a final compilation. When you compile a second time, the dynamic information files are merged into a summary file. Using the summary of the profile information in this file, the compiler attempts to optimize the execution of the most heavily traveled paths in the program.

See Profile-guided Optimization Quick Reference for information about the supported options and Profile an Application for specific details about using PGO from a command line.

PGO enables the compiler to take better advantage of the processor architecture, more effective use of instruction paging and cache memory, and make better branch predictions. PGO provides the following benefits:

- Use profile information for register allocation to optimize the location of spill code.
- Improve branch prediction for indirect function calls by identifying the most likely targets. (Some processors have longer pipelines, which improves branch prediction and translates into high performance gains.)

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

Interprocedural Optimization (IPO) and PGO can affect each other; using PGO can often enable the compiler to make better decisions about function inlining, which increases the effectiveness of interprocedural optimizations. Unlike other optimizations, such as those strictly for size or speed, the results of IPO and PGO vary. This variability is due to the unique characteristics of each program, which often include different profiles and different opportunities for optimizations.

Performance Improvements with PGO

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 infrequently executed (cold) error-handling code can be relocated so the branch is rarely predicted incorrectly. Minimizing cold code interleaved into the frequently executed (hot) code improves instruction cache behavior.

When you use PGO, consider the following guidelines:

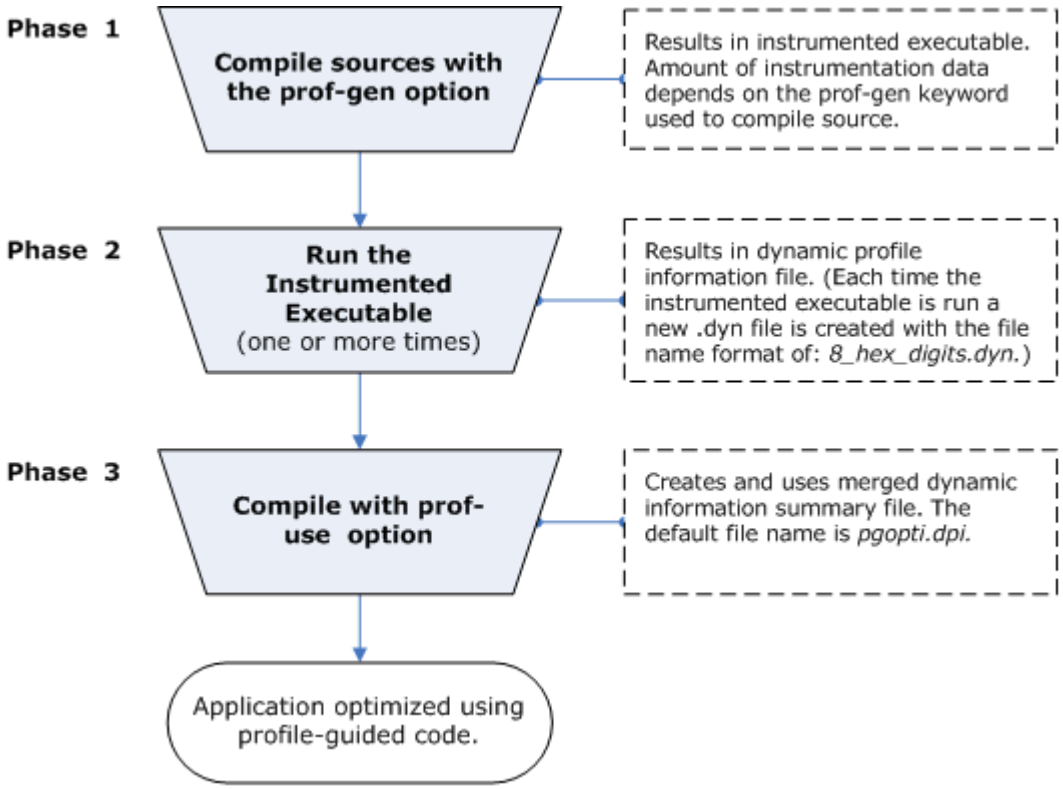
- Minimize changes to your program after you execute the instrumented code and before feedback compilation. During feedback compilation, the compiler ignores dynamic information for functions modified after that information was generated. (If you modify your program the compiler issues a warning that 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.
- Know the sections of your code that are the most heavily used. If the data set provided to your program is very consistent and displays similar behavior on every execution, then PGO can probably help optimize your program execution.
- Different data sets can result in different algorithms being called. The difference can cause the behavior of your program to vary for each execution. In cases where your code behavior differs greatly between executions, PGO may not provide noticeable benefits. If it takes multiple data sets to accurately characterize application performance then execute the application with all data sets then merge the dynamic profiles; this technique should result in an optimized application.

You must insure the benefit of the profiled information is worth the effort required to maintain up-to-date profiles.

Profile-Guided Optimization (PGO) Quick Reference
Profile-Guided Optimization consists of three phases (or steps):

1. Generating instrumented code by compiling with the `-prof-gen` (Linux* OS and Mac OS* X) or `/Qprof-gen` (Windows* OS) option when creating the instrumented executable.
2. Running the instrumented executable, which produces dynamic-information (.dyn) files.
3. Compiling the application using the profile information using the `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows) option.

The figure illustrates the phases and the results of each phase.



See Profile an Application for details about using each phase.

The following table lists the compiler options used in PGO:

Linux* and Mac OS* X	Windows*	Effect
`-prof-gen`	`/Qprof-gen`	Instruments a program for profiling to get the execution counts of each basic block. The option is used in phase 1 (instrumenting the code) to instruct the compiler to produce instrumented code for your object files in preparation for instrumented execution. By

		default, each instrumented execution creates one dynamic-information (dyn) file for each executable and (on Windows OS) one for each DLL invoked by the application. You can specify keywords, such as `-prof-gen=default` (Linux and Mac OS X) or `/Qprof-gen:default` (Windows). The keywords control the amount of source information gathered during phase 2 (run the instrumented executable). The `prof-gen` keywords are: • Specify `default` (or omit the keyword) to request profiling information for use with the `prof-use` option and optimization when the instrumented application is run (phase 2). • Specify `srcpos` or `globdata` to request additional profiling information for the `code coverage` and `test prioritization` tools when the instrumented application is run (phase 2). The phase 1 compilation creates an spi file. • Specify `globdata` to request additional profiling information for data ordering optimization when the instrumented application is run (phase 2). The phase 1 compilation creates an spi file. If you are performing a parallel make, this option will not affect it.
`-prof-use`	`/Qprof-use`	Instructs the compiler to produce a profile-optimized executable and merges available dynamic-information (dyn) files into a pgopti.dpi file. This option implicitly invokes the `profmerge` tool. The dynamic-information files are produced in phase 2 when you run the instrumented

executable.

If you perform multiple executions of the instrumented program to create additional dynamic-information files that are newer than the current summary pgopti.dpi file, this option merges the dynamic-information files again and overwrites the previous pgopti.dpi file (you can set the environment variable PROF_NO_CLOBBER to prevent the previous dpi file from being overwritten).

When you compile with prof-use, all dynamic information and summary information files should be in the same directory (current directory or the directory specified by the `prof-dir` option). If you need to use certain `profmerge` options not available with compiler options (such as specifying multiple directories), use the `profmerge` tool. For example, you can use `profmerge` to create a new summary dpi file before you compile with the `prof-use` option to create the optimized application.

You can specify keywords, such as `-prof-gen=weighted` (Linux and Mac OS X) or `/Qprof-gen:weighted` (Windows). If you omit the `weighted` keyword, the merged dynamic-information (dyn) files will be weighted proportionally to the length of time each application execution runs. If you specify the `weighted` keyword, the profiler applies an equal weighting (regardless of execution times) to the dyn file values to normalize the data counts. This keyword is useful when the execution runs have different time durations and you want them to be treated equally.

When you use `prof-use`, you can also specify the `prof-file` option to rename the summary dpi file and the `prof-dir` option to specify the directory for dynamic-information (dyn) and summary (dpi) files.

		Linux: • Using this option with `-prof-func-groups` allows you to control function grouping behavior.
`-no-fnsplit`	`/Qfnsplit-`	Disables function splitting. Function splitting is enabled by the `prof-use` option in phase 3 to improve code locality by splitting routines into different sections: one section to contain the cold or very infrequently executed (cold) code, and one section to contain the rest of the frequently executed (hot) code. You may want to disable function splitting for the following reasons: • Improve 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. • Account for the cases when the profile data does not represent the actual program behavior, that is, when the routine is actually used frequently rather than infrequently. This option is supported on IA-32 architecture for Windows OS and on IA-64 architecture for Windows and Linux OS. It is not supported on other platforms (Intel® 64 architecture, Mac OS X, and Linux on IA-32 architecture). Windows: This option behaves differently on systems based on IA-32 architecture than it does on systems based on IA-64 architecture. IA-32 architecture, Windows OS: • The option completely disables function splitting, placing all the code in one section. IA-64 architecture, Linux and Windows OS: • The 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.
`-prof-func-groups`	`/Qprof-func-order`	Enables ordering of program routines using profile information when specified with `prof-use` (phase 3). The instrumented program (phase 1) must have been compiled with the `prof-gen` option `srcpos` keyword. Not valid for multi-file compilation with the `ipo` options. Mac OS X: Not supported. IA-64 architecture: Not supported. For more information, see Using Function Ordering, Function Order Lists, Function Grouping, and Data Ordering Optimizations.
`-prof-data-order`	`/Qprof-data-order`	Enables ordering of static program data items based on profiling information when specified with `prof-use`. The instrumented program (phase 1) must have been compiled with the `prof-gen` option `srcpos` keyword. Not valid for multi-file compilation with the `ipo` options. Mac OS X: Not supported. For more information, see Using Function Ordering, Function Order Lists, Function Grouping, and Data Ordering Optimizations.
`-prof-src-dir`	`/Qprof-src-dir`	Controls whether full directory information for the source directory path is stored in or read from dynamic-information (dyn) files. When used during phase 1 compilation (`prof-gen`), this determines whether the full path is added into dyn file created during instrumented application execution. When used during `profmerge` or phase 3 compilation (`prof-use`), this determines whether the full path for source file names is used or ignored when

		reading the dyn or dpi files. Using the default `-prof-src-dir` (Linux and Mac OS X) or `/Qprof-src-dir` (Windows) uses the full directory information and also enables the use of the `prof-src-root and prof-src-cwd` options. If you specify `-no-prof-src-dir` (Linux and Mac OS X) `or /Qprof-src-dir-` (Windows), only the file name (and not the full path) is stored or used. If you do this, all dyn or dpi files must be in the current directory and the `prof-src-root and prof-src-cwd` options are ignored.
`-prof-dir`	`/Qprof-dir`	Specifies the directory in which dynamic information (dyn) files are created in, read from, and stored; otherwise, the dyn files are created in or read from the current directory used during compilation. For example, you can use this option when compiling in phase 1 (`prof-gen` option) to define where dynamic information files will be created when running the instrumented executable in phase 2. You also can use this option when compiling in phase 3 (`prof-use` option) to define where the dynamic information files will be read from and a summary file (dpi) created.
`-prof-src-root` or `-prof-src-cwd`	`/Qprof-src-root` or `/Qprof-src-cwd`	Specifies a directory path prefix for the root directory where the user's application files are stored: • To specify the directory prefix root where source files are stored, specify the `-prof-src-root` (Linux and Mac OS X) `or /Qprof-src-root` (Windows) option. • To use the current working directory, specify the `-prof-src-cwd` (Linux and Mac OS X) `or /Qprof-src-cwd` (Windows) option.

		This option is ignored if you specify `-no-prof-src-dir` (Linux and Mac OS X) or `/Qprof-src-dir-` (Windows).
`-prof-file`	`/Qprof-file`	Specifies file name for profiling summary file. If this option is not specified, the name of the file containing summary information will be pgopti.dpi.
`-prof-gen-sampling`	`/Qprof-gen-sampling`	IA-32 architecture. Prepares application executables for hardware profiling (sampling) and causes the compiler to generate source code mapping information. Mac OS X: This option is not supported.
`-ssp`	`/Qssp`	IA-32 architecture. Enables Software-based Speculative Pre-computation (SSP) optimization. Mac OS X: This option is not supported.

Refer to Quick Reference Lists for a complete listing of the quick reference topics.

Profile an Application

Profiling an application includes the following three phases:

- Instrumentation compilation and linking
- Instrumented execution
- Feedback compilation

This topic provides detailed information on how to profile an application by providing sample commands for each of the three phases (or steps).

1. Instrumentation compilation and linking

   Use `-prof-gen` (Linux* and Mac OS* X) or `/Qprof-gen` (Windows*) to produce an executable with instrumented information included.

Operating System	Commands
Linux and Mac OS X	`ifort -prof-gen -prof-dir/usr/profiled a1.f90 a2.f90 a3.f90` `ifort -oa1 a1.o a2.o a3.o`
Windows	`ifort /Qprof-gen /Qprof-dirc:\profiled a1.f90 a2.f90 a3.f90` `ifort a1.obj a2.obj a3.obj`

   Use the `-prof-dir` (Linux and Mac OS X) or `/Qprof-dir` (Windows) option if the application includes the source files located in multiple directories; using the option insures the profile information is generated in one consistent place. The

example commands demonstrate how to combine these options on multiple sources.

The compiler gathers extra information when you use the `-prof-gen=srcpos` (Linux and Mac OS X) or `/Qprof-gen:srcpos` (Windows) option; however, the extra information is collected to provide support only for specific Intel tools, like the code-coverage tool. If you do not expect to use such tools, do not specify `-prof-gen=srcpos` (Linux and Mac OS X) or `/Qprof-gen:srcpos` (Windows); the extended option does not provide better optimization and could slow parallel compile times.

