IBM XL Fortran for Blue Gene/Q, V14.1

Optimization and Programming Guide

Version 14.1

SC14-7370-00

IBM XL Fortran for Blue Gene/Q, V14.1

Optimization and Programming Guide

Version 14.1

SC14-7370-00

Note

Before using this information and the product it supports, read the information in “Notices” on page 311.

First edition

This edition applies to IBM XL Fortran for Blue Gene/Q, V14.1 (Program 5799-AH1) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Contents

About this information

This information is part of the IBM® XL Fortran for Blue Gene®/Q, V14.1 information suite. It provides both reference information and practical tips for using XL Fortran's optimization and tuning capabilities to maximize application performance, as well as expanding on programming concepts such as I/O and interlanguage calls.

Who should read this information

This information is for anyone who wants to exploit the XL Fortran compiler's capabilities for optimizing and tuning Fortran programs. Readers should be familiar with their Blue Gene/Q operating system and have extensive Fortran programming experience with complex applications. However, users new to XL Fortran can still use this information to help them understand how the compiler's features can be used for effective program optimization.

How to use this information

This guide focuses on specific programming and compilation techniques that can maximize XL Fortran application performance. It covers optimization and tuning strategies, recommended programming practices and compilation procedures, debugging, and information on using XL Fortran advanced language features. This guide also contains cross-references to relevant topics of other reference guides in the XL Fortran information suite.

Topics not described in this information are available as indicated in the following:

- Installation, system requirements, last-minute updates: see the *XL Fortran Installation Guide* and product README.
- Overview of XL Fortran features: see the *Getting Started with XL Fortran*.
- Syntax, semantics, and implementation of the XL Fortran programming language: see the *XL Fortran Language Reference*.
- Compiler setup, compiling and running programs, compiler options, diagnostics: see the *XL Fortran Compiler Reference*.
- Operating system commands related to the use of the compiler: consult your Linux-specific distribution's man page help and information.

How this information is organized

This guide includes the following topics:

- Chapter 1, "Optimizing your applications," on page 1 provides an overview of the optimization process.
- Chapter 2, "Tuning XL compiler applications," on page 31 discusses the compiler options available for optimizing and tuning code.
- Chapter 3, "Advanced optimization concepts," on page 41, Chapter 4, "Managing code size," on page 45, and "Debugging optimized code" on page 20 discuss advanced techniques like optimizing loops and inlining code, and debug considerations for optimized code.

- The following sections contain information on how to write optimization friendly, portable XL Fortran code, that is interoperable with other languages. Also included is a description of XL Fortran's OpenMP and SMP support with guidelines for writing parallel code.
  - Chapter 5, "Compiler-friendly programming techniques," on page 51
  - Chapter 6, "High performance libraries," on page 55
  - Chapter 7, "Parallel programming with XL Fortran," on page 69
  - Chapter 8, "Interlanguage calls," on page 253
- The following sections contain information about XL Fortran and its implementation that can be useful for new and experienced users alike, as well as those who want to move their existing Fortran applications to the XL Fortran compiler:
  - Chapter 9, "Implementation details of XL Fortran Input/Output (I/O)," on page 273
  - Chapter 10, "Implementation details of XL Fortran floating-point processing," on page 285
  - Chapter 11, "Porting programs to XL Fortran," on page 301

Conventions

Typographical conventions

The following table shows the typographical conventions used in the IBM XL Fortran for Blue Gene/Q, V14.1 information.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options, and directives.	The compiler provides basic invocation commands, **bgxlf**, along with several other compiler invocation commands to support various Fortran language levels and compilation environments.
italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the *size* parameter if you return more than the *size* requested.
<u>underlining</u>	The default setting of a parameter of a compiler option or directive.	nomaf | <u>maf</u>
`monospace`	Programming keywords and library functions, compiler builtins, examples of program code, command strings, or user-defined names.	To compile and optimize myprogram.f, enter: `bgxlf myprogram.f -O3`.
UPPERCASE bold	Fortran programming keywords, statements, directives, and intrinsic procedures. Uppercase letters may also be used to indicate the minimum number of characters required to invoke a compiler option/suboption.	The **ASSERT** directive applies only to the **DO** loop immediately following the directive, and not to any nested **DO** loops.

Qualifying elements (icons and bracket separators)

In descriptions of language elements, this information uses icons and marked bracket separators to delineate the Fortran language standard text as follows:

Table 2. Qualifying elements

Icon	Bracket separator text	Meaning
▶ F2008 F2008 ◀	N/A	The text describes an IBM XL Fortran implementation of the Fortran 2008 standard.
▶ F2003 F2003 ◀	Fortran 2003 begins / ends	The text describes an IBM XL Fortran implementation of the Fortran 2003 standard, and it applies to all later standards.
▶ IBM IBM ◀	IBM extension begins / ends	The text describes a feature that is an IBM XL Fortran extension to the standard language specifications.

Note: If the information is marked with a Fortran language standard icon or bracket separators, it applies to this specific Fortran language standard and all later ones. If it is not marked, it applies to all Fortran language standards.

Syntax diagrams

Throughout this information, diagrams illustrate XL Fortran syntax. This section will help you to interpret and use those diagrams.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ▶▶── symbol indicates the beginning of a command, directive, or statement.

  The ──▶ symbol indicates that the command, directive, or statement syntax is continued on the next line.

  The ▶── symbol indicates that a command, directive, or statement is continued from the previous line.

  The ──▶◀ symbol indicates the end of a command, directive, or statement.

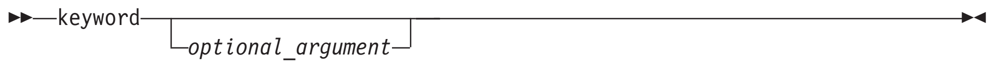
  Fragments, which are diagrams of syntactical units other than complete commands, directives, or statements, start with the |── symbol and end with the ──| symbol.

  IBM XL Fortran extensions are marked by a number in the syntax diagram with an explanatory note immediately following the diagram.

  Program units, procedures, constructs, interface blocks and derived-type definitions consist of several individual statements. For such items, a box encloses the syntax representation, and individual syntax diagrams show the required order for the equivalent Fortran statements.
- Required items are shown on the horizontal line (the main path):

```
▶▶──keyword──required_argument──────────────────────────────────────▶◀
```

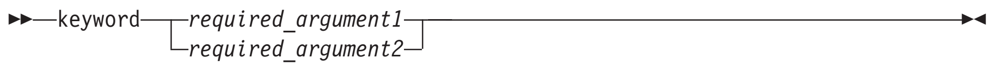
- Optional items are shown below the main path:



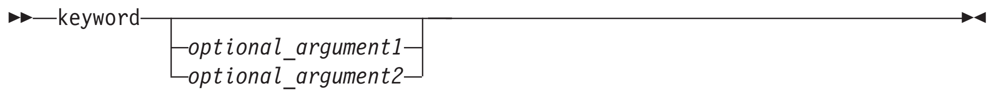
Note: Optional items (not in syntax diagrams) are enclosed by square brackets ([and]). For example, `[UNIT=]u`

- If you can choose from two or more items, they are shown vertically, in a stack.

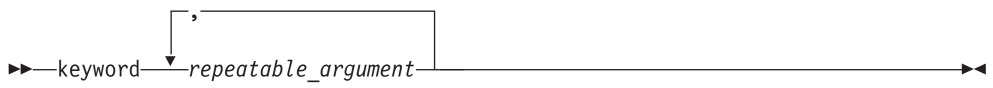
  If you *must* choose one of the items, one item of the stack is shown on the main path.



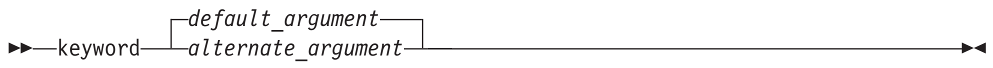
  If choosing one of the items is optional, the entire stack is shown below the main path.



- An arrow returning to the left above the main line (a repeat arrow) indicates that you can make more than one choice from the stacked items or repeat an item. The separator character, if it is other than a blank, is also indicated:



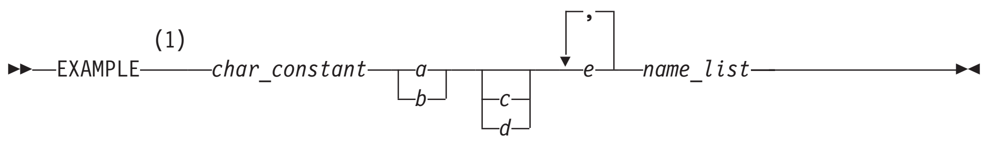
- The item that is the default is shown above the main path.



- Keywords are shown in nonitalic letters and should be entered exactly as shown.
- Variables are shown in italicized lowercase letters. They represent user-supplied names or values. If a variable or user-specified name ends in *_list*, you can provide a list of these terms separated by commas.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Sample syntax diagram

The following is an example of a syntax diagram with an interpretation:



Notes:

1 IBM extension

Interpret the diagram as follows:

- Enter the keyword EXAMPLE.
- EXAMPLE is an IBM extension.
- Enter a value for *char_constant*.
- Enter a value for *a* or *b*, but not for both.
- Optionally, enter a value for *c* or *d*.
- Enter at least one value for *e*. If you enter more than one value, you must put a comma between each.
- Enter the value of at least one *name* for *name_list*. If you enter more than one value, you must put a comma between each. (The *_list* syntax is equivalent to the previous syntax for *e*.)

How to read syntax statements

Syntax statements are read from left to right:

- Individual required arguments are shown with no special notation.
- When you must make a choice between a set of alternatives, they are enclosed by { and } symbols.
- Optional arguments are enclosed by [and] symbols.
- When you can select from a group of choices, they are separated by | characters.
- Arguments that you can repeat are followed by ellipses (...).

Example of a syntax statement

```
EXAMPLE char_constant {a|b}[c|d]e[,e]... name_list{name_list}...
```

The following list explains the syntax statement:

- Enter the keyword EXAMPLE.
- Enter a value for *char_constant*.
- Enter a value for *a* or *b*, but not for both.
- Optionally, enter a value for *c* or *d*.
- Enter at least one value for *e*. If you enter more than one value, you must put a comma between each.
- Optionally, enter the value of at least one *name* for *name_list*. If you enter more than one value, you must put a comma between each *name*.

Note: The same example is used in both the syntax-statement and syntax-diagram representations.

Examples in this information

The examples in this information, except where otherwise noted, are coded in a simple style that does not try to conserve storage, check for errors, achieve fast performance, or demonstrate all possible methods to achieve a specific result.

The examples for installation information are labelled as either *Example* or *Basic example*. *Basic examples* are intended to document a procedure as it would be performed during a basic, or default, installation; these need little or no modification.

Notes on the terminology used

Some of the terminology in this information is shortened as follows:

- The term *free source form format* often appears as *free source form*.
- The term *fixed source form format* often appears as *fixed source form*.
- The term *XL Fortran* often appears as *XLF*.

Related information

The following sections provide related information for XL Fortran:

IBM XL Fortran information

XL Fortran provides product information in the following formats:

- README files

  README files contain late-breaking information, including changes and corrections to the product information. README files are located by default in the XL Fortran directory and in the root directory of the installation CD.
- Installable man pages

  Man pages are provided for the compiler invocations and all command-line utilities provided with the product. Instructions for installing and accessing the man pages are provided in the *IBM XL Fortran for Blue Gene/Q, V14.1 Installation Guide*.
- Information center

  The information center of searchable HTML files can be launched on a network and accessed remotely or locally. Instructions for installing and accessing the online information center are provided in the *IBM XL Fortran for Blue Gene/Q, V14.1 Installation Guide*.

  The information center of searchable HTML files is viewable on the web at http://pic.dhe.ibm.com/infocenter/compbg/v121v141/index.jsp.
- PDF documents

  PDF documents are located by default in the /opt/ibmcmp/xlf/bg/14.1/doc/en_US/pdf/ directory. The PDF files are also available on the web at http://www.ibm.com/software/awdtools/fortran/xlfortran/features/bg/library/.

  The following files comprise the full set of XL Fortran product information:

Table 3. XL Fortran PDF files

Document title	PDF file name	Description
IBM XL Fortran for Blue Gene/Q, V14.1 Installation Guide, GC14-7367-00	install.pdf	Contains information for installing XL Fortran and configuring your environment for basic compilation and program execution.
Getting Started with IBM XL Fortran for Blue Gene/Q, V14.1, GC14-7366-00	getstart.pdf	Contains an introduction to the XL Fortran product, with information on setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL Fortran for Blue Gene/Q, V14.1 Compiler Reference, GC14-7368-00	compiler.pdf	Contains information about the various compiler options and environment variables.
IBM XL Fortran for Blue Gene/Q, V14.1 Language Reference, GC14-7369-00	langref.pdf	Contains information about the Fortran programming language as supported by IBM, including language extensions for portability and conformance to nonproprietary standards, compiler directives and intrinsic procedures.
IBM XL Fortran for Blue Gene/Q, V14.1 Optimization and Programming Guide, SC14-7370-00	proguide.pdf	Contains information on advanced programming topics, such as application porting, interlanguage calls, floating-point operations, input/output, application optimization and parallelization, and the XL Fortran high-performance libraries.

To read a PDF file, use the Adobe Reader. If you do not have the Adobe Reader, you can download it (subject to license terms) from the Adobe website at http://www.adobe.com.

More information related to XL Fortran including IBM Redbooks® publications, white papers, tutorials, and other articles, is available on the web at:

http://www.ibm.com/software/awdtools/fortran/xlfortran/features/bg/library/

Standards and specifications

XL Fortran is designed to support the following standards and specifications. You can refer to these standards for precise definitions of some of the features found in this information.

- *American National Standard Programming Language FORTRAN, ANSI X3.9-1978*.
- *American National Standard Programming Language Fortran 90, ANSI X3.198-1992*.
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*.
- *Federal (USA) Information Processing Standards Publication Fortran, FIPS PUB 69-1*.
- *Information technology - Programming languages - Fortran, ISO/IEC 1539-1:1991 (E)*. (This information uses its informal name, Fortran 90.)
- *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:1997*. (This information uses its informal name, Fortran 95.)
- *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:2004*. (This information uses its informal name, Fortran 2003.)
- *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:2010*. (This information uses its informal name, Fortran 2008.)

- *Military Standard Fortran DOD Supplement to ANSI X3.9-1978, MIL-STD-1753* (United States of America, Department of Defense standard). Note that XL Fortran supports only those extensions documented in this standard that have also been subsequently incorporated into the Fortran 90 standard.
- *OpenMP Application Program Interface Version 3.1*, available at http://www.openmp.org

Other IBM information

- *Blue Gene/Q Hardware Overview and Installation Planning, SG24-7872*, available at http://www.redbooks.ibm.com/redpieces/abstracts/sg247872.html?Open
- *Blue Gene/Q Hardware Installation and Maintenance Guide, SG24-7974*, available at http://www.redbooks.ibm.com/redpieces/abstracts/sg247974.html?Open
- *Blue Gene/Q High Availability Service Node, REDP-4657*, available at http://www.redbooks.ibm.com/redpieces/abstracts/redp4657.html?Open
- *Blue Gene/Q System Administration, SG24-7869*, available at http://www.redbooks.ibm.com/redpieces/abstracts/sg247869.html?Open
- *Blue Gene/Q Application Development, SG24-7948*, available at http://www.redbooks.ibm.com/redpieces/abstracts/sg247948.html?Open
- *Blue Gene/Q Code Development and Tools Interface, REDP-4659*, available at http://www.redbooks.ibm.com/redpieces/abstracts/redp4659.html?Open

Technical support

Additional technical support is available from the XL Fortran Support page at http://www.ibm.com/software/awdtools/fortran/xlfortran/features/bg/support/. This page provides a portal with search capabilities to a large selection of Technotes and other support information.

If you cannot find what you need, you can send email to compinfo@ca.ibm.com.

For the latest information about XL Fortran, visit the product information site at http://www.ibm.com/software/awdtools/fortran/xlfortran/features/bg/.

How to send your comments

Your feedback is important in helping to provide accurate and high-quality information. If you have any comments about this information or any other XL Fortran information, send your comments by email to compinfo@ca.ibm.com.

Be sure to include the name of the information, the part number of the information, the version of XL Fortran, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

Chapter 1. Optimizing your applications

The XL compilers enable development of high performance applications by offering a comprehensive set of performance enhancing techniques that exploit the multilayered Blue Gene® architecture. These performance advantages depend on good programming techniques, thorough testing and debugging, followed by optimization, and tuning.

Distinguishing between optimization and tuning

You can use optimization and tuning separately or in combination to increase the performance of your application. Understanding the difference between them is the first step in understanding how the different levels, settings, and techniques can increase performance.

Optimization

Optimization is a compiler driven process that searches for opportunities to restructure your source code and give your application better overall performance at run time, without significantly impacting development time. The XL compiler optimization suite, which you control using compiler options and directives, performs best on well-written source code that has already been through a thorough debugging and testing process. These optimization transformations can:

- Reduce the number of instructions your application executes to perform critical operations.
- Restructure your object code to make optimal use of the Blue Gene architecture.
- Improve memory subsystem usage.
- Exploit the ability of the architecture to handle large amounts of shared memory parallelization.

Each basic optimization technique can result in a performance benefit, although not all optimizations can benefit all applications. Consult the “Steps in the optimization process” on page 2 for an overview of the common sequence of steps you can use to increase the performance of your application.

Tuning

While optimization applies general transformations designed to improve the performance of any application in any supported environment, tuning offers you opportunities to adjust specific characteristics or target execution environments of your application to improve its performance. Even at low optimization levels, tuning for your application and target architecture can have a positive impact on performance. With proper tuning the compiler can:

- Select more efficient machine instructions.
- Generate instruction sequences that are more relevant to your application.
- Select from more focussed optimizations to improve your code.

Steps in the optimization process

As you begin the optimization process, consider that not all optimization techniques suit all applications. Trade-offs sometimes occur between an increase in compile time, a reduction in debugging capability, and the improvements that optimization can provide.

Learning about, and experimenting with different optimization techniques can help you strike the right balance for your XL compiler applications while achieving the best possible performance. Also, though it is unnecessary to hand-optimize your code, compiler-friendly programming can be extremely beneficial to the optimization process. Unusual constructs can obscure the characteristics of your application and make performance optimization difficult. Use the steps in this section as a guide for optimizing your application.

1. The Basic optimization step begins your optimization processes at levels 0 and 2.
2. The Advanced optimization step exposes your application to more intense optimizations at levels 3, 4.
3. The High-order transformation (HOT) step can help you limit loop execution time.
4. The Interprocedural analysis (IPA) step can optimize your entire application at once.
5. The Debugging optimized code step can help you identify issues and problems that can occur with optimized code.
6. The Getting more performance section offers other strategies and tuning alternatives to compiler-driven optimization.

The section Compiler-friendly programming techniques contains tips for writing more easily optimized source code.

Basic optimization

The XL compiler supports several levels of optimization, with each option level building on the levels below through increasingly aggressive transformations, and consequently using more machine resources.

Ensure that your application compiles and executes properly at low optimization levels before trying more aggressive optimizations. This topic discusses two optimizations levels, listed with complementary options in the *Basic optimizations* table. The table also includes a column for compiler options that can have a performance benefit at that optimization level for some applications.

Table 4. Basic optimizations

Optimization level	Additional options implied by default	Complementary options	Other options with possible benefits
-O0	**-qsimd=auto**	**-qarch**	
-O2	**-qmaxmem=**8192 **-qsimd=auto**	**-qarch** **-qtune**	**-qmaxmem=**-1 **-qhot=level=0**

Note: Specifying **-O** without including a level implies **-O2**.

Optimizing at level 0

Benefits at level 0

- Minimal performance improvement, with minimal impact on machine resources.
- Exposes some source code problems, helping in the debugging process.

Begin your optimization process at **-O0** which the compiler already specifies by default. This level performs basic analytical optimization by removing obviously redundant code, and can result in better compile time. It also ensures your code is algorithmically correct so you can move forward to more complex optimizations. **-O0** also includes some redundant instruction elimination and constant folding. The option **-qfloat=nofold** can be used to suppress folding floating-point operations. Optimizing at this level accurately preserves all debugging information and can expose problems in existing code, such as uninitialized variables.

Additionally, specifying **-qarch** at this level targets your application for a particular machine and can significantly improve performance by ensuring your application takes advantage of all applicable architectural benefits.

Note: For SMP programs, you need to add an additional option **-qsmp=noopt**.

For more information on tuning, consult Tuning for Your Target Architecture.

See "-O" in the *XL Fortran Compiler Reference* for information on the **-O** level syntax.

Optimizing at level 2

Benefits at level 2

- Eliminates redundant code
- Basic loop optimization
- Can structure code to take advantage of **-qarch** and **-qtune** settings

After successfully compiling, executing, and debugging your application using **-O0**, recompiling at **-O2** opens your application to a set of comprehensive low-level transformations that apply to subprogram or compilation unit scopes and can include some inlining. Optimizations at **-O2** are a relative balance between increasing performance while limiting the impact on compilation time and system resources. You can increase the memory available to some of the optimizations in the **-O2** portfolio by providing a larger value for the **-qmaxmem** option. Specifying **-qmaxmem=-1** allows the optimizer to use memory as needed without checking for limits but does not change the transformations the optimizer applies to your application at **-O2**.

Starting to tune at level 2

Choosing the right hardware architecture target or family of targets becomes even more important at **-O2** and higher. By targeting the proper hardware, the optimizer can make the best use of the hardware facilities available. If you choose a family of hardware targets, the **-qtune** option can direct the compiler to emit code consistent with the architecture choice, but executes optimally on the chosen tuning hardware target. With this option, you can compile for a general set of targets but have the code run best on a particular target.

See the Chapter 2, "Tuning XL compiler applications," on page 31 topics for details on the **-qarch** and **-qtune** options.

The **-O2** option can perform a number of additional optimizations, including:

- Common subexpression elimination: Eliminates redundant instructions.
- Constant propagation: Evaluates constant expressions at compile-time.
- Dead code elimination: Eliminates instructions that a particular control flow does not reach, or that generate an unused result.
- Dead store elimination: Eliminates unnecessary variable assignments.
- Graph coloring register allocation: Globally assigns user variables to registers.
- Value numbering: Simplifies algebraic expressions, by eliminating redundant computations.
- Instruction scheduling for the target machine.
- Loop unrolling and software pipelining.
- Moving invariant code out of loops.
- Simplifying control flow.
- Strength reduction and effective use of addressing modes.

Even with **-O2** optimizations, some useful information about your source code is made available to the debugger if you specify **-g**. Using a higher **-g** level increases the information provided to the debugger, but reduces the optimization that can be done. Conversely, higher optimization levels can transform code to an extent to which debugging information is no longer accurate. Use that information with discretion.

The section on "Debugging optimized code" on page 20 discusses other debugging strategies in detail.

See "-O" in the *XL Fortran Compiler Reference* for information on the **-O** level syntax.

Advanced optimization

Higher optimization levels can have a tremendous impact on performance, but some trade-offs can occur in terms of code size, compile time, resource requirements, and numeric or algorithmic precision.

After applying "Basic optimization" on page 2 and successfully compiling and executing your application, you can apply more powerful optimization tools. The XL compiler optimization portfolio includes many options for directing advanced optimization, and the transformations your application undergoes are largely under your control. The discussion of each optimization level in Table 5 includes information on not only the performance benefits, and the possible trade-offs as well, but information on how you can help guide the optimizer to find the best solutions for your application.

Table 5. Advanced optimizations

Optimization Level	Additional options implied	Complementary options	Options with possible benefits
-O3	**-qnostrict** **-qmaxmem=-1** **-qhot=level=0** **-qsimd=auto**	**-qarch** **-qtune**	

Table 5. Advanced optimizations (continued)

Optimization Level	Additional options implied	Complementary options	Options with possible benefits
-O4	**-qnostrict** **-qmaxmem=-1** **-qhot** **-qipa** **-qarch=auto** **-qtune=auto** **-qcache=auto** **-qsimd=auto**	**-qarch** **-qtune** **-qcache**	**-qsmp=auto**
-O5	All of **-O4** **-qipa=level=2**	**-qarch** **-qtune** **-qcache**	**-qsmp=auto**

When you compile programs with any of the following sets of options:

- **-qhot -qnostrict**
- **-qhot -O3**
- **-O4**
- **-O5**

the compiler automatically attempts to vectorize calls to system math functions by calling the equivalent vector functions in the Mathematical Acceleration Subsystem libraries (MASS), with the exceptions of functions `vatan2`, `vsatan2`, `vdnint`, `vdint`, `vcosisin`, `vscosisin`, `vqdrt`, `vsqdrt`, `vrqdrt`, `vsrqdrt`, `vpopcnt4`, and `vpopcnt8`. If the compiler cannot vectorize, it automatically tries to call the equivalent MASS scalar functions. For automatic vectorization or scalarization, the compiler uses versions of the MASS functions contained in the system library `libxlopt.a`.

In addition to any of the preceding sets of options, when the **-qipa** option is in effect, if the compiler cannot vectorize, it tries to inline the MASS scalar functions before deciding to call them.

Optimizing at level 3

Benefits at level 3

- Automatic generation of SIMD instructions (**-qhot=level=0**)
- In-depth “Aliasing” on page 41 analysis
- Better loop scheduling
- High-order loop analysis and transformations (**-qhot=level=0**)
- Inlining of small procedures within a compilation unit by default
- Eliminating implicit compile-time memory usage limits
- Widening, which merges adjacent load/stores and other operations
- Pointer aliasing improvements to enhance other optimizations

Specifying **-O3** initiates more intense low-level transformations that remove many of the limitations present at **-O2**. For instance, the optimizer no longer checks for memory limits, by defaulting to **-qmaxmem=-1**. Additionally, optimizations encompass larger program regions and attempt more in-depth analysis. While not all applications contain opportunities for the optimizer to provide a measurable increase in performance, most applications can benefit from this type of analysis.

Potential trade-offs at level 3

With the in-depth analysis of **-O3** comes a trade-off in terms of compilation time and memory resources. Also, since **-O3** implies **-qnostrict**, the optimizer can alter certain floating-point semantics in your application to gain execution speed. This typically involves precision trade-offs as follows:

- Reordering of floating-point computations.
- Reordering or elimination of possible exceptions, such as division by zero or overflow.
- Using alternative calculations that might give slightly less precise results or not handle infinities or NaNs in the same way.

You can still gain most of the **-O3** benefits while preserving precise floating-point semantics by specifying **-qstrict**. Compiling with **-qstrict** is necessary if you require the same absolute precision in floating-point computational accuracy as you get with **-O0**, **-O2**, or **-qnoopt** results. The option **-qstrict=ieeefp** also ensures adherence to all IEEE semantics for floating-point operations. If your application is sensitive to floating-point exceptions or the order of evaluation for floating-point arithmetic, compiling with **-qstrict**, **-qstrict=exceptions**, or **-qstrict=order** helps to ensure accurate results. You should also consider the impact of the **-qstrict=precision** suboption group on floating-point computational accuracy. The precision suboption group includes the individual suboptions: **subnormals**, **operationprecision**, **association**, **reductionorder**, and **library** (described in the **-qstrict** option in the *XL Fortran Compiler Reference*).

Without **-qstrict**, the difference in computation for any one source-level operation is very small in comparison to “Basic optimization” on page 2. Although a small difference can be compounded if the operation is in a loop structure where the difference becomes additive, most applications are not sensitive to the changes that can occur in floating-point semantics.

See "-O" in the *XL Fortran Compiler Reference* for information on the **-O** level syntax.

An intermediate step: adding -qhot suboptions at level 3

At **-O3**, the optimization includes minimal **-qhot** loop transformations at **level=0** to increase performance. You can further increase your performance benefit by increasing the level and therefore the aggressiveness of **-qhot**. Try specifying **-qhot** without any suboptions, or **-qhot=level=1**.

The following **-qhot** suboptions can also provide additional performance benefits, depending on the characteristics of your application:

- **-qhot=vector** to enable long vectorization
- **-qhot=arraypad** to enable array padding
- **-qhot=fastmath** to enable the replacement of math routines with those from the XLOPT library

For more information on **-qhot**, see “High-order transformation (HOT)” on page 9.

Conversely, if the application does not use loops processing arrays (which **-qhot** improves), you can improve compile speed with minimal performance loss by using **-qnohot** after **-O3**.

Optimizing at level 4

Benefits at level 4

- Propagation of global and argument values between compilation units
- Inlining code from one compilation unit to another
- Reorganization or elimination of global data structures
- An increase in the precision of aliasing analysis

Optimizing at **-O4** builds on **-O3** by triggering **-qipa=level=1** which performs interprocedural analysis (IPA), optimizing your entire application as a unit. This option is particularly pertinent to applications that contain a large number of frequently used routines.

To make full use of IPA optimizations, you must specify **-O4** on the compilation and link steps of your application build as interprocedural analysis occurs in stages at both compile and link time.

Potential trade-offs at level 4

In addition to the trade-offs already mentioned for **-O3**, specifying **-qipa** can significantly increase compilation time, especially at the link step.

See "-O" in the *XL Fortran Compiler Reference* for information on the **-O** level syntax.

The IPA process

1. At compile time optimizations occur on a file-by-file basis, as well as preparation for the link stage. IPA writes analysis information directly into the object files the compiler produces.
2. At the link stage, IPA reads the information from the object files and analyzes the entire application.
3. This analysis guides the optimizer on how to rewrite and restructure your application and apply appropriate **-O3** level optimizations.

The “Interprocedural analysis (IPA)” on page 11 section contains more information on IPA including details on IPA suboptions.

Beyond **-qipa**, **-O4** enables other optimization options:

- **-qhot**

  Enables more aggressive HOT transformations to optimize loop constructs and array language.
- **-qhot=vector**

  Optimizes array data to run mathematical operations in parallel where applicable.
- **-qarch=**auto and **-qtune=**auto

  Optimizes your application to execute on a hardware architecture identical to your build machine. If the architecture of your build machine is incompatible with your application's execution environment, you must specify a different **-qarch** suboption after the **-O4** option. This overrides **-qarch=auto**.
- **-qcache=auto**

  Optimizes your cache configuration for execution on specific hardware architecture. The **auto** suboption assumes that the cache configuration of your build machine is identical to the configuration of your execution architecture.

Specifying a cache configuration can increase program performance, particularly loop operations by blocking them to process only the amount of data that can fit into the data cache.

If you want to execute your application on a different machine, specify correct cache values.

Optimizing at level 5

Benefits at level 5

- Most aggressive optimizations available
- Makes full use of loop optimizations and "Interprocedural analysis (IPA)" on page 11

As the highest optimization level, **-O5** includes all **-O4** optimizations and deepens whole program analysis by increasing the **-qipa** level to 2. Compiling with **-O5** also increases how aggressively the optimizer pursues aliasing improvements. Additionally, if your application contains a mix of C/C++ and Fortran code that you compile using the XL compilers, you can increase performance by compiling and linking your code with the **-O5** option.

Potential trade-offs at level 5

Compiling at **-O5** requires more compile time and machine resources than any other optimization levels, particularly if you include **-O5** on the IPA link step. Compile at **-O5** as the final phase in your optimization process after successfully compiling and executing your application at **-O4**.

See "-O" in the *XL Fortran Compiler Reference* for information on the **-O** level syntax.

Specialized optimization techniques

While some optimization techniques are active at advanced optimization levels, certain types of applications can receive a performance benefit even when you apply only basic optimizations.

Table 6. Specialized optimization techniques

Technique	**Benefit**
HOT	Minimizes loop execution time which is beneficial to most applications that contain large loops, or many small loops. **HOT** also improves memory access patterns in your application.
IPA	Performs whole program analysis, providing the optimization suite with a complete view of your entire application. This applies performance enhancements with more focus and robustness.
Vector technology	Vector technology is a PowerPC® technology for accelerating the performance-driven, high-bandwidth communications and computing applications. You can use the vector technology to get dramatic performance improvement for your applications.

Table 6. Specialized optimization techniques (continued)

Technique	**Benefit**
Compiler reports	You can use the **-qlistfmt** option to generate a compiler report in XML 1.0 format that indicates some of the details of how your program was optimized. You can use this information to understand your application code and to tune your code for better performance.

High-order transformation (HOT)

As part of the XL compiler optimization suite, the HOT transformations focus specifically on loops which typically account for the majority of the execution time for most applications. HOT transformations perform in-depth loop analysis to minimize their execution time.

Loop optimization analysis includes:
- Interchange
- Fusion
- Unrolling loop nests
- Reducing the use of temporary arrays

The goals of these optimizations include:
- Reducing memory access costs through effective cache use and translation look-aside buffers (TLBs). Increasing memory locality reduces cache and TLB misses.
- Overlapping computation and memory access through effective utilization of the hardware data prefetching capabilities.
- Improving processor resource utilization by reordering and balancing the use of instructions with complementary resource requirements. Loop computation balance typically involves creating an equitable relationship between load/store operations and floating-point computations.

Compiling with **-O3** and higher triggers HOT transformations by default. You can also see performance benefits by specifying **-qhot** with **-O2**, or adding more **-qhot** optimizations than the default **level=0** at **-O3** .

You can see particular **-qhot** benefits if your application contains Fortran 90-style array language constructs, as HOT transformations include elimination of intermediate temporary variables and statement fusion.

You can also use directives to assist in loop analysis. Assertive directives such as **INDEPENDENT** or **CNCALL** allow you to describe important loop characteristics or behaviors that HOT transformations can exploit. Prescriptive directives such as **UNROLL** or **PREFETCH** allow you to direct the HOT transformations on a loop-by-loop basis. You can also specify the **-qreport** compiler option to generate information about loop transformations. The report can assist you in deciding where best to include directives to improve the performance of your application. For example, you can use this section of the listing to identify non-stride-one references that may prevent loop vectorization.

You can use the **-qreport** option in conjunction with **-qhot** or any optimization option that works with **-qhot** to produce a pseudo-Fortran report showing how the loops were transformed. The `LOOP TRANSFORMATION SECTION` of the listing file also contains information about data prefetch insertion locations.

When used with **-qsmp**, **-qhot=level=2** instructs the compiler to perform the transformations of **-qhot=level=1** plus some additional transformation on nested loops. The resulting loop analysis and transformations can lead to more cache reuse and loop parallelization. If you use **-qhot=level=2** and **-qsmp** together with **-qreport** or **-qlistfmt**, you can see this information on aggressive loop analysis performed on loop nests in the `LOOP TRANSFORMATION SECTION` of the listing file or compiler report.

With the **-qassert=refalign** suboption, the compiler might generate more efficient code. This assertion is particularly useful when you target a Single Instruction Multiple Data (SIMD) architecture with **-qhot=level=0** or **-qhot=level=1** with the **-qsimd=auto** option.

In addition to general loop transformation, **-qhot** supports suboptions that you can specify to enable additional transformations detailed in this section.

HOT short vectorization

When you are targeting a PowerPC processor that supports Quad Processing Extension (QPX) vectors, the default **-qsimd=auto** option enables the compiler to transform code into QPX vector instructions. These machine instructions can execute up to sixteen operations in parallel. This transformation mostly applies to the loops that iterate over contiguous array data and perform calculations on each element. You can use the **NOSIMD** directive to prevent the transformation of a particular loop.

HOT long vectorization

When you specify any of the following:
- **-O4** and higher
- **-qhot** with **-qnostrict**

you enable **-qhot=vector** by default. Specifying **-qnostrict** with optimizations other than **-O4** and **-O5** ensures that the compiler looks for long vectorization opportunities. This can optimize loops in source code for operations on array data by ensuring that operations run in parallel where applicable. The compiler uses standard machine registers for these transformations and does not restrict vector data size; supporting both single- and double-precision floating-point vectorization. Often, HOT vectorization involves transformations of loop calculations into calls to specialized mathematical routines supplied with the compiler such as the Mathematical Acceleration Subsystem (MASS) libraries. These mathematical routines use algorithms that calculate results more efficiently than executing the original loop code.

For more information about optimization levels like **-O4** and the other compiler options they imply, see "Advanced optimization" on page 4.

HOT array size adjustment

An array dimension that is a power of two can lead to a decrease in cache utilization. The **-qhot=arraypad** suboption allows the compiler to increase the dimensions of arrays where doing so could improve the efficiency of array-processing loops. Using this suboption can reduce cache misses and page faults that slow your array processing programs. The HOT transformations will not necessarily pad all arrays, and can pad different arrays by different amounts in order to gain performance. You can specify a padding factor to apply to all arrays. This value is typically a multiple of the largest array element size.

Use **-qhot=arraypad** with discretion as array padding uses more memory and the performance trade-off does not benefit all applications. Also, these HOT transformations do not include checks for array data overlay, as with Fortran **EQUIVALENCE**, or array shaping operations.

HOT fast scalar math routines

The XLOPT library contains faster versions of certain math functions that are normally provided by the operating system or in the default runtime. With **-qhot=fastmath**, the compiler replaces calls to the math functions with their faster counterparts in XLOPT library. This option requires **-qstrict=nolibrary** in effect.

Interprocedural analysis (IPA)

Interprocedural Analysis (IPA) can analyze and optimize your application as a whole, rather than on a file-by-file basis.

Run during the link step of an application build, the entire application, including linked libraries, is available for interprocedural analysis. This whole program analysis opens your application to a powerful set of transformations available only when more than one file or compilation unit is accessible. IPA optimizations are also effective on mixed language applications.