2. Instrumented execution

Run your instrumented program with a representative set of data to create one or more dynamic information files.

Operating System	Command
Linux and Mac OS X	`./a1.out`
Windows	`a1.exe`

Executing the instrumented applications generates dynamic information file that has a unique name and .dyn suffix. A new dynamic information file is created every time you execute the instrumented program.

You can run the program more than once with different input data.

3. Feedback compilation

Before this step, copy all .dyn and .dpi files into the same directory. Compile and link the source files with `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows); the option instructs the compiler to use the generated dynamic information to guide the optimization:

Operating System	Examples
Linux and Mac OS X	`ifort -prof-use -ipo -prof-dir/usr/profiled a1.f90 a2.f90 a3.f90`
Windows	`ifort /Qprof-use /Qipo /Qprof-dirc:\profiled a1.f90 a2.f90 a3.f90`

This final phase compiles and links the sources files using the data from the dynamic information files generated during instrumented execution (phase 2).

In addition to the optimized executable, the compiler produces a pgopti.dpi file.

Most of the time, you should specify the default optimizations, `-O2` (Linux and Mac OS X) or `/O2` (Windows), for phase 1, and specify more advanced optimizations, `-ipo` (Linux) or `/Qipo` (Windows), during the final (phase 3) compilation. For

example, the example shown above used `-O2` (Linux and Mac OS X) or `/O2` (Windows) in step 1 and `-ipo` (Linux or Mac OS X) or `/Qipo` (Windows) in step 3.

Note

The compiler ignores the `-ipo` or `-ip` (Linux and Mac OS X) or `/Qipo` or `/Qip` (Windows) option during phase 1 with `-prof-gen` (Linux and Mac OS X) or `/Qprof-gen` (Windows).

PGO Tools

PGO Tools Overview

This section describes the tools that take advantage or support the Profile-guided Optimizations (PGO) available in the compiler.

- code coverage tool
- test prioritization tool
- profmerge tool
- proforder tool

In addition to the tools, this section also contains information on using Software-based Speculative Precomputation, which will allow you to optimize applications using profiling- and sampling-feedback methods on IA-32 architectures.

code coverage Tool

The code coverage tool provides software developers with a view of how much application code is exercised when a specific workload is applied to the application. To determine which code is used, the code coverage tool uses Profile-guided Optimization (PGO) options and optimizations. The major features of the code coverage tool are:

- Visually presenting code coverage information for an application with a customizable code coverage coloring scheme
- Displaying dynamic execution counts of each basic block of the application
- Providing differential coverage, or comparison, profile data for two runs of an application

The information about using the code coverage tool is separated into the following sections:

- code coverage tool Requirements
- Visually Presenting Code Coverage for an Application
- Excluding Code from Coverage Analysis

- Exporting Coverage Data

The tool analyzes static profile information generated by the compiler, as well as dynamic profile information generated by running an instrumented form of the application binaries on the workload. The tool can generate the in HTML-formatted report and export data in both text-, and XML-formatted files. The reports can be further customized to show color-coded, annotated, source-code listings that distinguish between used and unused code.

The code coverage tool is available on IA-32, Intel® 64, and IA-64 architectures on Linux* and Windows*. The tool is available on IA-32 and Intel® 64 architectures on Mac OS* X.

You can use the tool in a number of ways to improve development efficiency, reduce defects, and increase application performance:

- During the project testing phase, the tool can measure the overall quality of testing by showing how much code is actually tested.
- When applied to the profile of a performance workload, the code coverage tool can reveal how well the workload exercises the critical code in an application. High coverage of performance-critical modules is essential to taking full advantage of the Profile-Guided Optimizations that Intel Compilers offer.
- The tool provides an option, useful for both coverage and performance tuning, enabling developers to display the dynamic execution count for each basic block of the application.
- The code coverage tool can compare the profile of two different application runs. This feature can help locate portions of the code in an application that are unrevealed during testing but are exercised when the application is used outside the test space, for example, when used by a customer.

code coverage tool Requirements

To run the code coverage tool on an application, you must have following items:

- The application sources.
- The .spi file generated by the Intel® compiler when compiling the application for the instrumented binaries using the `-prof-gen=srcpos` (Linux and Mac OS X) or `/Qprof-gen:srcpos` (Windows) options.
- A pgopti.dpi file that contains the results of merging the dynamic profile information (.dyn) files, which is most easily generated by the profmerge tool. This file is also generated implicitly by the Intel® compilers when compiling an application with `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows) options with available .dyn and .dpi files.

See Understanding Profile-guided Optimization and Example of Profile-guided Optimization for general information on creating the files needed to run this tool.

Using the Tool

The tool uses the following syntax:

Tool Syntax

```
codecov [-codecov option]
```

where `-codecov_option` is one or more optional parameters specifying the tool option passed to the tool. The available tool options are listed in code coverage tools Options (below). If you do not use any additional tool options, the tool will provide the top-level code coverage for the entire application.

In general, you must perform the following steps to use the code coverage tool:

1. Compile the application using `-prof-gen=srcpos` (Linux and Mac OS X) or `/Qprof-gen:srcpos` (Windows).
   This step generates an instrumented executable and a corresponding static profile information (pgopti.spi) file.
2. Run the instrumented application.
   This step creates the dynamic profile information (.dyn) file Each time you run the instrumented application, the compiler generates a unique .dyn file either in the current directory or the directory specified in by prof_dir.
3. Use the profmerge tool to merge all the .dyn files into one .dpi (pgopti.dpi) file.
   This step consolidates results from all runs and represents the total profile information for the application, generates an optimized binary, and creates the dpi file needed by the code coverage tool.

   You can use the profmerge tool to merge the .dyn files into a .dpi file without recompiling the application. The profmerge tool can also merge multiple .dpi files into one .dpi file using the `profmerge -a` option. Select the name of the output .dpi file using the `profmerge -prof_dpi` option.

 **Caution**

The profmerge tool merges all .dyn files that exist in the given directory. Make sure unrelated .dyn files, which may remain from unrelated runs, are not present. Otherwise, the profile information will be skewed with invalid profile data, which can result in misleading coverage information and adverse performance of the optimized code.

4. Run the code coverage tool. (The valid syntax and tool options are shown below.)
   This step creates a report or exported data as specified. If no other options are

specified, the code coverage tool creates a single HTML file (CODE_COVERAGE.HTML) and a sub-directory (CodeCoverage) in the current directory. Open the file in a web browser to view the reports.

Note

Windows* only: Unlike the compiler options, which are preceded by forward slash ("/"), the tool options are preceded by a hyphen ("-").

The code coverage tool allows you to name the project and specify paths to specific, necessary files. The following example demonstrates how to name a project and specify .dpi and .spi files to use:

Example: specify .dpi and .spi files

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

The tool can add a contact name and generate an email link for that contact at the bottom of each HTML page. This provides a way to send an electronic message to the named contact. The following example demonstrates how to add specify a contact and the email links:

Example: add contact information

```
codecov -prj myProject -mname JoeSmith -maddr js@company.com
```

This following example demonstrates how to use the tool to specify the project name, specify the dynamic profile information file, and specify the output format and file name.

Example: export data to text

```
codecov -prj test1 -dpi test1.dpi -txtbcvrg test1_bcvrg.txt
```

code coverage tool Options

The tool uses the options listed in the table:

Option	Default	Description
`-bcolor` *color*	`#FFFF99`	Specifies the HTML color name for code in the uncovered blocks.
`-beginblkdsbl` *string*		Specifies the comment that marks the beginning of the code fragment to be ignored by the coverage tool.
`-ccolor` *color*	`#FFFFFF`	Specifies the HTML color name or code of the covered code.
`-comp` *file*		Specifies the file name that contains the list of files being (or not) displayed.

`-counts`		Generates dynamic execution counts.
`-demang`		Demangles both function names and their arguments.
`-dpi` *`file`*	`pgopti.dpi`	Specifies the file name of the dynamic profile information file (.dpi).
`-endblkdsbl` *`string`*		Specifies the comment that marks the end of the code fragment to be ignored by the coverage tool.
`-fcolor` *`color`*	`#FFCCCC`	Specifies the HTML color name for code of the uncovered functions.
`-help, -h`		Prints tool option descriptions.
`-icolor` *`color`*	`#FFFFFF`	Specifies the HTML color name or code of the information lines, such as basic-block markers and dynamic counts.
`-maddr` *`string`*	`Nobody`	Sets the email address of the web-page owner
`-mname` *`string`*	`Nobody`	Sets the name of the web-page owner.
`-nopartial`		Treats partially covered code as fully covered code.
`-nopmeter`		Turns off the progress meter. The meter is enabled by default.
`-onelinedsbl` *`string`*		Specifies the comment that marks individual lines of code or the whole functions to be ignored by the coverage tool.
`-pcolor` *`color`*	`#FAFAD2`	Specifies the HTML color name or code of the partially covered code.
`-prj` *`string`*		Sets the project name.
`-ref`		Finds the differential coverage with respect to ref_dpi_file.
`-spi` *`file`*	`pgopti.spi`	Specifies the file name of the static profile

		information file (.spi).
`-srcroot dir`		Specifies a different top level project directory than was used during compiler instrumentation run to use for relative paths to source files in place of absolute paths.
`-txtbcvrg file`		Export block-coverage for covered functions as text format. The file parameter must be in the form of a valid file name.
`-txtbcvrgfull file`		Export block-coverage for entire application in text and HTML formats. The file parameter must be in the form of a valid file name.
`-txtdcg file`		Generates the dynamic call-graph information in text format. The file parameter must be in the form of a valid file name.
`-txtfcvrg file`		Export function coverage for covered function in text format. The file parameter must by in the form of a valid file name.
`-ucolor color`	`#FFFFFF`	Specifies the HTML color name or code of the unknown code.
`-xcolor color`	`#90EE90`	Specifies the HTML color of the code ignored.
`-xmlbcvrg file`		Export the block-coverage for the covered function in XML format. The file parameter must by in the form of a valid file name.
`-xmlbcvrgfull file`		Export function coverage for entire application in XML format in addition to HTML output. The file parameter must be in the form of a valid file name.
`-xmlfcvrg file`		Export function coverage for covered function in XML format. The file parameter must be in the form of a valid file name.

Visually Presenting Code Coverage for an Application

Based on the profile information collected from running the instrumented binaries when testing an application, the Intel® compiler will create HTML-formatted reports using the code coverage tool. These reports 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 views

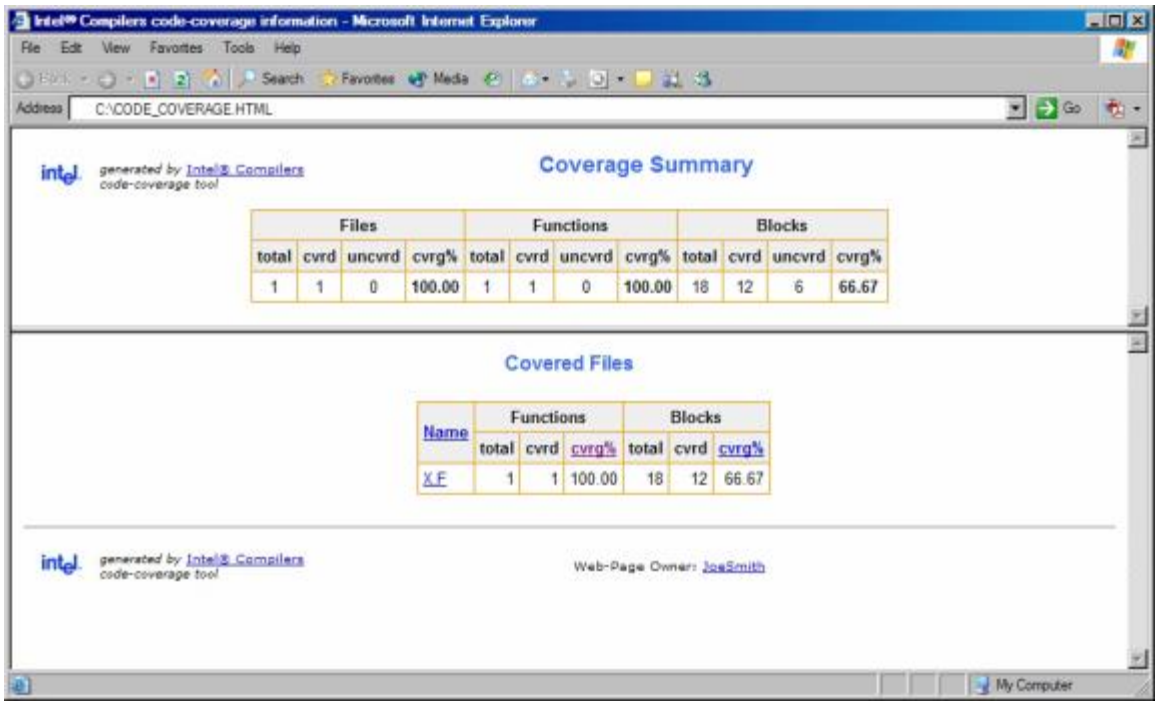
Top Level Coverage

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

- 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

By default, the code coverage tool generates a single HTML file (CODE_COVERAGE.HTML) and a subdirectory (CodeCoverage) in the current directory. The HTML file defines a frameset to display all of the other generated reports. Open the HTML file in a web-browser. The tool places all other generated report files in a CodeCoverage subdirectory.

If you choose to generate the html-formatted version of the report, you can view coverage source of that particular module directly from a browser. The following figure shows the top-level coverage report.



The coverage tool creates a frame set that allows quick 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, you can see the least-covered function in the list and by another click the browser displays the body of the function. You can 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:

- Number of blocks within uncovered functions
- Block coverage in the case of covered functions
- Function names

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, and unknown code. The default colors that the tool uses for presenting the coverage information are shown in the tables that follows:

Category	Default	Description
Covered code	#FFFFFF	Indicates code was exercised by the tests. You can override the default color with the -ccolor tool option.
Uncovered basic block	#FFFF99	Indicates the basic blocks that were not exercised by any of the tests. However, these blocks were within functions that were executed during the tests. You can override the default color with the -bcolor tool option.
Uncovered function	#FFCCCC	Indicates functions that were never called during the tests. You can override the default color with the -fcolor tool option.
Partially covered code	#FAFAD2	Indicates that more than one basic block was generated for the code at this position. Some of the blocks were covered and some were not. You can override the default color with the -pcolor tool option.
Ignored code	#90EE90	Indicates code that was specifically marked to be ignored. You can override this default color using the -xcolor tool option.
Information lines	#FFFFFF	Indicates basic-block markers and dynamic counts. You can override the default color with the -icolor tool option.
Unknown	#FFFFFF	Indicates that 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. You can override the default color with the -ucolor tool option.

The default colors can be customized to be any valid HTML color name or hexadecimal value 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-formatted report 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.

Dynamic Counters

The coverage tool can be configured to generate the information about the dynamic execution counts. This ability can display the dynamic execution count of each basic block of the application and is useful for both coverage and performance tuning.

The custom 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:

Example

```
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 from two runs of an application: a reference run and a new run identifying the code that is covered by the

new run but not covered by the reference run. Use this feature to find the portion of the applications code that is not covered by the applications 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 applications test space.

Generating Reference Data

Create the dynamic profile information for the reference data, which can be used in differential coverage reporting later, by using the `-ref` option. The following command demonstrate a typical command for generating the reference data:

Example: generating reference data

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

The coverage statistics of a differential-coverage run shows the percentage of the code exercised on a new run but missed in the reference run. In such cases, the tool shows only the modules that included the code that was not covered. Keep this in mind when viewing the coloring scheme in the source views.

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 Differential Coverage

To run the code coverage tool for differential coverage, you must have the application sources, the .spi file, and the .dpi file, as described in the code coverage tool Requirements section (above).

Once the required files are available, enter a command similar to the following begin the process of differential coverage analysis:

Example

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

Specify the .dpi and .spi files using the `-spi` and `-dpi` options.

Excluding Code from Coverage Analysis

The code coverage tool allows you to exclude portions of your code from coverage analysis. This ability can be useful during development; for example, certain portions of code might include functions used for debugging only. The test case should not include tests for functionality that will unavailable in the final application.

Another example of code that can be excluded is code that might be designed to deal with internal errors unlikely to occur in the application. In such cases, not having a test case lack of a test case is preferred. You might want to ignore infeasible (dead) code in the coverage analysis. The code coverage tool provides several options for marking

portions of the code infeasible (dead) and ignoring the code at the file level, function level, line level, and arbitrary code boundaries indicated by user-specific comments. The following sections explain how to exclude code at different levels.

Including and Excluding Coverage at the File Level

The code coverage tool provides the ability to selectively include or exclude files for analysis. Create a component file and add the appropriate string values that indicate the file and directory name for code you want included or excluded. Pass the file name as a parameter of the `-comp` option. The following example shows the general command:

Example: specifying a component file

```
codecov -comp file
```

where `file` is the name of a text file containing strings that ask as file and directory name masks for including and excluding file-level analysis. For example, assume that the following:

- You want to include all files with the string "source" in the file name or directory name.
- You create a component text file named `myComp.txt` with the selective inclusion string.

Once you have a component file, enter a command similar to the following:

Example

```
codecov -comp myComp.txt
```

In this example, individual files name including the string "source" (like source1.f and source2.f) and files in directories where the name contains the string "source" (like source/file1.f and source2\file2.f) are include in the analysis.

Excluding files is done in the same way; however, the string must have a tilde (~) prefix. The inclusion and exclusion can be specified in the same component file.

For example, assume you want to analyze all individual files or files contained in a directory where the name included the string "source", and you wanted to exclude all individual file and files contained in directories where the name included the string "skip". You would add content similar to the following to the component file (`myComp.txt`) and pass it to the `-comp` option:

Example: inclusion and exclusion strings

```
source
~skip
```

Entering the `codecov -comp myComp.txt` command with both instructions in the component file will instruct the tool to include individual files where the name contains "source" (like source1.f and source2.f) and directories where the name contains "source" (like source/file1.f and source2\file2.f), and exclude any individual files where the name contains "skip" (like skipthis1.f and skipthis2.f) or directories where the name contains "skip" (like skipthese1\debug1.f and skipthese2\debug2.f).

Excluding Coverage at the Line and Function Level

You can mark individual lines for exclusion my passing string values to the `-onelinedsbl` option. For example, assume that you have some code similar to the following:

Sample code

```
print*, "ERROR: n  = ", n  ! NO COVER
print*, "       n2 = ", n2  ! INF IA-32 architecture
```

If you wanted to exclude all functions marked with the comments NO_COVER or INF IA-32 architecture, you would enter a command similar to the following.

Example

```
codecov -onelinedsbl NO COVER -onelinedsbl "INF IA-32
architecture"
```

You can specify multiple exclusion strings simultaneously, and you can specify any string values for the markers; however, you must remember the following guidelines when using this option:

- Inline comments must occur at the end of the statement.
- The string must be a part of an inline comment.

An entire function can be excluded from coverage analysis using the same methods. For example, the following function will be ignored from the coverage analysis when you issue example command shown above.

Sample code

```
subroutine dumpInfo (n)
integer n ! NO COVER
...
end subroutine
```

Additionally, you can use the code coverage tool to color the infeasible code with any valid HTML color code by combining the `-onelinedsbl` and `-xcolor` options. The following example commands demonstrate the combination:

Example: combining tool options

```
codecov -onelinedsbl INF -xcolor lightgreen
codecov -onelinedsbl INF -xcolor #CCFFCC
```

Excluding Code by Defining Arbitrary Boundaries

The code coverage tool provides the ability to arbitrarily exclude code from coverage analysis. This feature is most useful where the excluded code either occur inside of a function or spans several functions.

Use the `-beginblkdsbl` and `-endblkdsbl` options to mark the beginning and end, respectively, of any arbitrarily defined boundary to exclude code from analysis. Remember the following guidelines when using these options:

- Inline comments must occur at the end of the statement.

- The string must be a part of an inline comment.

For example assume that you have the following code:

Sample code

```
integer n, n2
n = 123
n2 = n*n
if (n2 .lt. 0) then
! /* BINF */
  print*, "ERROR: n  = ", n
  print*, "       n2 = ", n2
! // EINF
else if (n2 .eq. 0) then
  print*, "zero: n = ", n, " n2 = ", n2
else
  print*, "positive: n = ", n, " n2 = ", n2
endif
end
```

The following example commands demonstrate how to use the `-beginblkdsbl` option to mark the beginning and the `-endblkdsbl` option to mark the end of code to exclude from the sample shown above.

Example: arbitrary code marker commands

```
codecov -xcolor #ccFFCC -beginblkdsbl BINF -endblkdsbl EINF
codecov -xcolor #ccFFCC -beginblkdsbl "BEGIN_INF" -endblkdsbl "END_INF"
```

Notice that you can combine these options in combination with the `-xcolor` option.

Exporting Coverage Data

The code coverage tool provides specific options to extract coverage data from the dynamic profile information (.dpi files) that result from running instrumented application binaries under various workloads. The tool can export the coverage data in various formats for post-processing and direct loading into databases: the default HTML, text, and XML. You can choose to export data at the function and basic block levels.

There are two basic methods for exporting the data: quick export and combined export. Each method has associated options supported by the tool

- **Quick export:** The first method is to export the data coverage to text- or XML-formatted files without generating the default HTML report. The application sources need not be present for this method. The code coverage tool creates reports and provides statistics only about the portions of the application executed. The resulting analysis and reporting occurs quickly, which makes it practical to apply the coverage tool to the dynamic profile information (the .dpi file) for every test case in a given test space instead of applying the tool to the profile of individual test suites or the merge of all test suites. The `-xmlfcvrg`, `-txtfcvrg`, `-xmlbcvrg` and `-txtbcvrg` options support the first method.
- **Combined export:** The second method is to generate the default HTML and simultaneously export the data to text- and XML-formatted files. This process is slower than first method since the application sources are parsed and reports

generated. The `-xmlbcvrgfull` and `-txtbcvrgfull` options support the second method.

These export methods provide the means to quickly extend the code coverage reporting capabilities by supplying consistently formatted output from the code coverage tool. You can extend these by creating additional reporting tools on top of these report files.

Quick Export

The profile of covered functions of an application can be exported quickly using the `-xmlfcvrg`, `-txtfcvrg`, `-xmlbcvrg`, and `-txtbcvrg` options. When using any of these options, specify the output file that will contain the coverage report. For example, enter a command similar to the following to generate a report of covered functions in XML formatted output:

Example: quick export of function data

```
codecov -prj test1 -dpi test1.dpi -xmlfcvrg test1_fcvrg.xml
```

The resulting report will show how many times each function was executed and the total number of blocks of each function together with the number of covered blocks and the block coverage of each function. The following example shows some of the content of a typical XML report.

XML-formatted report example

```
<PROJECT name = "test1">
   <MODULE name = "D:\SAMPLE.F">
      <FUNCTION name="f0" freq="2">
         <BLOCKS total="6" covered="5" coverage="83.33%"></BLOCKS>
      </FUNCTION>
      ...
   </MODULE>
   <MODULE name = "D:\SAMPLE2.F">
      ...
   </MODULE>
</PROJECT>
```

In the above example, we note that function f0, which is defined in file sample.f, has been executed twice. It has a total number of six basic blocks, five of which are executed, resulting in an 83.33% basic block coverage.

You can also export the data in text format using the `-txtfcvrg` option. The generated text report, using this option, for the above example would be similar to the following example:

Text-formatted report example

```
Covered Functions in File: "D:\SAMPLE.F"
"f0"    2       6       5       83.33
"f1"    1       6       4       66.67
"f2"    1       6       3       50.00
...
```

In the text formatted version of the report, the each line of the report should be read in the following manner:

Column 1	Column 2	Column 3	Column 4	Column 5

function name	execution frequency	line number of the start of the function definition	column number of the start of the function definition	percentage of basic-block coverage of the function

Additionally, the tool supports exporting the block level coverage data using the `-xmlbcvrg` option. For example, enter a command similar to the following to generate a report of covered blocks in XML formatted output:

Example: quick export of block data to XML

```
codecov -prj test1 -dpi test1.dpi -xmlbcvrg test1_bcvrg.xml
```

The example command shown above would generate XML-formatted results similar to the following:

XML-formatted report example

```
<PROJECT name = "test1">
  <MODULE name = "D:\SAMPLE.cF90">
    <FUNCTION name="f0" freq="2">
    ...
      <BLOCK line="11" col="2">
        <INSTANCE id="1" freq="1"> </INSTANCE>
      </BLOCK>
      <BLOCK line="12" col="3">
        <INSTANCE id="1" freq="2"> </INSTANCE>
        <INSTANCE id="2" freq="1"> </INSTANCE>
      </BLOCK>
```

In the sample report, notice that one basic block is generated for the code in function f0 at the line 11, column 2 of the file sample.f90. This particular block has been executed only once. Also notice that there are two basic blocks generated for the code that starts at line 12, column 3 of file. One of these blocks, which has id = 1, has been executed two times, while the other block has been executed only once. A similar report in text format can be generated through the `-txtbcvrg` option.

Combined Exports

The code coverage tool has also the capability of exporting coverage data in the default HTML format while simultaneously generating the text- and XML-formatted reports.

Use the `-xmlbcvrgfull` and `-txtbcvrgfull` options to generate reports in all supported formatted in a single run. These options export the basic-block level coverage data while simultaneously generating the HTML reports. These options generate more complete reports since they include analysis on functions that were not executed at all. However, exporting the coverage data using these options requires access to application source files and take much longer to run.

Dynamic Call Graphs

Using the `-txtdcg` option the tool can provide detailed information about the dynamic call graphs in an application. Specify an output file for the dynamic call-graph report. The resulting call graph report contains information about the percentage of static and dynamic calls (direct, indirect, and virtual) at the application, module, and function levels.

test prioritization Tool

The test prioritization tool enables the profile-guided optimizations on IA-32, Intel® 64, and IA-64 architectures, on Linux* and Windows*, to select and prioritize test for an application based on prior execution profiles. The tool is available on IA-32 and Intel® 64 architectures on Mac OS* X.

The tool offers a potential of significant time saving in testing and developing large-scale applications where testing is the major bottleneck.

Development often requires changing applications modules. As applications change, developers can have a difficult time retaining the quality of their functional and performance tests so they are current and on-target. The test prioritization tool lets software developers select and prioritize application tests as application profiles change.

The information about the tool is separated into the following sections:

- Features and benefits
- Requirements and syntax
- Usage model
- Tool options
- Running the tool

Features and Benefits

The test prioritization tool provides an effective testing hierarchy based on the code coverage for an application. The following list summarizes the advantages of using the tool:

- 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.

See Understanding Profile-guided Optimization and Example of Profile-guided Optimization for general information on creating the files needed to run this tool.

test prioritization Tool Requirements

The test prioritization tool needs the following items to work:

- The .spi file generated by Intel® compilers when compiling the application for the instrumented binaries with the `-prof-gen=srcpos` (Linux* and Mac OS* X) or `/Qprof-gen:srcpos` (Windows*) option.
- The .dpi files generated by the profmerge tool as a result of merging the dynamic profile information .dyn files of each of the application tests. (Run the profmerge tool on all .dyn files that are generated for each individual test and name the resulting .dpi in a fashion that uniquely identifies the test.)

 **Caution**

The profmerge tool merges all .dyn files that exist in the given directory. Make sure unrelated .dyn files, which may remain from unrelated runs, are not present. Otherwise, the profile information will be skewed with invalid profile data, which can result in misleading coverage information and adverse performance of the optimized code;

- User-generated file containing the list of tests to be prioritized. For successful instrumented code run, you should:
  - Name each test .dpi file so the file names uniquely identify each test.
  - Create a .dpi list file, which is a text file that contains the names of all .dpi test files.

    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` states 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.

The tool uses the following general syntax:

Tool Syntax