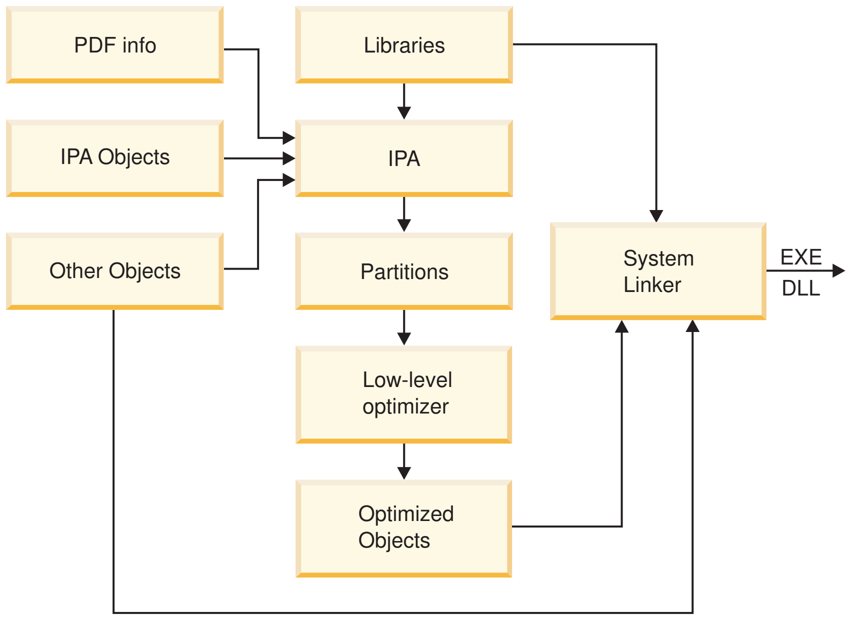


Figure 1. IPA at the link step

The following are some of the link-time transformations that IPA can use to restructure and optimize your application:

- Inlining between compilation units.
- Complex data flow analyses across subprogram calls to eliminate parameters or propagate constants directly into called subprograms.

- Improving parameter usage analysis, or replacing external subprogram calls to system libraries with more efficient inline code.
- Restructuring data structures to maximize access locality.

In order to maximize IPA link-time optimization, you must use IPA at both the compile and link step. Objects you do not compile with IPA can only provide minimal information to the optimizer, and receive minimal benefit. However when IPA is active on the compile step, the resulting object file contains program information that IPA can read during the link step. The program information is invisible to the system linker, and you can still use the object file and link without invoking IPA. The IPA optimizations use hidden information to reconstruct the original compilation and can completely analyze the subprograms the object contains in the context of their actual usage in your application.

During the link step, IPA restructures your application, partitioning it into distinct logical code units. After IPA optimizations are complete, IPA applies the same low-level compilation-unit transformations as the **-O2** and **-O3** base optimizations levels. Following those transformations, the compiler creates one or more object files and linking occurs with the necessary libraries through the system linker.

It is important that you specify a set of compilation options as consistent as possible when compiling and linking your application. This includes all compiler options, not just **-qipa** suboptions. When possible, specify identical options on all compilations and repeat the same options on the IPA link step. Incompatible or conflicting options that you specify to create object files, or link-time options in conflict with compile-time options can reduce the effectiveness of IPA optimizations.

Using IPA on the compile step only

About this task

IPA can still perform transformations if you do not specify IPA on the link step. Using IPA on the compile step initiates optimizations that can improve performance for an individual object file even if you do not link the object file using IPA. The primary focus of IPA is link-step optimization, but using IPA only on the compile-step can still be beneficial to your application without incurring the costs of link-time IPA.

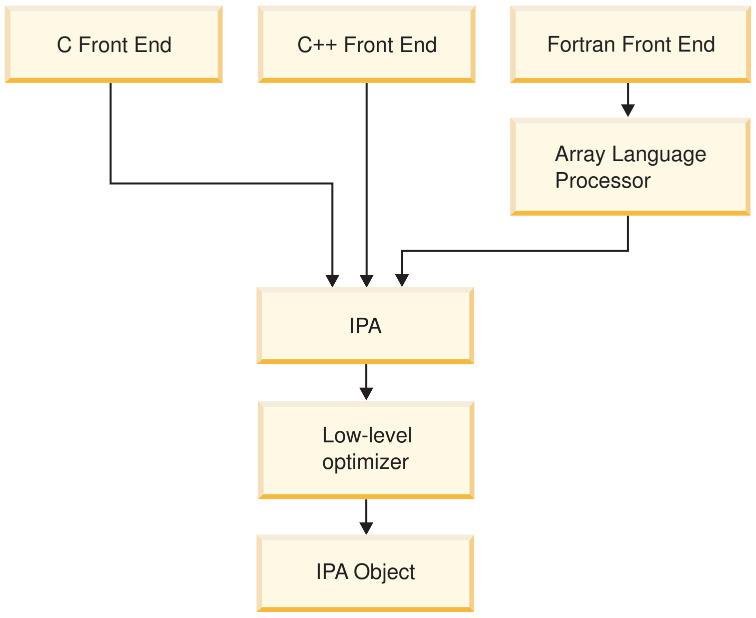


Figure 2. IPA at the compile step

IPA Levels and other IPA suboptions

You can control many IPA optimization functions using the **-qipa** option and suboptions. The most important part of the IPA optimization process is the level at which IPA optimization occurs. Default compilation does not invoke IPA. If you specify **-qipa** without a level, or specify **-O4**, IPA optimizations are at level one. If you specify **-O5**, IPA optimizations are at level two.

Table 7. The levels of IPA

IPA Level	Behaviors
qipa=level=0	• Automatically recognizes standard library functions • Localizes statically bound variables and procedures • Organizes and partitions your code according to call affinity, expanding the scope of the **-O2** and **-O3** low-level compilation unit optimizer • Lowers compilation time in comparison to higher levels, though limits analysis
qipa=level=1	• Level 0 optimizations • Performs procedure inlining across compilation units • Organizes and partitions static data according to reference affinity

Table 7. The levels of IPA (continued)

IPA Level	Behaviors
qipa=level=2	• Level 0 and level 1 optimizations • Performs whole program alias analysis which removes ambiguity between pointer references and calls, while refining call side effect information • Propagates interprocedural constants • Eliminates dead code • Performs pointer analysis • Performs procedure cloning • Optimizes intraprocedural operations, using specifically: – Value numbering – Code propagation and simplification – Code motion, into conditions and out of loops – Redundancy elimination techniques • Performs data reorganization

IPA includes many suboptions that can help you guide IPA to perform optimizations important to the particular characteristics of your application. Among the most relevant to providing information on your application are:

- **lowfreq**, with which you can specify a list of procedures that are likely to be called infrequently during the course of a typical program run. Performance can increase because optimization transformations will not focus on these procedures.
- **partition**, with which you can specify the size of the regions within the program to analyze. Larger partitions contain more procedures, which result in better interprocedural analysis but require more storage to optimize.
- **threads**, with which you can specify the number of parallel threads available to IPA optimizations. This can provide an increase in compilation-time performance on multi-processor systems.

Using IPA across the XL compiler family

About this task

The XL compiler family shares optimization technology. Object files you create using IPA on the compile step with the XL C, C++, and Fortran compilers can undergo IPA analysis during the link step. Where program analysis shows that objects were built with compatible options, such as **-qnostrict**, IPA can perform transformations such as inlining C functions into Fortran code, or propagating C++ constant data into C function calls.

Vector technology

Vector technology is a PowerPC technology for accelerating the performance-driven, high-bandwidth communications and computing applications. You can use the vector technology to get dramatic performance improvement for your applications.

There are two ways of using the vector technology: hand coding and automatic vectorization. Automatic vectorization often brings the best performance when you write the code in the right way, but appropriate hand coding can provide additional performance improvement.

The following example shows the difference between a simple array element addition and a vectorized version of the same loop.

Array element addition without using the vector technology:

```
subroutine myadd(n)
  integer :: i, n
  real(8), dimension(n) :: a, b, c

  do i=1, n
    a(i) = b(i) + c(i)
  enddo
end subroutine
```

Modified array element addition utilizing the vector technology:

```
subroutine myadd_vector(n)
  integer :: j, n
! vector_size is a constant
  vector(real(8)), dimension(n/vector_size) :: v_a, v_b, v_c

  do j=1, n/vector_size
    v_a(j) = vec_add(v_b(j), v_c(j))
  enddo
end subroutine
```

In the vectorized version of the code, the data type is replaced by the vector data type. The loop range is reduced from `n` to `n/vector_size`. Without the vector technology, multiple instructions cost many processor clock cycles. With the vector technology, the operation, `v_a(j)=vec_add(v_b(j), v_c(j))`, is executed in a single machine instruction for each vector. Therefore, the vector technology can improve the performance of an application.

This section provides general information about vector technology with the following three subsections:
- "Vector technology information"
- "Explicitly calling vector libraries for vectorization" on page 17
- "Auto-vectorization limitations" on page 17

Vector technology information

This section provides links to all of the information about the vector technology and categorize them into the following types:
- Using vector technology with hand coding
- Using vector technology with auto-vectorization

Using vector technology with hand coding

The following table lists the information about using the vector technology with hand coding and provides the links to the detailed information in different documents.

Table 8. Language features for using vector technology with hand coding:

Information you need	**Sections you can read**
Intrinsic data types	Vector (IBM extension) in *XL Fortran Language Reference*
Vector type declaration statement	Vector (IBM extension) in *XL Fortran Language Reference*

Table 8. Language features for using vector technology with hand coding: (continued)

Information you need	**Sections you can read**
Vector intrinsic procedures	Vector intrinsic procedures (IBM extension) in *XL Fortran Language Reference*
Using the vector libraries	Using the vector libraries

Using vector technology with auto-vectorization

The following table lists the information about compiler options for auto-vectorization and provides the links to the detailed information in different documents.

Table 9. Information about compiler options for auto-vectorization

To do...	**Read...**
Enable generation of vector instructions for processors that support them.	-qsimd in *XL Fortran Compiler Reference*
Perform high-order transformations (HOT) during optimization.	-qhot in *XL Fortran Compiler Reference*
Produce listing files and understand how sections of code have been optimized.	• -qlistfmt in *XL Fortran Compiler Reference* • -qreport in *XL Fortran Compiler Reference* • Using compiler reports to diagnose optimization opportunities • Parsing compiler reports with development tools
Ensure that optimizations done by default, do not alter certain program semantics related to strict IEEE floating-point conformance.	-qstrict in *XL Fortran Compiler Reference*
Tuning for your target architecture using -qarch and -qtune.	• Tuning for your target architecture • Using -qtune

The following table lists the directive and compiler option that you can use to prohibit auto-vectorization and provides the links to the detailed information in different documents.

Table 10. Directive and compiler option for auto-vectorization

To do...	**Read...**
Prohibit the compiler from auto-vectorizing the loop immediately following the directive.	NOVECTOR in *XL Fortran Language Reference*
Disable auto-vectorization.	-qsimd in *XL Fortran Compiler Reference*

Some optimization processes are related to auto-vectorization, you can use compiler options to control these optimizations. The following table lists these optimization processes and provides the links to the detailed information in different documents.

Table 11. Optimizations related to auto-vectorization

To learn about...	Read...
The High-order transformation (HOT)	• High-order transformation (HOT) • An intermediate step: adding -qhot suboptions at level 3
The Interprocedural analysis (IPA)	The IPA process

Explicitly calling vector libraries for vectorization

To use the vector technology in your applications, you can either rewrite the algorithm manually or rely on the automatic vectorization of the compiler. Although automatic vectorization can provide the highest performing solution, proper hand coding can also bring good performance.

The following example shows how to explicitly call the vector libraries to make use of the vector functionality provided by the target hardware.

```
function dotp(x,y,n) result(s)
       real*8 x(*),y(*),s
       vector(real(8)) sv,xv,yv
       integer i,n

       sv = vec_splats(0.0D0)
       do i=1,n,2
           xv = vec_ld2(0,x(i))
           yv = vec_ld2(0,y(i))
           sv = vec_madd(xv,yv,sv)
       enddo
       s = vec_extract(sv,0)+vec_extract(sv,1)
       if (mod(n,2) .eq. 1) then
           s = s + x(n)*y(n)
       endif
end function

program dot
       real*8 x(100),y(100),s
       integer i
       do i=1,100
           x(i)=0.5*i
           y(i)=2.0
       enddo
       s = dotp(x,y,100)
       print *,s
end
```

The program performs the dot product for two arrays of REAL. At each iteration, two elements from the arrays are loaded into two REAL vector variables. The program then uses a multiply add operation to calculate the product of the two vectors and add the product with the previous sum. At the end of the loop the two elements of the vector that hold the partial sums are added to form the complete sum value. If the size of the input vectors do not evenly fit in the vector variables, a single scalar product is performed to complete the dot product computation.

Auto-vectorization limitations

When you use the auto-vectorization, you might find that some transformations cannot be performed. If you compile with **-qhot** and **-qlistfmt=xml=transforms** or **-qlistfmt=xml=all**, you can get a compiler report that lists the reasons why some transformations were not performed. For detailed information about the possible reasons, see Using compiler reports to diagnose optimization opportunities.

This section uses two code examples to illustrate why auto-vectorization cannot be performed under certain situations.

Example 1:

```
program try
   real*8 x(100)
   integer i
   x(1)=9
   do i=2,100
     x(i)=x(i-1)
   enddo
end
```

The `x(i)=x(i-1)` statement violates the restriction that "a loop cannot be automatically parallelized if one of its variable carries a dependency". `x(i)` or `x(i-1)` depends on each other in this sample, which makes the loop non-vectorizable.

Example 2:

```
program try
   real*8 x(100)
   integer i
   do i=1,100,5
     x(i)=i + 8;
     x(i+1)=i + 9;
     x(i+2)=i + 12;
     x(i+3)=i + 15;
   enddo
end
```

The following statements violate the restriction that auto-vectorization cannot be performed if the loop contains a non stride one store.

```
     x(i)=i + 8;
     x(i+1)=i + 9;
     x(i+2)=i + 12;
     x(i+3)=i + 15;
```

In each iteration of the loop, four elements in the array x are accessed and one element is skipped. This continues until the end of the loop, which makes the loop cannot be vectorized.

Using compiler reports to diagnose optimization opportunities

You can use the **-qlistfmt** option to generate a compiler report in XML or HTML format that indicates some of the details of how your program was optimized. You can also use the **genhtml** tool to convert an existing XML report to HTML format. This information can be used to understand your application code and to tune your code for better performance.

The compiler report in XML format can be viewed in a browser that supports XSLT. If you compile with the stylesheet suboption, **-qlistfmt=xml=all:stylesheet=xlstyle.xsl**, the report contains a link to a stylesheet that renders the XML readable and provides you with opportunities to improve the optimization of your code. You can also create tools to parse this information.

Inline reports

If compiled with **-qinline** and one of **-qlistfmt=xml=inlines**, **-qlistfmt=html=inlines**, **-qlistfmt=xml** or **-qlistfmt=html**, the compiler report that is generated includes a list of inline attempts during the compilation. The report also specifies the type of attempt and its outcome.

For each function that the compiler has attempted to inline, there is an indication of whether the inline was successful. The report might contain any number of explanations for a named function that has not been successfully inlined. Some examples of these explanations are:

- FunctionTooBig - The function is too big to be inlined.
- RecursiveCall - The function is not inlined because it is recursive.
- ProhibitedByUser - Inlining was not performed because of a user specified pragma or directive.
- CallerIsNoopt - No inlining was performed because the caller was compiled without optimization.
- WeakAndNotExplicitlyInline - The calling function is weak and not marked as inline.

For a complete list of the possible explanations, see the `Inline optimization types` section of the XML schema help file called `XMLContent.html` in the `/opt/ibmcmp/xlf/bg/14.1/listings/` directory, which also includes its Japanese and Chinese version, `XMLContent-Japanese.utf8.html` and `XMLContent-Chinese.utf8.html`.

Loop transformations

If compiled with **-qhot** and one of **-qlistfmt=xml=transforms**, **-qlistfmt=html=transforms**, **-qlistfmt=xml** or **-qlistfmt=html**, the compiler report that is generated includes a list of the transformations performed on all loops in the file during the compilation. It also lists reasons why some transformations were not performed.

- Reasons why a loop cannot be automatically parallelized
- Reasons why a loop cannot be unrolled
- Reasons why SIMD vectorization failed

For a complete list of the possible transformation problems, see the `Loop transformation types` section of the XML schema help file called `XMLContent.html` in the `/opt/ibmcmp/xlf/bg/14.1/listings/` directory, which also includes its Japanese and Chinese version, `XMLContent-Japanese.utf8.html` and `XMLContent-Chinese.utf8.html`.

Data reorganizations

If compiled with **-qhot** and one of **-qlistfmt=xml=data**, **-qlistfmt=html=data**, **-qlistfmt=xml** or **-qlistfmt=html**, the compiler report that is generated includes a list of data reorganizations performed on the program during compilation. Here are some examples of data reorganizations:

- Array splitting
- Array coalescing
- Array interleaving
- Array transposition

- Common block splitting
- Memory merge

For each of these reorganizations, the report contains details about the name of the data, file names, line numbers, and the region names.

Parsing compiler reports with development tools

Software development tools can be created to parse the compiler reports produced in XML format. These tools can help direct you to opportunities to improve the performance of your application.

The compiler includes an XML schema that you can use to create a tool to parse the compiler reports and display aspects of your code that may represent performance improvement opportunities. The schema, `xllisting.xsd`, is located in the `/opt/ibmcmp/xlf/bg/14.1/listings/` directory. This schema helps to present the information from the report in a tree structure.

You can also find a schema help file called `XMLContent.html` that helps you understand the details of the schema.

Debugging optimized code

Debugging optimized programs presents special usability problems. Optimization can change the sequence of operations, add or remove code, change variable data locations, and perform other transformations that make it difficult to associate the generated code with the original source statements.

For example:

Data location issues
With an optimized program, it is not always certain where the most current value for a variable is located. For example, a value in memory may not be current if the most current value is being stored in a register. Most debuggers are incapable of following the removal of stores to a variable, and to the debugger it appears as though that variable is never updated, or possibly even set. This contrasts with no optimization where all values are flushed back to memory and debugging can be more effective and usable.

Instruction scheduling issues
With an optimized program, the compiler may reorder instructions. That is, instructions may not be executed in the order the programmer would expect based on the sequence of lines in their original source code. Also, the sequence of instructions may not be contiguous. As the user steps through their program with a debugger, it may appear as if they are returning to a previously executed line in their code (interleaving of instructions).

Consolidating variable values
Optimizations can result in the removal and consolidation of variables. For example, if a program has two expressions that assign the same value to two different variables, the compiler may substitute a single variable. This can inhibit debug usability because a variable that a programmer is expecting to see is no longer available in the optimized program.

There are a couple of different approaches you can take to improve debug capabilities while also optimizing your program:

Debug non-optimized code first
Debug a non-optimized version of your program first, then recompile it with your desired optimization options. See "Debugging in the presence of optimization" for some compiler options that are useful in this approach.

Use -g level
Use the **-g** level suboption to control the amount of debugging information made available. Increasing it improves debug capability, but prevents some optimizations.

Use -qoptdebug
When compiling with **-O3** optimization or higher, use the compiler option **-qoptdebug** to generate a pseudocode file that more accurately maps to how instructions and variable values will operate in an optimized program. With this option, when you load your program into a debugger, you will be debugging the pseudocode for the optimized program. See "Using -qoptdebug to help debug optimized programs" on page 22 for more information.

Understanding different results in optimized programs

Here are some reasons why an optimized program might produce different results from one that has not undergone the optimization process:

- Optimized code can fail if a program contains code that is not valid. For example, failure can occur if the program passes an actual argument that also appears in a common block in the called procedure, or if two or more dummy arguments are associated with the same actual argument. The optimization process relies on your application conforming to language standards.
- If a program that works without optimization fails when you optimize, check the cross-reference listing and the execution flow of the program for variables that are used before they are initialized. Compile with the **-qinitauto=***hex_value* or **-qinitalloc=***hex_value* option to try to produce the incorrect results consistently. For example, using **-qinitauto=FF** gives **REAL** and **COMPLEX** variables an initial value of "negative not a number" (-NAN). Any operations on these variables will also result in NAN values. Other bit patterns (***hex_value***) may yield different results and provide further clues as to what is going on. Programs with uninitialized variables can appear to work properly when compiled without optimization, because of the default assumptions the compiler makes, but can fail when you optimize. Similarly, a program can appear to execute correctly after optimization, but fails at lower optimization levels or when run in a different environment.
- A variation on uninitialized storage. Referring to an automatic-storage variable by its address after the owning function has gone out of scope leads to a reference to a memory location that can be overwritten as other auto variables come into scope as new functions are called.

Use with caution debugging techniques that rely on examining values in storage. The compiler might have deleted or moved a common expression evaluation. It might have assigned some variables to registers, so that they do not appear in storage at all.

Debugging in the presence of optimization

Debug and compile your program with your desired optimization options. Test the optimized program before placing it into production. If the optimized code does not produce the expected results, you can attempt to isolate the specific optimization problems in a debugging session.

The following list presents options that provide specialized information, which can be helpful during the development of optimized code:

-qlist
Instructs the compiler to emit an object listing. The object listing includes hex and pseudo-assembly representations of the generated instructions, traceback tables, and text constants.

-qreport
Instructs the compiler to produce a report of the loop transformations it performed and how the program was parallelized. For **-qreport** to generate a listing, the options **-qhot** or **-qsmp** should also be specified.

-qipa=list
Instructs the compiler to emit an object listing that provides information for IPA optimization.

-qcheck
Generates code that performs certain types of runtime checking.

-qsmp=noopt
If you are debugging SMP code, **-qsmp=noopt** ensures that the compiler performs only the minimum transformations necessary to parallelize your code and preserves maximum debug capability.

-qoptdebug
When used with high levels of optimization, produces files containing optimized pseudocode that can be read by a debugger.

-qkeepparm
Ensures that procedure parameters are stored on the stack even during optimization. This can negatively impact execution performance. The **-qkeepparm** option then provides access to the values of incoming parameters to tools, such as debuggers, simply by preserving those values on the stack.

-qinitalloc
Instructs the compiler to emit code that initializes all allocatable and pointer variables that are allocated but not initialized to a given value.

-qinitauto
Instructs the compiler to emit code that initializes all automatic variables to a given value.

-g, -qdbg
Generates debugging information for use by a symbolic debugger. You can use different **-g** or **-qdbg** levels to debug optimized code by viewing or possibly modifying accessible variables at selected source locations in the debugger. Higher **-g** or **-qdbg** levels provide a more complete debug support, while lower levels provide higher runtime performance. For details, see **-g** or **-qdbg**.

In addition, you can also use the **SNAPSHOT** directive to ensure that certain variables are visible to the debugger at points in your application. For details, see **SNAPSHOT**.

Using -qoptdebug to help debug optimized programs

The purpose of the **-qoptdebug** compiler option is to aid the debugging of optimized programs. It does this by creating pseudocode that maps more closely to the instructions and values of an optimized program than the original source code. When a program compiled with this option is loaded into a debugger, you will be debugging the pseudocode rather than your original source. By making optimizations explicit in pseudocode, you can gain a better understanding of how your program is really behaving under optimization. Files containing the pseudocode for your program are generated with the file suffix `.optdbg`. Only line debugging is supported for this feature.

Compile your program as in the following example:

```
bgxlf95 myprogram.f -O3 -qhot -g -qoptdebug -o myprogram
```

In this example, your source file is compiled to a.out. The pseudocode for the optimized program is written to a file called `myprogram.optdbg` which can be referred to while debugging your program.

Notes:

- The **-g** or the **-qlinedebug** option must also be specified in order for the compiled executable to be debuggable. However, if neither of these options are specified, the pseudocode file `<output_file>.optdbg` containing the optimized pseudocode is still generated.
- The **-qoptdebug** option only has an effect when one or more of the optimization options **-qhot**, **-qsmp**, or **-qipa** are specified, or when the optimization levels that imply these options are specified; that is, the optimization levels **-O3**, **-O4**, and **-O5**. The example shows the optimization options **-qhot** and **-O3**.

Debugging the optimized program

From the following examples, you can see how the compiler might apply optimizations to a simple program and how debugging it can differ from debugging your original source.

Example 1: Represents the original non-optimized code for a simple program. It presents an optimization opportunity to the compiler; the loop can be unrolled. In the optimized source, you can see iterations of the loop listed explicitly.

Example 2: Represents a listing of the optimized source as shown in the debugger. You can see the unrolled loop in the optimized source.

Example 3: Shows an example of stepping through the optimized source using the debugger. Note, there is no longer a correspondence between the line numbers for these statements in the optimized source as compared to the line numbers in the original source.

Example 1: Original code

```
FUNCTION FOO(X, Y) RESULT(R)
  IMPLICIT NONE
  REAL :: X, Y, Z, R
  INTEGER :: I
  Z = X + Y
  DO I = 1, 4
    PRINT *, Z + REAL(I)
  END DO
  R = Z
END FUNCTION

PROGRAM MAIN
  IMPPLICIT NONE
```

```
        REAL :: X, FOO
        X = FOO(3.0, 4.0)
        PRINT *, X
      END PROGRAM MAIN
```

Example 2: gdb debugger listing

```
(gdb) list 1,55
1
2
3           1|         REAL*4 FUNCTION foo (x, y)
4           8|           #2 = _xlfBeginIO(",257,#1,32768,NULL,0,NULL)
5                        T_2 = (x + y) +  1.00000000E+00
6                        CALL _xlfWriteLDReal(%VAL(#2),T_2,",")
7                        _xlfEndIO(%VAL(#2))
8                        #2.rnn5 = _xlfBeginIO(",257,#1,32768,NULL,0,NULL)
9                        T_2 = (x + y) +  2.00000000E+00
10                       CALL _xlfWriteLDReal(%VAL(#2.rnn5),T_2,",")
11                       _xlfEndIO(%VAL(#2.rnn5))
12                       #2.rnn4 = _xlfBeginIO(",257,#1,32768,NULL,0,NULL)
13                       T_2 = (x + y) +  3.00000000E+00
14                       CALL _xlfWriteLDReal(%VAL(#2.rnn4),T_2,",")
15                       _xlfEndIO(%VAL(#2.rnn4))
16                       #2.rnn3 = _xlfBeginIO(",257,#1,32768,NULL,0,NULL)
17                       T_2 = (x + y) +  4.00000000E+00
18                       CALL _xlfWriteLDReal(%VAL(#2.rnn3),T_2,",")
19                       _xlfEndIO(%VAL(#2.rnn3))
20         11|           RETURN
21                     END FUNCTION foo
22
23
24         13|         PROGRAM main ()
25         17|           T_4 =  3.00000000E+00
26                       T_5 =  4.00000000E+00
27                       x = foo(T_4,T_5)
28         19|           #2 = _xlfBeginIO(",257,#1,32768,NULL,0,NULL)
29                       #3 = x
30                       CALL _xlfWriteLDReal(%VAL(#2),#3,",")
31                       _xlfEndIO(%VAL(#2))
32         20|           CALL _xlfExit(0)
33                     END PROGRAM main
```

Example 3: Stepping through optimized source

```
(gdb) break foo
Breakpoint 1 at 0x100006a0: file myprogram.o.optdbg, line 3.
(gdb) run
Starting program: /home/user/myprogram
[Thread debugging using libthread_db enabled]
[New Thread -134438912 (LWP 24368)]
[Switching to Thread -134438912 (LWP 24368)]

Breakpoint 1, foo (x=Cannot access memory at address 0x0
) at myprogram.o.optdbg:3
3           1|         REAL*4 FUNCTION foo (x, y)
(gdb) step
4           8|           #2 = _xlfBeginIO(",257,#1,32768,NULL,0,NULL)
(gdb) step
5                        T_2 = (x + y) +  1.00000000E+00
(gdb) step
4           8|           #2 = _xlfBeginIO(",257,#1,32768,NULL,0,NULL)
(gdb) step
5                        T_2 = (x + y) +  1.00000000E+00
(gdb) step
6                        CALL _xlfWriteLDReal(%VAL(#2),T_2,",")
(gdb) step
 8.000000000
7                        _xlfEndIO(%VAL(#2))
```

```
(gdb) step
8                        #2.rnn5 = _xlfBeginIO(",257,#1,32768,NULL,0,NULL)
(gdb) cont
Continuing.
 9.000000000
 10.00000000
 11.00000000
 7.000000000

Program exited normally.
```

Tracing procedures in your code

You can instruct the compiler to insert calls to the tracing procedures that you have defined to aid in debugging or timing the execution of other procedures.

To trace procedures in your program, you must specify which procedures to trace. You must also provide your own tracing procedures. If you enable tracing without providing tracing procedures, you will get linker errors about undefined symbols called `__func_trace_enter`, `__func_trace_exit`, and possibly `__func_trace_catch`.

Specifying which procedures to trace

The **-qfunctrace** compiler option controls tracing for all non-inlined user-defined procedures and all outlined compiler-generated procedures in your program. If you are interested in tracing specific external or modules procedures, you can use the **-qfunctrace+** and **-qfunctrace-** compiler options. You can also specify the **NOFUNCTRACE** directive to disable the tracing of entire modules, external procedures, module procedures, or internal procedures.

What can be traced

Tracing applies to programs, external procedures, non-intrinsic module procedures, and internal procedures.

Compiler-generated procedures are not traced unless they were generated for outlined user code, such as an OpenMP program. In those cases, the name of the outlined procedure contains the name of the original user procedure as a prefix.

Inlined procedures and statement functions cannot be traced because they do not exist in the executable.

To avoid infinite recursion, user-defined tracing procedures cannot be traced. Similarly, tracing must be disabled for procedures called from user-defined tracing procedures.

How to write tracing procedures

You can implement the tracing procedures in Fortran, C, or C++.

To implement the tracing procedures in Fortran, the characteristics of the procedures must be the same as those specified in the following interface:

```
SUBROUTINE routine_name(procedure_name, file_name, line_number, id)
  USE, INTRINSIC :: iso_c_binding
  CHARACTER(*), INTENT(IN) :: procedure_name
  CHARACTER(*), INTENT(IN) :: file_name
  INTEGER(C_INT), INTENT(IN) :: line_number
  TYPE(C_PTR), INTENT(INOUT) :: id
END SUBROUTINE
```

where `routine_name` is the name of an external or module procedure.

You must then tell the compiler to use your subroutine as a tracing procedure in one of the following ways:

- Using the **-qfunctrace_xlf_enter**, **-qfunctrace_xlf_exit**, or **-qfunctrace_xlf_catch** compiler options.
- Using the **FUNCTRACE_XLF_ENTER**, **FUNCTRACE_XLF_EXIT**, or **FUNCTRACE_XLF_CATCH** directives.

When you specify these options or directives, XL Fortran generates wrapper procedures called `__func_trace_enter`, `__func_trace_exit`, and `__func_trace_catch` that call your corresponding tracing procedure. These wrappers allow interoperability with C and C++ by converting the dummy arguments from the C prototype to the interface described earlier. `routine_name` must therefore not be named `__func_trace_enter`, `__func_trace_exit`, or `__func_trace_catch`. In addition, your program must not contain more than one of each of the tracing procedures.

Writing the tracing procedures in C or C++ requires that you provide the `__func_trace_enter`, `__func_trace_exit`, and `__func_trace_catch` procedures directly. They must have the following prototypes:

- `void __func_trace_enter(const char *const procedure_name, const char *const file_name, int line_number, void **const id);`
- `void __func_trace_exit(const char *const procedure_name, const char *const file_name, int line_number, void **const id);`
- `void __func_trace_catch(const char *const procedure_name, const char *const file_name, int line_number, void **const id);`

Note: If you write the tracing procedures in C++, they must be declared `extern "C"`.

XL Fortran inserts calls to your tracing procedures on procedure entry and exit. It passes the name of the procedure being traced, the name of the file containing the entry or exit point being traced, and the line number. It also passes the address of a static pointer that is initialized to `C_NULL_PTR` at the beginning of the program. This pointer allows you to store arbitrary data in the entry tracing procedure and access this data in the exit and catch procedures. See the Examples section for detail. Because this pointer resides in static memory, extra steps might be needed when tracing threaded or recursive procedures.

Sample tracing procedures

XL Fortran provides sample tracing procedures in the `/opt/ibmcmp/xlf/bg/14.1/samples/functrace` directory. You can use these procedures for simple tracing, or you can modify them for more complex tracing.

- `tracing_routines.c`: Provides tracing procedures written in C. This file is useful when you do not require access to Fortran modules, and when there is a possibility of recursive input / output.

- `tracing_routines.f90`: Provides tracing procedures written in Fortran. This file is useful when you need access to Fortran modules or intrinsics in your tracing procedures.

The following example illustrates the use of the samples for simple tracing:

```
> cat helloworld.f
print *, 'hello world'
end
> cc -c /opt/ibmcmp/xlf/bg/14.1/samples/functrace/tracing_routines.c
> bgxlf95 helloworld.f -qfunctrace tracing_routines.o
** _main   === End of Compilation 1 ===
1501-510  Compilation successful for file helloworld.f.
> ./a.out
{ _main (helloworld.f:1)
 hello world
} _main (helloworld.f:2)

>
```

Tracing limitations

The procedure tracing functionality has the following limitations:

- A procedure cannot be traced separately from its **ENTRY** points. Either all are traced or none are. The name of the procedure is passed to the tracing procedure even when tracing the **ENTRY** point. The line number helps distinguish what is being traced in this case.
- The Fortran standard requires pure procedures to have no side effects. The compiler uses this assumption when optimizing your program. If you enable tracing of a pure procedure, your tracing procedure must not change the program state in a way that creates a side effect.
- The Fortran standard imposes limits on recursive input/output. If you write your tracing procedures in Fortran, you must be careful not to break these rules.

  The following example has a print statement where an I/O item is the result of a function call (foo). It is illegal for the tracing procedure in this case to have I/O on an external file:

  ```
  > cat recursive.f
  integer function test()
    test = 1
  end function

  integer test
  print *, test() ! test must not have I/O on external unit
  end
  > bgxlf95 -c /opt/ibmcmp/xlf/bg/14.1/samples/functrace/tracing_routines.f90
  ** my__func_trace_enter   === End of Compilation 1 ===
  ** my__func_trace_exit   === End of Compilation 2 ===
  ** my__func_trace_catch   === End of Compilation 3 ===
  1501-510  Compilation successful for file tracing_routines.f90.
  > bgxlf95 recursive.f tracing_routines.o -qfunctrace
  ** test   === End of Compilation 1 ===
  ** _main   === End of Compilation 2 ===
  1501-510  Compilation successful for file recursive.f.
  > ./a.out
  { _main (recursive.f:6)
  XL Fortran (I/O initialization): I/O recursion detected.
  Aborted
  >
  ```

  Note: You can work around this by writing the tracing procedure in C. For an example, see the `tracing_routines.c` sample file described in section "Sample tracing procedures" on page 26.

- When optimizing your program, the compiler reorders code and removes dead code. As a result, the line number passed to the tracing procedure might not be accurate when optimization is enabled.

Examples

In the following example, **-qfunctrace** is used to measure the time spent in each external procedure. The **FUNCTRACE_XLF_ENTER** and **FUNCTRACE_XLF_EXIT** directives are used to specify procedures `my_enter` and `my_exit` as the tracing procedures. The **NOFUNCTRACE** directive is used to disable tracing of `main_program`:

```
> cat example.f
! Designate my_enter as a tracing procedure that should be called
! on procedure entry
!ibm* functrace_xlf_enter
subroutine my_enter(procedure_name, file_name, line_number, id)
  use, intrinsic :: iso_c_binding
  use, intrinsic :: xlfutility
  character(*), intent(in) :: procedure_name, file_name
  integer(c_int), intent(in) :: line_number
  type(c_ptr), intent(inout) :: id

  integer(kind=time_size), pointer :: enter_count

  ! Store the time we entered the procedure being traced into id.
  if (.not. c_associated(id)) then
    allocate(enter_count)
    enter_count = time_()
    id = c_loc(enter_count)
  end if

  print *, 'Entered procedure ', procedure_name, ' at ( ',  &
           file_name, ' :', line_number, ').'
end subroutine

! Designate my_exit as a tracing procedure that should be called
! on procedure exit
!ibm* functrace_xlf_exit
subroutine my_exit(procedure_name, file_name, line_number, id)
  use, intrinsic :: iso_c_binding
  use, intrinsic :: xlfutility
  character(*), intent(in) :: procedure_name, file_name
  integer(c_int), intent(in) :: line_number
  type(c_ptr), intent(inout) :: id

  integer(kind=time_size), pointer :: enter_count
  integer(kind=time_size) exit_count, duration

  ! id should have been associated in my_enter with the time we
  ! entered the procedure being traced.  Find the elapsed time.
  if (c_associated(id)) then
    exit_count = time_()
    call c_f_pointer(id, enter_count)
    duration = exit_count - enter_count
  else
    stop "error!"
  endif

  print *, 'Leaving procedure ', procedure_name, ' at ( ',  &
           file_name, ' :', line_number, ').'
  print *, 'Spent', duration, 'seconds in ', procedure_name, '.'
end subroutine

! sub2 will be traced
subroutine sub2
```

```
  call sleep_(3)
end subroutine

! sub1 will be traced
subroutine sub1
  call sleep_(5)
  call sub2
end subroutine

! Do not want to trace main_program
!ibm* nofunctrace
program main_program
  call sub1
end program
> bgxlf95 example.f -qfunctrace
** my_enter   === End of Compilation 1 ===
** my_exit   === End of Compilation 2 ===
** sub2   === End of Compilation 3 ===
** sub1   === End of Compilation 4 ===
** main_program   === End of Compilation 5 ===
1501-510  Compilation successful for file example.f.
> ./a.out
 Entered procedure sub1 at ( example.f : 59 ).
 Entered procedure sub2 at ( example.f : 54 ).
 Leaving procedure sub2 at ( example.f : 55 ).
 Spent 3 seconds in sub2.
 Leaving procedure sub1 at ( example.f : 61 ).
 Spent 8 seconds in sub1.
>
```

Related information

- For details about the **-qfunctrace** compiler option, see -qfunctrace in the *XL Fortran Compiler Reference*.
- For details about **-qfunctrace_xlf_catch**, **-qfunctrace_xlf_enter**, or **-qfunctrace_xlf_exit** compiler options, see the Detailed descriptions of the XL Fortran compiler options section in the *XL Fortran Compiler Reference*.
- For details about the **FUNCTRACE_XLF_CATCH**, **FUNCTRACE_XLF_ENTER**, and **FUNCTRACE_XLF_EXIT** directives, see Detailed directive descriptions section in the *XL Fortran Language Reference*.
- For details about the **NOFUNCTRACE** directive, see NOFUNCTRACE in the *XL Fortran Language Reference*.