```
tselect -dpi_list file
```

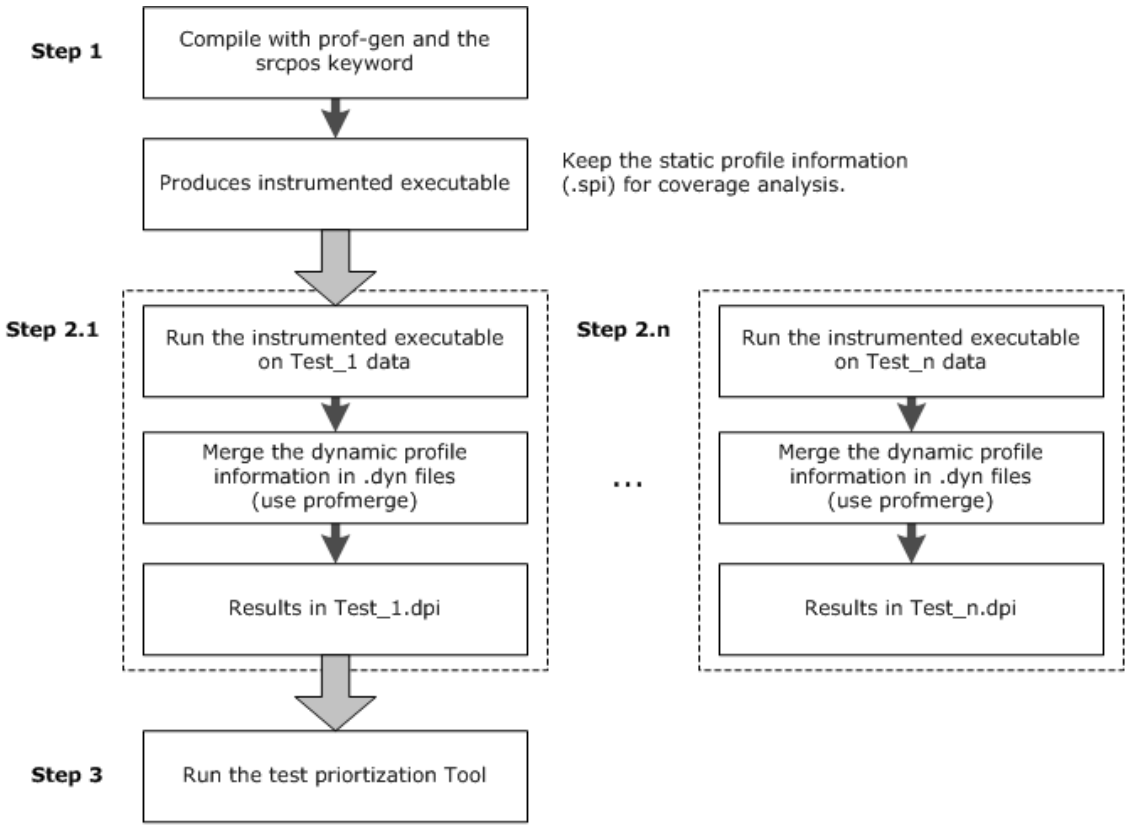
`-dpi_list` is a required tool option that sets the path to the list file containing the list of the all .dpi files. All other tool commands are optional.

Note

Windows* only: Unlike the compiler options, which are preceded by forward slash ("/"), the tool options are preceded by a hyphen ("-").

Usage Model

The following figure illustrates a typical test prioritization tool usage model.



test prioritization tool Options

The tool uses the options that are listed in the following table:

Option	Default	Description
`-help`		Prints tool option descriptions.

`-dpi_list` *file*		Required. Specifies the file name of the file that contains the names of the dynamic profile information (`.dpi`) files. Each line of the file must contain only one `.dpi` file name, which can be optionally followed by its execution time. The name must uniquely identify the test.
`-spi` *file*	`pgopti.spi`	Specifies the file name of the static profile information file (.SPI).
`-o` *file*		Specifies the file name of the output report file.
`-comp` *file*		Specifies the file name that contains the list of files of interest.
`-cutoff` *value*		Instructs the tool to terminate when the cumulative block coverage reaches a preset percentage, as specified by *value*, of pre-computed total coverage. *value* must be greater than 0.0 (for example, 99.00) but not greater than 100. *value* can be set to 100.
`-nototal`		Instructs the tool to ignore the pre-compute total coverage process.
`-mintime`		Instructs the tool to minimize 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.
`-srcbasedir` *dir*		Specifies a different top level project directory than was used during compiler instrumentation run with the `prof-src-root` compiler option to support relative paths to source files in place of absolute paths.
`-verbose`		Instructs the tool to generate more logging information about program progress.

Running the tool

The following steps demonstrate one simple example for running the tool on IA-32 architectures.

1. Specify the directory by entering a command similar to the following:

Example

```
set prof-dir=c:\myApp\prof-dir
```

2. Compile the program and generate instrumented binary by issuing commands similar to the following:

Operating System	Command
Linux and Mac OS X	`ifort -prof-gen=srcpos myApp.f90`
Windows	`ifort /Qprof-gen:srcpos myApp.f90`

This commands shown above compiles the program and generates instrumented binary `myApp` as well as the corresponding static profile information `pgopti.spi`.

3. Make sure that unrelated `.dyn` files are not present by issuing a command similar to the following:

Example

```
rm prof-dir \*.dyn
```

4. Run the instrumented files by issuing a command similar to the following:

Example

```
myApp < data1
```

The 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 the `-prof-dir` step above.

5. Merge all `.dyn` file into a single file by issuing a command similar to the following:

Example

```
profmerge -prof_dpi Test1.dpi
```

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. Again make sure there are no unrelated .dyn files present a second time by issuing a command similar to the following:

Example

```
rm prof-dir \*.dyn
```

7. Run the instrumented application and generate one or more new dynamic profile information files that have an extension .dyn in the directory specified the `prof-dir` step above by issuing a command similar to the following:

Example

```
myApp < data2
```

8. Merge all .dyn files into a single file, by issuing a command similar to the following

   Example

   ```
   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. Make sure that there are no unrelated .dyn files present for the final time, by issuing a command similar to the following:

   Example

   ```
   rm prof-dir \*.dyn
   ```

10. Run 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` by issuing a command similar to the following:

    Example

    ```
    myApp < data3
    ```

11. Merge all .dyn file into a single file, by issuing a command similar to the following:

    Example

    ```
    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`.

Tool Usage Examples

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.

Example 1: Minimizing the Number of Tests

The following example describes how minimize the number of test runs.

Example Syntax

```
tselect -dpi_list tests_list -spi pgopti.spi
```

where the `-spi` option specifies the path to the .spi file.

The following sample output shows typical results.

Sample Output

```
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   100.50    52.17     50.00    Test2.dpi
```

In this example, the results provide 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:

Sample Output

```
Test1.dpi 00:00:60:35
Test2.dpi 00:00:10:15
Test3.dpi 00:00:30:45
```

The following command minimizes the execution time by passing the `-mintime` option:

Sample Syntax

```
tselect -dpi_list tests_list -spi pgopti.spi -mintime
```

The following sample output shows possible results:

Sample Output

```
Total number of tests   =  3
Total block coverage    ~  52.17
Total function coverage ~  50.00
Total execution time    =  1:41:35
 num  elaspedTime  %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.

The order of tests 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 1 shown above: first Test3, then Test2. In Example 2, Test2 is the test that gives the highest coverage per execution time, so Test2 is picked as the first test to run.

Using Other Options

The `-cutoff` option enables the tool to exit when it reaches a given level of basic block coverage. The following example demonstrates how to the option:

Example

```
tselect -dpi_list tests_list -spi pgopti.spi -cutoff 85.00
```

If the tool is run with the cutoff value of 85.00, as 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 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.

profmerge and proforder Tools

profmerge Tool

Use the profmerge tool to merge dynamic profile information (.dyn) files and any specified summary files (.dpi). The compiler executes profmerge automatically during the feedback compilation phase when you specify `-prof-use` (Linux* and Mac OS* X) or `/Qprof-use` (Windows*).

The command-line usage for profmerge is as follows:

Syntax

```
profmerge [-prof_dir dir_name]
```

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

Note

The spelling of tools options may differ slightly from compiler options. Tools options use an underscore (for example -prof_dir) instead of the hyphen used by compiler options ompiler options (for example `-prof-dir` or `/Qprof-dir`) to join words. Also, on Windows* OS systems, the tool options are preceded by a hyphen ("-") unlike Windows compiler options, which are preceded by a forward slash ("/").

You can use profmerge tool to merge .dyn files into a .dpi file without recompiling the application. Thus, you can run the instrumented executable file on multiple systems to generate dyn files, and optionally use profmerge with the `-prof_dpi` option to name each summary dpi file created from the multiple dyn files.

Since the profmerge tool merges all the .dyn files that exist in the given directory, make sure unrelated .dyn files are not present; otherwise, profile information will be based on invalid profile data, which can negatively impact the performance of optimized code.

profmerge Options

The profmerge tool supports the following options:

Tool Option	Description
`-help`	Lists supported options.
`-nologo`	Disables version information. This option is supported on Windows* only.
`-exclude_funcs` *functions*	Excludes `functions` from the profile. The list items must be separated by a comma (","); you can use a period (".") as a wild card character in function names.
`-prof_dir` *dir*	Specifies the directory from which to read .dyn and .dpi files, and write the .dpi file. Alternatively, you can set the environment variable PROF_DIR.
`-prof_dpi` *file*	Specifies the name of the .dpi file being generated.
`-prof_file` *file*	Merges information from file matching: dpi_file_and_dyn_tag.
`-dump`	Displays profile information.
`-src_old` *dir* `-src new` *dir*	Changes the directory path stored within the .dpi file.
`-src_no_dir`	Uses only the file name and not the directory name when reading dyn/dpi records. If you specify `-src_no_dir`, the directory name of the source file will be ignored when deciding which profile data records correspond to a specific application routine, and the `-src-root` option is ignored.
`-src-root` *dir*	Specifies a directory path prefix for the root directory where the user's application files are stored. This option is ignored if you specify `-src_no_dir`.
`-a` *file1.dpi...fileN.dpi*	Specifies and merges available .dpi files.
`-verbose`	Instructs the tool to display full information during merge.
`-weighted`	Instructs the tool to apply an equal weighting (regardless of execution times) to the dyn file values to normalize the data counts. This keyword is useful when the execution

runs have different time durations and you want them to be treated equally.

Relocating source files using profmerge

The Intel® 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.

You can disable the use of directory names when reading dyn/dpi file records by specifying the `profmerge option -src_no_dir.` This `profmerge` option is the same as the compiler option `-no-prof-src-dir` (Linux and Mac OS X) and `/Qprof-src-dir-` (Windows).

To enable the movement of application sources, as well as the sharing of profile summary files, you can use the `profmerge` option `-src-root` to specify a directory path prefix for the root directory where the application files are stored. Alternatively, you can specify the option pair `-src_old -src_new` to modify the data in an existing summary dpi file. For example:

Example: relocation command syntax