Getting more performance

The XL compiler family offers other strategies and tuning alternatives for increasing performance.

Whether you are already optimizing at **-O5**, or you are looking for more opportunities to increase performance without the resource costs of optimizing at higher levels, the XL compiler family offers other strategies and tuning alternatives. For more information, see the following topics:

- Tuning XL compiler applications
- Advanced optimization concepts
- Optimizing your SMP code

Beyond performance: effective programming techniques

Applications that perform well begin with applications that are written well. See the following topics for information about writing better code; whether your goal is to make your code more portable, more easily optimized, or interoperable with other languages.

- Chapter 4, “Managing code size,” on page 45
- Chapter 5, “Compiler-friendly programming techniques,” on page 51
- Chapter 7, “Parallel programming with XL Fortran,” on page 69
- Chapter 8, “Interlanguage calls,” on page 253

Chapter 2. Tuning XL compiler applications

Included as part of the XL Fortran optimization suite are options you can use to instruct the compiler to generate code that executes optimally on a given processor or architecture family, and to instruct the compiler on the execution characteristics of your application.

The better you can convey those characteristics, the more precisely the compiler can tune and optimize your application. This section assumes that you have already begun optimizing your application using the strategies found in Optimizing your applications.

Tuning for your target architecture

By default, the compiler generates code that runs on all supported systems, though this code does not run optimally on all supported systems. By selecting options to target the appropriate architectures, you can optimize your application to suit the broadest possible selection of relevant processors, a range of processors within a given family, or a specific processor.

The compiler options in the *Options for targeting your architecture* table introduce how you can control optimizations affecting individual aspects of your target architecture. This section also goes into further detail on how you can use some of those options to ensure your application provides the best possible performance on those targets.

Table 12. Options for targeting your architecture

Option	Behavior
-q64	Generates code for a 64-bit addressing model (64-bit execution mode).
-qarch	Selects a family of processor architectures, or a specific architecture that the compiler will generate machine instructions for. **-qarch=qp** produces object code that runs on the Blue Gene/Q platforms.
-qtune	Focuses optimizations for execution on a given processor without restricting the processor architectures that your application can execute on. **–qtune=qp** specifies that optimizations are tuned for the Blue Gene/Q platforms.
-qcache	Defines a specific cache or memory geometry. Selecting a predefined optimization level like **-O2** sets default vales for **-qcache** suboptions.

In addition to targeting the correct architecture for your application, it is important to select the right level of optimization. Combining the appropriate architecture settings with an optimization level that fits your application can vastly enhance performance. If you have not already done so, consult Optimizing your applications in addition to this section.

Using -qarch

Using **-qarch** you can select a machine architecture or a family of architectures on which you can run your application. Selecting the correct **-qarch** suboption is crucial to influencing chip-level optimization as the choice of **-qarch** suboption controls:

- The list of machine instructions available to the compiler when generating object code.
- The characteristics and capabilities of the hardware the compiler will model when optimizing.
- Optimization trade-offs and opportunities in individual instruction selection and instruction sequence selection
- The default setting of the **-qtune** option.

Architecture selection is important at all optimization levels. Even at low optimization levels like **-O0** and **-O2**, specifying the correct target architecture can be beneficial to performance. Specifying the correct target allows the compiler to select more efficient machine instructions and generate instruction sequences that perform best for a particular machine.

The **-qarch** suboptions allow you to specify individual processors or a family of processors with common instruction sets or subsets. The choice of processor gives you the flexibility of compiling your application to execute optimally on a particular machine, or to execute on a wide variety of machines while still applying as much architecture-specific optimization as possible. The less specific your choice of architecture, the fewer machine instructions available to the compiler when generating code. A less specific choice can also limit the number of hardware intrinsic functions available to your application. A more specific choice of architecture, can make available more instructions and hardware intrinsic functions. The *XL Fortran Compiler Reference* details the specific chip architectures and architecture families available.

When compiling your application, using a consistent or compatible **-qarch** setting for all files will ensure that you are getting the most from your architecture targets. If you are using **-qipa** link-time optimizations, the architecture setting you specify on the link step overrides the compile step setting.

You must ensure that your application executes only on machines that support your **-qarch** settings. Executing your application on other machines can produce incorrect results, even if your application appears to run without trapping.

Using -qtune

The **-qtune** option focuses optimizations for execution on a given processor without restricting the processor architectures that your application can execute on, generating machine instructions consistent with your **-qarch** architecture choice. Using **-qtune** also guides the optimizer in performing transformations, such as instruction scheduling, so that the resulting code executes most efficiently on your chosen **-qtune** architecture. The **-qtune** option tunes code to run on one particular processor architecture, and includes only specific processors as suboptions. The **-qtune** option does not support suboptions representing families of processors.

Use **-qtune** to specify the most common or critical processor where your application executes.

The default **-qtune** setting depends on the **-qarch** setting. If the **-qarch** option is set to a particular machine architecture, this limits the range of available **-qtune** suboptions, and the default tune setting will be compatible with the selected target processor. If **-qarch** option is set to a family of processors, the range of values available for **-qtune** expands across that family, and the default is chosen from a commonly used machine in that family. If you compile with **-qtune=auto**, the default for optimization levels **-O4** and **-O5**, the compiler detects the machine characteristics on which you are compiling, and assumes you want to tune for that type of machine. You can override this behavior by specifying **-qtune** after the **-O4** or **-O5** compiler options.

Using -qcache

The -qcache option allows you to instruct the optimizer on the memory cache layout of your target architecture. There are several suboptions you can specify to describe cache characteristics such as:

- The types of cache available
- The cache size
- Cache-miss penalties

The **-qcache** option is only effective if you understand the cache characteristics of the execution environment of your application. Before using **-qcache**, look at the options section of the listing file with the **-qlist** option to see if the current cache settings are acceptable. The settings appear in the listing when you compile with **-qlistopt**. If you are unsure about how to interpret this information, do not use **-qcache**, and allow the compiler to use default cache settings.

If you do not specify **-qcache**, the compiler makes cache assumptions based on your **-qarch** and **-qtune** settings. If you compile with the **-qcache=auto** suboption, the default at optimization levels -O4 and -O5, the compiler detects the cache characteristics of your compilation machine and tunes cache optimizations for that cache layout. If you do specify **-qcache**, also specify **-qhot**, or an option such as **-O4** that implies **-qhot**. The optimizations that **-qhot** performs are designed to take advantage of your **-qcache** settings.

Before you finish tuning

Consult the following list to ensure that you are getting the most out of your target machine options.

- Do not specify a **-qarch** option that is incompatible with your hardware. This can produce unexpected results.
- Specify a **-qarch** setting that represents the largest common instruction set available to the machines that your application will execute on.
- If you are executing your application on multiple machines, choose the **-qtune** suboption that aligns with the machine you expect your application to run on most frequently or where performance is most important.
- If compiling with **-qcache**, specify **-qhot** as well, which can take advantage of your cache settings.

Tuning your code for Blue Gene

This section describes the strategy that you can use to facilitate the automatic single-instruction-multiple-data (SIMD) capabilities in XL Fortran on Blue Gene/Q platforms.

Blue Gene/Q provides SIMD instructions, which can operate quadrate double words (256 bits) in parallel with one instruction. SIMD exploits the instruction level parallelism (ILP) and can greatly improve the performance of your program.

IBM XL Fortran for Blue Gene/Q, V14.1 is able to automatically generate SIMD instructions (auto-SIMD) for your program when the conditions for SIMD are satisfied. Auto-SIMD is enabled at the following optimization levels when **-qsimd=auto** and **-qhot=level=1** is in effect. In particular, at **-O3**, **-qhot=level=0** is implied and auto-SIMD is enabled too.

- **-O2 -qhot**
- **-O3**
- **-O3 -qhot**
- **-O4 -qhot**
- **-O5 -qhot**

Notes:

- On Blue Gene/Q, **-qsimd=auto** is enabled by default at all optimization levels.
- Specifying **-qhot** without suboptions is equivalent to **-qhot=level=1**.

You can always turn off auto-SIMD with **-qsimd=noauto**.

Auto-SIMD for loops and basic blocks

When auto-SIMD is enabled, XL Fortran looks for two kinds of candidates, loops and basic blocks, and tries to group the operations on contiguous data into SIMD operations.

Example 1

This example shows a sample loop construct.

```
DO i = 0, 100
   a(i) = b(i) + c(i)
END DO
```

When auto-SIMD is enabled, XL Fortran can transform this loop into the following pseudo code. Four iterations of this loop are run in parallel with SIMD instructions.

```
VECTOR(REAL(8)) :: v1
VECTOR(REAL(8)) :: v2
VECTOR(REAL(8)) :: v3

DO i = 0, 100, 4
  v1 = VEC_LDS(0, b(i))       ! Loads b(i), b(i+1), b(i+2), and b(i+3).
  v2 = VEC_LDS(0, c(i))       ! Loads c(i), c(i+1), c(i+2), and c(i+3).
  v3 = VEC_ADD(v1, v2)        ! Adds the four elements.
  CALL VEC_ST(v3, 0, a(i))    ! Stores the result to a(i), a(i+1), a(i+2), and a(i+3).
END DO
```

Example 2

This example shows a sample basic block.

```
a(1) = b(1) + c(1)
a(2) = b(2) + c(2)
a(3) = b(3) + c(3)
a(4) = b(4) + c(4)
```

When auto-SIMD is enabled, XL Fortran can transform the block into the following pseudo code. The four statements are run in parallel with SIMD instructions.

```
VECTOR(REAL(8)) :: v1 = VEC_LDS(0, b(1))   ! Loads b(1), b(2), b(3), and b(4).
VECTOR(REAL(8)) :: v2 = VEC_LDS(0, c(1))   ! Loads c(1), c(2), c(3), and c(4).
VECTOR(REAL(8)) :: v3 = VEC_ADD(v1, v2)    ! Adds the four elements.
CALL VEC_ST(v3, 0, a(1))                   ! Stores the result to a(1), a(2), a(3), and a(4).
```

You can use the **-qreport** option to display which loops are optimized by auto-SIMD and why other loops are not.

Conditions for auto-SIMD transformation

In auto-SIMD transformation, XL Fortran verifies the following conditions that guarantee the correctness of the transformation:

- The dependence constraint for SIMD

  You can help the compiler by providing the alias information , or use the **INDEPENDENT** directive to specify that the loop iterations are independent.

  ```
  !IBM* INDEPENDENT
  DO i = 1, n
    a(i + l1) = a(i + l2) + a(i + l3)
  END DO
  ```
- The alignment of data access

  The alignment of pointers might become unknown to the compiler when pointer assignment or pointer arithmetic are involved. You can help the compiler by providing the alignment information with the **ALIGN** directive, or using the **-qassert=refalign** option to assert that all the pointers are naturally aligned to the type of data being pointed to.

  The following example shows how to use the **ALIGNX** intrinsic procedure to assert the alignment.

  ```
  CALL ALIGNX(32, a)
  CALL ALIGNX(32, b)
  CALL ALIGNX(32, c)

  DO n = 1, 200
    c(n) = a(n) + b(n)
  END DO
  ```

Related information

- -qsimd
- -qhot
- -qreport
- -qassert
- ALIGN
- INDEPENDENT
- ALIGNX(K,M)
- Vector intrinsic procedures (IBM extension)

Further option driven tuning

You can use options to convey the characteristics of your application to the compiler, tuning the optimizations that the compiler will apply. Option driven tuning is a process that can require experimentation to find the right combination of options to increase the performance of your application.

The XL compilers support many options that allow you to assert that your application will not follow certain standard language rules in some instances. The compiler assumes language standard compliance and can perform unsafe optimizations if your application is not compliant. Standards-conforming applications are more easily optimized and more portable, but when full compliance is not possible, use the appropriate options to ensure your code is optimized safely.

For complete compiler option syntax, see the *XL Fortran Compiler Reference*.

Options for providing application characteristics

This section provides a list of options that can dictate a wide variety of characteristics about your application to the compiler including floating-point and loop behaviors.

Option Description

-qalias
Supports several suboptions that can help the compiler analyze the characteristics of your application. For more information on aliasing, see Advanced optimization concepts.

noaryovrlp
Asserts that your application contains no array assignments between storage associated (overlapping) arrays.

nointptr
Asserts that your application does not make use of integer (Cray) pointers.

nopteovrlp
Asserts that your application does not contain pointee variables that refer to any data objects that are not pointee variables. Also, that your application does not contain two pointee variables that can refer to the same storage location.

std
Asserts that your application follows all language rules for variable aliasing. This is the default compiler setting. Specify **-qalias=nostd** if your application does not follow all variable aliasing rules.

-qassert
Includes the following suboptions that can be useful for providing some loop characteristics of your application.

nodeps
Asserts that the loops in your application do not contain loop carry dependencies.

itercnt=*number*
Gives the optimizer a value to use when estimating the number of iterations for loops where it cannot determine that value.

-qddim
Forces the compiler to reevaluate the bounds of a pointee array each time the application references the array. Specify this option only if your application performs dynamic dimensioning of pointee arrays.

-qdirectstorage
Asserts that your application accesses write-through-enabled or cache-inhibited storage.

-qfloat
Provides the compiler with floating-point characteristics for your application. The following suboptions are particularly useful.

nans
Asserts that your application makes use of signaling NaN (not-a-number) floating-point values. Normal floating-point operations do not create these values, your application must create signalling NaNs.

rrm Prohibits optimization transformations that assume the floating-point rounding mode must be the default setting round-to-nearest. If your application changes the rounding mode in any way, specify this option.

-qflttrap
Controls various aspects of floating-point exception handling that your application can require if it attempts to detect or handle such exceptions.

-qieee Specifies the preferred floating-point rounding mode when evaluating expressions at compile time. This option is important if your application requires a non-default rounding mode in order to have consistency between compile-time evaluation and runtime evaluation.

You can also specify **-y** to set the preferred floating-point rounding mode.

-qlibansi
Asserts that any external function calls in your compilation that have the same name as standard C library function calls, such as malloc or memcpy, are in fact those functions and are not a user-written function with that name.

-qlibessl
Asserts that your application will be linked with IBM's ESSL high-performance mathematical library and that mathematical operations can be transformed into calls to that library. For more information on ESSL, see the High performance libraries topic.

-qlibmpi
Asserts that all functions with Message Passing Interface (MPI) names are in fact MPI functions and not a user function with different semantics.

-qlibposix
Asserts that any external function calls in your application that have the same name as standard Posix library function calls are in fact those functions and are not a user-written function with that name.

-qonetrip
Asserts that all DO loops in your application will execute at least one iteration. You can also specify this behavior with **-1**.

-qnostrictieeemod
Relaxes certain rules required by the Fortran 2003 standard related to the use of the IEEE intrinsic modules. Specify this option if your application does not use these modules.

-qstrict_induction
Prevents optimization transformations that would be unsafe if DO loop integer iteration count variables overflow and become negative. Few applications contain algorithms that require this option.

-qthreaded
Informs the compiler that your application will execute in a multithreaded/SMP environment. Using an **_r** invocation, like **bgxlf_r**, adds this option automatically.

-qnounwind
Informs the compiler that the stack will not be unwound while any routine in your application is active. The **-qnounwind** option enables prologue tailoring optimization, which reduces the number of saves and restores of nonvolatile registers.

-qnozerosize
: Asserts that this application does not require checking for zero-sized arrays when performing array operations.

Options to control optimization transformations

There are many options available to you in addition to the base set found in the Optimizing your applications section. Some of these options prevent an optimization that can be unsafe for certain applications or enable one that is safe for your application, but is not normally available as part of the optimization process.

Option Description

-qcompact
: Chooses a reduction of final code size over a reduction in execution time. You can use this option to constrain the optimizations of **-O3** and higher. For more information on restriction code size, see the Managing code size section.

-qsimd=auto
: Makes use of the vector capabilities of chips.

-qfloat
: This option provides a number of suboptions for controlling the optimizations to your floating-point calculations.

  norsqrt
  : Prevents the replacement of the division of the result of a square-root calculation with a multiplication by the reciprocal of the square root.

  nostrictmaf
  : Prevents certain floating-point multiply-and-add instructions where the sign of signed zero value would not be preserved.

-qipa Includes many suboptions that can assist the IPA optimizations while analyzing your application. If you are using the **-qipa** option or higher optimization levels that imply IPA, it is to your benefit to examine the suboptions available.

-qmaxmem
: Limits the memory available to certain memory-intensive optimizations at low levels. Specify **-qmaxmem=-1** to remove these memory limits.

-qnoprefetch
: Prevents the the insertion of prefetching machine instructions into your application during optimization.

-qinline
: Exerts control over inlining optimization transformations. For more information on inlining, see the Advanced optimization concepts section.

-qsmallstack
: Instructs the compiler to limit the use of stack storage in your application. This can increase heap usage.

-qsmp Produces code for an SMP system. This option also searches for opportunities to increase performance by automatically parallelizing your code. The Parallel programming with XL Fortran section contains more information on writing parallel code.

-qstacktemp
: Limits certain compiler temporaries allocated on the stack. Those not allocated on the stack will be allocated on the heap. This option is useful for applications that use enough stack space to exceed stack user or system limits.

-qstrict
: Limits optimizations to strict adherence to implied program semantics. This often prevents the compiler from ignoring certain little-used rules in the IEEE floating-point specification that few applications require for correct behavior. For example, reordering or reassociating a sequence of floating-point calculations can cause floating-point exceptions at an unexpected location or mask them completely. The **-qstrict** option includes suboptions that refine the control of the transformations performed by the optimizers. Do not use this option unless your application requires strict adherence as **-qstrict** and its suboptions can severely inhibit optimization.

-qunroll
: Independently controls loop unrolling. At **-O3** and higher, **-qunroll** is a default setting.

Options to assist with performance analysis

The compiler provides a set of options that can help you analyze the performance aspects of your application. These options are most useful when you are selecting your level of optimization and tuning the optimization process to the particular characteristics of your application.

-d Informs the compiler that you want to preserve the preprocessed versions of your compilation files. Typically these files would have a .F extension.

-g inserts full debugging information into your object code. While the optimization process can obscure original program meaning, at least some of the information that this option produces is useful to performance analysis tools. You can also specify this behavior with **-qdbg**.

-p Inserts appropriate profiling information into your object to code to make using tools for performance analysis possible. You can also specify this behavior with **-pg**.

-qdpcl Prepares your object for processing by tools based on the Dynamic Probe Class Library (DPCL).

-qlinedebug
: An option similar to **-g**, this option inserts only minimal debugging information into your object code such as function names and line number information.

-qlist Produces a listing file containing a pseuo-assembly listing of your object code.

-qlistfmt
: Creates a compiler report to assist with finding optimization opportunities.

-qreport
: Inserts information in the listing file showing the transformations done by certain optimizations.

-S Produces a .s file containing the assembly version of the .o file produced by the compilation.

-qtbtable
Limits the amount of debugging traceback information in object files, which reduces the size of the program. Use **-qtbtable=full** if you intend to analyze your application with a profiling utility.

Options that can inhibit performance

Some compiler options are necessary for some applications to produce correct or repeatable results. Usually, these options instruct the compiler to enforce very strict language semantics that few applications require. Others are supported by the compiler to allow compilation of code that does not conform to language standards. Avoid these options if you are trying to increase the runtime performance of your application. In cases where these options are enabled by default, you must disable them to increase performance. You can specify **-qlistopt** to show, in the listing file, the settings of each of these options.

The following list summarizes the options that can inhibit performance. Each option is described in the *XL Fortran Compiler Reference*.

- **-qalias=nostd**
- **-qcompact**
- **-qfloat=nosqrt**, **-qfloat=nostrictmaf**, **-qfloat=rrm**
- **-qsimd=noauto**
- **-qnoprefetch**
- **-qnounroll**
- **-qsmallstack**
- **-qstacktemp**=[value other than 0 or -1]
- **-qstrict**
- **-qstrict_induction**
- **-qstrictieeemod**
- **-qunwind**
- **-qxlf2008=checkpresence**
- **-qzerosize**
- **-qnoinline**

Chapter 3. Advanced optimization concepts

After you apply command-line optimizations and tuning that are appropriate to your application and the constraints of your development cycle, you have opportunities to further improve the performance of your application through aliasing and inlining.

Aliasing

An alias occurs when different variables point directly or indirectly to a single area of storage. Aliasing refers to assumptions made during optimization about which variables can point to or occupy the same storage area.

When an alias exists, or the potential for an alias occurs during the optimization process, pessimistic aliasing occurs. This can inhibit optimizations like dead store elimination and loop transformations on aliased variables. Also, pessimistic aliasing can generate additional loads and stores as the compiler must ensure that any changes to the variable that occur through the alias are not lost.

When aliasing occurs there is less opportunity for optimization transformations to occur on and around aliased variables than variables where no aliasing has taken place. For example, if variables *A*, *B*, and *C* are all aliased, any optimization must assume that a store into or a use of *A* is also a store or a use of *B* and *C*, even if that is not the case. Some of the highest optimization levels can improve alias analysis and remove some pessimistic aliases. However, in all cases, when it is not proven during an optimization transformation that an alias can be removed that alias must be left in place.

Where possible, avoid programming techniques that lead to pessimistic aliasing assumptions. These aliasing assumptions are the single most limiting factor to optimization transformations. The following situations can lead to pessimistic aliasing:

- When you assign a pointer the address of any variable, the pointer can be aliased with globally visible variables and with static variables visible in the pointer's scope.
- When you call a procedure that has dummy arguments passed by reference, aliasing occurs for variables used as actual arguments, and for global variables.
- The compiler will make several worst-case aliasing assumptions concerning variables in common blocks and modules. These assumptions can inhibit optimization.

Some compiler options like **-qalias** can affect aliasing directly. For more information on how to tune the aliasing behavior in your application, see “Options for providing application characteristics” on page 36.

Inlining

Inlining is the process of replacing a subroutine or function call at the call site with the body of the subroutine or function being called. This eliminates call-linkage overhead and can expose significant optimization opportunities.

For example, with inlining, the compiler can replace the subroutine parameters in the function body with the actual arguments passed. Inlining trade-offs can include code bloat and an increase in the difficulty of debugging your source code.

If your application contains many calls to small procedures, the procedure call overhead can sometimes increase the execution time of the application considerably. Specifying the **-qinline** compiler option can reduce this overhead. Additionally, you can use the **-p** or **-pg** options and profiling tools to determine which subprograms your application calls most frequently, and use **-qinline** to list their names to ensure inlining.

The **-qinline** option can perform inlining where the calling and called procedures are in different compilation units. This applies to optimization level **-O5** only.

```
# Let the compiler decide what to inline.
bgxlf95 -O3 -qinline inline.f

# Encourage the compiler to inline particular subprograms.
bgxlf95 -O3 -qinline+called_100_times:called_1000_times inline.f
```

Note: **-qipa=inline** is deprecated and no longer supported; it is replaced by **-qinline**.

Finding the right level of inlining

A common occurrence in application optimization is excessive inlining. This can actually lead to a decrease in performance because running larger programs can cause more frequent cache misses and page faults. Because the XL compilers contain safeguards to prevent excessive inlining, this can lead to situations where subprograms you want to inline are not automatically inlined when you specify **-qinline**.

Some common conditions that prevent **-qinline** from inlining particular subprograms are:

- The calling and called procedures are in different compilation units. If so, you can use the **-qinline** option in the link step to enable cross-file inlining. This applies to optimization level **-O5** only.
- After inlining expands a subprogram to a particular limit, the optimizer does not inline subsequent calls to that subprogram.
- Any interface errors, such as different numbers, sizes, or types of arguments or return values, can prevent inlining for a subprogram call. You can also use interface blocks for the procedures being called.
- Actual or potential aliasing of dummy arguments or automatic variables can limit inlining. Consider the following cases:
  - There are more than 31 arguments to the procedure your application is calling.
  - Any automatic variables in the called procedures are involved in an **EQUIVALENCE** statement
  - The same variable argument is passed more than once in the same call. For example, `CALL SUB(X,Y,X)`.
- Some procedures that use computed **GO TO** statements, where any of the corresponding statement labels are also used in an **ASSIGN** statement.

To change the size limits that control inlining, you can specify **-qinline=level=*n***, where *n* is `0` through `10`. Larger values allow more inlining.

It is possible to inline C/C++ functions into Fortran programs and Fortran functions into C/C++ programs during link time optimizations. You must compile the C/C++ code using the IBM XL C/C++ compilers with **-qinline** and a compatible option set to that used in the IBM XL Fortran compilation.

Chapter 4. Managing code size

Code size is often not a detriment to performance for most XL compiler programmers. For some however, generating compact object code can be as important as generating efficient code.

Oversized programs can affect overall performance by creating a conflict for real storage between pages of virtual storage containing code, and pages of virtual storage containing data. On systems with a small, combined instruction and data cache, cache collisions between code and data can also reduce performance. This section provides suggestions on how to achieve a balance between code efficiency and object-module size, while identifying compiler options that can affect object-module size. Code size tuning is most effective once you have built a stable application and run optimization at **-O2** or higher.

Reasons for tuning for code size include:

- Your application design calls for an implementation with limited real memory, instruction-cache space, or disk space.
- When loading your application, it uses enough memory to create a conflict between code areas and data areas in real memory, and both code and data are frequently paged in and out.
- There are high activity areas in your code large enough that instruction cache and instruction Translation Lookaside Buffer (TLB) misses have a major effect on performance.
- You intend your application to run on a host that serves end users, or in a batch environment with limits on real memory.

Before tuning for code size, it is important for you to determine whether code size is the actual problem. Very large applications tend to have small clusters of high activity and large sections of infrequently accessed code. If a particular code page is not accessed in a particular run, that page is never loaded into memory, and has no negative impact on performance. If you are tuning for code size due to the high activity code segments that cause instruction cache and instruction TLB misses that have a major effect on performance, this can be symptomatic of a program structure that requires improvement or hardware not suited to the resource requirements of the application.

If your data takes up more real storage than is available, reducing code size can improve performance by ensuring that fewer pages of data are paged out as code is paged in. However, data blocking strategies are likely to prove both more effective and easier to implement. Processing data in each page as completely as possible before moving on to the next page can reduce the number of data page misses.

If you are coding an application for a machine with a combined instruction and data cache, you can improve performance by applying the techniques described later in this section, but tuning for data cache management can yield better results than code-size tuning. Also note that highly tuning your code for the cache characteristics of one system can lead to undesirable performance results if you execute your application elsewhere.

Steps for reducing code size

Reducing the code size of your application can have a positive effect on the performance of your application

Consider the following steps for reducing code size:

- Ensure that you have built a stable application that compiles at **-O2** or higher.
- Use performance analysis tools to isolate high activity code segments and tune for performance where appropriate. Basing decisions for code size tuning on an application that has already undergone performance analysis will give you more information on where your application could benefit from code size tuning.
- Use compiler options like **-qcompact** to help reduce code size. See Compiler option influences on code size for more information. Also see the following options in the *XL Fortran Compiler Reference*:
  - **-qinline**.
  - The *partition* parameter for **-qipa**.
  - **-qunroll**.

Be aware that optimization can cause code to expand significantly through loop unrolling, invariant **IF** floating, inlining, and other optimizations. The higher your optimization level, the more code size can increase. For more information on finding an optimization level appropriate for your application, see Chapter 1, "Optimizing your applications," on page 1.

Compiler option influences on code size

High optimization levels can increase code size. You can use other compiler options to influence the size of your code and improve performance.

The -qipa compiler option

The **-qipa** option enables interprocedural analysis (IPA) by the compiler. Interprocedural analysis analyzes the relationships between procedures and the code that references those procedures, so that more optimizations within procedures and across procedure references can take place. Interprocedural analysis can decrease code size and improve performance at the same time. In some cases however, IPA inlining can increase code size. Use with discretion.

Related reference:

See interprocedural analysis (IPA) in the Compiler Reference

The -qinline inlining option

Using the **-qinline** compiler option, you can specify that the compiler consider all Fortran 90 or Fortran 95 procedures, or a particular list of procedures for inlining. Inlining procedures can increase the performance of your application. However, if your program references a procedure from many different locations in the source code, inlining that procedure can increase code size dramatically. You can use **-qnoinline** to disable procedure inlining entirely. You can also partially disable inlining with **-qinline**-*procedure_name*.

Do not assume that all inlining increases code size. When your source code references a very small procedure many times, inlining can reduce code size, because inlining eliminates control transfer and data interface code. In addition, inlining code facilitates other optimizations at the point of inlining, by providing information on the values of arguments referencing the procedure. If a procedure is very small and is referenced from a number of places, inlining can also increase code locality and reduce code paging.

For details about the **-qinline** compiler option, see -qinline in the *XL Fortran Compiler Reference*.

The -qhot compiler option

The loop analysis and optimization available when you specify **-qhot** can increase code size. If your application contains many large loops and loop optimization opportunities exist, **-qhot** code size can increase significantly along with performance. Specifying **-qhot=level=0** will perform minimal high-order transformations if code size is an issue. The topic High-order transformation contains more information on using **-qhot** effectively.

The -qcompact compiler option

The **-qcompact** compiler option instructs the compiler to avoid certain optimizing transformations that expand the object code. Compiling with **-qcompact**, disables many transformations, including:

- Loop unrolling
- Expansion of fixed-point multiply by more than one instruction
- Inline expansion of some string and memory manipulation functions. In some cases **-qcompact** will avoid inlining opportunities entirely.

Specifying **-qcompact** creates a trade-off between the performance of individual routines in your application, and overall system performance. Suppressing transformations degrades the performance of individual routines, while overall system performance can increase as a more compact program can provide some or all of the following:

- Fewer instruction-cache misses
- Fewer TLB misses for pages of application code
- Fewer page faults for application code

Other influences on code size

In addition to compiler options, there are a number of ways programming and analysis can influence the size of your source code.

High activity areas

Once you apply the techniques discussed earlier in this section, your strategy for further code size reduction depends on your objective. Use profiling tools to locate hot spots in your program; then follow one of the following guidelines:

- If you want to reduce code size to reduce program paging, concentrate on minimizing branches and procedure references within those hot spots.
- If you want to reduce code size to reduce the size of your program's files on disk, concentrate on areas that are *not* hot spots. Remove any expansive optimizations from code that does not contain hot spots.

Computed GOTOs and CASE constructs

A sparse computed **GOTO** can increase code size considerably. In a sparse computed **GOTO**, most statement labels point to the default. Consider the following example where label `10` is the default:

```
      GOTO (10,10,10,10,20,10,10,10,10,30,20,10,10,10,10,
     +10,20,10,20,10,20,30,30,10,10,10,10,10,10,20,10,10,...
     +10,20,30,10,10,10,30,10,10,10,10,10,10,10,20,10,30) IA(I)

      GOTO 10
30    CONTINUE
      ! ...
      GOTO 10
20    CONTINUE
      ! ...
10    CONTINUE
```

Although fewer cases are shown, the following **CASE** construct is a functionally equivalent to the example above. N is the value of the largest integer that the computed **GOTO** or **CASE** construct is testing.

```
      INTEGER IA(10000)
      SELECT CASE (IA(I))
        CASE DEFAULT
          GOTO 10
        CASE (5)
          GOTO 20
        CASE (10)
          GOTO 30
        CASE (11)
          GOTO 20
        ! ...
        CASE (N-10)
          GOTO 30
        CASE (N-2)
          GOTO 20
        CASE (N)
          GOTO 30
      END SELECT
```

In both examples, the compiler builds a branch table in the object file that contains one entry for each possibility from 1 to N, where N is the largest integer value tested. The data section of the program stores this branch table. If N is very large, the table can increase both the size of the object file and the effects of data-cache misses.

If you use a **CASE** construct with a small number of cases and wide gaps between the test values of the cases, the compiler selects a different algorithm to dispatch to the appropriate location, and the resulting code can be more compact than a functionally equivalent computed **GOTO**. The compiler cannot determine that a computed **GOTO** has a default branch point, so the compiler assumes that any value in the range will be selected. In a **CASE** construct, the compiler assumes that cases you do not specify in the construct are handled as default.

Code size with dynamic or static linking

Dynamic or static linking each affect the size of your code, and the resulting performance of your application.

Dynamic linking and code size

When linking your programs, dynamic linking often ensures more compact code than linking statically. Dynamic linking does not include library procedures in your object file. Instead, a reference at runtime causes the operating system to locate the dynamic library that contains the procedure, and reference that procedure from the library on the system. Only one copy of the procedure is in memory, even if several programs, or copies of a single program, are accessing the procedure simultaneously. This can reduce paging overhead. However, any libraries your program references must be present in your application's execution environment.

Note that if your program references high performance libraries like BLAS or ESSL, these procedures are dynamically linked to your program by default.

Static linking and code size

Static linking binds library procedures into your application's object file. This can increase the size of your object file. If your program references only a small portion of the procedures available in a library, static linking can eliminate the need to provide the library to your users. However, static linking ties your application to one version of the library which can be detrimental in situations where your application will execute in different environments, such as different levels of the operating system.

Chapter 5. Compiler-friendly programming techniques

Writing compiler-friendly code, with both the optimizer and portability in mind, can be as important to the performance of your application as the compilation options that you specify.

General practices

It is not necessary to hand-optimize your code, as hand-optimizing can introduce unusual constructs that can obscure the intentions of your application from the compiler and limit optimization opportunities.

Large programs can use significant address space resources.

Avoid breaking your program into too many small functions, as this can increase the percentage of time the program spends in dealing with call overhead. If you choose to use many small functions, compiling with **-qipa** can help minimize the impact on performance. Attempting to optimize an application with many small functions without the benefit of **-qipa** can severely limit the scope of other optimizations.

Use command invocations like **bgxlf90** and **bgxlf95**, which use **-qnosave**. The **-qnosave** option sets the default storage class of all variables to automatic. This provides more opportunities for optimization. All compiler command invocations except **bgf77**, **bgfort77**, **bgxlf** and **bgxlf_r** use **-qnosave** by default.

Use modules to group related subroutines and functions.

Use module variables instead of common blocks for global storage.

Mark all code that accesses or manipulates data objects by independent I/O processes and independent, asynchronously interrupting processes as **VOLATILE**. For example, mark code that accesses shared variables and pointers to shared variables. Mark your code carefully however, as **VOLATILE** is a barrier to optimization as accessing a **VOLATILE** object forces the compiler to always load the value from storage. This prevents powerful optimizations such as constant propagation or invariant code motion.

The XL compilers support high performance libraries that can provide significant advantages over custom implementations or generic libraries.

Variables and pointers

The effective use of aliasing and of variables and pointers provides opportunities for improved performance and further optimization.

Obey all aliasing rules. Avoid specifying **-qalias=nostd**. For more information on aliasing and how it can affect performance, see “Aliasing” on page 41.

Avoid unnecessary use of global variables and pointers, including module variables and common blocks. When using global variables and pointers in a loop, load them into a local variable before the loop and store them back after. If you do not use the local variable somewhere other than in the loop body, the optimization process can usually recognize what you are doing and expose more optimization opportunities. Replacing a global variable in a loop with a local variable reduces the possibilities for aliasing.

Use the **INTENT** statement to describe the usage of dummy arguments.

Limit the use of **ALLOCATABLE** objects and **POINTER** variables to situations demanding dynamic memory allocation.

Arrays

Where possible, use local variables instead of global variables for loop index variables and bounds.

Whenever possible, ensure references to arrays or array sections refer to contiguous blocks of storage. Noncontiguous memory array references, when passed as parameters, lead to copy-in and copy-out operations.

▶ F2008 When declaring an array pointer or an assumed-shape array, you can use the **CONTIGUOUS** attribute to ensure that the array elements in order are stored in contiguous memory and not separated by other data objects. An array pointer with the **CONTIGUOUS** attribute can only be pointer associated with a contiguous target. An assumed-shape array with the **CONTIGUOUS** attribute is always contiguous; however, the corresponding actual argument can be contiguous or noncontiguous. If it is noncontiguous, the compiler makes it contiguous by creating a temporary contiguous argument. When the **CONTIGUOUS** attribute is used, the compiler can perform appropriate semantic check and detect invalid codes, which helps you write more optimized codes and enables the compiler to further optimize the runtime performance and storage layout. F2008 ◀

Keep your array expressions simple so that the optimizer can deduce access patterns more easily and reuse index calculations in whole or in part.

Frequent use of array-to-array assignment and **WHERE** constructs can impact performance by increasing temporary storage and creating loops. Using **-qlist** or **-qreport** can help you understand the performance characteristics of your code, and where applying **-qhot** could be beneficial. If you are already optimizing with **-qipa**, ensure you are using the **list=***filename* option, so that the **-qlist** listing file is not overwritten.

Related information

- ▶ F2008 The CONTIGUOUS attribute F2008 ◀

Choosing appropriate variable sizes

Improve the efficiency of your application by choosing the appropriate variable sizes.