```
profmerge -prof dir
<dir1> -src old <dir2>
-src_new <dir3>
```

where *<dir1>* is the full path to dynamic information file (.dpi), *<dir2>* is the old full path to source files, and *<dir3>* is the new full path to source files. The example command (above) reads the `pgopti.dpi` file, in the location specified in *<dir1>*. For each routine represented in the `pgopti.dpi` file, whose source path begins with the *<dir2>* prefix, profmerge replaces that prefix with *<dir3>*. The `pgopti.dpi` file is updated with the new source path information.

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

Operating System	Command Examples
Linux and Mac OS X	`profmerge -prof dir -src old /src/prog 1 -src new /src/prog 2` `profmerge -prof_dir -src_old /proj_1 -src_new /proj_2`
Windows	`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.

Note

Because the source relocation feature of profmerge modifies the `pgopti.dpi` file, consider making a backup copy of the file before performing the source relocation.

proforder Tool

The proforder tool is used as part of the feedback compilation phase, to improve program performance. Use proforder to generate a function order list for use with the `/ORDER` linker option. The tool uses the following syntax:

Syntax

`proforder [-prof_dir dir] [-o file]`

where *`dir`* is the directory containing the profile files (`.dpi` and `.spi`), and *`file`* is the optional name of the function order list file. The default name is `proford.txt`.

Note

The spelling of tools options may differ slightly from compiler options. Tools options use an underscore (for example -prof_dir) instead of the hyphen used by compiler options (for example `-prof-dir` or `/Qprof-dir`) to join words. Also, on Windows* OS systems, the tool options are preceded by a hyphen ("-") unlike Windows compiler options, which are preceded by a forward slash ("/").

proforder Options

The proforder tool supports the following options:

Tool Option	Default	Description
`-help`		Lists supported options.
`-nologo`		Disables version information. This option is supported on Windows* only.
`-omit_static`		Instructs the tool to omit static functions from function ordering.
`-prof_dir` *`dir`*		Specifies the directory where the .spi and .dpi file reside.
`-prof_dpi` *`file`*		Specifies the name of the .dpi file.
`-prof_file` *`string`*		Selects the .dpi and .spi files that include the substring value in the file name matching the values passed as string.

`-prof_spi` *file*		Specifies the name of the .spi file.
`-o` *file*	`proford.txt`	Specifies an alternate name for the output file.

Using Function Order Lists, Function Grouping, Function Ordering, and Data Ordering Optimizations

Instead of doing a full multi-file interprocedural build of your application by using the compiler option `-ipo` (Linux* OS) or `/Qipo` (Windows* OS), you can obtain some of the benefits by having the compiler and linker work together to make global decisions about where to place the procedures and data in your application. These optimizations are not supported on Mac OS* X systems.

The following table lists each optimization, the type of procedures or global data it applies to, and the operating systems and architectures that it is supported on.

Optimization	Type of Procedure or Data	Supported OS and Architectures
Function Order Lists: Specifies the order in which the linker should link the non-static routines (procedures) of your program. This optimization can improve application performance by improving code locality and reduce paging. Also see Comparison of Function Order Lists and IPO Code Layout.	EXTERNAL procedures and library procedures only (not other types of static procedures).	Windows OS: IA-32, Intel® 64, and IA-64 architectures Linux OS: not supported
Function Grouping: Specifies that the linker should place the extern and static routines (procedures) of your program into hot or cold program sections. This optimization can improve	EXTERNAL procedures and static procedures only (not library procedures).	Linux OS: IA-32 and Intel 64 architectures Windows OS: not supported

application performance by improving code locality and reduce paging.		
Function Ordering: Enables ordering of static and extern routines using profile information. Specifies the order in which the linker should link the routines (procedures) of your program. This optimization can improve application performance by improving code locality and reduce paging.	EXTERNAL procedures and static procedures only (not library procedures)	Linux and Windows OS: IA-32, Intel 64, and IA-64 architectures
Data Ordering: Enables ordering of static global data items (data in common blocks, module variables, and variables for which the compiler applied the SAVE attribute or statement) based on profiling information. Specifies the order in which the linker should link global data of your program. This optimization can improve application performance by improving the locality of static global data, reduce paging of large data sets, and improve data cache use.	Static global data (data in common blocks, module variables, and variables for which the compiler applied the SAVE attribute or statement) only	Linux and Windows OS, IA-32, Intel 64, and IA-64 architectures

You can use only one of the function-related ordering optimizations listed above. However, you can use the Data Ordering optimization with any one of the function-related ordering optimizations listed above, such as Data Ordering with Function Ordering, or Data Ordering with Function Grouping. In this case, specify the `prof-gen` option keyword `globdata` (needed for Data Ordering) instead of `srcpos` (needed for function-related ordering).

The following sections show the commands needed to implement each of these optimizations: function order list, function grouping, function ordering, and data ordering. For all of these optimizations, omit the `-ipo` (Linux* OS) or `/Qipo` (Windows OS) or equivalent compiler option.

Generating a Function Order List (Windows OS)

This section provides an example of the process for generating a function order list. Assume you have a Fortran program that consists of the following files: `file1.f90` and `file2.f90`. Additionally, assume you have created a directory for the profile data files called `c:\profdata`. You would enter commands similar to the following to generate and use a function order list for your Windows application.

1. Compile your program using the `/Qprof-gen:srcpos` option. Use the `/Qprof-dir` option to specify the directory location of the profile files. This step creates an instrumented executable.

   Example commands

   ```
   ifort /exe:myprog /Qprof-gen:srcpos /Qprof-dir c:\profdata file1.f90 file2.f90
   ```

2. Run the instrumented program with one or more sets of input data. Change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

   Example commands

   ```
   myprog.exe
   ```

3. Before this step, copy all .dyn and .dpi files into the same directory. Merge the data from one or more runs of the instrumented program by using the profmerge tool to produce the `pgopti.dpi` file. Use the `/prof_dir` option to specify the directory location of the .dyn files.

   Example commands

   ```
   profmerge /prof_dir c:\profdata
   ```

4. Generate the function order list using the proforder tool. (By default, the function order list is produced in the file `proford.txt`.)

   Example commands

   ```
   proforder /prof_dir c:\profdata /o myprog.txt
   ```

5. Compile the application with the generated profile feedback by specifying the `ORDER` option to the linker. Use the `/Qprof-dir` option to specify the directory location of the profile files.

   Example commands

   ```
   ifort /exe:myprog Qprof-dir c:\profdata file1.f90 file2.f90 /link /ORDER:@MYPROG.t
   ```

Using Function Grouping (Linux OS)

This section provides a general example of the process for using the function grouping optimization. Assume you have a Fortran program that consists of the following files: `file1.f90` and `file2.f90`. Additionally, assume you have created a directory for the profile data files called `profdata`. You would enter commands similar to the following to use a function grouping for your Linux application.

1. Compile your program using the `-prof-gen` option. Use the `-prof-dir` option to specify the directory location of the profile files. This step creates an instrumented executable.

   Example commands

   ```
   ifort -o myprog -prof-gen -prof-dir ./profdata file1.f90 file2.f90
   ```

2. Run the instrumented program with one or more sets of input data. Change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

   Example commands

   ```
   ./myprog
   ```

3. Copy all .dyn and .dpi files into the same directory. If needed, you can merge the data from one or more runs of the instrumented program by using the profmerge tool to produce the `pgopti.dpi` file.
4. Compile the application with the generated profile feedback by specifying the `-prof-func-group` option to request the function grouping as well as the `-prof-use` option to request feedback compilation. Again, use the `-prof-dir` option to specify the location of the profile files.

   Example commands

   ```
   ifort /exe:myprog file1.f90 file2.f90 -prof-func-group -prof-use -prof-dir ./profd
   ```

Using Function Ordering

This section provides an example of the process for using the function ordering optimization. Assume you have a Fortran program that consists of the following files: `file1.f90` and `file2.f90`. Additionally, assume you have created a directory for the profile data files called `c:\profdata` (on Windows) or `./profdata` (on Linux). You would enter commands similar to the following to generate and use function ordering for your application.

1. Compile your program using the `-prof-gen=srcpos` (Linux) or `/Qprof-gen:srcpos` (Windows) option. Use the `-prof-dir` (Linux) or `/Qprof-dir` (Windows) option to specify the directory location of the profile files. This step creates an instrumented executable.

Operating System	Example commands
Linux	`ifort -o myprog -prof-gen=srcpos -prof-dir ./profdata file1.f90 file2.f90`

Windows	`ifort /exe:myprog /Qprof-gen:srcpos /Qprof-dir c:\profdata file1.f90 file`

2. Run the instrumented program with one or more sets of input data. Change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

Operating System	Example commands
Linux	`./myprog`
Windows	`myprog.exe`

3. Copy all .dyn and .dpi files into the same directory. If needed, you can merge the data from one or more runs of the instrumented program by using the profmerge tool to produce the `pgopti.dpi` file.
4. Compile the application with the generated profile feedback by specifying the `-prof-func-order` (Linux) or `/Qprof-func-order` (Windows) option to request the function ordering as well as the `-prof-use` (Linux) or `/Qprof-use` (Windows) option to request feedback compilation. Again, use the `-prof-dir` (Linux) or `/Qprof-dir` (Windows) option to specify the location of the profile files.

Operating System	Example commands
Linux	`ifort -o myprog -prof-dir ./profdata file1.f90 file2.f90 -prof-func-order`
Windows	`ifort /exe:myprog /Qprof-dir c:\profdata file1.f90 file2.f90 /Qprof-func-`

Using Data Ordering

This section provides an example of the process for using the data order optimization. Assume you have a Fortran program that consists of the following files: `file1.f90` and `file2.f90`. Additionally, assume you have created a directory for the profile data files called `c:\profdata` (on Windows) or `./profdata` (on Linux). You would enter commands similar to the following to use data ordering for your application.

1. Compile your program using the `-prof-gen=globdata` (Linux) or `/Qprof-gen:globdata` (Windows) option. Use the `-prof-dir` (Linux) or `/Qprof-dir` (Windows) option to specify the directory location of the profile files. This step creates an instrumented executable.

Operating System	Example commands
Linux	`ifort -o myprog -prof-gen=globdata -prof-dir ./profdata file1.f90 file2.f90`
Windows	`ifort /exe:myprog /Qprof-gen:globdata /Qprof-dir c:\profdata file1.f90 file2.f90`

2. Run the instrumented program with one or more sets of input data. If you specified a location other than the current directory, change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

Operating System	Example commands
Linux	`./myprog`
Windows	`myprog.exe`

3. Copy all .dyn and .dpi files into the same directory. If needed, you can merge the data from one or more runs of the instrumented program by using the profmerge tool to produce the `pgopti.dpi` file.
4. Compile the application with the generated profile feedback by specifying the `-prof-data-order` (Linux) or `/Qprof-data-order` option to request the data ordering as well as the `-prof-use` (Linux) or `/Qprof-use` (Windows) option to request feedback compilation. Again, use the `-prof-dir` (Linux) or `/Qprof-dir` (Windows) option to specify the location of the profile files.

Operating System	Example commands
Linux	`ifort -o myprog -prof-dir ./profdata file1.f90 file2.f90 -prof-data-orde`
Windows	`ifort /exe:myprog Qprof-dir c:\profdata file1.f90 file2.f90 /Qprof-data-o use`

Comparison of Function Order Lists and IPO Code Layout

The Intel® compiler provides two methods of optimizing the layout of functions in the executable:

- Using a function order list
- Using the `/Qipo` (Windows) compiler option

Each method has its advantages. A function order list, created with proforder, lets you optimize the layout of non-static functions: that is, external and library functions whose names are exposed to the linker.

The compiler cannot affect the layout order for functions it does not compile, such as library functions. The function layout optimization is performed automatically when IPO is active.

Function Order List Effects

Function Type	IPO Code Layout	Function Ordering with proforder
Static	X	No effect
Extern	X	X
Library	No effect	X

Function Order List Usage Guidelines (Windows*)

Use the following guidelines to create a function order list:

- The order list only affects the order of non-static functions.
- You must compile with `/Gy` to enable function-level linking. (This option is active if you specify either option `/O1` or `/O2`.)

PGO API Support

API Support Overview

The Profile Information Generation Support (Profile IGS) lets you control the generation of profile information during the instrumented execution phase of profile-guided optimizations.

A set of functions and an environment variable comprise the Profile IGS. The remaining topics in this section describe the associated functions and environment variables.

The compiler sets a `define` for `_PGO_INSTRUMENT` when you compile with either `-prof-gen` (Linux* and Mac OS* X) or `/Qprof-gen` (Windows*). Without instrumentation, the Profile IGS functions cannot provide PGO API support.

Note

The Profile IGS functions are written in C. Fortran applications must call C/C++ functions.

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.

You can use the Profile IGS functions in your application by including a header file at the top of any source file where the functions may be used.

Example
`INCLUDE "pgouser.h"`

The Profile IGS Environment Variable

The environment variable for Profile IGS is `INTEL_PROF_DUMP_INTERVAL`. This environment variable may be used to initiate Interval Profile Dumping in an instrumented user application. See the recommended usage of `_PGOPTI_Set_Interval_Prof_Dump()` for more information.

PGO Environment Variables

The environment variables determine the directory in which to store dynamic information files, control the creation of one or multiple dyn files to collect profiling information, and determine whether to overwrite `pgopti.dpi`.

The environment variables are described in the table below.

Variable	Description
`INTEL_PROF_DUMP_CUMULATIVE`	When using interval profile dumping (initiated by `INTEL_PROF_DUMP_INTERVAL` or the function `_PGOPTI_Set_Interval_Prof_Dump`) during the execution of an instrumented user application, allows creation of a single .dyn file to contain profiling information instead of multiple .dyn files. If this environment variable is not set, executing an instrumented user application creates a new .dyn file for each interval. Setting this environment variable is useful for applications that do not terminate or those that terminate abnormally (bypass the normal exit code).
`INTEL_PROF_DUMP_INTERVAL`	Initiates interval profile dumping in an instrumented user application. This environment variable may be used to initiate Interval Profile Dumping in an instrumented application. See Interval Profile Dumping for more information.
`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	Deprecated. Please use INTEL_PROF_DUMP_INTERVAL instead.
PROF_NO_CLOBBER	Alters the feedback compilation phase slightly. By default, during the feedback compilation phase, the compiler merges data from all dynamic information files and creates a new pgopti.dpi file if the .dyn files are newer than an existing pgopti.dpi file. When this variable is set the compiler does not overwrite the existing pgopti.dpi file. Instead, the compiler issues a warning. You must remove the pgopti.dpi file if you want to use additional dynamic information files.

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

Dumping Profile Information

The _PGOPTI_Prof_Dump_All() function dumps the profile information collected by the instrumented application. The prototype of the function call is listed below.

Syntax

```
void _PGOPTI_Prof_Dump_All(void);
```

An older version of this function, _PGOPTI_Prof_Dump(), which will also dump profile information is still available; the older function operates much like _PGOPTI_Prof_Dump_All(), except on Linux when used in connection with shared libraries (.so) and _exit() to terminate a program. When _PGOPTI_Prof_Dump_All() is called before_exit() to terminate the program, the new function insures that a .dyn file is created for all shared libraries needing to create a .dyn file. Use _PGOPTI_Prof_Dump_All() on Linux to insure portability and correct functionality.

The profile information is generated in a .dyn file (generated in phase 2 of 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_All() should be called just once. It is also possible to use this function in conjunction with _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 All()
  input data = get input data()
end do
end program
```

Dumping Profile Data

This discussion provides an example of how to call the C PGO API routines from Fortran.

As part of the instrumented execution phase of PGO, the instrumented program writes profile data to the dynamic information file (.dyn file).

The profile information file is written after the instrumented program returns normally from `PROGRAM()` or calls the standard exit function.

Programs that do not terminate normally, can use the `_PGOPTI_Prof_Dump_All` function. During the instrumentation compilation, using the `-prof-gen` (Linux* and Mac OS* X) or `/Qprof-gen` (Windows*) option, you can add a call to this function to your program using a strategy similar to the one shown below:

Example

```
interface
  subroutine PGOPTI Prof Dump All()
!DEC$attributes c,alias:'PGOPTI Prof Dump All'::PGOPTI Prof Dump All
  end subroutine
  subroutine PGOPTI Prof Reset()
!DEC$attributes c,alias:'PGOPTI Prof Reset'::PGOPTI Prof Reset
  end subroutine
end interface
call PGOPTI_Prof_Dump_All()
```

Caution

You must remove the call or comment it out prior to the feedback compilation with `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows).

Interval Profile Dumping

The `_PGOPTI_Set_Interval_Prof_Dump()` function activates Interval Profile Dumping and sets the approximate frequency at which dumps occur. This function is used in non-terminating applications.

The prototype of the function call is listed below.

Syntax

```
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.

Setting the interval to zero or a negative number will disable interval profile dumping, and setting a very small value for the 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.

You can use the profmerge tool to merge the .dyn files.

Recommended usage

Call this function 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 `INTEL_PROF_DUMP_INTERVAL` to the desired interval value prior to starting the application.

Using interval profile dumping, you should be able to profile a non-terminating application with minimal changes to the application source code.

Resetting the Dynamic Profile Counters

The `_PGOPTI_Prof_Reset()` function resets the dynamic profile counters. The prototype of the function call is listed below.

Syntax

```
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 Dumping Profile Information.

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 the function call is listed below.

Syntax

```
void _PGOPTI_Prof_Dump_And_Reset(void);
```

This function is used in non-terminating applications and may be called more than once. Each call will dump the profile information to a new .dyn file.

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.

Using High-Level Optimization (HLO)

High-Level Optimizations (HLO) Overview

HLO exploits the properties of source code constructs (for example, loops and arrays) in applications developed in high-level programming languages. Within HLO, loop transformation techniques include:

- Loop Permutation or Interchange
- Loop Distribution
- Loop Fusion
- Loop Unrolling
- Data Prefetching
- Scalar Replacement
- Unroll and Jam
- Loop Blocking or Tiling
- Partial-Sum Optimization
- Loadpair Optimization
- Predicate Optimization
- Loop Versioning with Runtime Data-Dependence Check (IA-64 architecture only)
- Loop Versioning with Low Trip-Count Check
- Loop Reversal
- Profile-Guided Loop Unrolling
- Loop Peeling
- Data Transformation: Malloc Combining and Memset Combining
- Loop Rerolling
- Memset and Memcpy Recognition
- Statement Sinking for Creating Perfect Loopnests

While the default optimization level, `-O2` (Linux* OS and Mac OS* X) or `/O2` (Windows* OS) option, performs some high-level optimizations (for example, prefetching, complete unrolling, etc.), specifying `-O3` (Linux and Mac OS X) or `/O3` (Windows) provides the best chance for performing loop transformations to optimize memory accesses; the scope of optimizations enabled by these options is different for IA-32 architecture, Intel® 64, and IA-64 architectures.

Applications for the IA-32 and Intel® 64 architectures

In conjunction with the vectorization options, `-ax` and `-x` (Linux and Mac OS X) or `/Qax` and `/Qx` (Windows), the `-O3` (Linux and Mac OS X) or `/O3` (Windows) option causes the compiler to perform more aggressive data dependency analysis than the default `-O2` (Linux and Mac OS X) or `/O2` (Windows).

Compiler prefetching is disabled in favor of the prefetching support available in the processors.

Applications for the IA-32 and IA-64 architectures

The `-O3` (Linux and Mac OS X) or `/O3` (Windows) option enables the `-O2` (Linux and Mac OS X) or `/O2` (Windows) option and adds more aggressive optimizations (like loop transformations); `O3` optimizes for maximum speed, but may not improve performance for some programs.

Applications for the IA-64 architecture

The `-ivdep-parallel` (Linux) or `/Qivdep-parallel` (Windows) option implies there is no loop-carried dependency in the loop where an IVDEPdirective is specified. (This strategy is useful for sparse matrix applications.)

Tune applications for IA-64 architecture by following these general steps:

1. Compile your program with `-03` (Linux) or `/O3` (Windows) and `-ipo` (Linux) or `/Qipo` (Windows). Use profile guided optimization whenever possible.
2. Identify hot spots in your code.
3. Generate a high-level optimization report.
4. Check why loops are not software pipelined.
5. Make the changes indicated by the results of the previous steps.
6. Repeat these steps until you achieve the desired performance.

General Application Tuning

In general, you can use the following strategies to tune applications for multiple architectures:

- Use `!DEC$ ivdep` to tell the compiler there is no dependency. You may also need the `-ivdep-parallel` (Linux and Mac OS X) or `/Qivdep-parallel` (Windows) option to indicate there is no loop carried dependency.
- Use `!DEC$ swp` to enable software pipelining (useful for lop-sided control and unknown loop count).
- Use `!DEC$ loop count(n)` when needed.
- If cray pointers are used, use `-safe-cray-ptr` (Linux and Mac OS X) or `/Qsafe-cray-ptr` (Windows) to indicate there is no aliasing.
- Use `!DEC$ distribute point` to split large loops (normally, this is automatically done).
- Check that the prefetch distance is correct. Use `CDEC$ prefetch` to override the distance when it is needed.

Loop Unrolling
The benefits of loop unrolling are as follows:

- 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.
- For processors based on the IA-32 architectures, the processor 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 the processors, until they have a maximum of 16 iterations
- A potential limitation is that excessive unrolling, or unrolling of very large loops, can lead to increased code size.

The `-unroll[n]` (Linux* and Mac OS* X) or `/Qunroll:[n]` (Windows*) option controls how the Intel® compiler handles loop unrolling.

Refer to Applying Optimization Strategies for more information.

Linux and Mac OS X	Windows	Description
`-unrolln`	`/Qunroll:n`	Specifies the maximum number of times you want to unroll a loop. The following examples unrolls a loop four times: `ifort -unroll4 a.f90` (Linux and Mac OS X) `ifort /Qunroll:4 a.f90` (Windows) **Note**

The compilers for IA-64 architecture recognizes only *n* = 0; any other value is ignored.

Omitting a value for *n* lets the compiler decide whether to perform unrolling or not. This is the default; the compiler uses default heuristics or defines `n`.

Passing 0 as *n* disables loop unrolling; the following examples disables loop unrolling:

`ifort -unroll0 a.f90` (Linux and Mac OS X)

`ifort /Qunroll:0 a.f90` (Windows)

Loop Independence

Loop independence is important since loops that are independent can be parallelized. Independent loops can be parallelized in a number of ways, from the course-grained parallelism of OpenMP*, to fine-grained Instruction Level Parallelism (ILP) of vectorization and software pipelining.

Loops are considered independent when the computation of iteration Y of a loop can be done independently of the computation of iteration X. In other words, if iteration 1 of a loop can be computed and iteration 2 simultaneously could be computed without using any result from iteration 1, then the loops are independent.

Occasionally, you can determine if a loop is independent by comparing results from the output of the loop with results from the same loop written with a decrementing index counter.

For example, the loop shown in example 1 might be independent if the code in example 2 generates the same result.

Example

```
subroutine loop independent A (a,b,MAX)
  implicit none
  integer :: j, MAX, a(MAX), b(MAX)
  do j=0, MAX
    a(j) = b(j)
  end do
end subroutine loop independent A
subroutine loop independent B (a,b,MAX)
  implicit none
  integer :: j, MAX, a(MAX), b(MAX)
  do j=MAX, 0, -1
    a(j) = b(j)
  end do
end subroutine loop_independent_B
```

When loops are dependent, improving loop performance becomes much more difficult. Loops can be dependent in several, general ways.

- Flow Dependency
- Anti Dependency

- Output Dependency
- Reductions

The following sections illustrate the different loop dependencies.

Flow Dependency - Read After Write

Cross-iteration flow dependence is created when variables are written then read in different iterations, as shown in the following example:

Example

```
subroutine flow dependence (A,MAX)
  implicit none
  integer :: J,MAX
  real :: A(MAX)
  do j=1, MAX
    A(J)=A(J-1)
  end do
end subroutine flow_dependence
```

The above example is equivalent to the following lines for the first few iterations:

Sample iterations

```
A[1]=A[0];
A[2]=A[1];
```

Recurrence relations feed information forward from one iteration to the next.

Example

```
subroutine time stepping loops (a,b,MAX)
  implicit none
  integer :: J,MAX
  real :: a(MAX), b(MAX)
  do j=1, MAX
    a(j) = a(j-1) + b(j)
  end do
end subroutine time_stepping_loops
```

Most recurrences cannot be made fully parallel. Instead, look for a loop further out or further in to parallelize. You might be able to get more performance gains through unrolling.

Anti Dependency - Write After Read

Cross-iteration anti-dependence is created when variables are read then written in different iterations, as shown in the following example:

Example

```
subroutine anti dependence (A,MAX)
  implicit none
  integer :: J,MAX
  real :: a(MAX), b(MAX)
  do J=1, MAX
    A(J)=A(J+1)
  end do
end subroutine anti_dependence
```

The above example is equivalent to the following lines for the first few iterations:

Sample interations

```
A[1]=A[2];
A[2]=A[3];
```

Output Dependency - Write After Write

Cross-iteration output dependence is where variables are written then rewritten in a different iteration. The following example illustrates this type of dependency:

Example

```
subroutine output dependence (A,B,C,MAX)
  implicit none
  integer :: J,MAX
  real :: a(MAX), b(MAX), c(MAX)
  do J=1, MAX
    A(J)=B(J)
    A(J+1)=C(J)
  end do
end subroutine output_dependence
```

The above example is equivalent to the following lines for the first few iterations:

Sample interations

```
A[1]=B[1];
A[2]=C[1];
A[2]=B[2];
A[3]=C[2];
```

Reductions

The Intel® compiler can successfully vectorize or software pipeline (SWP) most loops containing reductions on simple math operators like multiplication (*), addition (+), subtraction (-), and division (/). Reductions collapse array data to scalar data by using associative operations:

Example

```
subroutine reduction (sum,c,MAX)
  implicit none
  integer :: j,MAX
  real :: sum, c(MAX)
  do J=0, MAX
    sum = sum + c(j)
  end do
end subroutine reduction
```

The compiler might occasionally misidentify a reduction and report flow-, anti-, output-dependencies or sometimes loop-carried memory-dependency-edges; in such cases, the compiler will not vectorize or SWP the loop.

Prefetching with Options

The goal of prefetch insertion optimization is to reduce cache misses by providing hints to the processor about when data should be loaded into the cache. The prefetch optimization is enabled or disabled by the `-opt-prefetch` (Linux* and Mac OS* X) or

`/Qopt-prefetch` (Windows*) compiler option. This option also allows you to specify the level of software prefetching.

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.

In addition to the `-opt-prefetch` (Linux and Mac OS X) or `/Qopt-prefetch` (Windows) option, an intrinsic subroutine `mm_prefetch` and compiler directive `prefetch` are also available.

The architecture affects the option behavior. For more information about the differences, see the following topic:

- `-opt-prefetch` compiler option

Additionally, you can refer to the hardware and software programming resources listed in Other Resources.

Optimization Support Features

Prefetching Support

Data prefetching refers to loading data from a relatively slow memory into a relatively fast cache before the data is needed by the application. Data prefetch behavior depends on the architecture:

- IA-64 architecture: The Intel® compiler generally issues prefetch instructions when you specify `-O1`, `-O2`, and `-O3` (Linux*) or `/O1`, `/O2`, and `/O3` (Windows*).
- IA-32 and Intel® 64 architectures: The processor identifies simple, regular data access patterns and performs a hardware prefetch. The compiler will only issue prefetch instructions for more complicated data access patterns where a hardware prefetch is not expected.

Issuing prefetches improves performance in most cases; however, there are cases where issuing prefetch instructions might slow application performance. Experiment with prefetching; it can be helpful to turn prefetching on or off with a compiler option while leaving all other optimizations unaffected to isolate a suspected prefetch performance issue. See Prefetching with Options for information on using compiler options for prefetching data.

There are two primary methods of issuing prefetch instructions. One is by using compiler directives and the other is by using compiler intrinsics.

PREFETCH and NOPREFETCH Directives

The PREFETCH and NOPREFETCH directives are supported by Itanium® processors only. These 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 the number of iterations ahead to prefetch the data and is determined by the compiler. This directive is supported with optimization levels of -O1 (Linux*) or /O1 (Windows*) or higher. Remember that -O2 or /O2 is the default optimization level.

Example

```
!DEC$ NOPREFETCH c
!DEC$ PREFETCH a
  do i = 1, m
    b(i) = a(c(i)) + 1
  enddo
```

The following example is for IA-64 architecture only:

Example

```
do j=1,lastrow-firstrow+1
i = rowstr(j)
iresidue = mod( rowstr(j+1)-i, 8 )
sum = 0.d0
!DEC$ NOPREFETCH a,p,colidx
do k=i,i+iresidue-1
  sum = sum +  a(k)*p(colidx(k))
enddo
!DEC$ NOPREFETCH colidx
!DEC$ PREFETCH a:1:40
!DEC$ 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
```

Intrinsics

Before inserting compiler intrinsics, experiment with all other supported compiler options and directives. Compiler intrinsics are less portable and less flexible than either a compiler option or compiler directives.