When programming SMP applications, use the **CONTAINS** statement only to share thread local storage.

In most cases using **INTEGER(8)** for scalars improves the efficiency of DO loops, subscripting, mathematical calculations and calling conventions when passing objects. However, if your code contains large arrays with values that can fit in an **INTEGER(4)**, using smaller kind parameters can actually improve memory efficiency by reducing memory traffic to load or store data.

Use the lowest floating-point precision appropriate to your application. Higher precisions can reduce performance, so use the **REAL(16)**, or **COMPLEX(16)** data types only when you require extremely high precision.

On systems with QPX, **-qsimd=auto** provides opportunities for vectorization on **REAL(8)** types.

Chapter 6. High performance libraries

XL Fortran is shipped with a set of libraries for high-performance mathematical computing.

The set of libraries for high-performance mathematical computing are:

- The Mathematical Acceleration Subsystem (MASS) is a set of libraries of tuned mathematical intrinsic routines that provide improved performance over the corresponding standard system math library routines. MASS is described in "Using the Mathematical Acceleration Subsystem libraries (MASS)."
- The Basic Linear Algebra Subprograms (BLAS) are a subset of routines from IBM's Engineering and Scientific Subroutine Library (ESSL) library, which provides matrix/vector multiplication functions tuned for Blue Gene architectures. The BLAS functions are described in "Using the Basic Linear Algebra Subprograms – BLAS" on page 65.

Note that if you are going to link your application with the ESSL libraries, using **-qessl** and IPA allows the optimizer to automatically use ESSL routines.

Using the Mathematical Acceleration Subsystem libraries (MASS)

XL Fortran is shipped with a set of Mathematical Acceleration Subsystem (MASS) libraries for high-performance mathematical computing.

The MASS libraries consist of a library of scalar Fortran routines described in "Using the scalar library" on page 56, a set of vector libraries tuned for specific architectures described in "Using the vector libraries" on page 58, and a SIMD library described in "Using the SIMD library" on page 62. The functions contained in both scalar and vector libraries are automatically called at certain levels of optimization, but you can also call them explicitly in your programs. Note that the accuracy and exception handling might not be identical in MASS functions and system library functions.

The MASS functions must run with the default rounding mode and floating-point exception trapping settings.

When you compile programs with any of the following sets of options:

- **-qhot -qnostrict**
- **-qhot -O3**
- **-O4**
- **-O5**

the compiler automatically attempts to vectorize calls to system math functions by calling the equivalent MASS vector functions (with the exceptions of functions `vatan2`, `vsatan2`, `vdnint`, `vdint`, `vcosisin`, `vscosisin`, `vqdrt`, `vsqdrt`, `vrqdrt`, `vsrqdrt`, `vpopcnt4`, `vpopcnt8`, `vexp2`, `vexp2m1`, `vsexp2`, `vsexp2m1`, `vlog2`, `vlog21p`, `vslog2`, and `vslog21p`). If it cannot vectorize, it automatically tries to call the equivalent MASS scalar functions. For automatic vectorization or scalarization, the compiler uses versions of the MASS functions contained in the XLOPT library `libxlopt.a`.

In addition to any of the preceding sets of options, when the **-qipa** option is in effect, if the compiler cannot vectorize, it tries to inline the MASS scalar functions before deciding to call them.

"Compiling and linking a program with MASS" on page 64 describes how to compile and link a program that uses the MASS libraries, and how to selectively use the MASS scalar library functions in conjunction with the regular system libraries.

Related external information

Mathematical Acceleration Subsystem website, available at http://www.ibm.com/software/awdtools/mass/

Using the scalar library

The MASS scalar library `libmass.a` contains an accelerated set of frequently used math intrinsic functions that provide improved performance over the corresponding standard system library functions. The MASS scalar functions are used when explicitly linking `libmass.a`.

If you want to explicitly call the MASS scalar functions, you can take the following steps:

1. Link the MASS scalar library libmass.a with your application. For instructions, see "Compiling and linking a program with MASS" on page 64
2. All the MASS scalar routines, except those listed in step 3 are recognized by XL Fortran as intrinsic functions, so no explicit interface block is needed. To provide an interface block for the functions listed in step 3, include `mass.include` in your source file.
3. Include `mass.include` in your source file for the following functions:
   - `acosf`, `acosh`, `acoshf`, `asinf`, `asinh`, `asinhf`, `atan2f`, `atanf`, `atanh`, `atanhf`, `cbrt`, `cbrtf`, `copysign`, `copysignf`, `cosf`, `coshf`, `cosisin`, `erff`, `erfcf`, `expf`, `expm1f`, `hypot`, `hypotf`, `lgammaf`, `logf`, `log10f`, `log1pf`, `rsqrt`, `sinf`, `sincos`, `sinhf`, `tanf`, `tanhf`, and `x**y`

The MASS scalar functions accept double-precision parameters and return a double-precision result, or accept single-precision parameters and return a single-precision result, except `sincos` which gives 2 double-precision results. They are summarized in Table 13.

Table 13. MASS scalar functions

Double-precision function	Single-precision function	Arguments	Description
acos	acosf	(x)	Returns the arccosine of x
acosh	acoshf	(x)	Returns the hyperbolic arccosine of x
	anint	(x)	Returns the rounded integer value of x
asin	asinf	(x)	Returns the arcsine of x
asinh	asinhf	(x)	Returns the hyperbolic arcsine of x
atan2	atan2f	(x,y)	Returns the arctangent of x/y
atan	atanf	(x)	Returns the arctangent of x
atanh	atanhf	(x)	Returns the hyperbolic arctangent of x
cbrt	cbrtf	(x)	Returns the cube root of x

Table 13. MASS scalar functions (continued)

Double-precision function	Single-precision function	Arguments	Description
copysign	copysignf	(x,y)	Returns x with the sign of y
cos	cosf	(x)	Returns the cosine of x
cosh	coshf	(x)	Returns the hyperbolic cosine of x
cosisin		(x)	Returns a complex number with the real part the cosine of x and the imaginary part the sine of x.
dnint		(x)	Returns the nearest integer to x (as a double)
erf	erff	(x)	Returns the error function of x
erfc	erfcf	(x)	Returns the complementary error function of x
exp	expf	(x)	Returns the exponential function of x
expm1	expm1f	(x)	Returns (the exponential function of x) - 1
hypot	hypotf	(x,y)	Returns the square root of $x^2 + y^2$
lgamma	lgammaf	(x)	Returns the natural logarithm of the absolute value of the Gamma function of x
log	logf	(x)	Returns the natural logarithm of x
log10	log10f	(x)	Returns the base 10 logarithm of x
log1p	log1pf	(x)	Returns the natural logarithm of (x + 1)
rsqrt		(x)	Returns the reciprocal of the square root of x
sin	sinf	(x)	Returns the sine of x
sincos		(x,s,c)	Sets s to the sine of x and c to the cosine of x
sinh	sinhf	(x)	Returns the hyperbolic sine of x
sqrt		(x)	Returns the square root of x
tan	tanf	(x)	Returns the tangent of x
tanh	tanhf	(x)	Returns the hyperbolic tangent of x
x**y		(x,y)	Returns x raised to the power y

The following example shows the XL Fortran interface declaration for the `rsqrt` scalar function:

```
interface

real*8 function rsqrt (%val(x))
  real*8 x      ! Returns the reciprocal of the square root of x.
end function rsqrt

end interface
```

Notes:

- The trigonometric functions (`sin`, `cos`, `tan`) return NaN (Not-a-Number) for large arguments (where the absolute value is greater than 2^{50}pi).

- In some cases, the MASS functions are not as accurate as the `libm.a` library, and they might handle edge cases differently (`sqrt(Inf)`, for example).
- See the *Mathematical Acceleration Subsystem website* for accuracy comparisons with `libm.a`.

Related external information

Mathematical Acceleration Subsystem website, available at http://www.ibm.com/software/awdtools/mass/

Using the vector libraries

If you want to explicitly call any of the MASS vector functions, you can do so by including `massv.include` in your source files and linking your application with the appropriate vector library. (Information about linking is provided in "Compiling and linking a program with MASS" on page 64.)

libmassv.a
Contains functions that have been tuned for the Blue Gene/Q architecture.

The single-precision and double-precision floating-point functions contained in the vector libraries are summarized in Table 14 on page 59. The integer functions contained in the vector libraries are summarized in Table 15 on page 60.

With the exception of a few functions (described in the following paragraph), all of the floating-point functions in the vector libraries accept three arguments:

- A double-precision (for double-precision functions) or single-precision (for single-precision functions) vector output argument.
- A double-precision (for double-precision functions) or single-precision (for single-precision functions) vector input argument.
- An integer vector-length argument.

The functions are of the form

```
function_name (y,x,n)
```

where *y* is the target vector, *x* is the source vector, and *n* is the vector length. The arguments *y* and *x* are assumed to be double-precision for functions with the prefix `v`, and single-precision for functions with the prefix `vs`. As an example, the following code:

```
include 'massv.include'

real*8 x(500), y(500)
integer n
n = 500
...
call vexp (y, x, n)
```

outputs a vector *y* of length 500 whose elements are exp(x(i)), where i=1,...,500.

The functions `vdiv`, `vsincos`, `vpow`, and `vatan2` (and their single-precision versions, `vsdiv`, `vssincos`, `vspow`, and `vsatan2`) take four arguments. The functions `vdiv`, `vpow`, and `vatan2` take the arguments (*z*,*x*,*y*,*n*). The function `vdiv` outputs a vector *z* whose elements are x(i)/y(i), where i=1,...,n. The function `vpow` outputs a vector *z* whose elements are $x(i)^{y(i)}$, where i=1,..,n. The function `vatan2` outputs a vector *z* whose elements are atan(x(i)/y(i)), where i=1,..,n. The function `vsincos` takes the arguments (*y*,*z*,*x*,*n*), and outputs two vectors, *y* and *z*, whose elements are sin(x(i)) and cos(x(i)), respectively.

In `vcosisin(y,x,n)` and `vscosisin(y,x,n)`, *x* is a vector of *n* elements and the function outputs a vector *y* of *n* `complex(8)(for vcosisin)` or `complex(4)(for vscosisin)` elements of the form (cos(x(i)),sin(x(i))).

Table 14. MASS floating-point vector library functions

Double-precision function	Single-precision function	Arguments	Description
vacos	vsacos	`(y,x,n)`	Sets y(i) to the arc cosine of x(i), for i=1,..,n
vacosh	vsacosh	`(y,x,n)`	Sets y(i) to the hyperbolic arc cosine of x(i), for i=1,..,n
vasin	vsasin	`(y,x,n)`	Sets y(i) to the arc sine of x(i), for i=1,..,n
vasinh	vsasinh	`(y,x,n)`	Sets y(i) to the arc hyperbolic sine of x(i), for i=1,..,n
vatan2	vsatan2	`(z,x,y,n)`	Sets z(i) to the arc tangent of x(i)/y(i), for i=1,..,n
vatanh	vsatanh	`(y,x,n)`	Sets y(i) to the arc hyperbolic tangent of x(i), for i=1,..,n
vcbrt	vscbrt	`(y,x,n)`	Sets y(i) to the cube root of x(i), for i=1,..,n
vcos	vscos	`(y,x,n)`	Sets y(i) to the cosine of x(i), for i=1,..,n
vcosh	vscosh	`(y,x,n)`	Sets y(i) to the hyperbolic cosine of x(i), for i=1,..,n
vcosisin	vscosisin	`(y,x,n)`	Sets the real part of y(i) to the cosine of x(i) and the imaginary part of y(i) to the sine of x(i), for i=1,..,n
vdint		`(y,x,n)`	Sets y(i) to the integer truncation of x(i), for i=1,..,n
vdiv	vsdiv	`(z,x,y,n)`	Sets z(i) to x(i)/y(i), for i=1,..,n
vdiv_fast (1)	vsdiv_fast (2)	`(z,x,y,n)`	Sets z(i) to x(i)/y(i), for i=1,..,n.
vdnint		`(y,x,n)`	Sets y(i) to the nearest integer to x(i), for i=1,..,n
verf	vserf	`(y,x,n)`	Sets y(i) to the error function of x(i), for i=1,..,n
verfc	vserfc	`(y,x,n)`	Sets y(i) to the complimentary error function of x(i), for i=1,..,n
vexp	vsexp	`(y,x,n)`	Sets y(i) to the exponential function of x(i), for i=1,..,n
vexp2	vsexp2	`(y,x,n)`	Sets y(i) to 2 raised to the power of x(i), for i=1,..,n
vexpm1	vsexpm1	`(y,x,n)`	Sets y(i) to (the exponential function of x(i)) -1, for i=1,..,n
vexp2m1	vsexp2m1	`(y,x,n)`	Sets y(i) to (2 raised to the power of x(i)) -1, for i=1,..,n
vhypot	vshypot	`(z,x,y,n)`	Sets z(i) to the square root of the sum of the squares of x(i) and y(i), for i=1,..,n
vlog	vslog	`(y,x,n)`	Sets y(i) to the natural logarithm of x(i), for i=1,..,n
vlog2	vslog2	`(y,x,n)`	Sets y(i) to the base-2 logarithm of x(i), for i=1,..,n
vlog10	vslog10	`(y,x,n)`	Sets y(i) to the base-10 logarithm of x(i), for i=1,..,n
vlog1p	vslog1p	`(y,x,n)`	Sets y(i) to the natural logarithm of (x(i)+1), for i=1,..,n
vlog21p	vslog21p	`(y,x,n)`	Sets y(i) to the base-2 logarithm of (x(i)+1), for i=1,..,n
vpow	vspow	`(z,x,y,n)`	Sets z(i) to x(i) raised to the power y(i), for i=1,..,n
vqdrt	vsqdrt	`(y,x,n)`	Sets y(i) to the 4th root of x(i), for i=1,..,n
vrcbrt	vsrcbrt	`(y,x,n)`	Sets y(i) to the reciprocal of the cube root of x(i), for i=1,..,n
vrec	vsrec	`(y,x,n)`	Sets y(i) to the reciprocal of x(i), for i=1,..,n

Table 14. MASS floating-point vector library functions (continued)

Double-precision function	Single-precision function	Arguments	Description
vrec_fast [(3)]	vsrec_fast [(4)]	`(y,x,n)`	Sets y(i) to the reciprocal of x(i), for i=1,..,n
vrqdrt	vsrqdrt	`(y,x,n)`	Sets y(i) to the reciprocal of the 4th root of x(i), for i=1,..,n
vrsqrt	vsrsqrt	`(y,x,n)`	Sets y(i) to the reciprocal of the square root of x(i), for i=1,..,n
vsin	vssin	`(y,x,n)`	Sets y(i) to the sine of x(i), for i=1,..,n
vsincos	vssincos	`(y,z,x,n)`	Sets y(i) to the sine of x(i) and z(i) to the cosine of x(i), for i=1,..,n
vsinh	vssinh	`(y,x,n)`	Sets y(i) to the hyperbolic sine of x(i), for i=1,..,n
vsqrt	vssqrt	`(y,x,n)`	Sets y(i) to the square root of x(i), for i=1,..,n
vtan	vstan	`(y,x,n)`	Sets y(i) to the tangent of x(i), for i=1,..,n
vtanh	vstanh	`(y,x,n)`	Sets y(i) to the hyperbolic tangent of x(i), for i=1,..,n

Notes:

1. `vdiv_fast` arguments must satisfy all the following conditions for `i=1,...,n`:
   - $2^{-1021} \le |y(i)| \le 2^{1020}$
   - If `x(i)` is not zero, $2^{-969} \le |x(i)| < \infty$ and $2^{-1020} \le |x(i)/y(i)| \le 2^{1022}$
2. `vsdiv_fast` arguments must satisfy all the following conditions for `i=1,...,n`:
   - $2^{-125} \le |y(i)| \le 2^{124}$
   - If `x(i)` is not zero, $2^{-102} \le |x(i)| < \infty$ and $2^{-124} \le |x(i)/y(i)| \le 2^{126}$
3. `vrec_fast` arguments must satisfy the following condition for `i=1,...,n`:
   - $2^{-1021} \le |x(i)| \le 2^{1020}$
4. `vsrec_fast` arguments must satisfy the following condition for `i=1,...,n`:
   - $2^{-125} \le |x(i)| \le 2^{124}$

Integer functions are of the form *function_name* (*x*, *n*), where *x* is a vector of 4-byte (for `vpopcnt4`) or 8-byte (for `vpopcnt8`) numeric objects (integer or floating-point), and *n* is the vector length.

Table 15. MASS integer vector library functions

Function	Description	Interface

vpopcnt4	Returns the total number of 1 bits in the concatenation of the binary representation of x(i), for i=1,...,n, where `x` is vector of 32-bit objects	`integer*4 function vpopcnt4 (x, n)`
		`integer*4 x(*), n`

vpopcnt8	Returns the total number of 1 bits in the concatenation of the binary representation of x(i), for i=1,...,n, where `x` is vector of 64-bit objects	`integer*4 function vpopcnt8 (x, n)`
		`integer*8 x(*)`
		`integer*4 n`

The following example shows XL Fortran interface declarations for some of the MASS double-precision vector routines:

```
interface

subroutine vsqrt (y, x, n)
  real*8 y(*), x(*)
  integer n        ! Sets y(i) to the square root of x(i), for i=1,..,n
end subroutine vsqrt
```

```
subroutine vrsqrt (y, x, n)
  real*8 y(*), x(*)
  integer n       ! Sets y(i) to the reciprocal of the square root of x(i),
                  ! for i=1,..,n
end subroutine vrsqrt

end interface
```

The following example shows XL Fortran interface declarations for some of the MASS single-precision vector routines:

```
interface

subroutine vssqrt (y, x, n)
  real*4 y(*), x(*)
  integer n       ! Sets y(i) to the square root of x(i), for i=1,..,n
end subroutine vssqrt

subroutine vsrsqrt (y, x, n)
  real*4 y(*), x(*)
  integer n       ! Sets y(i) to the reciprocal of the square root of x(i),
                  ! for i=1,..,n
end subroutine vsrsqrt

end interface
```

Overlap of input and output vectors

In most applications, the MASS vector functions are called with disjoint input and output vectors; that is, the two vectors do not overlap in memory. Another common usage scenario is to call them with the same vector for both input and output parameters (for example, `vsin (y, y, n)`). Other kinds of overlap (where input and output vectors are neither disjoint nor identical) should be avoided, since they may produce unexpected results:

- For calls to vector functions that take one input and one output vector (for example, `vsin (y, x, n)`):

  The vectors `x(1:n)` and `y(1:n)` must be either disjoint or identical, or unexpected results may be obtained.
- For calls to vector functions that take two input vectors (for example, `vatan2 (y, x1, x2, n)`):

  The previous restriction applies to both pairs of vectors `y`,`x1` and `y`,`x2`. That is, `y(1:n)` and `x1(1:n)` must be either disjoint or identical; and `y(1:n)` and `x2(1:n)` must be either disjoint or identical.
- For calls to vector functions that take two output vectors (for example, `vsincos (y1, y2, x, n)`):

  The above restriction applies to both pairs of vectors `y1`,`x` and `y2`,`x`. That is, `y1(1:n)` and `x(1:n)` must be either disjoint or identical; and `y2(1:n)` and `x(1:n)` must be either disjoint or identical. Also, the vectors `y1(1:n)` and `y2(1:n)` must be disjoint.

Alignment of input and output vectors

To get the best performance from the vector library, align the input and output vectors as follows:

- 16-byte for single precision
- 32-byte for double precision

Consistency of MASS vector functions

All the functions in the MASS vector libraries are consistent, in the sense that a given input value will always produce the same result, regardless of its position in the vector, and regardless of the vector length.

Related external information

Mathematical Acceleration Subsystem website, available at http://www.ibm.com/software/awdtools/mass/

Using the SIMD library

The MASS SIMD library libmass_simd.a contains a set of frequently used math intrinsic functions that provide improved performance over the corresponding standard system library functions. If you want to use the MASS SIMD functions, you can do so as follows:

1. Provide the interfaces for the functions by including mass_simd.include in your source files.
2. Link the MASS SIMD library libmass_simd.a with your application. For instructions, see "Compiling and linking a program with MASS" on page 64.

The single/double-precision MASS SIMD functions accept single/double-precision arguments and return single/double-precision results. They are summarized in Table 16.

Table 16. MASS SIMD functions

Double-precision function	Single-precision function	Description	Double-precision function interface	Single-precision function interface
acosd4	acosf4	Computes the arc cosine of each element of vx.	vector(real(8)) function acosd4(vx) vector(real(8)), value :: vx	vector(real(8)) function acosf4(vx) vector(real(8)), value :: vx
acoshd4	acoshf4	Computes the arc hyperbolic cosine of each element of vx.	vector(real(8)) function acoshd4(vx) vector(real(8)), value :: vx	vector(real(8)) function acoshf4(vx) vector(real(8)), value :: vx
asind4	asinf4	Computes the arc sine of each element of vx.	vector(real(8)) function asind4(vx) vector(real(8)), value :: vx	vector(real(8)) function asinf4(vx) vector(real(8)), value :: vx
asinhd4	asinhf4	Computes the arc hyperbolic sine of each element of vx.	vector(real(8)) function asinhd4(vx) vector(real(8)), value :: vx	vector(real(8)) function asinhf4(vx) vector(real(8)), value :: vx
atand4	atanf4	Computes the arc tangent of each element of vx.	vector(real(8)) function atand4(vx) vector(real(8)), value :: vx	vector(real(8)) function atanf4(vx) vector(real(8)), value :: vx
atan2d4	atan2f4	Computes the arc tangent of each element of vx/vy.	vector(real(8)) function atan2d4(vx,vy) vector(real(8)), value :: vx, vy	vector(real(8)) function atan2f4(vx,vy) vector(real(8)), value :: vx, vy
atanhd4	atanhf4	Computes the arc hyperbolic tangent of each element of vx.	vector(real(8)) function atanhd4(vx) vector(real(8)), value :: vx	vector(real(8)) function atanhf4(vx) vector(real(8)), value :: vx
cbrtd4	cbrtf4	Computes the cube root of each element of vx	vector(real(8)) function cbrtd4(vx) vector(real(8)), value :: vx	vector(real(8)) function cbrtf4(vx) vector(real(8)), value :: vx
cosd4	cosf4	Computes the cosine of each element of vx.	vector(real(8)) function cosd4(vx) vector(real(8)), value :: vx	vector(real(8)) function cosf4(vx) vector(real(8)), value :: vx
coshd4	coshf4	Computes the hyperbolic cosine of each element of vx.	vector(real(8)) function coshd4(vx) vector(real(8)), value :: vx	vector(real(8)) function coshf4(vx) vector(real(8)), value :: vx
cosisind4	cosisinf4	Computes the cosine and sine of each element of x, and stores the results in y and z as follows: Sets the elements of y to cos(x1), sin(x1), cos(x2), sin(x2), and the elements of z to cos(x3), sin(x3), cos(x4), sin(x4), where x1, x2, x3, x4 are the elements of x.	subroutine cosisind4 (x, y, z) vector(real(8)), value :: x vector(real(8)) y, z	subroutine cosisinf4 (x, y, z) vector(real(8)), value :: x vector(real(8)) y, z
divd4	divf4	Computes the quotient vx/vy.	vector(real(8)) function divd4(vx, vy) vector(real(8)), value :: vx, vy	vector(real(8)) function divf4(vx, vy) vector(real(8)), value :: vx, vy
div_fastd4 (1)	div_fastf4 (2)	Computes the quotient vx/vy.	vector(real(8)) function div_fastd4(vx, vy) vector(real(8)), value :: vx, vy	vector(real(8)) function div_fastf4(vx, vy) vector(real(8)), value :: vx, vy
erfcd4	erfcf4	Computes the complementary error function of each element of vx.	vector(real(8)) function erfcd4(vx) vector(real(8)), value :: vx	vector(real(8)) function erfcf4(vx) vector(real(8)), value :: vx
erfd4	erff4	Computes the error function of each element of vx.	vector(real(8)) function erfd4(vx) vector(real(8)), value :: vx	vector(real(8)) function erff4(vx) vector(real(8)), value :: vx

Table 16. MASS SIMD functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function interface	Single-precision function interface
expd4	expf4	Computes the exponential function of each element of vx.	vector(real(8)) function expd4(vx) vector(real(8)), value :: vx	vector(real(8)) function expf4(vx) vector(real(8)), value :: vx
exp2d4	exp2f4	Computes 2 raised to the power of each element of vx.	vector(real(8)) function exp2d4(vx) vector(real(8)), value :: vx	vector(real(8)) function exp2f4(vx) vector(real(8)), value :: vx
expm1d4	expm1f4	Computes (the exponential function of each element of vx) - 1.	vector(real(8)) function expm1d4(vx) vector(real(8)), value :: vx	vector(real(8)) function exp2m1f4(vx) vector(real(8)), value :: vx
exp2m1d4	exp2m1f4	Computes (2 raised to the power of each element of vx) - 1.	vector(real(8)) function exp2m1d4(vx) vector(real(8)), value :: vx	vector(real(8)) function exp2m1f4(vx) vector(real(8)), value :: vx
hypotd4	hypotf4	For each element of vx and the corresponding element of vy, computes sqrt(vx*vx +vy*vy).	vector(real(8)) function hypotd4(vx,vy) vector(real(8)), value :: vx, vy	vector(real(8)) function hypotf4(vx,vy) vector(real(8)), value :: vx, vy
lgammad4	lgammaf4	Computes the natural logarithm of the absolute value of the Gamma function of each element of vx .	vector(real(8)) function lgammad4(vx) vector(real(8)), value :: vx	vector(real(8)) function lgammaf4(vx) vector(real(8)), value :: vx
logd4	logf4	Computes the natural logarithm of each element of vx.	vector(real(8)) function logd4(vx) vector(real(8)), value :: vx	vector(real(8)) function logf4(vx) vector(real(8)), value :: vx
log2d4	log2f4	Computes the base-2 logarithm of each element of vx.	vector(real(8)) function log2d4(vx) vector(real(8)), value :: vx	vector(real(8)) function log2f4(vx) vector(real(8)), value :: vx
log10d4	log10f4	Computes the base-10 logarithm of each element of vx.	vector(real(8)) function log10d4(vx) vector(real(8)), value :: vx	vector(real(8)) function log10f4(vx) vector(real(8)), value :: vx
log1pd4	log1pf4	Computes the natural logarithm of each element of (vx +1).	vector(real(8)) function log1pd4(vx) vector(real(8)), value :: vx	vector(real(8)) function log1pf4(vx) vector(real(8)), value :: vx
log21pd4	log21pf4	Computes the base-2 logarithm of each element of (vx +1).	vector(real(8)) function log21pd4(vx) vector(real(8)), value :: vx	vector(real(8)) function log21pf4(vx) vector(real(8)), value :: vx
powd4	powf4	Computes each element of vx raised to the power of the corresponding element of vy.	vector(real(8)) function powd4(vx, vy) vector(real(8)), value :: vx, vy	vector(real(8)) function powf4(vx, vy) vector(real(8)), value :: vx, vy
qdrtd4	qdrtf4	Computes the quad root of each element of vx.	vector(real(8)) function qdrtd4(vx) vector(real(8)), value :: vx	vector(real(8)) function qdrtf4(vx) vector(real(8)), value :: vx
rcbrtd4	rcbrtf4	Computes the reciprocal of the cube root of each element of vx.	vector(real(8)) function rcbrtd4(vx) vector(real(8)), value :: vx	vector(real(8)) function rcbrtf4(vx) vector(real(8)), value :: vx
recipd4	recipf4	Computes the reciprocal of each element of vx.	vector(real(8)) function recipd4(vx) vector(real(8)), value :: vx	vector(real(8)) function recipf4(vx) vector(real(8)), value :: vx
recip_fastd4 (3)	recip_fastf4 (4)	Computes the reciprocal of each element of vx.	vector(real(8)) function recip_fastd4(vx) vector(real(8)), value :: vx	vector(real(8)) function recip_fastf4(vx) vector(real(8)), value :: vx
rqdrtd4	rqdrtf4	Computes the reciprocal of the quad root of each element of vx.	vector(real(8)) function rqdrtd4(vx) vector(real(8)), value :: vx	vector(real(8)) function rqdrtf4(vx) vector(real(8)), value :: vx
rsqrtd4	rsqrtf4	Computes the reciprocal of the square root of each element of vx.	vector(real(8)) function rsqrtd4(vx) vector(real(8)), value :: vx	vector(real(8)) function rsqrtf4(vx) vector(real(8)), value :: vx
sincosd4	sincosf4	Computes the sine and cosine of each element of vx.	subroutine sincosd4(vx, vs, vc) vector(real(8)), value :: vx vector(real(8)) vs, vc	subroutine sincosf4(vx, vs, vc) vector(real(8)), value :: vx vector(real(8)) vs, vc
sind4	sinf4	Computes the sine of each element of vx.	vector(real(8)) function sind4(vx) vector(real(8)), value :: vx	vector(real(8)) function sinf4(vx) vector(real(8)), value :: vx
sinhd4	sinhf4	Computes the hyperbolic sine of each element of vx.i	vector(real(8)) function sinhd4(vx) vector(real(8)), value :: vx	vector(real(8)) function sinhf4(vx) vector(real(8)), value :: vx
sqrtd4	sqrtf4	Computes the square root of each element of vx.	vector(real(8)) function sqrtd4(vx) vector(real(8)), value :: vx	vector(real(8)) function sqrtf4(vx) vector(real(8)), value :: vx
tand4	tanf4	Computes the tangent of each element of vx.	vector(real(8)) function tand4(vx) vector(real(8)), value :: vx	vector(real(8)) function tanf4(vx) vector(real(8)), value :: vx
tanhd4	tanhf4	Computes the hyperbolic tangent of each element of vx.	vector(real(8)) function tanhd4(vx) vector(real(8)), value :: vx	vector(real(8)) function tanhf4(vx) vector(real(8)), value :: vx

Notes:

1. `div_fastd4` arguments must satisfy all the following conditions for each element `xi` of `x` and `yi` of `y`:
   - $2^{-1021} \le |yi| \le 2^{1020}$
   - If `xi` is not zero, $2^{-969} \le |xi| < \infty$ and $2^{-1020} \le |xi/yi| \le 2^{1022}$
2. `div_fastf4` arguments must satisfy all the following conditions for each element `xi` of `x` and `yi` of `y`:
   - $2^{-125} \le |yi| \le 2^{124}$

- If `xi` is not zero, $2^{-102} \leq |xi| < \infty$ and $2^{-124} \leq |xi/yi| \leq 2^{126}$

3. `recip_fastd4` arguments must satisfy the following condition for each element `xi` of `x`:
   - $2^{-1021} \leq |xi| \leq 2^{1020}$
4. `recip_fastf4` arguments must satisfy the following condition for each element `xi` of `x`:
   - $2^{-125} \leq |xi| \leq 2^{124}$

Compiling and linking a program with MASS

To compile an application that calls the functions in the MASS libraries, specify one or more of the following keywords on the **-l** linker option:

- **mass**
- **massv**
- **mass_simd**

For example, if the MASS libraries are installed in the default directory, you can specify one of the following:

Link with scalar library libmass.a and vector library libmassv.a

```
bgxlf progf.f -o progf -lmass -lmassv
```

Link with SIMD library libmass_simd.a

```
bgxlf progf.f -o progf -lmass_simd
```

Using libmass.a with the math system library

If you want to use the `libmass.a` scalar library for some functions and the normal math library `libm.a` for other functions, follow this procedure to compile and link your program:

1. Use the **ar** command to extract the object files of the desired functions from libmass.a. For most functions, the object file name is the function name followed by `.s64.o`. [1] For example, to extract the object file for the `tan` function, the command would be:
   ```
   ar -x tan.s64.o libmass.a
   ```
2. Archive the extracted object files into another library:
   ```
   ar -qv libfasttan.a tan.s64.o
   ranlib libfasttan.a
   ```
3. Create the final executable using **bgxlf**, specifying **-lfasttan** instead of **-lmass**:
   ```
   bgxlf sample.f -o sample -Ldir_containing_libfasttan -lfasttan
   ```

   This links only the `tan` function from MASS (now in `libfasttan.a`) and the remainder of the math functions from the standard system library.

Exceptions:

1. The `sin` and `cos` functions are both contained in the object file sincos.s64.o. The `cosisin` and `sincos` functions are both contained in the object file cosisin.s64.o.
2. The XL Fortran ** (exponentiation) operator is contained in the object file dxy.s64.o.

Note: The `cos` and `sin` functions will both be exported if either one is exported. `cosisin` and `sincos` will both be exported if either one is exported.

Using the Basic Linear Algebra Subprograms – BLAS

Four Basic Linear Algebra Subprograms (BLAS) functions are shipped with XL Fortran in the `libxlopt` library.

The functions consist of the following:
- SGEMV (single-precision) and DGEMV (double-precision), which compute the matrix-vector product for a general matrix or its transpose
- SGEMM (single-precision) and DGEMM (double-precision), which perform combined matrix multiplication and addition for general matrices or their transposes

Note: Some error-handling code has been removed from the BLAS functions in `libxlopt`, and no error messages are emitted for calls to the these functions.

"BLAS function syntax" describes the interfaces for the XL Fortran BLAS functions, which are similar to those of the equivalent BLAS functions shipped in IBM's Engineering and Scientific Subroutine Library (ESSL); for more detailed information and examples of usage of these functions, you may wish to consult the *Engineering and Scientific Subroutine Library Guide and Reference*, available at the Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL web page.

"Linking the libxlopt library" on page 67 describes how to link to the XL Fortran `libxlopt` library if you are also using a third-party BLAS library.