Directives enable compiler optimizations while intrinsics perform optimizations. As a result, programs with directives are more portable, because the compiler can adapt to different processors, while the programs with intrinsics may have to be rewritten/ported for different processors. This is because intrinsics are closer to assembly programming.

The compiler supports an intrinsic subroutine mm_prefetch. In contrast the way the prefetch directive enables a data prefetch from memory, the subroutine mm_prefetch prefetches data from the specified address on one memory cache line. The mm_prefetch subroutine is described in the *Intel® Fortran Language Reference.*

About Register Allocation

The Intel® Compiler for IA-32 and Intel® 64 architectures contains an advanced, region-based register allocator. Register allocation can be influenced using the `-opt-ra-region-strategy` (Linux* and Mac OS* X) and `/Qopt-ra-region-strategy` (Windows*) option.

The register allocation high-level strategy when compiling a routine is to partition the routine into regions, assign variables to registers or memory within each region, and resolve discrepancies at region boundaries. The overall quality of the allocation depends heavily on the region partitioning.

By default, the Intel Compiler selects the best region partitioning strategy, but the `-opt-ra-region-strategy` (Linux* and Mac OS X) and `/Qopt-ra-region-strategy` (Windows) option allows you to experiment with the other available allocation strategies, which might result in better performance in some cases. The option provides several different arguments that allow you to specify the following allocation strategies:

- routine = a region for each routine
- trace = a region for each trace
- loop = a region for each loop
- block = a region for each block
- default = the compiler selects best allocation strategy

See the `/Qopt-ra-region-strategy-opt-ra-region-strategy/Qopt-ra-region-strategy-opt-ra-region-strategy` compiler option for additional information.

The option can affect compile time. Register allocation is a relatively costly operation, and the time spent in register allocation tends to grow as the number of regions increases. Expect relative compile time to increase in the order listed, from shortest to longest:

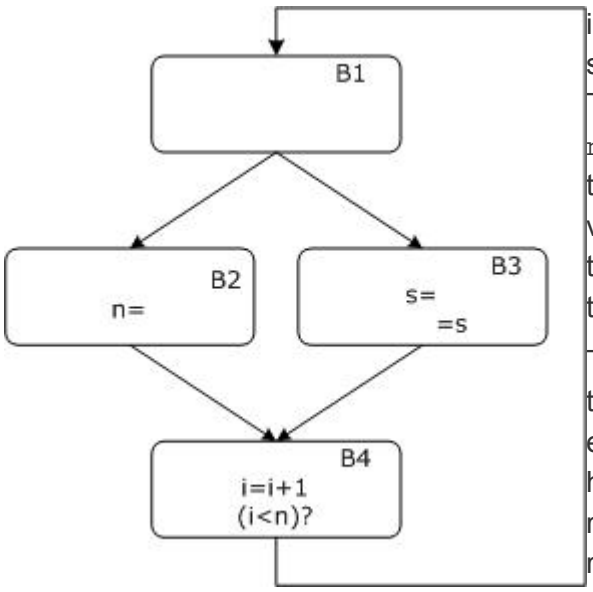
1. routine-based regions (shortest)
2. loop-based regions
3. trace-based regions
4. block-based regions (longest)

Trace-based regions tend to work very well when profile guided optimizations are enabled. The allocator is able to construct traces that accurately reflect the hot paths through the routine.

In the absence of profile information, loop-based regions tend to work well because the execution profile of a program tends to match its loop structure. In programs where the execution profile does not match the loop structure, routine- or block-based regions can produce better allocations.

Block-based regions provide maximum flexibility to the allocator and in many cases can produce the best allocation; however, the allocator is sometimes over-aggressive with block-based regions about allocating variables to registers; the behavior can lead to poor allocations in the IA-32 and Intel® 64 architectures where registers can be scarce resources.

Register Allocation Example Scenarios



Consider the following example, which illustrates a control flow that results from a simple if-then-else statement within a loop. There are 3 variables in this loop: `i`, `s`, and `n`. For this example, assume there are only two registers available to hold these three variables; one, or more, variable will need to be stored in memory for at least part of the loop.

The best choice depends on which path through the loop is more frequently executed. For example, if B1, B2, B4 is the hot path, keeping variables `i` and `n` in registers along that path and saving and restoring one of them in B3 to free up a register for `s` is the best strategy. That scenario avoids all memory accesses on the hot path. If B1, B3, B4 is the hot path, the best strategy is to keep variables `i` and `s` in registers and to store n in memory, since there are no assignments to `n` along the path. This strategy results in a single memory read on the hot path. If both paths are executed with equal frequency, the best strategy is to save and restore either `i` or `n` around B3 just like in the B1, B2, B4 case. That case avoids all memory accesses on one path and results in a single memory write and a single memory read on the other path.

The compiler generates two significantly different allocations for this example depending on the region strategy; the preferred result depends on the runtime behavior of the program, which may not be known at compile time.

With a routine- or a loop-based strategy, all four blocks will be allocated together. The compiler picks a variable to store in memory based on expected costs. In this example, the allocator will probably select variable `n`, resulting in a memory write in B2 and a memory read in B3.

With a trace-based strategy, the compiler uses estimates of execution frequency to select the most frequently executed path through the loop. When profile guided optimizations are enabled the estimates are based on the concrete information about the runtime behavior of the instrumented program. If the PGO information accurately reflects

typical application behavior, the compiler produces highly accurate traces. In other cases, the traces are not necessarily an accurate reflection of the hot paths through the code.

Suppose in this example that the compiler selects B1, B2, B4 path as the hot trace. The compiler will assign these three blocks to one region, and B3 will be in a separate region. There are only two variables in the larger region, so both may be kept in registers. In the region containing just B3 either `i` or `n` are stored in memory, and the compiler makes an arbitrary choice between the two variables. In a block-based strategy, each block is a region. In the B1, B2, B4 path there are sufficient registers to hold the two variables: `i` and `n`. The region containing B3 is treated just like the trace-based case; either `i` or `n` will be stored in memory.

Programming Guidelines

Understanding Run-time Performance

The information in this topic assumes that you are using a performance optimization methodology and have analyzed the application type you are optimizing.

After profiling your application to determine where best to spend your time, attempt to discover what optimizations and what limitations have been imposed by the compiler. Use the compiler reports to determine what to try next.

Depending on what you discover from the reports you may be able to help the compiler through options, directives, and slight code modifications to take advantage of key architectural features to achieve the best performance.

The compiler reports can describe what actions have been taken and what actions cannot be taken based on the assumptions made by the compiler. Experimenting with options and directives allows you to use an understanding of the assumptions and suggest a new optimization strategy or technique.

Helping the Compiler

You can help the compiler in some important ways:

- Read the appropriate reports to gain an understanding of what the compiler is doing for you and the assumptions the compiler has made with respect to your code.
- Use specific options, intrinsics, libraries, and directives to get the best performance from your application.

Use the Math Kernel Library (MKL) instead of user code, or calling F90 intrinsics instead of user code.

See Applying Optimization Strategies for other suggestions.

Memory Aliasing For IA-64 Architectures

Memory aliasing is the single largest issue affecting the optimizations in the Intel® compiler for IA-64 architecture based systems. Memory aliasing is writing to a given memory location with more than one pointer. The compiler is cautious to not optimize too aggressively in these cases; if the compiler optimizes too aggressively, unpredictable behavior can result (for example, incorrect results, abnormal termination, etc.).

Since the compiler usually optimizes on a module-by-module, function-by-function basis, the compiler does not have an overall perspective with respect to variable use for global variables or variables that are passed into a function; therefore, the compiler usually assumes that any pointers passed into a function are likely to be aliased. The compiler makes this assumption even for pointers you know are not aliased. This behavior means that perfectly safe loops do not get pipelined or vectorized, and performance suffers.

There are several ways to instruct the compiler that pointers are not aliased:

- Use a comprehensive compiler option, such as `-fno-alias` (Linux*) or `/Oa` (Windows*). These options instruct the compiler that no pointers in any module are aliased, placing the responsibility of program correctness directly with the developer.
- Use a less comprehensive option, like `-fno-fnalias` (Linux) or `/Ow` (Windows). These options instruct the compiler that no pointers passed through function arguments are aliased.
  Function arguments are a common example of potential aliasing that you can clarify for the compiler. You may know that the arguments passed to a function do not alias, but the compiler is forced to assume so. Using these options tells the compiler it is now safe to assume that these function arguments are not aliased. This option is still a somewhat bold statement to make, as it affects all functions in the module(s) compiled with the `-fno-nalias` (Linux) or `/Ow` (Windows) option.
- Use the `IDVEP` directive. Alternatively, you might use a directive that applies to a specified loop in a function. This is more precise than specifying an entire function. The directive asserts that, for a given loop, there are no vector dependencies. Essentially, this is the same as saying that no pointers are aliasing in a given loop.

Non-Unit Stride Memory Access

Another issue that can have considerable impact on performance is accessing memory in a non-Unit Stride fashion. This means that as your inner loop increments consecutively, you access memory from non adjacent locations. For example, consider the following matrix multiplication code:

Example

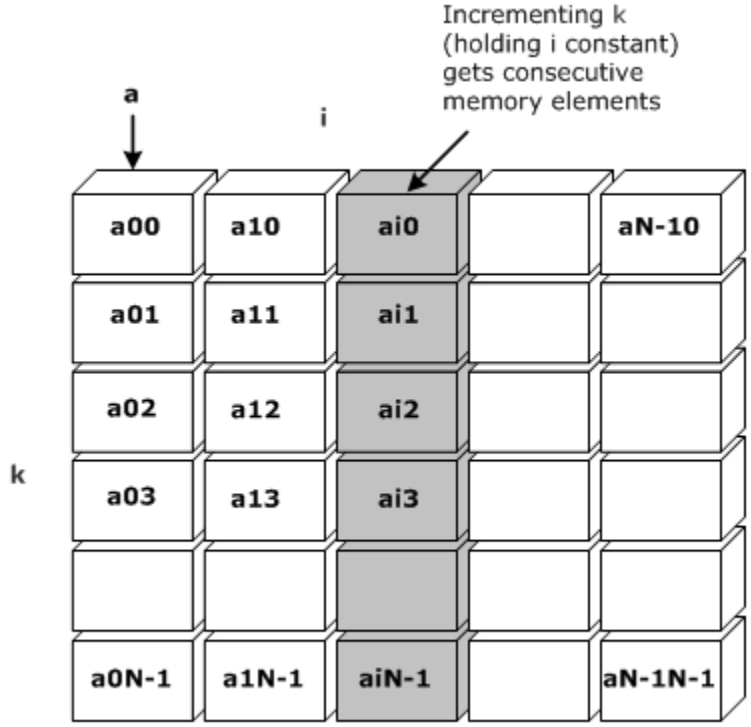
```
!Non-Unit Stride Memory Access
subroutine non_unit_stride_memory_access(a,b,c, NUM)
```

```
  implicit none
  integer :: i,j,k,NUM
  real :: a(NUM,NUM), b(NUM,NUM), c(NUM,NUM)
! loop before loop interchange
  do i=1,NUM
    do j=1,NUM
      do k=1,NUM
        c(j,i) = c(j,i) + a(j,k) * b(k,i)
      end do
    end do
  end do
end subroutine non_unit_stride_memory_access
```

Notice that `c[i][j]`, and `a[i][k]` both access consecutive memory locations when the inner-most loops associated with the array are incremented. The `b` array however, with its loops with indexes k and j, does not access Memory Unit Stride. When the loop reads `b[k=0][j=0]` and then the k loop increments by one to `b[k=1][j=0]`, the loop has skipped over NUM memory locations having skipped `b[k][1], b[k][2] .. b[k][NUM]`.

Loop transformation (sometimes called loop interchange) helps to address this problem. While the compiler is capable of doing loop interchange automatically, it does not always recognize the opportunity.

The memory access pattern for the example code listed above is illustrated in the following figure:

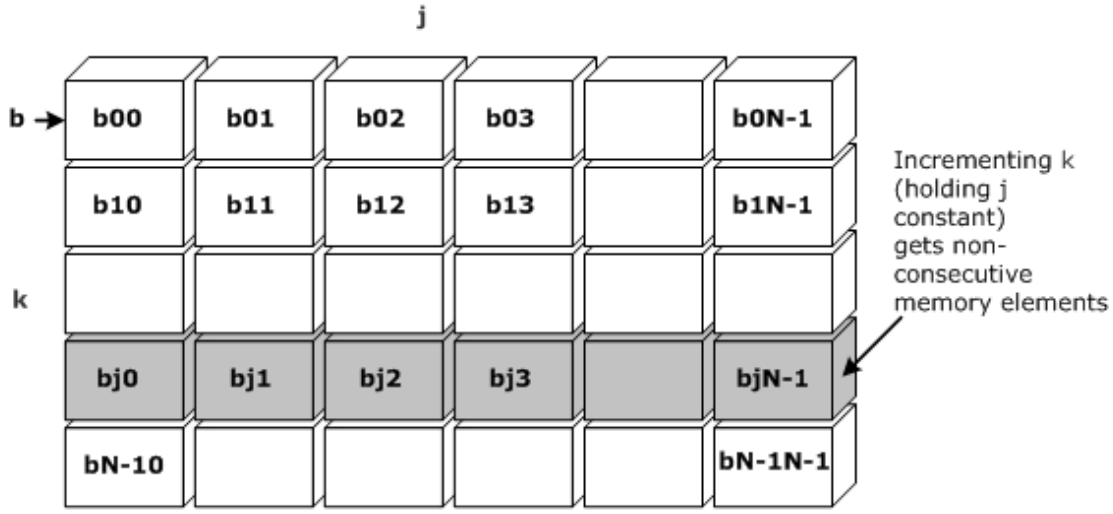


Assume you modify the example code listed above by making the following changes to introduce loop interchange:

Example

```
subroutine unit stride memory access(a,b,c, NUM)
  implicit none
  integer :: i,j,k,NUM
  real :: a(NUM,NUM), b(NUM,NUM), c(NUM,NUM)
! loop after interchange
  do i=1,NUM
    do k=1,NUM
      do j=1,NUM
        c(j,i) = c(j,i) + a(j,k) * b(k,i)
      end do
    end do
  end do
end subroutine unit_stride_memory_access
```

After the loop interchange the memory access pattern might look the following figure:



Understanding Data Alignment

Aligning data on boundaries can help performance. The Intel® compiler attempts to align data on boundaries for you. However, as in all areas of optimization, coding practices can either help or hinder the compiler and can lead to performance problems.