BLAS function syntax

The interfaces for the SGEMV and DGEMV functions are as follows:

```
CALL SGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
CALL DGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

The parameters are as follows:

trans
is a single character indicating the form of the input matrix *a*, where:
- `'N'` or `'n'` indicates that *a* is to be used in the computation
- `'T'` or `'t'` indicates that the transpose of *a* is to be used in the computation

m represents:
- the number of rows in input matrix *a*
- the length of vector *y*, if `'N'` or `'n'` is used for the *trans* parameter
- the length of vector *x*, if `'T'` or `'t'` is used for the *trans* parameter

The number of rows must be greater than or equal to zero, and less than or equal to the leading dimension of the matrix *a* (specified in *lda*)

n represents:
- the number of columns in input matrix *a*
- the length of vector *x*, if `'N'` or `'n'` is used for the *trans* parameter
- the length of vector *y*, if `'T'` or `'t'` is used for the *trans* parameter

The number of columns must be greater than or equal to zero.

alpha
is the scaling constant α

a is the input matrix of single-precision (for SGEMV) or double-precision (for DGEMV) real values

lda
is the leading dimension of the array specified by *a*. The number of rows must be greater than or equal to zero, and less than the leading dimension of the matrix *a* (specified in *lda*).

x is the input vector of single-precision (for SGEMV) or double-precision (for DGEMV) real values.

incx
is the stride for vector *x*. It can have any value.

beta
is the scaling constant β

y is the output vector of single-precision (for SGEMV) or double-precision (for DGEMV) real values.

incy
is the stride for vector *y*. It must not be zero.

Note: Vector *y* must have no common elements with matrix *a* or vector *x*; otherwise, the results are unpredictable.

The prototypes for the SGEMM and DGEMM functions are as follows:

```
CALL SGEMM(transa, transb, l, n, m, alpha, a, lda, b, ldb, beta, c, ldc)
CALL DGEMM(transa, transb, l, n, m, alpha, a, lda, b, ldb, beta, c, ldc)
```

The parameters are as follows:

transa
is a single character indicating the form of the input matrix *a*, where:
- `'N'` or `'n'` indicates that *a* is to be used in the computation
- `'T'` or `'t'` indicates that the transpose of *a* is to be used in the computation

transb
is a single character indicating the form of the input matrix *b*, where:
- `'N'` or `'n'` indicates that *b* is to be used in the computation
- `'T'` or `'t'` indicates that the transpose of *b* is to be used in the computation

l represents the number of rows in output matrix *c*. The number of rows must be less than or equal to the leading dimension of *c*.

n represents the number of columns in output matrix *c*. The number of columns must be greater than or equal to zero.

m represents:
- the number of columns in matrix *a*, if `'N'` or `'n'` is used for the *transa* parameter
- the number of rows in matrix *a*, if `'T'` or `'t'` is used for the *transa* parameter

and:
- the number of rows in matrix *b*, if `'N'` or `'n'` is used for the *transb* parameter
- the number of columns in matrix *b*, if `'T'` or `'t'` is used for the *transb* parameter

m must be greater than or equal to zero.

alpha
: is the scaling constant α

a
: is the input matrix *a* of single-precision (for SGEMM) or double-precision (for DGEMM) real values

lda
: is the leading dimension of the array specified by *a*. The leading dimension must be greater than zero. If *transa* is specified as `'N'` or `'n'`, the leading dimension must be greater than or equal to 1. If *transa* is specified as `'T'` or `'t'`, the leading dimension must be greater than or equal to the value specified in *m*.

b
: is the input matrix *b* of single-precision (for SGEMM) or double-precision (for DGEMM) real values.

ldb
: is the leading dimension of the array specified by *b*. The leading dimension must be greater than zero. If *transb* is specified as `'N'` or `'n'`, the leading dimension must be greater than or equal to the value specified in *m*. If *transa* is specified as `'T'` or `'t'`, the leading dimension must be greater than or equal to the value specified in *n*.

beta
: is the scaling constant β

c
: is the output matrix *c* of single-precision (for SGEMM) or double-precision (for DGEMM) real values.

ldc
: is the leading dimension of the array specified by *c*. The leading dimension must be greater than zero. If *transb* is specified as `'N'` or `'n'`, the leading dimension must be greater than or equal to the value specified in *l*.

Note: Matrix *c* must have no common elements with matrices *a* or *b*; otherwise, the results are unpredictable.

Linking the libxlopt library

By default, the `libxlopt` library is linked with any application you compile with XL Fortran. However, if you are using a third-party BLAS library, but want to use the BLAS routines shipped with `libxlopt`, you must specify the `libxlopt` library before any other BLAS library on the command line at link time. For example, if your other BLAS library is called `libblas`, you would compile your code with the following command:

```
bgxlf app.f -lxlopt -lblas
```

The compiler will call the SGEMV, DGEMV, SGEMM, and DGEMM functions from the `libxlopt` library, and all other BLAS functions in the `libblas` library.

Chapter 7. Parallel programming with XL Fortran

Parallel programming with XL Fortran involves a combination of compiling, setting of runtime options, and optimization of your code, by incorporating SMP directives and by using the pthreads library module.

XL Fortran supports the OpenMP specification, as understood and interpreted by IBM as well as the POSIX 1003.1-1996 standard, the transactional memory feature, and the thread-level speculative execution feature.

Note: IBM implementation of OpenMP in XL Fortran is the extension to the standard Fortran language.

Thread-level speculative execution

Thread-level speculative execution overcomes the analysis problems of compiler-directed code parallelization.

In compiler-directed code parallelization, the compiler must analyze the code to ensure that the code can be parallelized. However, the C-style pointers and array subscript arithmetic operations hamper the compiler analysis. Furthermore, the compiler generates parallel code only if dependencies can be ruled out; otherwise, it generates sequential code.

The XL Fortran compiler supports the programming paradigm of thread-level speculative execution. Thread-level speculative execution uses hardware support that dynamically detects thread conflicts and rolls back conflicting threads for re-execution. You can get significant performance gains in your applications by adding the compiler directives of thread-level speculative execution to the existing program code.

Thread-level speculative execution is enabled with the "-qsmp=speculative" compiler option.

Rules for committing data

With thread-level speculative execution, tasks are committed according to the following rules:

- Before a task is committed, the data is in a speculative state.
- Tasks are committed in program order.
- Therefore, a later task in program order can only be committed when all the earlier tasks have been committed. If a thread running a task encounters a conflict, all the threads running later tasks must roll back and retry. Eventually, all tasks are committed.

Thread-level speculative execution and OpenMP

All the OpenMP restrictions that apply to the **PARALLEL DO** and **PARALLEL SECTIONS** directives apply to the **SPECULATIVE DO** and **SPECULATIVE SECTIONS** directives.

Speculative threads are not able to detect access conflicts in OpenMP **THREADPRIVATE** data. Accessing such data inside regions of thread-level speculative execution does not guarantee the same behavior as regions being run by one thread.

For the restrictions of the OpenMP loop construct, see the Loop Construct section in the OpenMP specification 3.1.

For the restrictions of the OpenMP sections construct, see the sections Construct section in the OpenMP specification 3.1.

Related information

- The "-qsmp" compiler option
- Routines for thread-level speculative execution
- Environment variables for thread-level speculative execution
- SPECULATIVE DO / END SPECULATIVE DO
- SPECULATIVE SECTIONS
- THREADPRIVATE

Transactional memory

Transactional memory is a model for controlling concurrent memory accesses in the scope of parallel programming.

In parallel programming, concurrency control ensures that threads running in parallel do not update the same resources at the same time. Traditionally, the concurrency control of shared memory data is through locks, for example, mutex locks. A thread acquires a lock before modifying the shared data, and releases the lock afterward. A lock-based synchronization can lead to some performance issues because threads might need to wait to update lock-protected data.

Transactional memory is an alternative to lock-based synchronization. It attempts to simplify parallel programming by grouping read and write operations and running them like a single operation. Transactional memory is like database transactions where all shared memory accesses and their effects are either committed all together or discarded as a group. All threads can enter the critical region simultaneously. If there are conflicts in accessing the shared memory data, threads try accessing the shared memory data again or are stopped without updating the shared memory data. Therefore, transactional memory is also called a lock-free synchronization. Transactional memory can be a competitive alternative to lock-based synchronization.

A transactional memory system must hold the following properties across the entire execution of a concurrent program:

Atomicity
All speculative memory updates of a transaction are either committed or discarded as a unit.

Consistency
The memory operations of a transaction take place in order. Transactions are committed one transaction at a time.

Isolation
Memory updates are not visible outside of a transaction until the transaction commits data.

Transactional memory on Blue Gene/Q

On Blue Gene/Q, the transactional memory model is implemented in the hardware to access all the memory up to the 16 GB boundary.

Transactions are implemented through regions of code that you can designate to be single operations for the system. The regions of code that implement the transactions are called transactional atomic regions.

Transactional memory is enabled with the "-qtm" compiler option, and requires thread safe compilation mode.

Execution modes

When transactional memory is activated on Blue Gene/Q, transactions are run in one of the following operating modes:

- Speculation mode
  - Long running speculation mode (default)
  - Short running speculation mode
- Irrevocable mode

Each mode applies to an entire transactional atomic region.

Speculation mode

Under speculation mode, Kernel address space, devices I/Os, and most memory-mapped I/Os are protected from the irrevocable actions except when the **safe_mode** clause is specified. The transaction goes into irrevocable mode if such an action occurs to guarantee the correct result.

Blue Gene/Q supports two hardware implementation of transaction memory: long and short running speculation mode. If the transactional atomic region is large and many reuses are among the references inside the transaction, it is recommended that you use the default long running speculation mode. Otherwise, use the short running speculation mode.

Irrevocable mode

System calls, irrevocable operations such as I/O operations, and OpenMP constructs trigger transactions to go into irrevocable mode, which serializes transactions. Transactions are also running in irrevocable mode when the maximum number of transaction rollbacks has been reached.

Under irrevocable mode, each memory update of a thread is committed instantaneously instead of at the end of the transaction. Therefore, memory updates are immediately visible to other threads. If the transaction becomes irrevocable, the threads run nonspeculatively.

Using variables and synchronization constructs with transactional memory

The semantics of transactional memory ensure that the effects of transactions of a thread are visible to other threads only after the transactions commit data or become irrevocable. When you use variables or synchronization constructs inside transactions, be careful when the volatile and regular variables that are visible to other threads are updated.

Data races when using transactional memory

A data race might happen if a memory location is accessed concurrently from both the following types of code sections:

- A transactional atomic region that is not nested in other critical sections
- A lock-based critical section of another thread

For example, the atomicity of a lock-based critical section might be broken when the transaction happens in the middle of the critical section. The atomicity of the transaction might also be broken if the transaction becomes irrevocable and is interleaved with the critical section.

The data race happens because each transactional atomic region can be thought of as using a different lock. In contrast, the **!$omp critical** directive uses one lock for all critical regions in the same parallel region.

Related information

- The "-qtm" compiler option
- Routines for transactional memory
- Environment variables for transactional memory
- TM_ATOMIC / END TM_ATOMIC

Profiler for OpenMP

Profiler for OpenMP (POMP) is a profiling mechanism for OpenMP runtime. It inserts callbacks at key points in an OpenMP program; for example, when a parallel region is entered or exited. In these callbacks, you can run your code for various purposes, such as to increment profiling counters, or record timestamps.

For further details about POMP, see the Profiler for OpenMP section in *XL C/C++ Optimization and Programming Guide*.

Compiling your parallelized code

To compile parallelized code, you must specify the **-qsmp** compiler option. When compiling with **-qsmp**, the driver links the libraries found on the **smplibraries** line in the active stanza of your configuration file.

If you specify **-qsmp**, you must use an appropriate invocation command. Use any of the following invocations to compile SMP code or to ensure that the compiler links threadsafe libraries:

- **bgxlf_r**
- **bgxlf90_r**
- **bgxlf95_r**
- **bgxlf2003_r**
- **bgxlf2008_r**

Related reference:

See -qsmp in the Compiler Reference

The _OPENMP C preprocessor macro and conditional compilation

You can use sentinels to mark specific lines of an XL Fortran program for conditional compilation. This allows you to port code that contains statements that are only valid or applicable in an SMP environment to a non-SMP environment. You can do this using conditional compilation lines, or the **_OPENMP** C preprocessor macro. This macro is defined when the C preprocessor is invoked and you specify the **-qsmp=omp** compiler option. See Passing Fortran files through the C preprocessor in the *Editing, Compiling, Linking, and Running XL Fortran Programs* section of the *XL Fortran Compiler Reference* for an example of using this macro.

The following example uses conditional compilation lines to hide OpenMP runtime routines. You cannot easily compile code that calls OpenMP runtime routines in a non-OpenMP environment without using conditional compilation. Since calls to the runtime routines are not directives, they cannot be hidden by the **!$OMP** trigger. If you do not compile the example with **-qsmp=omp**, the variable that stores the number of threads is assigned the value of 8.

Example of conditional compilation lines

```
      PROGRAM PAR_MAT_MUL
!$    USE OMP_LIB
      IMPLICIT NONE
      INTEGER(KIND=8)                ::I,J,NTHREADS
      INTEGER(KIND=8),PARAMETER      ::N=60
      INTEGER(KIND=8),DIMENSION(N,N) ::AI,BI,CI
      INTEGER(KIND=8)                ::SUMI

      COMMON/DATA/ AI,BI,CI
!$OMP THREADPRIVATE (/DATA/)

!$OMP PARALLEL
      FORALL(I=1:N,J=1:N) AI(I,J) = (I-N/2)**2+(J+N/2)
      FORALL(I=1:N,J=1:N) BI(I,J) = 3-((I/2)+(J-N/2)**2)
!$OMP MASTER
      NTHREADS=8
!$    NTHREADS=OMP_GET_NUM_THREADS()
!$OMP END MASTER
!$OMP END PARALLEL

!$OMP PARALLEL DEFAULT(PRIVATE),COPYIN(AI,BI),SHARED(NTHREADS)
!$OMP DO
      DO I=1,NTHREADS
      CALL IMAT_MUL(SUMI)
      ENDDO
!$OMP END DO
!$OMP END PARALLEL

      END
```

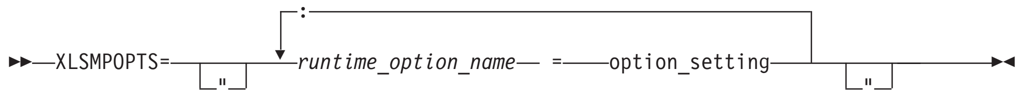
For information on using sentinels, see Conditional compilation in the *XL Fortran Language Reference*.

Setting runtime options

When you write parallel code, set the necessary XLSMPOPTS environment variables, and the environment variables for OpenMP, transactional memory, and thread-level speculative execution.

XLSMPOPTS

The **XLSMPOPTS** environment variable allows you to specify options that affect SMP execution. You can declare **XLSMPOPTS** by using the following **bash** command format:



You can specify option names and settings in uppercase or lowercase. You can add blanks before and after the colons and equal signs to improve readability. However, if the **XLSMPOPTS** option string contains imbedded blanks, you must enclose the entire option string in double quotation marks (**"**).

You can specify the following runtime options with the **XLSMPOPTS** environment variable:

schedule
: Selects the scheduling type and chunk size to be used as the default at run time. The scheduling type that you specify will only be used for loops that were not already marked with a scheduling type at compilation time.

  Work is assigned to threads in a different manner, depending on the scheduling type and chunk size used. A brief description of the scheduling types and their influence on how work is assigned follows:

  dynamic *or* **guided**
  : The runtime library dynamically schedules parallel work for threads on a "first-come, first-do" basis. "Chunks" of the remaining work are assigned to available threads until all work has been assigned. Work is not assigned to threads that are asleep.

  static
  : Chunks of work are assigned to the threads in a "round-robin" fashion. Work is assigned to all threads, both active and asleep. The system must activate sleeping threads in order for them to complete their assigned work.

  affinity
  : The runtime library performs an initial division of the iterations into *number_of_threads* partitions. The number of iterations that these partitions contain is:

    ```
    CEILING(number_of_iterations / number_of_threads)
    ```

    These partitions are then assigned to each of the threads. It is these partitions that are then subdivided into chunks of iterations. If a thread is asleep, the threads that are active will complete their assigned partition of work.

  Choosing chunking granularity is a tradeoff between overhead and load balancing. The syntax for this option is **schedule**=*suboption*, where the suboptions are defined as follows:

affinity[=*n*]
: As described previously, the iterations of a loop are initially divided into partitions, which are then preassigned to the threads. Each of these partitions is then further subdivided into chunks that contain *n* iterations. If you have not specified *n*, a chunk consists of CEILING(number_of_iterations_left_in_local_partition / 2) loop iterations.

  When a thread becomes available, it takes the next chunk from its preassigned partition. If there are no more chunks in that partition, the thread takes the next available chunk from a partition preassigned to another thread.

auto
: With **auto**, scheduling is delegated to the compiler and runtime system. The compiler and runtime system can choose any possible mapping of iterations to threads (including all possible valid schedules) and these may be different in different loops. Do not specify chunk size (*n*) when you use **auto**. If chunk size (*n*) is specified, the compiler issues a severe error message.

  Note: When both the **-qsmp=schedule** option and *OMP_SCHEDULE* are used, the option will override the environment variable.

dynamic[=*n*]
: The iterations of a loop are divided into chunks that contain *n* iterations each. If you have not specified *n*, a chunk consists of CEILING(number_of_iterations / number_of_threads) iterations.

guided[=*n*]
: The iterations of a loop are divided into progressively smaller chunks until a minimum chunk size of *n* loop iterations is reached. If you have not specified *n*, the default value for *n* is 1 iteration.

  The first chunk contains CEILING(number_of_iterations / number_of_threads) iterations. Subsequent chunks consist of CEILING(number_of_iterations_left / number_of_threads) iterations.

static[=*n*]
: The iterations of a loop are divided into chunks that contain *n* iterations. Threads are assigned chunks in a "round-robin" fashion. This is known as block cyclic scheduling. If the value of *n* is 1, the scheduling type is specifically referred to as cyclic scheduling.

  If you have not specified *n*, the chunks will contain CEILING(number_of_iterations / number_of_threads) iterations. Each thread is assigned one of these chunks. This is known as *block scheduling.*

If you have not specified **schedule**, the default is set to **schedule=static**, resulting in block scheduling. For more information, see the description of the **SCHEDULE** directive in the *XL Fortran Language Reference*.

Parallel execution options

parthds=*num*
: Specifies the number of threads (*num*) to be used for parallel execution of code that you compiled with the **-qsmp** option. By default, this is equal to the number of online processors. There are some applications that cannot use more than some maximum number of processors. There are also some applications that can achieve performance gains if they use more threads than there are processors.

This option allows you full control over the number of execution threads. The default value for *num* is 1 if you did not specify **-qsmp**. Otherwise, it is the number of online processors on the machine. For more information, see the **NUM_PARTHDS** intrinsic function in the *XL Fortran Language Reference*.

usrthds=*num*
: Specifies the maximum number of threads (*num*) that you expect your code will explicitly create if the code does explicit thread creation. The default value for *num* is 0. For more information, see the **NUM_PARTHDS** intrinsic function in the *XL Fortran Language Reference*.

stack=*num*
: Specifies the largest amount of space in bytes (*num*) that a thread's stack will need. The default value for *num* is 4194304.

  Set **stack**=*num* so it is within the acceptable upper limit. *num* can be up to the limit imposed by system resources or the stack size ulimit, whichever is smaller. An application that exceeds the upper limit may cause a segmentation fault.

stackcheck[=*num***]**
: Enables stack overflow checking for worker threads at runtime. *num* is the size in bytes that you specify; when the remaining stack size is less than *num*, a runtime warning message is issued. If you do not specify a value for *num*, the default value is 4096 bytes. Note that this option only has an effect when **-qsmp=stackcheck** has also been specified at compile time. See **-qsmp** in the *XL Fortran Compiler Reference* for more information.

startproc=*cpu_id*
: Enables thread binding and specifies the *cpu_id* to which the first thread binds. If the value provided is outside the range of available processors, the SMP run time issues a warning message and no threads are bound.

procs=*cpu_id[,cpu_id,...]*
: Enables thread binding and specifies a list of *cpu_id* to which the threads are bound. If the number of CPU IDs specified is less than the number of threads used by the program, the remaining threads are not bound.

stride=*num*
: Specifies the increment used to determine the *cpu_id* to which subsequent threads bind. *num* must be greater than or equal to 1. If the value provided causes a thread to bind to a CPU outside the range of available processors, a warning message is issued and no threads are bound.

Performance tuning options

When a thread completes its work and there is no new work to do, it can go into either a "busy-wait" state or a "sleep" state. In "busy-wait", the thread keeps executing in a tight loop looking for additional new work. This state is highly responsive but harms the overall utilization of the system. When a thread sleeps, it completely suspends execution until another thread signals it that there is work to do. This state provides better utilization of the system but introduces extra overhead for the application.

The **xlsmp** runtime library routines use both "busy-wait" and "sleep" states in their approach to waiting for work. You can control these states with the **spins**, **yields**, and **delays** options.

During the busy-wait search for work, the thread repeatedly scans the work queue up to *num* times, where *num* is the value that you specified for the option **spins**. If a thread cannot find work during a given scan, it intentionally wastes cycles in a delay loop that executes *num* times, where *num* is the value that you specified for the option **delays**. This delay loop consists of a single meaningless iteration. The length of actual time this takes will vary among processors. If the value **spins** is exceeded and the thread still cannot find work, the thread will yield the current time slice (time allocated by the processor to that thread) to the other threads. The thread will yield its time slice up to *num* times, where *num* is the number that you specified for the option **yields**. If this value *num* is exceeded, the thread will go to sleep.

In summary, the ordered approach to looking for work consists of the following steps:

1. Scan the work queue for up to **spins** number of times. If no work is found in a scan, then loop **delays** number of times before starting a new scan.
2. If work has not been found, then yield the current time slice.
3. Repeat the above steps up to **yields** number of times.
4. If work has still not been found, then go to sleep.

The syntax for specifying these options is as follows:

spins[=*num***]**
: where *num* is the number of spins before a yield. The default value for **spins** is **100**.

yields[=*num***]**
: where *num* is the number of yields before a sleep. The default value for **yields** is **10**.

delays[=*num***]**
: where *num* is the number of delays while busy-waiting. The default value for **delays** is **500**.

Zero is a special value for **spins** and **yields**, as it can be used to force complete busy-waiting. Normally, in a benchmark test on a dedicated system, you would set both options to zero. However, you can set them individually to achieve other effects.

For instance, on a dedicated 8-way SMP, setting these options to the following:

```
parthds=8 : schedule=dynamic=10 : spins=0 : yields=0
```

results in one thread per CPU, with each thread assigned chunks consisting of 10 iterations each, with busy-waiting when there is no immediate work to do.

Options to enable and control dynamic profiling

You can use dynamic profiling to reevaluate the compiler's decision to parallelize loops in a program. The three options you can use to do this are: **parthreshold**, **seqthreshold**, and **profilefreq**.

parthreshold=*num*
: Specifies the time, in milliseconds, below which each loop must execute serially. If you set **parthreshold** to 0, every loop that has been parallelized by the compiler will execute in parallel. The default setting is 0.2 milliseconds, meaning that if a loop requires fewer than 0.2 milliseconds to execute in parallel, it should be serialized.

Typically, **parthreshold** is set to be equal to the parallelization overhead. If the computation in a parallelized loop is very small and the time taken to execute these loops is spent primarily in the setting up of parallelization, these loops should be executed sequentially for better performance.

seqthreshold=*num*
: Specifies the time, in milliseconds, beyond which a loop that was previously serialized by the dynamic profiler should revert to being a parallel loop. The default setting is 5 milliseconds, meaning that if a loop requires more than 5 milliseconds to execute serially, it should be parallelized.

  seqthreshold acts as the reverse of **parthreshold**.

profilefreq=*num*
: Specifies the frequency with which a loop should be revisited by the dynamic profiler to determine its appropriateness for parallel or serial execution. Loops in a program can be data dependent. The loop that was chosen to execute serially with a pass of dynamic profiling may benefit from parallelization in subsequent executions of the loop, due to different data input. Therefore, you need to examine these loops periodically to reevaluate the decision to serialize a parallel loop at run time.

  The allowed values for this option are the numbers from 0 to 32. If you set **profilefreq** to one of these values, the following results will occur.

  - If **profilefreq** is 0, all profiling is turned off, regardless of other settings. The overheads that occur because of profiling will not be present.
  - If **profilefreq** is 1, loops parallelized automatically by the compiler will be monitored every time they are executed.
  - If **profilefreq** is 2, loops parallelized automatically by the compiler will be monitored every other time they are executed.
  - If **profilefreq** is greater than or equal to 2 but less than or equal to 32, each loop will be monitored once every *n*th time it is executed.
  - If **profilefreq** is greater than 32, then 32 is assumed.

  It is important to note that dynamic profiling is not applicable to user-specified parallel loops (for example, loops for which you specified the **PARALLEL DO** directive).

BG_SMP_FAST_WAKEUP

The BG_SMP_FAST_WAKEUP environment variable enables or disables the fast wake-up support, which allows the SMP runtime to use the hardware mechanism that puts waiting threads to sleep and wakes them up efficiently.

```
                             ┌─NO──┐
►►──BG_SMP_FAST_WAKEUP──=────┴─YES─┴──────────────────────────────────────►◄
```

The default value is **NO**. The value is not case sensitive.

BG_SMP_FAST_WAKEUP=YES enables the fast wake-up support.

When a thread is waiting for work or spinning at a barrier, it puts itself to sleep so that it does not consume processing resources. Due to the Blue Gene/Q threading model, the sleeping cannot be interrupted. This causes an issue when the hardware threads are oversubscribed; that is, multiple user threads are bound to the same hardware thread. This issue might cause deadlocks between user threads, in particular between OpenMP threads in the SMP runtime.

To avoid oversubscription of hardware threads when the fast wake-up support is enabled, you must also set the OMP_PROC_BIND environment variable to TRUE. This setting ensures that each user thread created by the SMP runtime is bound to a different hardware thread.

Related information

- OMP_PROC_BIND

BG_SPEC_SCRUB_CYCLE

The BG_SPEC_SCRUB_CYCLE environment variable sets the L2 background scrub cycle, which determines how quick a speculation ID can be reclaimed by the L2 and used again for speculation.

```
                              ┌─66─┐
►►──BG_SPEC_SCRUB_CYCLE──=────┴─n──┴──────────────────────────────────────────►◄
```

The default value is 66. The valid value range is 6-100.

Setting the scrub cycle low means the L2 needs to perform scrub more frequently.

The interval defines the period in 800MHz cycles (two processor cycles) for directory look-ups of the scrub process. For the default setting, the L2 examines one of the 1024 sets every 132 processor cycles, resulting in a complete cache scrub every 135168 processor cycles.

After a speculation ID has been used and committed or recycled, it takes at most 135168 cycles until the ID is available again for allocation. It can take less if all sets that were accessed with the ID are visited by either core memory accesses or the scrub earlier.

For example, programming the interval to 6 changes the duration for a full cache scrub from 135168 to 12288 processor cycles.

Environment variables for OpenMP

The following environment variables, which are included in the OpenMP standard, allow you to control the execution of parallel code.

Note: If you specify both the **XLSMPOPTS** environment variable and an OpenMP environment variable, the OpenMP environment variable takes precedence.

OMP_DYNAMIC

The **OMP_DYNAMIC** environment variable enables or disables dynamic adjustment of the number of threads available for the execution of parallel regions. The syntax is as follows:

```
►►──OMP_DYNAMIC=──┬─TRUE──┬──────────────────────────────────────────────►◄
                  └─FALSE─┘
```

If you set this environment variable to **TRUE**, the runtime environment can adjust the number of threads it uses for executing parallel regions so that it makes the most efficient use of system resources. If you set this environment variable to **FALSE**, dynamic adjustment is disabled.

The default value for **OMP_DYNAMIC** is **FALSE**. If your code needs to use a specific number of threads to run correctly, you should disable dynamic thread adjustment.

The **omp_set_dynamic** subroutine takes precedence over the **OMP_DYNAMIC** environment variable.

OMP_MAX_ACTIVE_LEVELS

The **OMP_MAX_ACTIVE_LEVELS** environment variable controls the maximum number of nested active parallel regions. The syntax is as follows:

```
►►──OMP_MAX_ACTIVE_LEVELS=──n────────────────────────────────────────────►◄
```

n is the maximum number of nested active parallel regions. It must be a positive scalar integer. XL Fortran does not support OpenMP nested parallelism. This environment variable has no effects to the nested parallel constructs in the program.

OMP_NESTED

The **OMP_NESTED** environment variable enables or disables nested parallelism. The syntax is as follows:

```
►►──OMP_NESTED=──┬─TRUE──┬───────────────────────────────────────────────►◄
                 └─FALSE─┘
```

If you set this environment variable to **TRUE**, nested parallelism is enabled. This means that the runtime environment might deploy extra threads to form the team of threads for the nested parallel region. If you set this environment variable to **FALSE**, nested parallelism is disabled.

The default value for **OMP_NESTED** is **FALSE**.

The **omp_set_nested** subroutine takes precedence over the **OMP_NESTED** environment variable.

Currently, XL Fortran does not support OpenMP nested parallelism.

OMP_NUM_THREADS

The **OMP_NUM_THREADS** environment variable sets the number of threads to use for parallel regions. The syntax of the environment variable is as follows:

```
►►──OMP_NUM_THREADS=──num_list───────────────────────────────────────────►◄
```

num_list
A list of one or more positive integer values separated by commas.

If you do not set the **OMP_NUM_THREADS** environment variable, the number of processors available is the default value to form a new team for the first encountered parallel construct. By default, any nested constructs are run by one thread.

If *num_list* contains a single value, dynamic adjustment of the number of threads is enabled (**OMP_DYNAMIC** is set to true), a parallel construct without a **NUM_THREADS** clause is encountered, the value is the maximum number of threads that can be used to form a new team for the encountered parallel construct.

If *num_list* contains a single value, dynamic adjustment of the number of threads is not enabled (**OMP_DYNAMIC** is set to false), a parallel construct without a **NUM_THREADS** clause is encountered, the value is the exact number of threads that can be used to form a new team for the encountered parallel construct.

If *num_list* contains multiple values, dynamic adjustment of the number of threads is enabled (**OMP_DYNAMIC** is set to true), a parallel construct without a **NUM_THREADS** clause is encountered, the first value is the maximum number of threads that can be used to form a new team for the encountered parallel construct. After the encountered construct is entered, the first value is removed and the remaining values form a new *num_list*. The new *num_list* is in turn used in the same way for any closely nested parallel constructs inside the encountered parallel construct.

If *num_list* contains multiple values, dynamic adjustment of the number of threads is not enabled (**OMP_DYNAMIC** is set to false), a parallel construct without a **NUM_THREADS** clause is encountered, the first value is the exact number of threads that can be used to form a new team for the encountered parallel construct. After the encountered construct is entered, the first value is removed and the remaining values form a new *num_list*. The new *num_list* is in turn used in the same way for any closely nested parallel constructs inside the encountered parallel construct.

Note: If the number of parallel regions is equal to or greater than the number of values in *num_list*, the **omp_get_max_threads** routine returns the last value of *num_list* in the parallel region.

If the number of threads requested exceeds the system resources available, the program stops.

The **omp_set_num_threads** routine sets the first value of *num_list*. The **omp_get_max_threads** routine returns the first value of *num_list*.

If you specify the number of threads for a given parallel region more than once with different settings, the compiler uses the following precedence order to determine which setting takes effect:

1. The number of threads set using the **NUM_THREADS** clause takes precedence over that set using the **omp_set_num_threads** routine.
2. The number of threads set using the **omp_set_num_threads** routine takes precedence over that set using the **OMP_NUM_THREADS** environment variable.

3. The number of threads set using the **OMP_NUM_THREADS** environment variable takes precedence over that set using the **PARTHDS** suboption of the **XLSMPOPTS** environment variable.

Note: In a given parallel region, the **omp_get_max_threads** routine returns the first value of *num_list*, even though the actual number of threads running that parallel region might be different from the first value of *num_list*.

The following example shows how you can set the **OMP_NUM_THREADS** environment variable.

```
export OMP_NUM_THREADS=5,10
export OMP_DYNAMIC=false

! OMP_GET_MAX_THREADS() returns 5 threads
!$omp parallel
! OMP_GET_MAX_THREADS() returns 10 threads
  !$omp parallel
  ! OMP_GET_MAX_THREADS() returns 10 threads
    !$omp parallel
    ! OMP_GET_MAX_THREADS() returns 10 threads
    !$omp end parallel
  !$omp end parallel
!$omp end parallel
```

OMP_PROC_BIND

The **OMP_PROC_BIND** environment variable controls whether OpenMP threads can be moved between processors. The syntax of the environment variable is as follows:

```
►►──OMP_PROC_BIND=──┬─TRUE──┬──────────────────────────────────────►◄
                    └─FALSE─┘
```

By default, the **OMP_PROC_BIND** environment variable is set to **TRUE**. If you set **OMP_PROC_BIND** to **FALSE**, the threads can be moved between processors.

Restriction: On the BG/Q system, the **OMP_PROC_BIND** environment variable is always set to **TRUE**. If you set **OMP_PROC_BIND** to **FALSE**, the compiler ignores it.

Note: The **OMP_PROC_BIND** environment variable provides a portable way to control whether OpenMP threads can be migrated.

OMP_SCHEDULE

The **OMP_SCHEDULE** environment variable applies to the **PARALLEL DO** and work-sharing **DO** directives that have a schedule type of **RUNTIME**. The syntax is as follows:

```
►►──OMP_SCHEDULE=──sched_type──┬──────────────┬────────────────────►◄
                               └─,──chunk_size─┘
```

sched_type
: is either **AUTO**, **DYNAMIC**, **GUIDED**, or **STATIC**. See the "SCHEDULE" on page 167 clause for a description of these scheduling parameters.

chunk_size
: is a positive, scalar integer that represents the chunk size.

This environment variable is ignored for the **PARALLEL DO** and work-sharing **DO** directives that have a schedule type other than **RUNTIME**.

If you do not specify a schedule type either at compile time through a directive, or at run time through the **OMP_SCHEDULE** environment variable or the **SCHEDULE** option of the **XLSMPOPTS** environment variable, the default schedule type is **AUTO**, which delegates scheduling decision to the compiler and runtime system. You cannot specify *chunk_size* when the schedule type is set to **AUTO**.

If you specify both the **SCHEDULE** option of the **XLSMPOPTS** environment variable and the **OMP_SCHEDULE** environment variable, the **OMP_SCHEDULE** environment variable takes precedence.

The following examples show how you can set the **OMP_SCHEDULE** environment variable:

```
export OMP_SCHEDULE="DYNAMIC"
export OMP_SCHEDULE="GUIDED,4"
export OMP_SCHEDULE="STATIC"
export OMP_SCHEDULE="AUTO"
```

OMP_STACKSIZE

The **OMP_STACKSIZE** environment variable specifies the size of the stack for threads created by the OpenMP runtime. The syntax is as follows:

```
►►──OMP_STACKSIZE=──┬─size──┬──────────────────────────────────────────►◄
                    ├─sizeB─┤
                    ├─sizeK─┤
                    ├─sizeM─┤
                    └─sizeG─┘
```

size is a positive integer that specifies the size of the stack for threads that are created by the OpenMP runtime.

"B", "K", "M", "G"
are letters that specify whether the given size is in Bytes, Kilobytes, Megabytes, or Gigabytes.

If only size is specified and none of *"B", "K", "M", "G"* is specified, size is in Kilobytes by default. This environment variable does not control the size of the stack for the initial thread.

The value assigned to the **OMP_STACKSIZE** environment variable is case insensitive and might have leading and trailing white space. The following examples show how you can set the **OMP_STACKSIZE** environment variable.

```
export OMP_STACKSIZE="10M"
export OMP_STACKSIZE=" 10 M "
```

If the value of **OMP_STACKSIZE** is not set, the initial value is set to the default value for 64-bit mode, up to the limit imposed by system resources).

If the compiler cannot deliver the stack size specified by the environment variable, or if **OMP_STACKSIZE** does not conform to the valid format, the compiler sets the environment variable to the default value.

The **OMP_STACKSIZE** environment variable takes precedence over the **stack** suboption of the **XLSMPOPTS** environment variable.

OMP_THREAD_LIMIT

The **OMP_THREAD_LIMIT** environment variable sets the number of OpenMP threads to use for the whole program. The syntax is as follows:

```
►►──OMP_THREAD_LIMIT=──n──────────────────────────────────────────────────►◄
```

n The number of OpenMP threads to use for the whole program. It must be a positive scalar integer.

The value for **OMP_THREAD_LIMIT** is a positive integer.

If the **OMP_THREAD_LIMIT** environment variable is not set and the **OMP_NUM_THREADS** environment variable is set to a single value, the default value for **OMP_THREAD_LIMIT** is the value of **OMP_NUM_THREADS** or the number of available processors, whichever is greater.

If the **OMP_THREAD_LIMIT** environment variable is not set and the **OMP_NUM_THREADS** environment variable is set to a list, the default value for **OMP_THREAD_LIMIT** is the multiplication of all the numbers in the list or the number of available processors, whichever is greater.

If both the **OMP_THREAD_LIMIT** and **OMP_NUM_THREADS** environment variables are not set, the default value for **OMP_THREAD_LIMIT** is the number of available processors.

OMP_WAIT_POLICY

The **OMP_WAIT_POLICY** environment variable provides hints about the preferred behavior of waiting threads during program execution. The syntax is as follows:

```
                      ┌─PASSIVE─┐
►►──OMP_WAIT_POLICY=──┴─ACTIVE──┴─────────────────────────────────────────►◄
```

Use **ACTIVE** if you want waiting threads to mostly be active. That is, the threads consume processor cycles while waiting. For example, waiting threads can spin while waiting. The **ACTIVE** wait policy is recommended for maximum performance on the dedicated machine.

Note: If the hardware threads are oversubscribed or multiple user threads are bound to the same hardware thread, **OMP_WAIT_POLICY=ACTIVE** might cause dead lock.

Use **PASSIVE** if you want waiting threads to mostly be passive. That is, the threads do not consume processor cycles while waiting. For example, waiting threads can sleep or yield the processor to other threads.

The default value of **OMP_WAIT_POLICY** is **PASSIVE**.

Note: If you set the **OMP_WAIT_POLICY** environment variable and specify the **SPINS**, **YIELDS**, or **DELAYS** suboptions of the **XLSMPOPTS** environment variable, **OMP_WAIT_POLICY** takes precedence.

Environment variables for thread-level speculative execution

The environment variables for thread-level speculative execution have no effect unless the thread-level speculative execution is enabled with the "-qsmp=speculative" compiler option.

SE_MAX_NUM_ROLLBACK

The SE_MAX_NUM_ROLLBACK environment variable indicates the maximum number a speculative thread of a particular thread-level speculative execution region can roll back before the region is run by one thread.

```
                           ┌─10─┐
►►──SE_MAX_NUM_ROLLBACK──=─┴─n──┴──────────────────────────────────────────►◄
```

n
: The maximum number of times a thread for a speculative region can roll back before being run by a single thread. It is a positive integer and must not exceed 2^{32}. The default value is 10.

SE_MAX_NUM_ROLLBACK = 0 forces the region of thread-level speculative execution to be run by one thread.

Related information

- Thread-level speculative execution

SE_REPORT_STAT_ENABLE

The SE_REPORT_STAT_ENABLE environment variable enables or disables the statistics query routine for thread-level speculative execution.

```
                             ┌─NO──┐
►►──SE_REPORT_STAT_ENABLE──=─┴─YES─┴───────────────────────────────────────►◄
```

The default value is **NO**. The value is not case sensitive.

When SE_REPORT_STAT_ENABLE is set to YES, you can retrieve the statistics through the **se_get_all_stats** routine.

Related information

- Routines for thread-level speculative execution

SE_REPORT_NAME

The SE_REPORT_NAME environment variable specifies the name of the statistics log file and the location where it is created for thread-level speculative execution.

```
►►──SE_REPORT_NAME──=──file_name───────────────────────────────────────────►◄
```

file_name
: The name of the log file, or the directory and name of the log file. If you specify the name only, the file is placed in the current working directory.

If SE_REPORT_NAME is not set, the log file is placed in the current working directory and is named `se_report.log.`*rank* where *rank* is the MPI rank of the process that called the **se_print_stats** routine.

Related information

- Routines for thread-level speculative execution

SE_REPORT_LOG

The SE_REPORT_LOG environment variable specifies how to create the statistics log files for thread-level speculative execution.

```
►►──SE_REPORT_LOG──=──┬─SUMMARY─┬──────────────────────────────────────────►◄
                      ├─FUNC────┤
                      ├─ALL─────┤
                      └─VERBOSE─┘
```

The value is not case sensitive.

`SUMMARY`
: The statistics log file is generated only at the end of the program. It contains the statistics of all the regions of thread-level speculative execution for all hardware threads.

`FUNC`
: The statistics log file is generated and updated at each call to the **se_print_stats** routine. It contains all the statistics at the time when the **se_print_stats** routine is called.

`ALL`
: The statistics log file is generated and updated at each call to the **se_print_stats** routine and at the end of the program.

`VERBOSE`
: The statistics log file is generated and updated at each call to the **se_print_stats** routine and at the end of the program. The generated report file also includes the addresses of memory access conflicts during the speculation.

Related information

- Routines for thread-level speculative execution

Environment variables for transactional memory

The environment variables for transactional memory have no effect unless transactional memory is enabled with the "-qtm" compiler option.

TM_MAX_NUM_ROLLBACK

The TM_MAX_NUM_ROLLBACK environment variable specifies the maximum number a thread of a particular transactional atomic region can roll back before the thread goes into irrevocable mode.