Always attempt to optimize using compiler options first.

To avoid performance problems you should keep the following guidelines in mind, which are separated by architecture:

IA-32, Intel® 64, and IA-64 architectures:

- Do not access or create data at large intervals that are separated by exactly 2^n (for example, 1 KB, 2 KB, 4 KB, 16 KB, 32 KB, 64 KB, 128 KB, 512 KB, 1 MB, 2 MB, 4 MB, 8 MB, etc.).
- Align data so that memory accesses does not cross cache lines (for example, 32 bytes, 64 bytes, 128 bytes).
- Use Application Binary Interface (ABI) for the Itanium® compiler to insure that ITP pointers are 16-byte aligned.

IA-32 and Intel® 64 architectures:

- Align data to correspond to the SIMD or Streaming SIMD Extension registers sizes.

IA-64 architecture:

- Avoid using packed structures.
- Avoid casting pointers of small data elements to pointers of large data elements.
- Do computations on unpacked data, then repack data if necessary, to correctly output the data.

In general, keeping data in cache has a better performance impact than keeping the data aligned. Try to use techniques that conform to the rules listed above.

See Setting Data Type and Alignment for more detailed information on aligning data.

Timing Your Application

You can start collecting information about your application performance by timing your application. More sophisticated and helpful data can be collected by using performance analysis tools.

Considerations on Timing Your Application

One of the performance indicators is your application timing. 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 system (processor 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. Certain overhead functions like loading libraries might influence short timings considerably.
- If your program displays a lot of text, consider redirecting the output from the program. Redirecting output from the program will change the times reported because of reduced screen I/O.
  Timings that show a large amount of system time may indicate a lot of time spent doing I/O, which might be worth investigating.

- 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.

Methods of Timing Your Application

To perform application timings, use the VTune™ Performance Analyzer or a version of the `TIME` command in a `.BAT` file (or the function timing profiling option). You might consider modifying the program to call routines within the program to measure execution time (possibly using conditionally compiled lines).

For example:

- Intel Fortran intrinsic procedures, such as `SECNDS`, `CPU_TIME`, `SYSTEM_CLOCK`, `TIME`, and `DATE_AND_TIME`.
- Portability library routines, such as `DCLOCK`, `ETIME`, `SECNDS`, or `TIME`.

Whenever possible, perform detailed performance analysis on a system that closely resembles the system(s) that will be used for actual application use.

Sample Timing

The following program template could be run by a `.BAT` file that executes the `TIME` command both before and after execution, to provide an approximate wall-clock time for the execution of the entire program. The Fortran intrinsic `CPU_TIME` can be used at selected points in the program to collect the CPU time between the start and end of the task to be timed.

Example

```
REAL time begin, time end
  ...
  CALL CPU TIME ( time begin )
  !
  !task to be timed
  !
  CALL CPU TIME ( time end )
  PRINT *, 'Time of operation was ', &
time_end - time_begin, ' seconds'
```

Considerations for Linux*

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

Bourne* shell example

```
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:

Bourne* shell example

```
$ 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:

C shell l example

```
% 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:

bash shell l example

```
[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 intrinsic procedures, such as `SECNDS`, `DCLOCK`, `CPU_TIME`, `SYSTEM_CLOCK`, `TIME`, and `DATE_AND_TIME`.

Applying Optimization Strategies

The compiler may or may not apply the following optimizations to your loop: Interchange, Unrolling, Cache Blocking, and LoadPair. These transformations are discussed in the following sections, including how to transform loops manually and how to control them with directives or internal options.

Loop Interchange

Loop Interchange is a nested loop transformation applied by High-level Optimization (HLO) that swaps the order of execution of two nested loops. Typically, the transformation is performed to provide sequential Unit Stride access to array elements used inside the loop to improve cache locality. The compiler `-O3` (Linux* and Mac OS* X) or `/O3` (Windows*) optimization looks for opportunities to apply loop interchange for you.

The following is an example of a loop interchange

Example

```
subroutine loop interchange(a,b,c, NUM)
implicit none
integer :: i,j,k,NUM
real :: a(NUM,NUM), b(NUM,NUM), c(NUM,NUM)
! loop before loop interchange
do i=1,NUM
  do j=1,NUM
    do k=1,NUM
        c(j,i) = c(j,i) + a(j,k) * b(k,i)
    end do
  end do
end do
! loop after interchange
do i=1,NUM
  do k=1,NUM
    do j=1,NUM
        c(j,i) = c(j,i) + a(j,k) * b(k,i)
    end do
  end do
end do
end subroutine loop_interchange
```

Unrolling

Loop unrolling is a loop transformation generally used by HLO that can take better advantage of Instruction-Level Parallelism (ILP), keeping as many functional units busy doing useful work as possible during single loop iteration. In loop unrolling, you add more work to the inside of the loop while doing fewer loop iterations in exchange.

Example

```
subroutine loop unroll before(a,b,c,N,M)
implicit none
integer :: i,j,N,M
real :: a(N,M), b(N,M), c(N,M)
N=1025
M=5
do i=1,N
  do j=1,M
    a(j,i) = b(j,i) + c(j,i)
```

```
  end do
end do
end subroutine loop_unroll_before
```

Example

```
subroutine loop unroll after(a,b,c,N,M)
implicit none
integer :: i,j,K,N,M
real :: a(N,M), b(N,M), c(N,M)
N=1025
M=5
K=MOD(N,4)  !K= N MOD 4
! main part of loop
do i=1,N-K,4
  do j=1,M
    a(j,i) = b(j,i) + c(j,i)
    a(j,i+1) = b(j,i+1) + c(j,i+1)
    a(j,i+2) = b(j,i+2) + c(j,i+2)
    a(j,i+3) = b(j,i+3) + c(j,i+3)
  end do
end do
! post conditioning part of loop
do i= N-K+2, N, 4
  do j=1,M
    a(j,i) = b(j,i) + c(j,i)
  end do
end do
end subroutine loop_unroll_after
```

Post conditioning is preferred over pre-conditioning because post conditioning will preserve the data alignment and avoid the cost of memory alignment access penalties.

Cache Blocking

Cache blocking involves structuring data blocks so that they conveniently fit into a portion of the L1 or L2 cache. By controlling data cache locality, an application can minimize performance delays due to memory bus access. The application controls the behavior by dividing a large array into smaller blocks of memory so a thread can make repeated accesses to the data while the data is still in cache.

For example, image processing and video applications are well suited to cache blocking techniques because an image can be processed on smaller portions of the total image or video frame. Compilers often use the same technique, by grouping related blocks of instructions close together so they execute from the L2 cache.

The effectiveness of the cache blocking technique depends on data block size, processor cache size, and the number of times the data is reused. Cache sizes vary based on processor. An application can detect the data cache size using the `CPUID` instruction and dynamically adjust cache blocking tile sizes to maximize performance. As a general rule, cache block sizes should target approximately one-half to three-quarters the size of the physical cache. For systems that are Hyper-Threading Technology (HT Technology) enabled target one-quarter to one-half the physical cache size. Designing for Hyper-Threading Technology

Cache blocking is applied in HLO and is used on large arrays where the arrays cannot all fit into cache simultaneously. This method is one way of pulling a subset of data into cache (in a small region), and using this cached data as effectively as possible before the data is replaced by new data from memory.

Example

```
subroutine cache blocking before(a,b,N)
  implicit none
  integer :: i,j,k,N
  real :: a(N,N,N), b(N,N,N), c(N,N,N)
  N=1000
  do i = 1, N
    do j = 1, N
      do k = 1, N
        a(i,j,k) = a(i,j,k)  + b(i,j,k)
      end do
    end do
  end do
end subroutine cache blocking before

subroutine cache blocking after(a,b,N)
  implicit none
  integer :: i,j,k,u,v,N
  real :: a(N,N,N), b(N,N,N), c(N,N,N)
  N=1000
  do v = 1, N, 20
    do u = 1, N, 20
      do k = v, v+19
        do j = u, u+19
          do i = 1, N
            a(i,j,k) = a(i,j,k)  + b(i,j,k)
          end do
        end do
      end do
    end do
  end do
end subroutine cache_blocking_after
```

The cache block size is set to 20. The goal is to read in a block of cache, do every bit of computing we can with the data in this cache, then load a new block of data into cache. There are 20 elements of `A` and 20 elements of `B` in cache at the same time and you should do as much work with this data as you can before you increment to the next cache block.

Blocking factors will be different for different architectures. Determine the blocking factors experimentally. For example, different blocking factors would be required for single precision versus double precision. Typically, the overall impact to performance can be significant.

Loop Distribution

Loop distribution is a high-level loop transformation that splits a large loop into two smaller loops. It can be useful in cases where optimizations like software-pipelining (SWP) or vectorization cannot take place due to excessive register usage. By splitting a loop into smaller segments, it may be possible to get each smaller loop or at least one of the smaller loops to SWP or vectorize. An example is as follows:

Example

```
subroutine loop distribution before(a,b,c,x,y,z,N)
  implicit none
  integer :: i,N
  real :: a(N), b(N), c(N), x(N), y(N), z(N)
  N=1024
  do i = 1, N
    a(i) = a(i)  + i
    b(i) = b(i)  + i
    c(i) = c(i)  + i
    x(i) = x(i)  + i
    y(i) = y(i)  + i
    z(i) = z(i)  + i
  end do
end subroutine loop distribution before

subroutine loop distribution after(a,b,c,x,y,z,N)
  implicit none
  integer :: i,N
  real :: a(N), b(N), c(N), x(N), y(N), z(N)
  N=1024
  do i = 1, N
    a(i) = a(i)  + i
    b(i) = b(i)  + i
    c(i) = c(i)  + i
  end do
  do i = 1, N
    x(i) = x(i)  + i
    y(i) = y(i)  + i
    z(i) = z(i)  + i
  end do
end subroutine loop_distribution_after
```

There are directives to suggest distributing loops to the compiler as follows:

Example

```
!DEC$ distribute point
```

Placed outside a loop, the compiler will attempt to distribute the loop based on an internal heuristic. The following is an example of using the pragma outside the loop:

Example

```
subroutine loop distribution pragma1(a,b,c,x,y,z,N)
  implicit none
  integer :: i,N
  real :: a(N), b(N), c(N), x(N), y(N), z(N)
  N=1024
!DEC$ distribute point
  do i = 1, N
    a(i) = a(i)  + i
    b(i) = b(i)  + i
    c(i) = c(i)  + i
    x(i) = x(i)  + i
    y(i) = y(i)  + i
    z(i) = z(i)  + i
  end do
end subroutine loop_distribution_pragma1
```

Placed within a loop, the compiler will attempt to distribute the loop at that point. All loop-carried dependencies will be ignored. The following example uses the directive within a loop to precisely indicate where the split should take place:

Example

```
subroutine loop distribution pragma2(a,b,c,x,y,z,N)
  implicit none
```

```
  integer :: i,N
  real :: a(N), b(N), c(N), x(N), y(N), z(N)
  N=1024
  do i = 1, N
    a(i) = a(i)  + i
    b(i) = b(i)  + i
    c(i) = c(i)  + i
!DEC$  distribute point
    x(i) = x(i)  + i
    y(i) = y(i)  + i
    z(i) = z(i)  + i
  end do
end subroutine loop_distribution_pragma2
```

Load Pair (Itanium® Compiler)

Load pairs (ldfp) are instructions that load two contiguous single or double precision values from memory in one move. Load pairs can significantly improve performance.

Manual Loop Transformations

There might be cases where these manual transformations are called acceptable or even preferred. As a general rule, you should let the compiler transform loops for you. Manually transform loops as a last resort; use this strategy only in cases where you are attempting to gain performance increases.

Manual loop transformations have many disadvantages, which include the following:

- Application code becomes harder to maintain over time.
- New compiler features can cause you to lose any optimization you gain by manually transforming the loop.
- Architectural requirements might restrict your code to a specific architecture unintentionally.

The HLO report can give you an idea of what loop transformations have been applied by the compiler.

Experimentation is a critical component in manually transforming loops. You might try to apply a loop transformation that the compiler ignored. Sometimes, it is beneficial to apply a manual loop transformation that the compiler has already applied with `-O3` (Linux) or `/O3` (Windows).

Optimizing the Compilation Process

Symbol Visibility Attribute Options (Linux* and Mac OS* X)

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.

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 the associated 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 and described in the following topic:

- `-fvisibility` compiler option


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`.


Note

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, you can override this default.

Specifying Symbol Visibility Explicitly

The Intel® compiler has visibility attribute options that provide command-line control of the visibility attributes in addition to 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.

Example

```
-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).

Specifying Visibility without the Symbol File

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

Example

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

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

Examples

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