```
                          ┌─10─┐
►►──TM_MAX_NUM_ROLLBACK──=──┴─n──┴─────────────────────────────────────────►◄
```

n The maximum number of times a thread of a transactional atomic region can roll back before executing in irrevocable mode. It is a positive integer and must not exceed 2^{32}. The default value is 10.

TM_MAX_NUM_ROLLBACK = 0 forces the thread to enter irrevocable mode.

TM_ENABLE_INTERRUPT_ON_CONFLICT

The TM_ENABLE_INTERRUPT_ON_CONFLICT environment variable indicates whether to generate interrupts on conflicts between speculative access.

```
                                    ┌─YES─┐
►►──TM_ENABLE_INTERRUPT_ON_CONFLICT──=───┴─NO──┴────────────────────────────────►◄
```

The default value is **YES**. The value is not case sensitive.

Conflict resolution usually happens at the end of transactions. Use this environment variable to generate interrupts for access conflicts in transactional atomic regions. This option is useful for long running transactions because the conflict resolution logic immediately starts inside the interrupt handler of the kernel.

Notes:
- Interrupts are always generated for speculative buffer overflows, supervised-mode violations, or conflicts between speculative and non-speculative access.
- Use with discretion if the TM_ENABLE_INTERRUPT_ON_CONFLICT environment variable is set to NO. If a transaction atomic region contains control flow that depends on some bad speculative data, the program may hang or the speculative thread may branch to invalid locations.

TM_REPORT_STAT_ENABLE

The TM_REPORT_STAT_ENABLE environment variable enables or disables statistics query routines for transactional memory.

```
                           ┌─NO──┐
►►──TM_REPORT_STAT_ENABLE──=───┴─YES─┴──────────────────────────────────────────►◄
```

The default value is **NO**. The value is not case sensitive.

When TM_REPORT_STAT_ENABLE is set to YES, you can retrieve the statistics through the following routines:
- "tm_get_stats(stats)" on page 198
- "tm_get_all_stats(stats)" on page 199

Related information
- Routines for transactional memory

TM_REPORT_NAME

The TM_REPORT_NAME environment variable specifies the name of the statistics log file and the location where it is created for transactional memory.

```
►►──TM_REPORT_NAME──=──file_name──────────────────────────────────────────────►◄
```

file_name
: The name of the log file, or the directory and name of the log file. If you specify the name only, the file is placed in the current working directory.

If TM_REPORT_NAME is not set, the log file is placed in the current working directory and is named `tm_report.log.`*`rank`* where *rank* in the extension is the MPI rank of the process that called the **tm_print_stats**.

Related information

- Routines for transactional memory

TM_REPORT_LOG

The TM_REPORT_LOG environment variable specifies how to create the statistics log file for transactional memory.

By default, no statistics log file is created.

```
►►──TM_REPORT_LOG──=──┬─SUMMARY─┬─────────────────────────────────────────────►◄
                      ├─FUNC────┤
                      ├─ALL─────┤
                      └─VERBOSE─┘
```

The value is not case sensitive.

`SUMMARY`
: The statistics log file is generated only at the end of the program.

`FUNC`
: The statistics log file is generated and updated at each call to the **tm_print_stats** routine.

`ALL`
: The statistics log file is generated and updated at each call to the **tm_print_stats** routine and at the end of the program.

`VERBOSE`
: The statistics log file is generated and updated at each call to the **tm_print_stats** routine and at the end of the program. The generated report file contains the address of transactions that enter into irrevocable mode, and other useful information, such as the configuration of the runtime.

If TM_REPORT_LOG is set to either SUMMARY, ALL, or VERBOSE, a log file that contains the following statistics is also created when the program ends:

- It contains the cumulative statistics for transactional atomic regions that each hardware thread has run. The statistics follow the Structure of statistic counters.
- It contains a summary that shows the sum of all the statistic counters for all the hardware threads.

Related information

- Routines for transactional memory

TM_SHORT_TRANSACTION_MODE

The TM_SHORT_TRANSACTION_MODE environment variable indicates whether to use the short running speculation mode in the hardware.

```
                                  ┌─NO──┐
►►──TM_SHORT_TRANSACTION_MODE──=──┴─YES─┴─────────────────────────────────►◄
```

The default value is **NO**. The value is not case sensitive.

TM_SHORT_TRANSACTION_MODE = YES enables the short running speculation mode. It is recommended if the transactional atomic region is short.

TM_SHORT_TRANSACTION_MODE = NO enables the long running speculation mode. It is recommended if the transactional atomic region is large and many reuses are among the references inside the transaction.

Optimizing your SMP code

Most IBM processors are capable of shared-memory parallel processing. Compile with **-qsmp** to generate the threaded code needed to exploit this capability. The option implies a **-O2** optimization level. The default behavior for the option without suboptions is to do automatic parallelization with optimization.

The most commonly used **-qsmp** suboptions are summarized in the following table.

Commonly used **-qsmp** suboptions

Suboption	**Behavior**
auto	Instructs the compiler to automatically generate parallel code where possible without user assistance. This option also recognizes all the SMP directives.
omp	Enforces compliance with the OpenMP API for specifying explicit parallelism.
opt	Instructs the compiler to optimize as well as parallelize. The optimization is equivalent to **-O2 –qhot** in the absence of other optimization options. The default setting of **-qsmp** is **-qsmp=auto:noomp:opt**.
suboptions	Other values for the suboption provide control over thread scheduling, nested parallelism, locking, and so on.

Developing and running SMP applications

- By default, the parallelization performed is both user-directed and automatic. Use **-qsmp=omp:noauto** if you are compiling an OpenMP program and do not want automatic parallelization.
- Before using **-qsmp** with automatic parallelization, test your programs using optimization and **-qhot** in a single-threaded manner.
- Always use the reentrant compiler invocations (the **_r** command invocations, like **bgxlf_r**) when using **-qsmp**.
- By default, the runtime uses all available processors. Do not set the **XLSMPOPTS=PARTHDS** or **OMP_NUM_THREADS** variables unless you want to use fewer than the number of available processors. You might want to set the number of executing threads to a small number or to 1 to ease debugging.
- If you are using a dedicated machine or node, consider one of the following environment variable settings:

  - Set **BG_SMP_FAST_WAKEUP** to **YES**.
  - Set **OMP_WAIT_POLICY** to **ACTIVE** .
  - Set **SPINS** and **YIELDS** (suboptions of **XLSMPOPTS**) to 0.

  These settings prevent the operating system from intervening in the scheduling of threads across synchronization boundaries, such as barriers.
- When debugging an SMP program, try using **-qsmp=noopt** (without **-O**) to make the debugging information produced by the compiler more precise. You can also use the **SNAPSHOT** directive to create additional program points for storage visibility by flushing registers to memory.

Parallelization directives

You can use parallelization directives to exert control over parallelization. This section describes all parallelization directives supported by XL Fortran on Blue Gene/Q platforms.

An introduction to parallelization directives

These directives allow you to exert control over parallelization. For example, the **PARALLEL DO** directive specifies that the loop immediately following the directive should be run in parallel. All parallelization directives are comment form directives.

For more information on rules and syntax for comment form directives, see Comment and noncomment form directives in the *XL Fortran Language Reference*.

XL Fortran supports a number of SMP directives, divided as follows. To ensure the greatest portability of code, OpenMP directives are recommended where possible. Use the OpenMP *trigger_constant*, **$OMP** for OpenMP directives, but do not use this *trigger_constant* with any other directive. OpenMP directives must not appear in **PURE** and **ELEMENTAL** procedures.

Parallel region construct

Parallel constructs form the foundation of OpenMP based parallel execution in XL Fortran. The **PARALLEL/END PARALLEL** directive pair forms a basic parallel construct. Each time an executing thread enters a parallel region, it creates a team of threads and becomes master of that team. This allows parallel execution to take place within that construct by the threads in that team. The following directives are necessary for a parallel region:

PARALLEL	**END PARALLEL**

Nesting OpenMP, transactional memory, and thread-level speculative execution

This section describes how you can mix parallel regions. The following types of parallel regions can be used without restrictions in the same program if they are not nested:
- OpenMP
- Transactional memory (TM)
- Thread-level speculative execution (SE)

They can also be nested but with some restrictions. The following table describes the behavior of different nesting scenarios.

Table 17. Nesting rules for OpenMP, TM, and SE

Scenario	Description	Runtime action
`BEGIN SE` `BEGIN SE` `END SE` `END SE`	An SE region is nested inside an SE region.	The SE nesting is flattened. The inner nested SE region is run speculatively by one thread as part of the parallel outer SE region. The clauses specified on the inner SE region are still effective.
`BEGIN SE` `BEGIN TM` `END TM` `END SE`	A TM region is nested inside an SE region.	The TM region is run speculatively in SE mode as part of the outer SE region.
`BEGIN SE` `BEGIN OpenMP` `END OpenMP` `END SE`	An OpenMP region is nested inside an SE region.	An OpenMP region running in parallel inside the speculative SE region causes the SE region to be stopped. The stopped SE region is rolled back and run nonspeculatively. The inner OpenMP region is run nonspeculatively by multiple threads.
`BEGIN TM` `BEGIN SE` `END SE` `END TM`	An SE region is nested inside a TM region.	The inner SE region is run speculatively in TM mode by one thread as part of the outer TM region.
`BEGIN TM` `BEGIN TM` `END TM` `END TM`	A TM region is nested inside a TM region.	The TM nesting is flattened. The inner nested TM regions are run speculatively as part of the outer TM region.
`BEGIN TM` `BEGIN OpenMP` `END OpenMP` `END TM`	An OpenMP region is nested inside a TM region.	An OpenMP region running in parallel inside the speculative TM region causes the TM region to be stopped. The stopped transaction is then rolled back and run nonspeculatively. The inner OpenMP region is run nonspeculatively by multiple threads.
`BEGIN OpenMP` `BEGIN SE` `END SE` `END OpenMP`	An SE region is nested inside an OpenMP region.	The SE region is run by one thread in nonspeculative mode. The outer OpenMP region is run in parallel.
`BEGIN OpenMP` `BEGIN TM` `END TM` `END OpenMP`	A TM region is nested inside an OpenMP region.	The outer OpenMP region is run in parallel and the TM region is run speculatively.
`BEGIN TM` `END TM` `BEGIN SE` `END SE`	A program contains separate TM and SE regions.	The first region is run in TM mode. The second region is run in SE mode by one thread. If the **reset_speculation_mode** routine is called after the first TM region, the second SE region is run in parallel speculatively.
`BEGIN SE` `END SE` `BEGIN TM` `END TM`	A program contains separate SE and TM regions.	The first region is run in SE mode. The second region is run in TM irrevocable mode. If the **reset_speculation_mode** routine is called after the first SE region, the second TM region is run in parallel speculatively.

Work-sharing constructs

Work-sharing constructs divide the execution of code enclosed by the construct between threads in a team. For work-sharing to take place, the construct must be enclosed within the dynamic extent of a parallel region. For further information on work-sharing constructs, see the following directives:

DO	**END DO**
SECTIONS	**END SECTIONS**
WORKSHARE	**END WORKSHARE**
SINGLE	**END SINGLE**

Combined parallel work-sharing constructs

A combined parallel work-sharing construct allows you to specify a parallel region that already contains a single work-sharing construct. These combined constructs are semantically identical to specifying a parallel construct enclosing a single work-sharing construct. For more information on implementing combined constructs, see the following directives:

PARALLEL DO	**END PARALLEL DO**
PARALLEL SECTIONS	**END PARALLEL SECTIONS**
PARALLEL WORKSHARE	**END PARALLEL WORKSHARE**

Synchronization constructs

The following directives allow you to synchronize the execution of a parallel region by multiple threads in a team:

ATOMIC	
BARRIER	
CRITICAL	**END CRITICAL**
FLUSH	
ORDERED	**END ORDERED**
TASKWAIT	

Other OpenMP directives

The following OpenMP directives provide additional SMP functionality:

MASTER	**END MASTER**
TASK	**END TASK**
THREADPRIVATE	

Non-OpenMP SMP directives

The following directives provide additional SMP functionality:

DO SERIAL	
SCHEDULE	
THREADLOCAL	

Deprecated directive

The SMP directive listed in the following table has been deprecated and might be removed in a future release. Use the corresponding OpenMP directive or clause to obtain the same behavior.

Table 18. Deprecated SMP directive

SMP directive name	OpenMP directive/clause name
SCHEDULE	SCHEDULE

The following example shows how to replace the deprecated SMP **SCHEDULE** directive with the OpenMP **SCHEDULE** clause.

The original code that uses the SMP **SCHEDULE** directive is as follows:

```
      program loop
      integer, parameter :: N=500
      integer :: i
!SMP$ SCHEDULE(STATIC)
      real :: arr(N)

!SMP$ parallel do
      do i=1, N
        arr(i) = real(i-1)
      enddo
      end program
```

To obtain the same behavior, you can use the OpenMP **SCHEDULE** clause, as shown below:

```
      program loop
      integer, parameter :: N=500
      integer :: i
      real :: arr(N)

!$OMP parallel do schedule(static)
      do i=1, N
        arr(i) = real(i-1)
      enddo
      end program
```

Detailed descriptions of parallelization directives

See the alphabetical list of all parallelization directives supported by XL Fortran.

For information on directive clauses, see “Directive clauses” on page 146.

ATOMIC

Purpose

You can use the **ATOMIC** directive to access a specific memory location safely within a parallel region. When you use the **ATOMIC** directive, you ensure that only one thread is accessing the memory location at a time, avoiding errors that might occur from simultaneous reads or writes to the same memory location.

Atomic operations are useful when creating multi-threaded or concurrent algorithms and data structures. Using atomic constructs, you can write more efficient concurrent algorithms with fewer locks.

An atomic construct supports the following kinds of atomic access:

- Atomic update

Updates the value of a variable atomically. Allows only one thread to write to a shared variable at a time, avoiding errors from simultaneous writes to the same variable.

- Atomic read

  Reads the value of a variable atomically. The value of a shared variable can be read safely, avoiding the danger of reading an intermediate value of the variable when it is accessed simultaneously by a concurrent thread.
- Atomic write

  Writes the value of a variable atomically. The value of a shared variable can be written exclusively to avoid errors from simultaneous writes.
- Atomic capture

  Updates the value of a variable while capturing the original or final value of the variable atomically.

The **ATOMIC** directive takes effect only if you specify the **-qsmp** compiler option.

Syntax

Atomic update

```
          ┌UPDATE┐
►►──ATOMIC┴──────┴──────────────────────────────────────────────────────►◄

►►──atomic_update_statement──────────────────────────────────────────────►◄

►►──┬──────────┬─────────────────────────────────────────────────────────►◄
    └END ATOMIC┘
```

Atomic read

```
►►──ATOMIC──READ─────────────────────────────────────────────────────────►◄

►►──atomic_capture_statement─────────────────────────────────────────────►◄

►►──┬──────────┬─────────────────────────────────────────────────────────►◄
    └END ATOMIC┘
```

Atomic write

```
►►──ATOMIC──WRITE──────────────────────────────────────────────►◄

►►──atomic_write_statement─────────────────────────────────────►◄

►►──┬────────────┬─────────────────────────────────────────────►◄
    └─END ATOMIC─┘
```

Atomic capture

```
►►──ATOMIC──CAPTURE────────────────────────────────────────────►◄

►►──atomic_update_statement────────────────────────────────────►◄

►►──atomic_capture_statement───────────────────────────────────►◄

►►──END ATOMIC─────────────────────────────────────────────────►◄
```

Or

```
►►──ATOMIC──CAPTURE────────────────────────────────────────────►◄

►►──atomic_capture_statement───────────────────────────────────►◄

►►──atomic_update_statement────────────────────────────────────►◄

►►──END ATOMIC─────────────────────────────────────────────────►◄
```

where *atomic_update_statement* is one of the following statements:

```
update_variable = update_variable operator expression
update_variable = expression operator update_variable
update_variable = intrinsic(update_variable, expression_list)
update_variable = intrinsic(expression_list, update_variable)
```

atomic_write_statement is:

```
update_variable = expression
```

atomic_capture_statement is:

```
capture_variable = update_variable
```

where:

update_variable, *capture_variable*
: are both nonpointer, nonallocatable scalar variables of intrinsic type.

intrinsic
: is one of **MAX**, **MIN**, **IAND**, **IOR** or **IEOR**.

operator
: is one of **+**, **–**, *****, **/**, **.AND.**, **.OR.**, **.EQV.**, **.NEQV.** or **.XOR.**.

expression
: is a scalar expression that does not reference *update_variable*.

expression_list
: is a comma-separated, non-empty list of scalar expressions that do not reference *update_variable*.

  Note: If the intrinsic is **IAND**, **IOR**, or **IEOR**, *expression_list* can only contain one expression.

Rules

An **ATOMIC** directive without a clause is equivalent to atomic update, and applies only to the statement that immediately follows it.

All accesses to a certain storage location throughout a concurrent program must be atomic. A non-atomic access to a memory location might break the expected atomic behavior of all atomic accesses to that storage location.

The *expression* in an atomic statement is not evaluated atomically. You must ensure that no race conditions exist in the calculation.

Within the entire program, if you use the **ATOMIC** directive to make references to the storage location of an *update_variable*, all these references must have the same type and type parameters.

capture_variable, *expression*, and *expression_list* must not access the same storage location as *update_variable*.

For atomic capture access, the operation of writing the captured value to the storage location represented by *capture_variable* is not atomic.

The function *intrinsic*, the operator *operator*, and the assignment must be the intrinsic function, operator and assignment, and not a redefined intrinsic function, defined operator or defined assignment.

Examples

Example 1: In this example, multiple threads are updating a counter. **ATOMIC** is used to ensure that no updates are lost.

```
PROGRAM P
  R = 0.0
!$OMP PARALLEL DO SHARED(R)
  DO I = 1, 10
!$OMP ATOMIC
    R = R + 1.0
  END DO
  PRINT *,R
END PROGRAM P
```

Expected output:

```
10.0
```

Example 2: In this example, an **ATOMIC** directive is required, because it is uncertain which element of array `Y` is updated in each iteration.

```
PROGRAM P
  INTEGER, DIMENSION(10) :: Y, INDEX
  INTEGER B

  Y = 5
  READ(*,*) INDEX, B
!$OMP PARALLEL DO SHARED(Y)
  DO I = 1, 10
!$OMP ATOMIC
    Y(INDEX(I)) = MIN(Y(INDEX(I)),B)
  END DO
  PRINT *, Y
END PROGRAM P
```

Input data:

```
10 10 8 8 6 6 4 4 2 2   4
```

Expected output:

```
5 4 5 4 5 4 5 4 5 4
```

Example 3: This example demonstrates the usage of atomic capture.

```
FUNCTION fnc(upper) RESULT(ret)
  INTEGER, INTENT(IN) :: upper
  INTEGER :: ret
  INTEGER, SAVE :: iter = 0

!$OMP ATOMIC CAPTURE
  iter = iter + 1
  ret = iter
!$OMP END ATOMIC

  IF (ret .GT. upper) THEN
    ret = -1
  ENDIF
END FUNCTION fnc
```

Related reference:

"CRITICAL / END CRITICAL" on page 99

"PARALLEL / END PARALLEL" on page 111

See -qsmp in the Compiler Reference

BARRIER

Purpose

The **BARRIER** directive enables you to synchronize all threads in a team. When a thread encounters a **BARRIER** directive, it will wait until all other threads in the team reach the same point.

Type

The **BARRIER** directive only takes effect if you specify the **-qsmp** compiler option.

Syntax

```
►►──BARRIER──────────────────────────────────────────────────►◄
```

Rules

A **BARRIER** region binds to the closest enclosing **PARALLEL** region.

A **BARRIER** region must not be closely nested inside a **CRITICAL**, **MASTER**, **ORDERED**, **TASK** or work-sharing region.

All threads in the team of the binding parallel region must execute the **BARRIER** region and complete execution of all explicit tasks in the binding parallel region up to this point before any threads in the team proceed beyond the barrier.

All **BARRIER** regions and work-sharing region must be encountered in the same order by all threads in the team.

Each **BARRIER** region must be encountered by all threads in a team or by none at all.

In addition to synchronizing the threads in a team, the **BARRIER** directive implies the **FLUSH** directive without the *variable_name_list*.

Examples

An example of the **BARRIER** construct binding to the **PARALLEL** construct. Note: To calculate *C*, we need to ensure that *A* and *B* have been completely assigned to, so threads need to wait.

```
      SUBROUTINE SUB1
      INTEGER A(1000), B(1000), C(1000)
!$OMP PARALLEL
!$OMP DO
      DO I = 1, 1000
        A(I) = SIN(I*2.5)
      END DO
!$OMP END DO NOWAIT
!$OMP DO
      DO J = 1, 10000
        B(J) = X + COS(J*5.5)
      END DO
!$OMP END DO NOWAIT
         ...
!$OMP BARRIER
      C = A + B
!$OMP END PARALLEL
      END
```

Related reference:
"FLUSH" on page 105
"PARALLEL / END PARALLEL" on page 111
See -qsmp in the Compiler Reference

CRITICAL / END CRITICAL

Purpose

The **CRITICAL** construct allows you to define independent blocks of code that are to be run by at most one thread at a time. It includes a **CRITICAL** directive that is followed by a block of code and ends with an **END CRITICAL** directive.

Type

The **CRITICAL** and **END CRITICAL** directives only take effect if you specify the **-qsmp** compiler option.

Syntax

```
►►──CRITICAL──┬─────────────────┬──────────────────────────────────►◄
              └─(──lock_name──)─┘

►►──block──────────────────────────────────────────────────────────►◄

►►──END CRITICAL──┬─────────────────┬──────────────────────────────►◄
                  └─(──lock_name──)─┘
```

lock_name
provides a way of distinguishing different **CRITICAL** constructs of code.

block represents the block of code to be executed by at most one thread at a time.

Rules

The optional *lock_name* is a name with global scope. You must not use the *lock_name* to identify any other global entity in the same executable program.

If you specify the *lock_name* on the **CRITICAL** directive, you must specify the same *lock_name* on the corresponding **END CRITICAL** directive.

If you specify the same *lock_name* for more than one **CRITICAL** construct, the compiler will allow only one thread to execute any one of these **CRITICAL** constructs at any one time. **CRITICAL** constructs that have different *lock_names* may be run in parallel.

The same lock protects all **CRITICAL** constructs that do not have an explicit *lock_name*. In other words, the compiler will assign the same *lock_name*, thereby ensuring that only one thread enters any unnamed **CRITICAL** construct at a time.

The *lock_name* must not share the same name as any local entity of Class 1.

It is illegal to branch into or out of a **CRITICAL** construct.

The **CRITICAL** construct may appear anywhere in a program.

Although it is possible to nest a **CRITICAL** construct within a **CRITICAL** region, a deadlock situation may result. The **-qsmp=rec_locks** compiler option can be used to prevent deadlocks. See the *XL Fortran Compiler Reference* for more information. The OpenMP API does not allow nested **CRITICAL** regions to have the same name.

CRITICAL and **END CRITICAL** directives imply the **FLUSH** directive without the *variable_name_list*.

Examples

Example 1: This example illustrates the use of a **CRITICAL** construct to update a shared variable inside a parallel region. The **CRITICAL** construct restricts only one thread to execute the code at a time.

```
      EXPR=0
!$OMP PARALLEL DO PRIVATE (I)
      DO I = 1, 100
!$OMP   CRITICAL
          EXPR = EXPR + A(I) * I
!$OMP   END CRITICAL
      END DO
```

Example 2: An example specifying a *lock_name* on the **CRITICAL** construct.

```
!$OMP PARALLEL DO PRIVATE(T)
      DO I = 1, 100
        T = B(I) * B(I-1)
!$OMP   CRITICAL (LOCK)
          SUM = SUM + T
!$OMP   END CRITICAL (LOCK)
      END DO
```

Related reference:

"ATOMIC" on page 93

"FLUSH" on page 105

See Global entity in the Language Reference

See Local entity in the Language Reference

"PARALLEL / END PARALLEL" on page 111

See -qsmp in the Compiler Reference

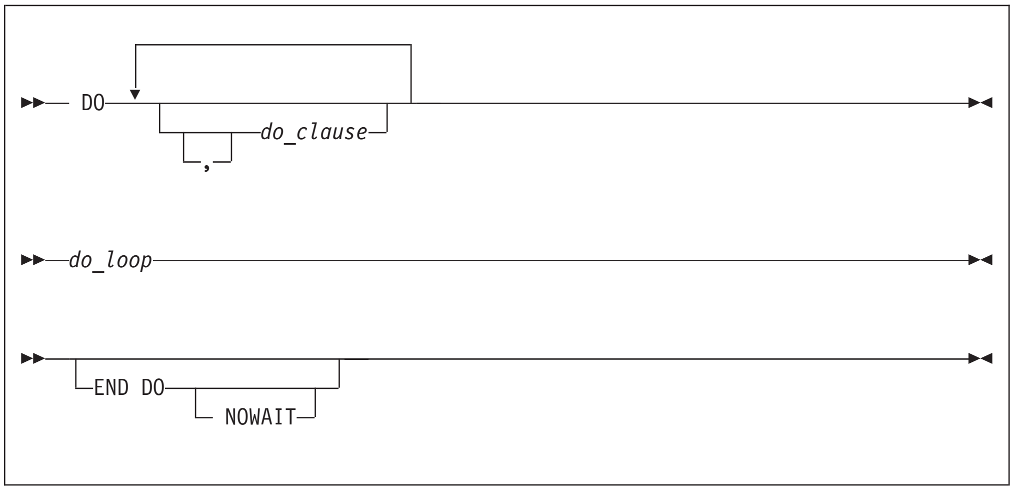
DO / END DO

Purpose

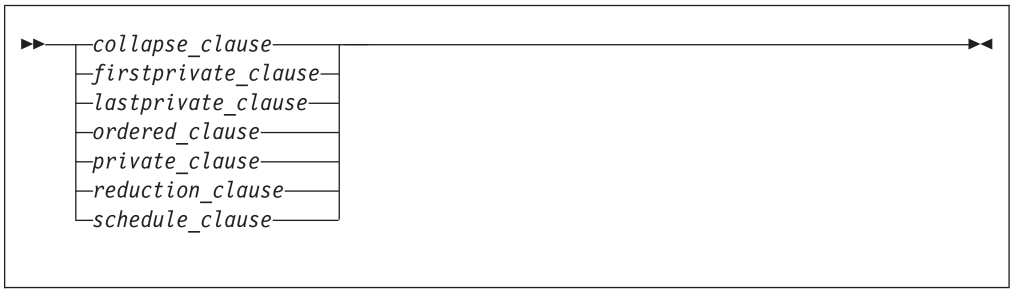
The **DO** (work-sharing) construct enables you to divide the execution of the loop among the members of the team that encounter it. The **END DO** directive enables you to indicate the end of a **DO** loop that is specified by the **DO** (work-sharing) directive.

The **DO** (work-sharing) and **END DO** directives only take effect when you specify the **-qsmp** compiler option.

Syntax



where *do_clause* is:



collapse_clause
: See — "COLLAPSE" on page 148.

firstprivate_clause
: See — "FIRSTPRIVATE" on page 155.

lastprivate_clause
: See — "LASTPRIVATE" on page 157.

ordered_clause
: See — "ORDERED" on page 160

private_clause
: See — "PRIVATE" on page 161.

reduction_clause
: See — "REDUCTION" on page 164

schedule_clause
: See — "SCHEDULE" on page 167

Rules

The first noncomment line (not including other directives) that follows the **DO** (work-sharing) directive must be a **DO** loop. This line cannot be an infinite **DO** or **DO WHILE** loop. The **DO** (work-sharing) directive applies only to the **DO** loop that is immediately following the directive, and not to any nested **DO** loops, unless the **COLLAPSE** clause is specified.

The **END DO** directive is optional. If you use the **END DO** directive, it must immediately follow the end of the **DO** loop.

You may have a **DO** construct that contains several **DO** statements. If the **DO** statements share the same **DO** termination statement, and an **END DO** directive follows the construct, you can only specify a work-sharing **DO** directive for the outermost **DO** statement of the construct.

If you specify **NOWAIT** on the **END DO** directive, a thread that completes its iterations of the loop early will proceed to the instructions following the loop. The thread will not wait for the other threads of the team to complete the **DO** loop. If you do not specify **NOWAIT** on the **END DO** directive, each thread will wait for all other threads within the same team at the end of the **DO** loop.

If you do not specify the **NOWAIT** clause, the **END DO** directive implies the **FLUSH** directive without the *variable_name_list*.

All threads in the team must encounter the **DO** (work-sharing) directive if any thread encounters it. A **DO** loop must have the same loop boundary and step value for each thread in the team. All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

A **DO** (work-sharing) directive must not appear within a **CRITICAL**, **MASTER**, or **ORDERED** region. In addition, it must not appear within a work-sharing region or a **TASK** region unless it is bound to another parallel region.

You cannot follow a **DO** (work-sharing) directive by another **DO** (work-sharing) directive. You can only specify one **DO** (work-sharing) directive for a given **DO** loop.

The **DO** (work-sharing) directive cannot appear with either an **INDEPENDENT** or **DO SERIAL** directive for a given **DO** loop.

To ensure that the same assignment of logical iteration numbers to threads is used in two work-sharing loop regions, you can use the **STATIC** schedule of the **SCHEDULE** clause. For details, see "SCHEDULE" on page 167.

Examples

Example 1: An example of several independent **DO** loops within a **PARALLEL** construct. No synchronization is performed after the first work-sharing **DO** loop, because **NOWAIT** is specified on the **END DO** directive.

```
!$OMP PARALLEL
!$OMP DO
      DO I = 2, N
        B(I)= (A(I) + A(I-1)) / 2.0
      END DO
!$OMP END DO NOWAIT
!$OMP DO
      DO J = 2, N
         C(J) = SQRT(REAL(J*J))
      END DO
!$OMP END DO
      C(5) = C(5) + 10
!$OMP END PARALLEL
      END
```

Example 2: An example of **SHARED**, and **SCHEDULE** clauses.

```
!$OMP PARALLEL SHARED(A)
!$OMP DO SCHEDULE(STATIC,10)
      DO I = 1, 1000
         A(I) = I * 4
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

Example 3: An example of both a **MASTER** and a **DO** (work-sharing) directive that bind to the closest enclosing **PARALLEL** directive.

```
!$OMP PARALLEL DEFAULT(PRIVATE), SHARED(X)
      Y = 100
!$OMP MASTER
      PRINT *, Y
!$OMP END MASTER
!$OMP DO
      DO I = 1, 10
         X(I) = I
         X(I) = X(I) + Y
      END DO
!$OMP END PARALLEL
      END
```

Example 4: An example of both the **FIRSTPRIVATE** and the **LASTPRIVATE** clauses on **DO** (work-sharing) directives.

```
      X = 100

!$OMP PARALLEL PRIVATE(I), SHARED(X,Y)
!$OMP DO FIRSTPRIVATE(X), LASTPRIVATE(X)
      DO I = 1, 80
         Y(I) = X + I
         X = I
      END DO
!$OMP END PARALLEL
      END
```

Related reference:

"COLLAPSE" on page 148

See DO in the Language Reference

"DO SERIAL"

"FLUSH" on page 105

See INDEPENDENT in the Language Reference

"ORDERED / END ORDERED" on page 108

"PARALLEL / END PARALLEL" on page 111

"PARALLEL DO / END PARALLEL DO" on page 113

DO SERIAL

Purpose

The **DO SERIAL** directive indicates to the compiler that the **DO** loop that is immediately following the directive must not be parallelized. This directive is useful in blocking automatic parallelization for a particular **DO** loop. The **DO SERIAL** directive only takes effect if you specify the **-qsmp** compiler option.

Syntax

```
►►──DO SERIAL──────────────────────────────────────────────────────►◄
```

Rules

The first noncomment line (not including other directives) that is following the **DO SERIAL** directive must be a **DO** loop. The **DO SERIAL** directive applies only to the **DO** loop that immediately follows the directive and not to any loops that are nested within that loop.

You can only specify one **DO SERIAL** directive for a given **DO** loop. The **DO SERIAL** directive must not appear with the **DO** (work-sharing), or **PARALLEL DO** directive on the same **DO** loop.

White space is optional between **DO** and **SERIAL.**

You should not use the OpenMP trigger constant with this directive.

Examples

Example 1: An example with nested DO loops where the inner loop (the J loop) is not parallelized.

```
!$OMP PARALLEL DO PRIVATE(S,I), SHARED(A)
      DO I=1, 500
        S=0
        !SMP$ DOSERIAL
        DO J=1, 500
          S=S+1
        ENDDO
        A(I)=S+I
      ENDDO
```

Example 2: An example with the **DOSERIAL** directive applied in nested loops. In this case, if automatic parallelization is enabled the I or K loop may be parallelized.

```
      DO I=1, 100
!SMP$ DOSERIAL
        DO J=1, 100
          DO K=1, 100
            ARR(I,J,K)=I+J+K
          ENDDO
        ENDDO
      ENDDO
```

Related reference:
"DO / END DO" on page 100

See DO in the Language Reference
"PARALLEL DO / END PARALLEL DO" on page 113

See -qdirective in the Compiler Reference

See -qsmp in the Compiler Reference

FLUSH

Purpose

The **FLUSH** directive ensures that each thread has access to data generated by other threads. This directive is required because the compiler may keep values in processor registers if a program is optimized. The **FLUSH** directive ensures that the memory images that each thread views are consistent.

The **FLUSH** directive only takes effect if you specify the **-qsmp** compiler option.

You might be able to improve the performance of your program by using the **FLUSH** directive instead of the **VOLATILE** attribute. The **VOLATILE** attribute causes variables to be flushed after every update and before every use, while **FLUSH** causes variables to be written to or read from memory only when specified.

Syntax

```
>>—FLUSH—┬──────────────────────────┬────────────────────────────────><
         └─(—variable_name_list—)─┘
```

Rules

You can specify this directive anywhere in your code; however, if you specify it outside a parallel region, it is ignored.

If you specify a *variable_name_list*, only the variables in that list are written to or read from memory (assuming that they have not been written or read already). All variables in the *variable_name_list* must be at the current scope and must be thread visible. Thread visible variables can be any of the following:
- Globally visible variables (common blocks and module data)
- Local and host-associated variables with the **SAVE** attribute
- Local variables without the **SAVE** attribute that are specified in a **SHARED** clause in a parallel region within the subprogram
- Local variables without the **SAVE** attribute that have had their addresses taken
- All pointer dereferences
- Dummy arguments

If an item or a subobject of an item in the *variable_name_list* has the **POINTER** attribute, the allocation and association status of the **POINTER** item is flushed, but the pointer target is not. If an item in the *variable_name_list* is an integer pointer, the pointer is flushed, but the object to which it points is not. If an item in the *variable_name_list* has the **ALLOCATABLE** attribute and the item is allocated, the allocated object is flushed. Otherwise, the allocation status is flushed

If you do not specify a *variable_name_list*, all thread visible variables are written to or read from memory.

When a thread encounters the **FLUSH** directive, it writes into memory the modifications to the affected variables. The thread also reads the latest copies of the variables from memory if it has local copies of those variables: for example, if it has copies of the variables in registers.

It is not mandatory for all threads in a team to use the **FLUSH** directive. However, to guarantee that all thread visible variables are current, any thread that modifies a thread visible variable should use the **FLUSH** directive to update the value of that variable in memory. If you do not use **FLUSH** or one of the directives that implies **FLUSH** (see below), the value of the variable might not be the most recent one.

The **FLUSH** directive does not imply any ordering between the directive and operations on variables not in the *variable_name_list*. The **FLUSH** directive does not imply any ordering between two or more **FLUSH** constructs if the constructs do not have any variables in common in the *variable_name_list*.

Note that **FLUSH** is not atomic. You must **FLUSH** shared variables that are controlled by a shared lock variable with one directive and then **FLUSH** the lock variable with another. This guarantees that the shared variables are written before the lock variable.

The following directives imply a **FLUSH** directive without the *variable_name_list* unless you specify a **NOWAIT** clause for those directives to which it applies:

- **BARRIER**
- **CRITICAL/END CRITICAL**
- **END DO**
- **END SECTIONS**
- **END SINGLE**
- **END WORKSHARE**
- **PARALLEL/END PARALLEL**
- **PARALLEL DO/END PARALLEL DO**
- **PARALLEL SECTIONS/END PARALLEL SECTIONS**
- **PARALLEL WORKSHARE/END PARALLEL WORKSHARE**
- **ORDERED/END ORDERED**

The **ATOMIC** directive implies a **FLUSH** directive with the *variable_name_list*. The *variable_name_list* contains only the object updated in the **ATOMIC** construct

The following routines imply a **FLUSH** directive without the *variable_name_list*:

- During **OMP_SET_LOCK**, and **OMP_UNSET_LOCK** regions.
- During **OMP_TEST_LOCK**, **OMP_SET_NEST_LOCK**, **OMP_UNSET_NEST_LOCK** and **OMP_TEST_NEST_LOCK** regions, if the region causes the lock to be set or unset.

Examples

In the following example, two threads perform calculations in parallel and are synchronized when the calculations are complete:

```
      PROGRAM P
      USE OMP_LIB
      INTEGER INSYNC(0:1), IAM

!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(INSYNC) NUM_THREADS(2)"
      IAM = OMP_GET_THREAD_NUM()
      INSYNC(IAM) = 0
!$OMP BARRIER
      CALL WORK
!$OMP FLUSH(INSYNC)
      INSYNC(IAM) = 1                        ! Each thread sets a flag
                                             ! once it has
!$OMP FLUSH(INSYNC)                           ! completed its work.
      DO WHILE (INSYNC(1-IAM) .eq. 0)        ! One thread waits for
                                             ! another to complete
!$OMP   FLUSH(INSYNC)                        ! its work.
      END DO

!$OMP END PARALLEL

      END PROGRAM P

      SUBROUTINE WORK                        ! Each thread does indep-
                                             ! endent calculations.
!       ...
!$OMP FLUSH                                  ! flush work variables
                                             ! before INSYNC
                                             ! is flushed.
      END SUBROUTINE WORK
```

MASTER / END MASTER

Purpose

The **MASTER** construct enables you to define a block of code that will be run by only the master thread of the team. It includes a **MASTER** directive that precedes a block of code and ends with an **END MASTER** directive.

The **MASTER** and **END MASTER** directives only take effect if you specify the **-qsmp** compiler option.

Syntax

```
>>—MASTER—————————————————————————————————————————————————————><

>>—block——————————————————————————————————————————————————————><

>>—END MASTER—————————————————————————————————————————————————><
```

block represents the block of code that will be run by the master thread of the team.

Rules

It is illegal to branch into or out of a **MASTER** construct.

A **MASTER** directive binds to the closest enclosing **PARALLEL** region, if one exists.

A **MASTER** directive cannot appear within a work-sharing region or a **TASK** region.

No implied barrier exists on entry to, or exit from, the **MASTER** construct.

Examples

Example 1: An example of the **MASTER** directive binding to the **PARALLEL** directive.

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP MASTER
        Y = 10.0
        X =  0.0
        DO I = 1, 4
            X = X + COS(Y) + I
        END DO
!$OMP END MASTER
!$OMP BARRIER
!$OMP DO PRIVATE(J)
        DO J = 1, 10000
             A(J) = X + SIN(J*2.5)
        END DO
!$OMP END DO
!$OMP END PARALLEL
        END
```

Related reference:

See -qsmp in the Compiler Reference

"PARALLEL / END PARALLEL" on page 111

"DO / END DO" on page 100

ORDERED / END ORDERED

Purpose

The **ORDERED / END ORDERED** directives cause the iterations of a block of code within a parallel loop to be executed in the order that the loop would execute in if it was run sequentially. You can force the code inside the **ORDERED** construct to run in a predictable order while code outside of the construct runs in parallel.

The **ORDERED** and **END ORDERED** directives only take effect if you specify the **-qsmp** compiler option.

Syntax

```
►►──ORDERED──────────────────────────────────────────────────────►◄

►►──block────────────────────────────────────────────────────────►◄

►►──END ORDERED──────────────────────────────────────────────────►◄
```

block represents the block of code that will be executed in sequence.

Rules

The **ORDERED** directive can only appear in the dynamic extent of a **DO** or **PARALLEL DO** directive. It is illegal to branch into or out of an **ORDERED** construct.

The **ORDERED** directive binds to the nearest dynamically enclosing **DO** or **PARALLEL DO** directive. You must specify the **ORDERED** clause on the **DO** or **PARALLEL DO** directive to which the **ORDERED** construct binds.

ORDERED constructs that bind to different **DO** directives are independent of each other.

Only one thread can execute an **ORDERED** construct at a time. Threads enter the **ORDERED** construct in the order of the loop iterations. A thread will enter the **ORDERED** construct if all of the previous iterations have either executed the construct or will never execute the construct.

Each iteration of a parallel loop with an **ORDERED** construct can only execute that **ORDERED** construct once. Each iteration of a parallel loop can execute at most one **ORDERED** directive. An **ORDERED** construct cannot appear within the dynamic extent of a **CRITICAL** construct.

The **END ORDERED** directive implies the **FLUSH** directive without the *variable_name_list*

Examples

Example 1: In this example, an **ORDERED** parallel loop counts down.

```
      PROGRAM P
!$OMP PARALLEL DO ORDERED
      DO I = 3, 1, -1
!$OMP ORDERED
      CALL C_PRINT(I) ! print I using routine written in C
!$OMP END ORDERED
      END DO
      END PROGRAM P
```

The expected output of this program is:

```
3
2
1
```

Example 2: This example shows a program with two **ORDERED** constructs in a parallel loop. Each iteration can only execute a single section.

```
      PROGRAM P
!$OMP PARALLEL DO ORDERED
      DO I = 1, 3
        IF (MOD(I,2) == 0) THEN
!$OMP     ORDERED
            CALL C_PRINT(I*10) ! print I*10 using routine written in C
!$OMP     END ORDERED
        ELSE
!$OMP     ORDERED
            CALL C_PRINT(I) ! print I using routine written in C
!$OMP     END ORDERED
        END IF
      END DO
      END PROGRAM P
```

The expected output of this program is:

```
1
20
3
```

Example 3: In this example, the program computes the sum of all elements of an array that are greater than a threshold. **ORDERED** is used to ensure that the results are always reproducible: roundoff will take place in the same order every time the program is executed, so the program will always produce the same results.

```
      PROGRAM P
        REAL :: A(1000)
        REAL :: THRESHOLD = 999.9
        REAL :: SUM = 0.0

!$OMP   PARALLEL DO ORDERED
        DO I = 1, 1000
          IF (A(I) > THRESHOLD) THEN
!$OMP       ORDERED
              SUM = SUM + A(I)
!$OMP       END ORDERED
          END IF
        END DO
      END PROGRAM P
```

Note: To avoid bottleneck situations when using the **ORDERED** clause, you can try using **DYNAMIC** scheduling or **STATIC** scheduling with a small chunk size. For more information on scheduling parameters, see the “SCHEDULE” on page 167 clause.

Related reference:

See -qsmp in the Compiler Reference
"PARALLEL DO / END PARALLEL DO" on page 113
"DO / END DO" on page 100
"CRITICAL / END CRITICAL" on page 99

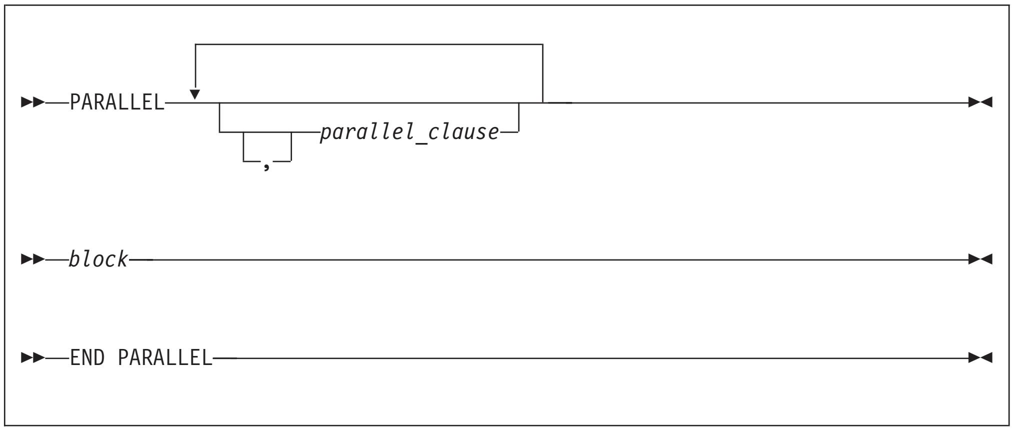
PARALLEL / END PARALLEL

Purpose

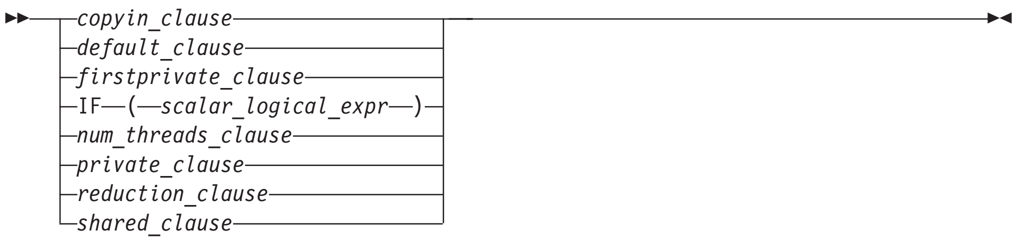
The **PARALLEL** construct enables you to define a block of code that can be executed by a team of threads concurrently. The **PARALLEL** construct includes a **PARALLEL** directive that is followed by one or more blocks of code, and ends with an **END PARALLEL** directive.

The **PARALLEL** and **END PARALLEL** directives only take effect if you specify the **-qsmp** compiler option.

Syntax



where *parallel_clause* is:



copyin_clause
See — "COPYIN" on page 150

default_clause
See — "DEFAULT" on page 152

if_clause
See — "IF" on page 156

firstprivate_clause
See — "FIRSTPRIVATE" on page 155.

num_threads_clause
    See — "NUM_THREADS" on page 160.

private_clause
    See — "PRIVATE" on page 161.

reduction_clause
    See — "REDUCTION" on page 164

shared_clause
    See — "SHARED" on page 170

Rules

It is illegal to branch into or out of a **PARALLEL** construct.

The **IF** and **DEFAULT** clauses can appear at most once in a **PARALLEL** directive.

You should be careful when you perform input/output operations in a parallel region. If multiple threads execute a Fortran I/O statement on the same unit, you should make sure that the threads are synchronized. If you do not, the behavior is undefined. See "Parallel I/O issues" on page 281 for more information. Also note that although in the XL Fortran implementation each thread has exclusive access to the I/O unit, the OpenMP specification does not require exclusive access.

Directives that bind to a parallel region will bind to that parallel region even if it is serialized.

The **END PARALLEL** directive implies the **FLUSH** directive without the *variable_name_list* and a **BARRIER** directive.

Examples

Example 1: An example of an outer **PARALLEL** directive with the **PRIVATE** clause enclosing the **PARALLEL** construct. Note: The **SHARED** clause is present on the inner **PARALLEL** construct.

```
!$OMP PARALLEL PRIVATE(X)
!$OMP DO
         DO I = 1, 10
           X(I) = I
!$OMP PARALLEL SHARED (X,Y)
!$OMP DO
      DO K = 1, 10
         Y(K,I)= K * X(I)
      END DO
!$OMP END DO
!$OMP END PARALLEL
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

Example 2: This example demonstrates the use of the **COPYIN** clause. Each thread created by the **PARALLEL** directive has its own copy of the common block **BLOCK**. The **COPYIN** clause causes the initial value of *FCTR* to be copied into the threads that execute iterations of the **DO** loop.

```
      PROGRAM TT
      COMMON /BLOCK/ FCTR
      INTEGER :: I, FCTR
!$OMP THREADPRIVATE(/BLOCK/)
      INTEGER :: A(100)
      FCTR = -1
      A = 0

!$OMP PARALLEL COPYIN(FCTR)
!$OMP DO
      DO I=1, 100
         FCTR = FCTR + I
         CALL SUB(A(I), I)
      ENDDO
!$OMP END PARALLEL

      PRINT *, A
      END PROGRAM

      SUBROUTINE SUB(AA, J)
      INTEGER :: FCTR, AA, J
      COMMON /BLOCK/ FCTR
!$OMP THREADPRIVATE(/BLOCK/)    ! EACH THREAD GETS ITS OWN COPY
                                ! OF BLOCK.
      AA = FCTR
      FCTR = FCTR - J
      END SUBROUTINE SUB
```

The expected output is:

```
0 1 2 3 ... 96 97 98 99
```

Related reference:

"FLUSH" on page 105

"PARALLEL DO / END PARALLEL DO"

See INDEPENDENT in the Language Reference

"THREADPRIVATE" on page 138

"DO / END DO" on page 100

See -qdirective in the Compiler Reference

See -qsmp in the Compiler Reference

PARALLEL DO / END PARALLEL DO

Purpose

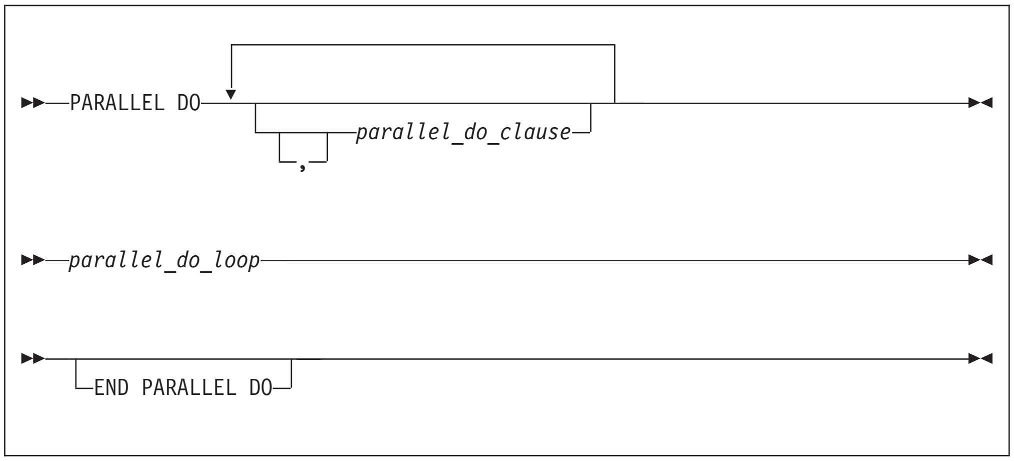
The **PARALLEL DO** directive enables you to specify which loops the compiler should parallelize. This is semantically equivalent to:

```
!$OMP PARALLEL
!$OMP DO
 ...
!$OMP ENDDO
!$OMP END PARALLEL
```

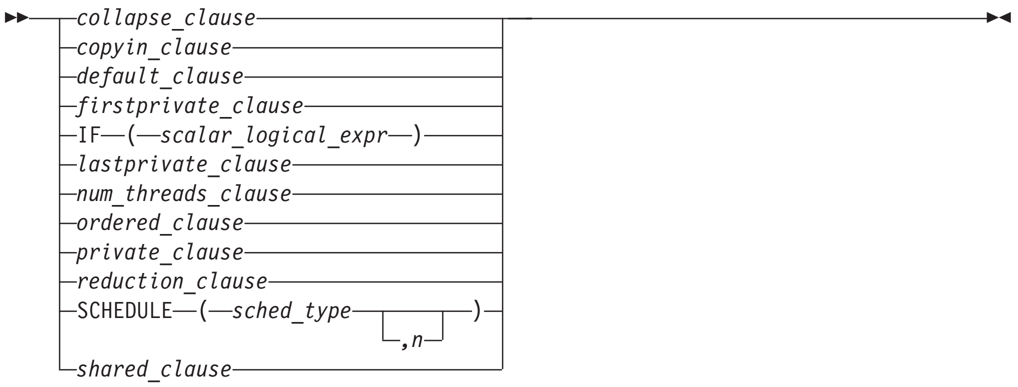
and is a convenient way of parallelizing loops. The **END PARALLEL DO** directive allows you to indicate the end of a **DO** loop that is specified by the **PARALLEL DO** directive.

The **PARALLEL DO** and **END PARALLEL DO** directives only take effect if you specify the **-qsmp** compiler option.

Syntax



where *parallel_do_clause* is:



collapse_clause
See — "COLLAPSE" on page 148

copyin_clause
See — "COPYIN" on page 150

default_clause
See — "DEFAULT" on page 152

if_clause
See — "IF" on page 156.

firstprivate_clause
See — "FIRSTPRIVATE" on page 155.

lastprivate_clause
See — "LASTPRIVATE" on page 157.

num_threads_clause
See — "NUM_THREADS" on page 160.

ordered_clause
See — "ORDERED" on page 160

private_clause
See — "PRIVATE" on page 161

reduction_clause
See — “REDUCTION” on page 164

schedule_clause
See — “SCHEDULE” on page 167

shared_clause
See — “SHARED” on page 170

Rules

The first noncomment line (not including other directives) that is following the **PARALLEL DO** directive must be a **DO** loop. This line cannot be an infinite **DO** or **DO WHILE** loop. The **PARALLEL DO** directive applies only to the **DO** loop that is immediately following the directive, and not to any nested **DO** loops, unless the **COLLAPSE** clause is specified.

If you specify a **DO** loop by a **PARALLEL DO** directive, the **END PARALLEL DO** directive is optional. If you use the **END PARALLEL DO** directive, it must immediately follow the end of the **DO** loop.

You may have a **DO** construct that contains several **DO** statements. If the **DO** statements share the same **DO** termination statement, and an **END PARALLEL DO** directive follows the construct, you can only specify a **PARALLEL DO** directive for the outermost **DO** statement of the construct.

You must not follow the **PARALLEL DO** directive by a **DO** (work-sharing) or **DO SERIAL** directive. You can specify only one **PARALLEL DO** directive for a given **DO** loop.

All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

The **PARALLEL DO** directive must not appear with the **INDEPENDENT** directive for a given **DO** loop.

Note: You should use the **PARALLEL DO** directive for maximum portability across multiple vendors. The **PARALLEL DO** directive is a prescriptive directive, while the **INDEPENDENT** directive is an assertion about the characteristics of the loop. (See the **INDEPENDENT** directive in the *XL Fortran Language Reference* for more information.)

The **IF** clause may appear at most once in a **PARALLEL DO** directive.

An **IF** expression is evaluated outside of the context of the parallel construct. Any function reference in the **IF** expression must not have side effects.

By default, a nested parallel loop is serialized, regardless of the setting of the **IF** clause. You can change this default by using the **-qsmp=nested_par** compiler option.

If the **REDUCTION** variable of an inner **DO** loop appears in the **PRIVATE** or **LASTPRIVATE** clause of an enclosing **DO** loop or **PARALLEL SECTIONS** construct, the variable must be initialized before the inner **DO** loop.

A variable that appears in the **REDUCTION** clause of an **INDEPENDENT** directive of an enclosing **DO** loop must not also appear in the *data_scope_entity_list* of the **PRIVATE** or **LASTPRIVATE** clause.

Within a **PARALLEL DO** construct, variables that do not appear in the **PRIVATE** clause are assumed to be shared by default.

You should be careful when you perform input/output operations in a parallel region. If multiple threads execute a Fortran I/O statement on the same unit, you should make sure that the threads are synchronized. If you do not, the behavior is undefined. Also note that although in the XL Fortran implementation each thread has exclusive access to the I/O unit, the OpenMP specification does not require exclusive access.

Directives that bind to a parallel region will bind to that parallel region even if it is serialized.

The **END PARALLEL DO** directive implies the **FLUSH** directive without the *variable_name_list* and a **BARRIER** directive.

Examples

Example 1: A valid example with the **LASTPRIVATE** clause.

```
!$OMP PARALLEL DO PRIVATE(I), LASTPRIVATE (X)
      DO I = 1,10
        X = I * I
        A(I) = X * B(I)
      END DO
      PRINT *, X                     ! X has the value 100
```

Example 2: A valid example with the **REDUCTION** clause.

```
!$OMP PARALLEL DO PRIVATE(I), REDUCTION(+:MYSUM)
      DO I = 1, 10
        MYSUM = MYSUM + IARR(I)
      END DO
```

Example 3: A valid example where more than one thread accesses a variable that is marked as **SHARED**, but the variable is used only in a **CRITICAL** construct.

```
!$OMP  PARALLEL DO SHARED (X)
     DO I = 1, 10
             A(I) = A(I) * I
!$OMP        CRITICAL
               X = X + A(I)
!$OMP        END CRITICAL
          END DO
```

Example 4: A valid example of the **END PARALLEL DO** directive.

```
         REAL A(100), B(2:100), C(100)
!$OMP PARALLEL DO
         DO I = 2, 100
             B(I) = (A(I) + A(I-1))/2.0
         END DO
!$OMP END PARALLEL DO
!$OMP PARALLEL DO
         DO  J = 1, 100
            C(J) = X + COS(J*5.5)
         END DO
!$OMP END PARALLEL DO
         END
```

Related reference:
"COLLAPSE" on page 148

See -qdirective in the Compiler Reference

See -qsmp in the Compiler Reference

See DO in the Language Reference
"DO / END DO" on page 100

See INDEPENDENT in the Language Reference
"ORDERED / END ORDERED" on page 108
"PARALLEL / END PARALLEL" on page 111
"PARALLEL SECTIONS / END PARALLEL SECTIONS"
"SCHEDULE" on page 120
"THREADPRIVATE" on page 138

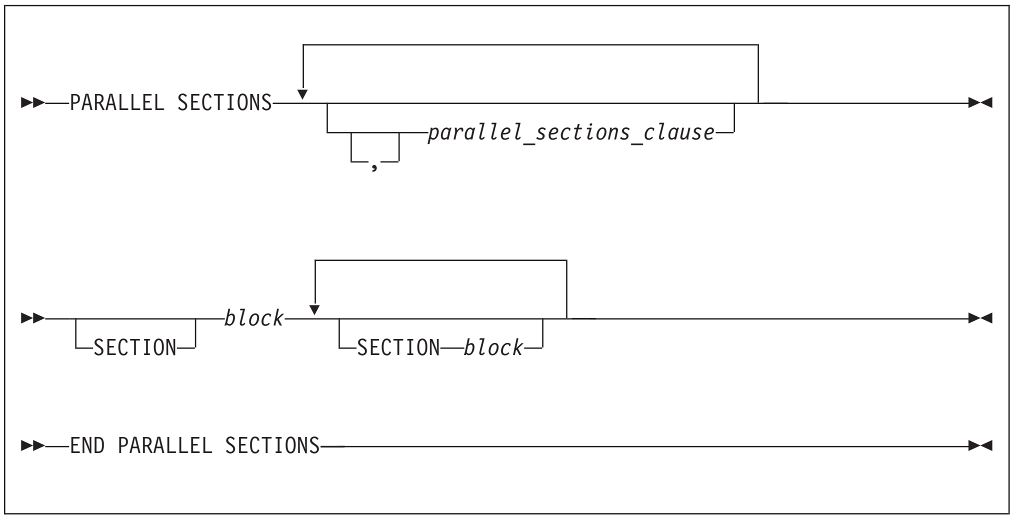
PARALLEL SECTIONS / END PARALLEL SECTIONS

Purpose

The **PARALLEL SECTIONS** construct provides a short form method for including **SECTIONS** directive inside a **PARALLEL** construct.

The **PARALLEL SECTIONS**, **SECTION** and **END PARALLEL SECTIONS** directives only take effect if you specify the **-qsmp** compiler option.

Syntax



where *parallel_sections_clause* is:

```
►►─┬─copyin_clause──────────────────┬──────────────────────────────────►◄
   ├─default_clause─────────────────┤
   ├─firstprivate_clause────────────┤
   ├─IF──(──scalar_logical_expr──)──┤
   ├─lastprivate_clause─────────────┤
   ├─num_threads_clause─────────────┤
   ├─private_clause─────────────────┤
   ├─reduction_clause───────────────┤
   └─shared_clause──────────────────┘
```

copyin_clause
    See — "COPYIN" on page 150

default_clause
    See — "DEFAULT" on page 152

firstprivate_clause
    See — "FIRSTPRIVATE" on page 155.

if_clause
    See — "IF" on page 156

lastprivate_clause
    See — "LASTPRIVATE" on page 157.

num_threads_clause
    See — "NUM_THREADS" on page 160.

private_clause
    See — "PRIVATE" on page 161.

reduction_clause
    See — "REDUCTION" on page 164

shared_clause
    See — "SHARED" on page 170

Rules

See the Rules section in "SECTIONS / END SECTIONS" on page 123.

In a **PARALLEL SECTIONS** construct, a variable that appears in the **REDUCTION** clause of an **INDEPENDENT** directive or the **PARALLEL DO** directive of an enclosing **DO** loop must not also appear in the *data_scope_entity_list* of the **PRIVATE** clause.

If the **REDUCTION** variable of the inner **PARALLEL SECTIONS** construct appears in the **PRIVATE** clause of an enclosing **DO** loop or **PARALLEL SECTIONS** construct, the variable must be initialized before the inner **PARALLEL SECTIONS** construct.

Examples

Example 1:

```
!$OMP PARALLEL SECTIONS
!$OMP   SECTION
          DO I = 1, 10
            C(I) = MAX(A(I),A(I+1))
          END DO
```

```
!$OMP   SECTION
          W = U + V
          Z = X + Y
!$OMP END PARALLEL SECTIONS
```

Example 2: In this example, the index variable I is declared as **PRIVATE**. Note also that the first optional **SECTION** directive has been omitted.

```
!$OMP PARALLEL SECTIONS PRIVATE(I)
          DO I = 1, 100
            A(I) = A(I) * I
          END DO
!$OMP   SECTION
          CALL NORMALIZE (B)
          DO I = 1, 100
            B(I) = B(I) + 1.0
          END DO
!$OMP   SECTION
          DO I = 1, 100
            C(I) = C(I) * C(I)
          END DO
!$OMP END PARALLEL SECTIONS
```

Related reference:

"PARALLEL / END PARALLEL" on page 111

"SECTIONS / END SECTIONS" on page 123

See INDEPENDENT in the Language Reference

See -qdirective in the Compiler Reference

See -qsmp in the Compiler Reference

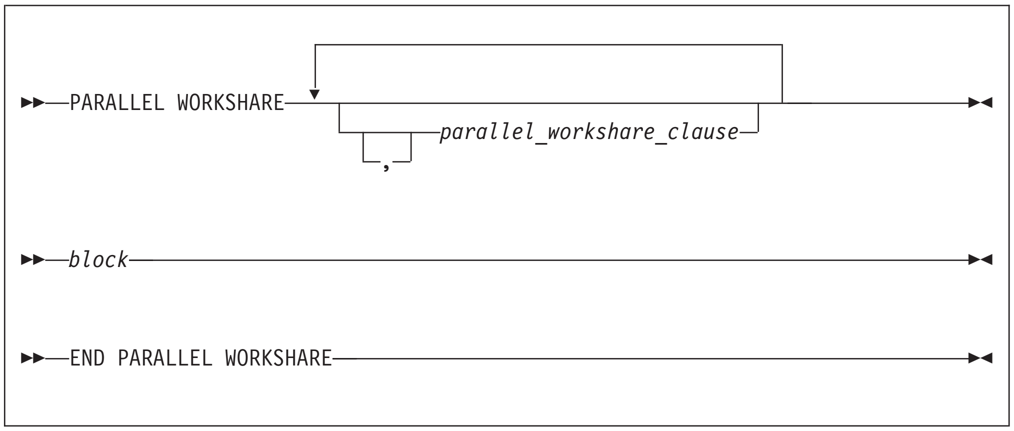
PARALLEL WORKSHARE / END PARALLEL WORKSHARE

Purpose

The **PARALLEL WORKSHARE** construct provides a short form method for including a **WORKSHARE** directive inside a **PARALLEL** construct.

The **PARALLEL WORKSHARE** / **END PARALLEL WORKSHARE** directives only take effect if you specify the **-qsmp** compiler option

Syntax



where *parallel_workshare_clause* is any of the directives accepted by either the **PARALLEL** or **WORKSHARE** directives.

Related reference:
"PARALLEL / END PARALLEL" on page 111
"WORKSHARE / END WORKSHARE" on page 144

SCHEDULE

Purpose

Note: The **SCHEDULE** directive has been deprecated and might be removed in a future release. Use the corresponding OpenMP SCHEDULE clause. For more information about the deprecated SMP directives and deprecation examples, see "Deprecated directive" on page 93.

The **SCHEDULE** directive allows the user to specify the chunking method for parallelization. Work is assigned to threads in different manners depending on the scheduling type or chunk size used.

The **SCHEDULE** directive only takes effect if you specify the **-qsmp** compiler option.

Syntax

```
►►──SCHEDULE──(──sched_type──┬─────────┬──)──────────────────────────────────────►◄
                             └─,──n────┘
```

n
: *n* must be a positive, specification expression. You must not specify *n* for the *sched_type* **RUNTIME**.

sched_type
: is **AFFINITY**, **DYNAMIC**, **GUIDED**, **RUNTIME**, or **STATIC**

For more information on *sched_type* parameters, see the **SCHEDULE** clause.

number_of_iterations
: is the number of iterations in the loop to be parallelized.

number_of_threads
: is the number of threads used by the program.

Rules

The **SCHEDULE** directive must appear in the specification part of a scoping unit.

Only one **SCHEDULE** directive may appear in the specification part of a scoping unit.

The **SCHEDULE** directive applies to the situation when all loops in the scoping unit do not already have explicit scheduling types specified. Individual loops can have scheduling types specified using the **SCHEDULE** clause of the **PARALLEL DO** directive.

Any dummy arguments appearing or referenced in the specification expression for the chunk size *n* must also appear in the **SUBROUTINE** or **FUNCTION** statement and in all **ENTRY** statements appearing in the given subprogram.

If the specified chunk size *n* is greater than the number of iterations, the loop will not be parallelized and will execute on a single thread.

If you specify more than one method of determining the chunking algorithm, the compiler will follow, in order of precedence:

1. **SCHEDULE** clause to the **PARALLEL DO** directive.
2. **SCHEDULE** directive.
3. **schedule** suboption to the **-qsmp** compiler option. See the **-qsmp** option in the *XL Fortran Compiler Reference*.
4. **XLSMPOPTS** runtime option. See "XLSMPOPTS" on page 74.
5. runtime default (that is, **STATIC**).

Examples

Example 1. Given the following information:

```
number of iterations = 1000
number of threads = 4
```

and using the **GUIDED** scheduling type, the chunk sizes would be as follows:

```
250 188 141 106 79 59 45 33 25 19 14 11 8 6 4 3 3 2 1 1 1 1
```

The iterations would then be divided into the following chunks:

```
chunk  1 = iterations    1 to  250
chunk  2 = iterations  251 to  438
chunk  3 = iterations  439 to  579
chunk  4 = iterations  580 to  685
chunk  5 = iterations  686 to  764
chunk  6 = iterations  765 to  823
chunk  7 = iterations  824 to  868
chunk  8 = iterations  869 to  901
chunk  9 = iterations  902 to  926
chunk 10 = iterations  927 to  945
chunk 11 = iterations  946 to  959
chunk 12 = iterations  960 to  970
chunk 13 = iterations  971 to  978
chunk 14 = iterations  979 to  984
chunk 15 = iterations  985 to  988
chunk 16 = iterations  989 to  991
chunk 17 = iterations  992 to  994
chunk 18 = iterations  995 to  996
chunk 19 = iterations  997 to  997
chunk 20 = iterations  998 to  998
chunk 21 = iterations  999 to  999
chunk 22 = iterations 1000 to 1000
```

A possible scenario for the division of work could be:

```
thread 1 executes chunks 1 5 10 13 18 20
thread 2 executes chunks 2 7  9 14 16 22
thread 3 executes chunks 3 6 12 15 19
thread 4 executes chunks 4 8 11 17 21
```

Example 2. Given the following information:

```
number of iterations = 100
number of threads = 4
```

and using the **AFFINITY** scheduling type, the iterations would be divided into the following partitions:

```
partition 1 = iterations  1 to 25
partition 2 = iterations 26 to  50
partition 3 = iterations 51 to  75
partition 4 = iterations 76 to 100
```

The partitions would be divided into the following chunks:

```
chunk 1a = iterations   1 to  13
chunk 1b = iterations  14 to  19
chunk 1c = iterations  20 to  22
chunk 1d = iterations  23 to  24
chunk 1e = iterations  25 to  25

chunk 2a = iterations  26 to  38
chunk 2b = iterations  39 to  44
chunk 2c = iterations  45 to  47
chunk 2d = iterations  48 to  49
chunk 2e = iterations  50 to  50

chunk 3a = iterations  51 to  63
chunk 3b = iterations  64 to  69
chunk 3c = iterations  70 to  72
chunk 3d = iterations  73 to  74
chunk 3e = iterations  75 to  75

chunk 4a = iterations  76 to  88
chunk 4b = iterations  89 to  94
chunk 4c = iterations  95 to  97
chunk 4d = iterations  98 to  99
chunk 4e = iterations 100 to 100
```

A possible scenario for the division of work could be:

```
thread 1 executes chunks 1a 1b 1c 1d 1e 4d
thread 2 executes chunks 2a 2b 2c 2d
thread 3 executes chunks 3a 3b 3c 3d 3e 2e
thread 4 executes chunks 4a 4b 4c 4e
```

In this scenario, thread 1 finished executing all the chunks in its partition and then grabbed an available chunk from the partition of thread 4. Similarly, thread 3 finished executing all the chunks in its partition and then grabbed an available chunk from the partition of thread 2.

Example 3. Given the following information:

```
number of iterations = 1000
number of threads = 4
```

and using the **DYNAMIC** scheduling type and chunk size of 100, the chunk sizes would be as follows:

```
100 100 100 100 100 100 100 100 100 100
```

The iterations would be divided into the following chunks:

```
chunk  1 = iterations   1 to  100
chunk  2 = iterations 101 to  200
chunk  3 = iterations 201 to  300
chunk  4 = iterations 301 to  400
chunk  5 = iterations 401 to  500
chunk  6 = iterations 501 to  600
chunk  7 = iterations 601 to  700
chunk  8 = iterations 701 to  800
chunk  9 = iterations 801 to  900
chunk 10 = iterations 901 to 1000
```

A possible scenario for the division of work could be:

```
thread 1 executes chunks 1  5  9
thread 2 executes chunks 2  8
thread 3 executes chunks 3  6  10
thread 4 executes chunks 4  7
```

Example 4. Given the following information:

```
number of iterations = 100
number of threads = 4
```

and using the **STATIC** scheduling type, the iterations would be divided into the following chunks:

```
chunk 1 = iterations  1 to  25
chunk 2 = iterations 26 to  50
chunk 3 = iterations 51 to  75
chunk 4 = iterations 76 to 100
```

A possible scenario for the division of work could be:

```
thread 1 executes chunks 1
thread 2 executes chunks 2
thread 3 executes chunks 3
thread 4 executes chunks 4
```

Related reference:

See DO in the Language Reference

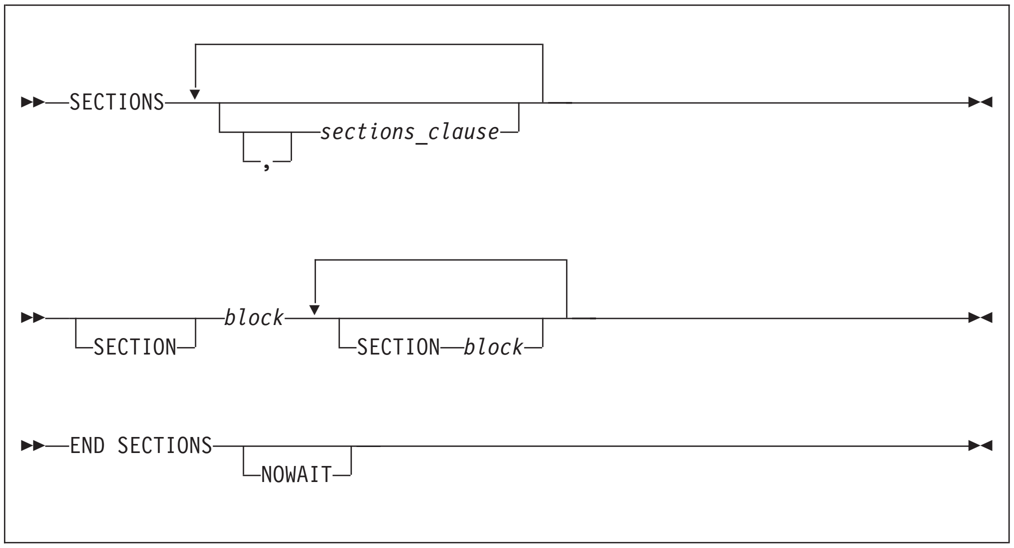
SECTIONS / END SECTIONS

Purpose

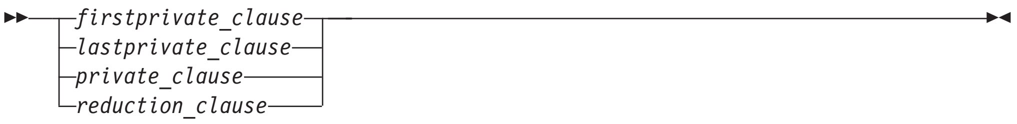
The **SECTIONS** construct defines distinct blocks of code to be executed in parallel by threads in the team.

The **SECTIONS** and **END SECTIONS** directives only take effect if you specify the **-qsmp** compiler option.

Syntax



where *sections_clause* is:



firstprivate_clause
: See — "FIRSTPRIVATE" on page 155.

lastprivate_clause
: See — "LASTPRIVATE" on page 157.

private_clause
: See — "PRIVATE" on page 161.

reduction_clause
: See — "REDUCTION" on page 164

Rules

The **SECTIONS** construct must be encountered by all threads in a team or by none of the threads in a team. All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

The **SECTIONS** construct includes the delimiting directives, and the blocks of code they enclose. At least one block of code must appear in the construct.

You must specify the **SECTION** directive at the beginning of each block of code except for the first. The end of a block is delimited by either another **SECTION** directive or by the **END SECTIONS** directive.

It is illegal to branch into or out of any block of code that is enclosed in the **SECTIONS** construct. All **SECTION** directives must appear within the lexical extent of the **SECTIONS/END SECTIONS** directive pair.

The scheduling of structured blocks among threads in the team is set so that the first thread arriving is the first thread to execute the block. The compiler determines how to divide the work among the threads based on a number of factors, such as the number of threads in the team and the number of sections to be executed in parallel. In a **SECTIONS** construct, a single thread might execute more than one **SECTION**. It is also possible that a thread in the team might not execute any **SECTION**.

In order for the directive to execute in parallel, you must place the **SECTIONS/END SECTIONS** pair within a parallel region. Otherwise, the blocks will be executed serially.

If you specify **NOWAIT** on the **SECTIONS** directive, a thread that completes its sections early will proceed to the instructions following the **SECTIONS** construct. If you do not specify the **NOWAIT** clause, each thread will wait for all of the other threads in the same team to reach the **END SECTIONS** directive. However, there is no implied **BARRIER** at the start of the **SECTIONS** construct.

You cannot specify a **SECTIONS** directive within the dynamic extent of a **CRITICAL**, **MASTER**, **ORDERED**, or **TASK** directive.

You cannot nest **SECTIONS**, **DO** or **SINGLE** directives that bind to the same **PARALLEL** directive.

BARRIER and **MASTER** directives are not permitted in the dynamic extent of a **SECTIONS** directive.

The **END SECTIONS** directive implies the **FLUSH** directive.

Examples

Example 1: This example shows a valid use of the **SECTIONS** construct within a **PARALLEL** region.

```
      INTEGER :: I, B(500), S, SUM
! ...
      S = 0
      SUM = 0
!$OMP PARALLEL SHARED(SUM), FIRSTPRIVATE(S)
!$OMP SECTIONS REDUCTION(+: SUM), LASTPRIVATE(I)
!$OMP SECTION
        S = FCT1(B(1::2))  ! Array B is not altered in FCT1.
        SUM = SUM + S
! ...
!$OMP SECTION
        S = FCT2(B(2::2))  ! Array B is not altered in FCT2.
        SUM = SUM + S
! ...
!$OMP SECTION
        DO I = 1, 500      ! The local copy of S is initialized
          S = S + B(I)     ! to zero.
        END DO
        SUM = SUM + S
! ...
!$OMP END SECTIONS
! ...
!$OMP DO REDUCTION(-: SUM)
      DO J=I-1, 1, -1    ! The loop starts at 500 -- the last
                         ! value from the previous loop.
        SUM = SUM - B(J)
      END DO

!$OMP MASTER
      SUM = SUM - FCT1(B(1::2)) - FCT2(B(2::2))
!$OMP END MASTER
!$OMP END PARALLEL
! ...
                    ! Upon termination of the PARALLEL
                    ! region, the value of SUM remains zero.
```

Example 2: This example shows a valid use of nested **SECTIONS**.

```
!$OMP PARALLEL
!$OMP MASTER
      CALL RANDOM_NUMBER(CX)
      CALL RANDOM_NUMBER(CY)
      CALL RANDOM_NUMBER(CZ)
!$OMP END MASTER

!$OMP SECTIONS
!$OMP SECTION
!$OMP   PARALLEL
!$OMP   SECTIONS PRIVATE(I)
!$OMP   SECTION
          DO I=1, 5000
            X(I) = X(I) + CX
          END DO
!$OMP   SECTION
          DO I=1, 5000
            Y(I) = Y(I) + CY
          END DO
!$OMP   END SECTIONS
!$OMP   END PARALLEL

!$OMP SECTION
!$OMP   PARALLEL SHARED(CZ,Z)
```

```
!$OMP   DO
        DO I=1, 5000
          Z(I) = Z(I) + CZ
        END DO
!$OMP   END DO
!$OMP   END PARALLEL
!$OMP END SECTIONS NOWAIT

! The following computations do not
! depend on the results from the
! previous section.

!$OMP DO
      DO I=1, 5000
        T(I) = T(I) * CT
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

Related reference:

"PARALLEL / END PARALLEL" on page 111

"BARRIER" on page 97

"PARALLEL DO / END PARALLEL DO" on page 113

See INDEPENDENT in the Language Reference

"THREADPRIVATE" on page 138

See -qdirective in the Compiler Reference

See -qsmp in the Compiler Reference

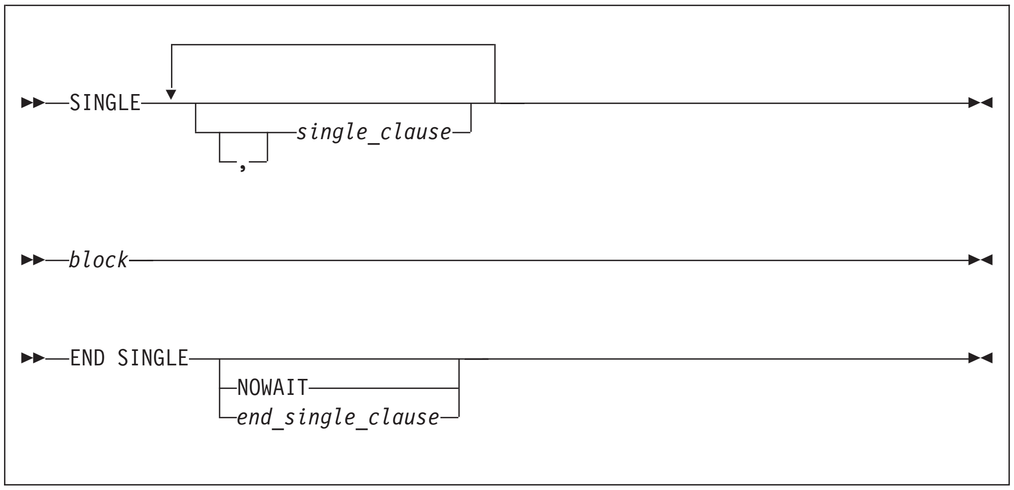
SINGLE / END SINGLE

Purpose

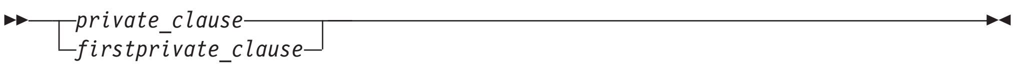
You can use the **SINGLE / END SINGLE** directive construct to specify that the enclosed code should only be executed by one thread in the team.

The **SINGLE** directive only takes effect if you specify the **–qsmp** compiler option.

Syntax



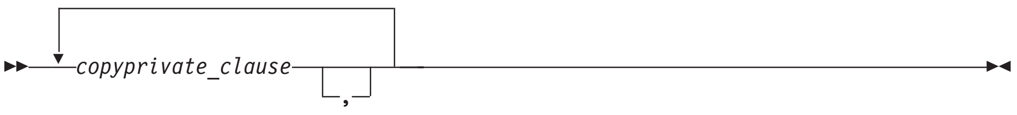
where *single_clause* is:



private_clause
: See — "PRIVATE" on page 161.

firstprivate_clause
: See — "FIRSTPRIVATE" on page 155.

where *end_single_clause* is:



NOWAIT

copyprivate_clause
: See — "COPYPRIVATE" on page 151.

Rules

It is illegal to branch into or out of a block that is enclosed within the **SINGLE** construct.

The **SINGLE** construct must be encountered by all threads in a team or by none of the threads in a team. The first thread to encounter the **SINGLE** construct will execute it. All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

If you specify **NOWAIT** on the **END SINGLE** directive, the threads that are not executing the **SINGLE** construct will proceed to the instructions following the **SINGLE** construct. If you do not specify the **NOWAIT** clause, each thread will wait at the **END SINGLE** directive until the thread executing the construct reaches the **END SINGLE** directive. You may not specify **NOWAIT** and **COPYPRIVATE** as part of the same **END SINGLE** directive.

There is no implied **BARRIER** at the start of the **SINGLE** construct. If you do not specify the **NOWAIT** clause, the **BARRIER** directive is implied at the **END SINGLE** directive.

You cannot nest work-sharing constructs inside one another if they bind to the same **PARALLEL** directive.

SINGLE directives are not permitted within the **CRITICAL**, **MASTER**, **ORDERED**, or **TASK** regions. **BARRIER** and **MASTER** directives are not permitted within the **SINGLE** regions.

If you have specified a variable as **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE**, or **REDUCTION** in the **PARALLEL** construct which encloses your **SINGLE** construct, you cannot specify the same variable in the **PRIVATE** or **FIRSTPRIVATE** clause of the **SINGLE** construct.

The **SINGLE** directive binds to the closest enclosing **PARALLEL** region, if one exists.

Examples

Example 1: In this example, the **BARRIER** directive is used to ensure that all threads finish their work before entering the **SINGLE** construct.

```
      REAL :: X(100), Y(50)
!     ...
!$OMP PARALLEL DEFAULT(SHARED)
      CALL WORK(X)

!$OMP BARRIER
!$OMP SINGLE
      CALL OUTPUT(X)
      CALL INPUT(Y)
!$OMP END SINGLE

      CALL WORK(Y)
!$OMP END PARALLEL
```

Example 2: In this example, the **SINGLE** construct ensures that only one thread is executing a block of code. In this case, array *B* is initialized in the **DO** (work-sharing) construct. After the initialization, a single thread is employed to perform the summation.

```
      INTEGER :: I, J
      REAL :: B(500,500), SM
!     ...

      J = ...
      SM = 0.0
!$OMP PARALLEL
!$OMP DO PRIVATE(I)
      DO I=1, 500
        CALL INITARR(B(I,:), I)       ! initialize the array B
      ENDDO
!$OMP END DO

!$OMP SINGLE                          ! employ only one thread
      DO I=1, 500
        SM = SM + SUM(B(J:J+1,I))
      ENDDO
!$OMP END SINGLE

!$OMP DO PRIVATE(I)
      DO I=500, 1, -1
        CALL INITARR(B(I,:), 501-I)   ! re-initialize the array B
      ENDDO
!$OMP END PARALLEL
```

Example 3: This example shows a valid use of the **PRIVATE** clause. Array *X* is **PRIVATE** to the **SINGLE** construct. If you were to reference array *X* immediately following the construct, it would be undefined.

```
      REAL :: X(2000), A(1000), B(1000)

!$OMP PARALLEL
!     ...
!$OMP SINGLE PRIVATE(X)
      CALL READ_IN_DATA(X)
      A = X(1::2)
      B = X(2::2)
!$OMP END SINGLE
!     ...
!$OMP END PARALLEL
```

Example 4: In this example, the **LASTPRIVATE** variable *I* is used in allocating *TMP*, the **PRIVATE** variable in the **SINGLE** construct.

```
      SUBROUTINE ADD(A, UPPERBOUND)
        INTEGER :: A(UPPERBOUND), I, UPPERBOUND
        INTEGER, ALLOCATABLE :: TMP(:)
!  ...
!$OMP   PARALLEL
!$OMP   DO LASTPRIVATE(I)
        DO I=1, UPPERBOUND
          A(I) = I + 1
        ENDDO
!$OMP   END DO

!$OMP   SINGLE FIRSTPRIVATE(I), PRIVATE(TMP)
        ALLOCATE(TMP(0:I-1))
        TMP = (/ (A(J),J=I,1,-1) /)
!  ...
        DEALLOCATE(TMP)
!$OMP   END SINGLE
!$OMP   END PARALLEL
!  ...
      END SUBROUTINE ADD
```

Example 5: In this example, a value for the variable *I* is entered by the user. This value is then copied into the corresponding variable *I* for all other threads in the team using a **COPYPRIVATE** clause on an **END SINGLE** directive.

```
        INTEGER I
  !$OMP PARALLEL PRIVATE (I)
  !     ...
  !$OMP SINGLE
        READ (*, *) I
  !$OMP END SINGLE COPYPRIVATE (I)   ! In all threads in the team, I
                                     ! is equal to the value
  !     ...                          ! that you entered.
  !$OMP END PARALLEL
```

Example 6: In this example, variable *J* with a **POINTER** attribute is specified in a **COPYPRIVATE** clause on an **END SINGLE** directive. The value of *J*, not the value of the object that it points to, is copied into the corresponding variable *J* for all other threads in the team. The object itself is shared among all the threads in the team.

```
        INTEGER, POINTER :: J
  !$OMP PARALLEL PRIVATE (J)
  ! ...
  !$OMP SINGLE
        ALLOCATE (J)
        READ (*, *) J
  !$OMP END SINGLE COPYPRIVATE (J)
  !$OMP ATOMIC
        J = J + OMP_GET_THREAD_NUM()
  !$OMP BARRIER
  !$OMP SINGLE
        WRITE (*, *) 'J = ', J   ! The result is the sum of all values added to
                                 ! J. This result shows that the pointer object
                                 ! is shared by all threads in the team.
        DEALLOCATE (J)
  !$OMP END SINGLE
  !$OMP END PARALLEL
```

Related reference:
"BARRIER" on page 97
"CRITICAL / END CRITICAL" on page 99
"FLUSH" on page 105
"MASTER / END MASTER" on page 107
"PARALLEL / END PARALLEL" on page 111

SPECULATIVE DO / END SPECULATIVE DO

The **SPECULATIVE DO** directive instructs the compiler to speculatively parallelize a **DO** loop.

Syntax

```
                                 ┌──────────────────────────┐
                                 │  ┌─,─┘                   │
                                 ▼                          │
►►──!SEP$──SPECULATIVE DO──────────────────────────────────┴──────────────────────────►◄
                                   └─speculative_do_clause─┘


►►──────────────────────────────────────────────────────────────────────────────────►◄
    └─!SEP$──END SPECULATIVE DO─┘
```

where *speculative_do_clause* is:

```
►►──┬─default_clause──────┬─────────────────────────────────────────────────────────►◄
    ├─private_clause──────┤
    ├─firstprivate_clause─┤
    ├─lastprivate_clause──┤
    ├─shared_clause───────┤
    ├─num_threads_clause──┤
    ├─reduction_clause────┤
    └─schedule_clause─────┘
```

speculative_do_clause is any of the following clauses:

default_clause
For details, see "DEFAULT" on page 152.

private_clause
For details, see "PRIVATE" on page 161.

firstprivate_clause
For details, see "FIRSTPRIVATE" on page 155.

lastprivate_clause
For details, see "LASTPRIVATE" on page 157.

shared_clause
For details, see "SHARED" on page 170.

num_threads_clause
For details, see "NUM_THREADS" on page 160.

reduction_clause
For details, see "REDUCTION" on page 164.

schedule_clause
For details, see "SCHEDULE" on page 167.

Usage

The directives for thread-level speculative execution only take effect if you specify the -qsmp=speculative compiler option.

The **SPECULATIVE DO** directive precedes a **DO** loop. The optional **END SPECULATIVE DO** directive indicates the end of the **DO** loop.

Example

```
INTEGER :: p = 0
INTEGER :: i = 1

!SEP$ SPECULATIVE DO FIRSTPRIVATE(p)
DO i = 1, 10
   p = p + 1
END DO

END
```

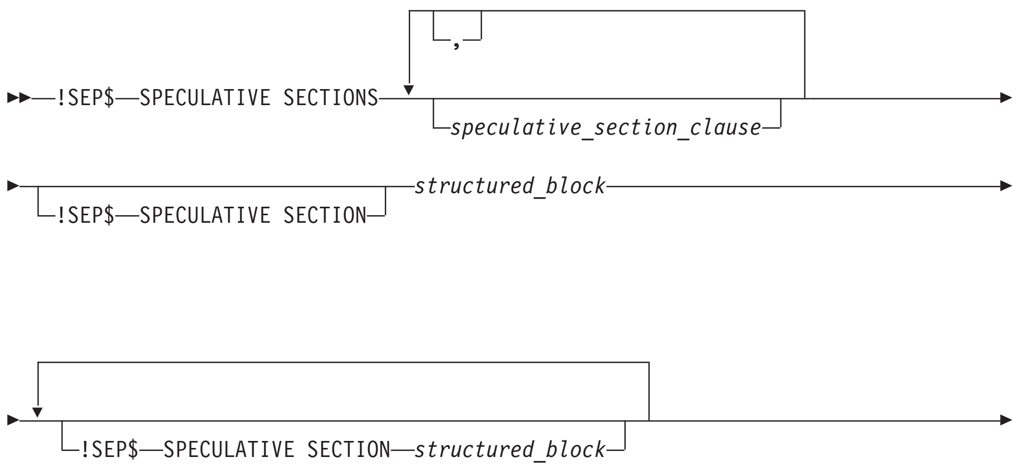
Related information

- The "-qsmp" compiler option
- Thread-level speculative execution
- SPECULATIVE SECTIONS
- Routines for thread-level speculative execution
- Environment variables for thread-level speculative execution

SPECULATIVE SECTIONS / END SPECULATIVE SECTIONS

The **SPECULATIVE SECTIONS** directive instructs the compiler to speculatively parallelize sections of the code. In code blocks delimited by **SPECULATIVE SECTIONS** and **END SPECULATIVE SECTIONS**, you can use the **SPECULATIVE SECTION** directive to delimit program code segments.

Syntax


!SEP$—SPECULATIVE SECTIONS
,
speculative_section_clause
!SEP$—SPECULATIVE SECTION
structured_block
!SEP$—SPECULATIVE SECTION—structured_block

```
►─!SEP$──END SPECULATIVE SECTIONS──────────────────────────────────────────►◄
```

where *speculative_section_clause* is:

```
►►──┬─default_clause──────┬────────────────────────────────────────────────►◄
    ├─private_clause──────┤
    ├─firstprivate_clause─┤
    ├─lastprivate_clause──┤
    ├─shared_clause───────┤
    ├─num_threads_clause──┤
    └─reduction_clause────┘
```

speculative_section_clause is any of the following clauses:

default_clause
For details, see "DEFAULT" on page 152.

private_clause
For details, see "PRIVATE" on page 161.

firstprivate_clause
For details, see "FIRSTPRIVATE" on page 155.

lastprivate_clause
For details, see "LASTPRIVATE" on page 157.

shared_clause
For details, see "SHARED" on page 170.

num_threads_clause
For details, see "NUM_THREADS" on page 160.

reduction_clause
For details, see "REDUCTION" on page 164.

Usage

The directives for thread-level speculative execution only take effect if you specify the "-qsmp=speculative" compiler option.

The **SPECULATIVE SECTION** directive is optional for the first code segment inside the **SPECULATIVE SECTIONS** directive. The code segments after the first one must be preceded by a **SPECULATIVE SECTION** directive. You must use all the **SPECULATIVE SECTION** directives only in the lexical construct of the code segment that is delimited by the **SPECULATIVE SECTIONS** and **END SPECULATIVE SECTIONS** directives.

Example

```
INTEGER :: p = 0
INTEGER :: i = 1

!SEP$ SPECULATIVE SECTIONS FIRSTPRIVATE(p)
   p = p + 1
!SEP$ SPECULATIVE SECTION
```

```
   i = i - 1
!SEP$ END SPECULATIVE SECTIONS

END
```

Related information

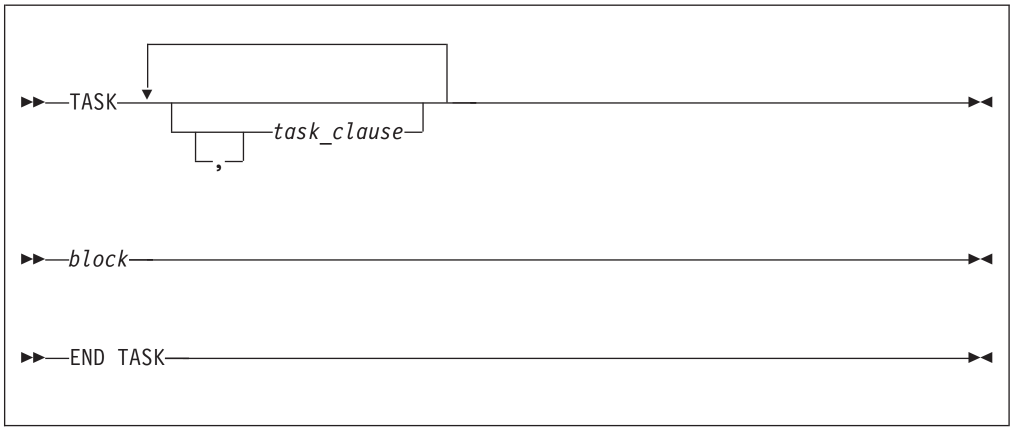
- The "-qsmp" compiler option
- Thread-level speculative execution
- SPECULATIVE DO / END SPECULATIVE DO
- Routines for thread-level speculative execution
- Environment variables for thread-level speculative execution

TASK / END TASK

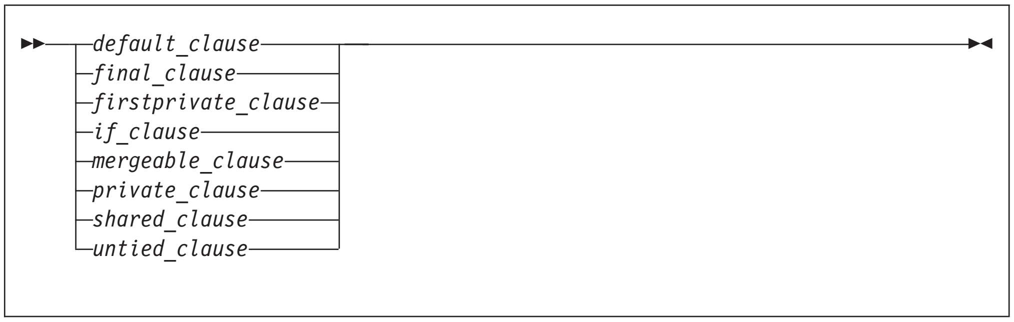
Purpose

The **TASK** directive instructs the compiler to run a block of code in parallel with the code outside the task region. The **TASK** directive can be useful for parallelizing irregular algorithms such as pointer chasing or recursive algorithms. The **TASK** directive takes effect only if you specify the **-qsmp** compiler option.

Syntax



where *task_clause* is:



default_clause
: See "DEFAULT" on page 152.

final_clause
: See "FINAL" on page 154.

firstprivate_clause
See “FIRSTPRIVATE” on page 155.

if_clause
See “IF” on page 156.

mergeable_clause
See “MERGEABLE” on page 159.

private_clause
See “PRIVATE” on page 161.

shared_clause
See “SHARED” on page 170.

untied_clause
See “UNTIED” on page 172.

Rules

A final task is a task that makes all its child tasks become final and included tasks. A final task is generated when either of the following conditions is true:

- A **FINAL** clause is specified on a task construct and the **FINAL** clause expression evaluates to `.TRUE.`.
- The generated task is a child task of a final task.

An undeferred task is a task whose execution is not deferred with respect to its generating task region. In other words, the generating task region is suspended until the undeferred task has finished running. An undeferred task is generated when an **IF** clause is specified on a task construct and the **IF** clause expression evaluates to `.FALSE.`.

An included task is a task whose execution is sequentially included in the generating task region. In other words, an included task is undeferred and executed immediately by the encountering thread. An included task is generated when the generated task is a child task of a final task.

A merged task is a task that has the same data environment as that of its generating task region. A merged task might be generated when both the following conditions are true:

- A **MERGEABLE** clause is specified on a task construct.
- The generated task is an undeferred task or an included task.

The following rules are true if no **DEFAULT** clause is specified with the enclosing **TASK** construct:

- If the enclosing **TASK** construct is not lexically enclosed by a parallel region, dummy arguments that do not appear in any **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE**, or **SHARED** clause of the enclosing **TASK** construct are firstprivate.
- A variable that is private in the innermost enclosing parallel construct is firstprivate in the **TASK** construct.
- Local variables of a routine are firstprivate if there is no enclosing parallel construct.
- A variable that is determined to be shared in all of the enclosing constructs, up to and including the innermost enclosing parallel construct, is shared.

The **IF** clause expression and the **FINAL** clause expression are evaluated outside of the task construct, and the evaluation order is not specified.

Related reference:
"FINAL" on page 154
"FIRSTPRIVATE" on page 155
"IF" on page 156
"MERGEABLE" on page 159
"DEFAULT" on page 152
"PRIVATE" on page 161
"SHARED" on page 170
"TASKWAIT"
"UNTIED" on page 172

TASKWAIT

Purpose

The **TASKWAIT** directive specifies a *wait* for child tasks to be completed that are generated by the current task.

Syntax

```
►►──TASKWAIT─────────────────────────────────────────────────────►◄
```

Related reference:
"TASK / END TASK" on page 133

TASKYIELD

Purpose

The **TASKYIELD** directive instructs the compiler that it can suspend the current task in favor of running a different task. The **TASKYIELD** region includes an explicit task scheduling point in the current task region.

Syntax

```
►►──TASKYIELD────────────────────────────────────────────────────►◄
```

THREADLOCAL

Purpose

You can use the **THREADLOCAL** directive to declare thread-specific common data. It is a possible method of ensuring that access to data that is contained within **COMMON** blocks is serialized.

In order to make use of this directive it is not necessary to specify the **-qsmp** compiler option, but the invocation command must be **bgxlf_r**, **bgxlf90_r**, **bgxlf95_r**, **bgxlf2003_r**, or **bgxlf2008_r** to link the necessary libraries.

Syntax

```
                         ┌─,──────────────────────┐
                         ▼                        │
►►──THREADLOCAL──┬────┬──/──common_block_name──/──┴──────────────────►◄
                 └─::─┘
```

Rules

You can only declare named blocks as **THREADLOCAL**. All rules and constraints that normally apply to named common blocks apply to common blocks that are declared as **THREADLOCAL**. See the **COMMON** statement in the *XL Fortran Language Reference* for more information on the rules and constraints that apply to named common blocks.

The **THREADLOCAL** directive must appear in the *specification_part* of the scoping unit. If a common block appears in a **THREADLOCAL** directive, it must also be declared within a **COMMON** statement in the same scoping unit. The **THREADLOCAL** directive may occur before or after the **COMMON** statement. See Main program in the *XL Fortran Language Reference* for more information on the *specification_part* of the scoping unit.

A common block cannot be given the **THREADLOCAL** attribute if it is declared within a **PURE** subprogram.

Members of a **THREADLOCAL** common block must not appear in **NAMELIST** statements.

A common block that is use-associated must not be declared as **THREADLOCAL** in the scoping unit that contains the **USE** statement.

Any pointers declared in a **THREADLOCAL** common block are not affected by the **-qinit=f90ptr** compiler option.

Objects within **THREADLOCAL** common blocks may be used in parallel loops and parallel sections. However, these objects are implicitly shared across the iterations of the loop, and across code blocks within parallel sections. In other words, within a scoping unit, all accessible common blocks, whether declared as **THREADLOCAL** or not, have the **SHARED** attribute within parallel loops and sections in that scoping unit.

If a common block is declared as **THREADLOCAL** within a scoping unit, any subprogram that declares or references the common block, and that is directly or indirectly referenced by the scoping unit, must be executed by the same thread executing the scoping unit. If two procedures that declare common blocks are executed by different threads, then they would obtain different copies of the common block, provided that the common block had been declared **THREADLOCAL**. Threads can be created in one of the following ways:

- Explicitly, via *pthreads* library calls
- Implicitly by the compiler for parallel loop execution
- Implicitly by the compiler for parallel section execution.

If a common block is declared to be **THREADLOCAL** in one scoping unit, it must be declared to be **THREADLOCAL** in every scoping unit that declares the common block.

If a **THREADLOCAL** common block that does not have the **SAVE** attribute is declared within a subprogram, the members of the block become undefined at subprogram RETURN or END, unless there is at least one other scoping unit in which the common block is accessible that is making a direct or indirect reference to the subprogram.

You cannot specify the same *common_block_name* for both a **THREADLOCAL** directive and a **THREADPRIVATE** directive.

Example 1: The following procedure "FORT_SUB" is invoked by two threads:

```
SUBROUTINE FORT_SUB(IARG)
  INTEGER IARG

  CALL LIBRARY_ROUTINE1()
  CALL LIBRARY_ROUTINE2()
  ...
END SUBROUTINE FORT_SUB
SUBROUTINE LIBRARY_ROUTINE1()
  COMMON /BLOCK/ R               ! The SAVE attribute is required for the
  SAVE /BLOCK/                   ! common block because the program requires
                                 ! that the block remain defined after
  !IBM* THREADLOCAL /BLOCK/      ! library_routine1 is invoked.
    R = 1.0
    ...
END SUBROUTINE LIBRARY_ROUTINE1
SUBROUTINE LIBRARY_ROUTINE2()
  COMMON /BLOCK/ R
  SAVE /BLOCK/
  !IBM* THREADLOCAL /BLOCK/

  ... = R
  ...
END SUBROUTINE LIBRARY_ROUTINE2
```

Example 2: "FORT_SUB" is invoked by multiple threads. This is an invalid example because "FORT_SUB" and "ANOTHER_SUB" both declare /BLOCK/ to be THREADLOCAL. They intend to share the common block, but they are executed by different threads.

```
SUBROUTINE FORT_SUB()
  COMMON /BLOCK/ J
  INTEGER :: J
  !IBM* THREADLOCAL /BLOCK/        ! Each thread executing FORT_SUB
                                   ! obtains its own copy of /BLOCK/
  INTEGER A(10)

  ...
  !IBM* INDEPENDENT
  DO INDEX = 1,10
    CALL ANOTHER_SUB(A(I))
  END DO
  ...

END SUBROUTINE FORT_SUB
SUBROUTINE ANOTHER_SUB(AA)         ! Multiple threads
are used to execute ANOTHER_SUB
  INTEGER AA
  COMMON /BLOCK/ J                 ! Each thread obtains a new copy of the
  INTEGER :: J                     !  common block /BLOCK/
  !IBM* THREADLOCAL /BLOCK/
  ...
  AA = J                           ! The value of 'J' is undefined.
END SUBROUTINE ANOTHER_SUB
```

One or more sample programs in the directory `/opt/ibmcmp/xlf/bg/14.1/samples/modules/threadlocal` illustrate how to use the **THREADLOCAL** directive and create threads in C.

Related reference:

See -qdirective in the Compiler Reference

See -qinit in the Compiler Reference

See COMMON in the Language Reference

See Main program in the Language Reference

THREADPRIVATE

Purpose

The **THREADPRIVATE** directive allows you to specify named common blocks and named variables as private to a thread but global within that thread. Once you declare a common block or variable **THREADPRIVATE**, each thread in the team maintains a separate copy of that common block or variable. Data written to a **THREADPRIVATE** common block or variable remains private to that thread and is not visible to other threads in the team.

In the serial and **MASTER** sections of a program, only the master thread's copy of the named common block and variable is accessible.

Use the **COPYIN** clause on the **PARALLEL**, **PARALLEL DO**, **PARALLEL SECTIONS** or **PARALLEL WORKSHARE** directives to specify that upon entry into a parallel region, data in the master thread's copy of a named common block or named variable is copied to each thread's private copy of that common block or variable.

The **THREADPRIVATE** directive only takes effect if you specify the **-qsmp** compiler option.

Syntax

```
►►──THREADPRIVATE──(──threadprivate_entity_list──)─────────────────────►◄
```

where *threadprivate_entity_list* is:

```
►►──┬─variable_name─────────┬──────────────────────────────────────────►◄
    └─/ common_block_name /─┘
```

common_block_name
: is the name of a common block to be made private to a thread.

variable_name
is the name of a variable to be made private to a thread.

Rules

You cannot specify a **THREADPRIVATE** variable, common block, or the variables that comprise that common block in a **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE**, **SHARED**, or **REDUCTION** clause.

A **THREADPRIVATE** variable must have the **SAVE** attribute. For variables or common blocks declared in the scope of a module, the **SAVE** attribute is implied. If you declare the variable outside of the scope of the module, the **SAVE** attribute must be specified.

In **THREADPRIVATE** directives, you can only specify named variables and named common blocks.

A variable can only appear in a **THREADPRIVATE** directive in the scope in which it is declared, and a **THREADPRIVATE** variable or common block may only appear once in a given scope. The variable must not be an element of a common block, or be declared in an **EQUIVALENCE** statement.

You cannot specify the same *common_block_name* for both a **THREADPRIVATE** directive and a **THREADLOCAL** directive.

All rules and constraints that apply to named common blocks also apply to common blocks declared as **THREADPRIVATE**. See the **COMMON** statement in the *XL Fortran Language Reference*.

If you declare a common block as **THREADPRIVATE** in one scoping unit, you must declare it as **THREADPRIVATE** in all other scoping units in which it is declared.

On entry into any parallel region, a **THREADPRIVATE** variable, or a variable in a **THREADPRIVATE** common block specified in a **COPYIN** clause is subject to the criteria stated in the **Rules** section for the **COPYIN** clause.

On entry into the first parallel region of the program, **THREADPRIVATE** variables or variables within a **THREADPRIVATE** common block not specified in a **COPYIN** clause are subject to the following criteria:
- If the variable has the **ALLOCATABLE** attribute, the initial allocation status of each copy of that variable is not currently allocated.
- If the variable has the **POINTER** attribute, and that pointer is disassociated through either explicit or default initialization, the association status of each copy of that variable is disassociated. Otherwise, the association status of the pointer is undefined.
- If the variable has neither the **ALLOCATABLE** nor the **POINTER** attribute and is defined through either explicit or default initialization, then each copy of that variable is defined. If the variable is undefined, then each copy of that variable is undefined.

On entry into subsequent parallel regions of the program, **THREADPRIVATE** variables, or variables within a **THREADPRIVATE** common block not specified in a **COPYIN** clause, are subject to the following criteria:

- If you are using the **OMP_DYNAMIC** environment variable, or the **omp_set_dynamic** subroutine to enable dynamic threads and:
  - If the number of threads is smaller than the number of threads in the previous region, and if a **THREADPRIVATE** object is referenced in both regions, then threads with the same thread number in their respective regions will reference the same copy of that variable.
  - If the number of threads is larger than the number of threads in the previous region, then the definition and association status of a **THREADPRIVATE** object is undefined, and the allocation status is undefined.
- If dynamic threads are disabled, the definition, association, or allocation status and definition, if the thread's copy of the variable was defined, is retained.

You cannot access the name of a common block by use association or host association. Thus, a named common block can only appear on a **THREADPRIVATE** directive if the common block is declared in the scoping unit that contains the **THREADPRIVATE** directive. However, you can access the variables in the common block by use association or host association. For more information, see Host and Use association in the *XL Fortran Language Reference*.

The **-qinit=f90ptr** compiler option does not affect pointers that you have declared in a **THREADPRIVATE** common block.

The **DEFAULT** clause does not affect variables in **THREADPRIVATE** common blocks.

Examples

Example 1: In this example, the **PARALLEL DO** directive invokes multiple threads that call **SUB1**. The common block **BLK** in **SUB1** shares the data that is specific to the thread with subroutine **SUB2**, which is called by **SUB1**.

```
      PROGRAM TT
        INTEGER :: I, B(50)

!$OMP   PARALLEL DO SCHEDULE(STATIC, 10)
        DO I=1, 50
          CALL SUB1(I, B(I))       ! Multiple threads call SUB1.
        ENDDO
      END PROGRAM TT

      SUBROUTINE SUB1(J, X)
        INTEGER :: J, X, A(100)
        COMMON /BLK/ A
!$OMP   THREADPRIVATE(/BLK/)   ! Array a is private to each thread.
!  ...
        CALL SUB2(J)
        X = A(J) + A(J + 50)
!  ...
      END SUBROUTINE SUB1

      SUBROUTINE SUB2(K)
        INTEGER :: C(100)
        COMMON /BLK/ C
!$OMP   THREADPRIVATE(/BLK/)
!  ...
        C = K
!  ...                       ! Since each thread has its own copy of
                             ! common block BLK, the assignment of
                             ! array C has no effect on the copies of
                             ! that block owned by other threads.
END SUBROUTINE SUB2
```

Example 2: In this example, each thread has its own copy of the common block **ARR** in the parallel section. If one thread initializes the common block variable **TEMP**, the initial value is not visible to other threads.

```
      PROGRAM ABC
        INTEGER :: I, TEMP(100), ARR1(50), ARR2(50)
        COMMON /ARR/ TEMP
!$OMP   THREADPRIVATE(/ARR/)
        INTERFACE
          SUBROUTINE SUBS(X)
            INTEGER :: X(:)
          END SUBROUTINE
        END INTERFACE
! ...
!$OMP   PARALLEL SECTIONS
!$OMP   SECTION                   ! The thread has its own copy of the
! ...                             ! common block ARR.
          TEMP(1:100:2) = -1
          TEMP(2:100:2) = 2
          CALL SUBS(ARR1)
! ...
!$OMP   SECTION                   ! The thread has its own copy of the
! ...                             ! common block ARR.
          TEMP(1:100:2) = 1
          TEMP(2:100:2) = -2
          CALL SUBS(ARR2)
! ...
!$OMP    END PARALLEL SECTIONS
! ...
        PRINT *, SUM(ARR1), SUM(ARR2)
      END PROGRAM ABC

      SUBROUTINE SUBS(X)
        INTEGER :: K, X(:), TEMP(100)
        COMMON /ARR/ TEMP
!$OMP   THREADPRIVATE(/ARR/)
!  ...
        DO K = 1, UBOUND(X, 1)
          X(K) = TEMP(K) + TEMP(K + 1)    ! The thread is accessing its
                                          ! own copy of
                                          ! the common block.
        ENDDO
! ...
      END SUBROUTINE SUBS
```

The expected output for this program is:

```
50 -50
```

Example 3: In the following example, local variables outside of a common block are declared **THREADPRIVATE**.

```
      MODULE MDL
        INTEGER          :: A(2)
        INTEGER, POINTER :: P
        INTEGER, TARGET  :: T
!$OMP THREADPRIVATE(A, P)
      END MODULE MDL


      PROGRAM MVAR
      USE OMP_LIB
      USE MDL

      INTEGER :: I

      CALL OMP_SET_NUM_THREADS(2)
      A = (/1, 2/)
      T = 4
      P => T

!$OMP PARALLEL PRIVATE(I) COPYIN(A, P)
      I = OMP_GET_THREAD_NUM()
      IF (I .EQ. 0) THEN
        A(1) = 100
        T = 5
      ELSE IF (I .EQ. 1) THEN
        A(2) = 200
      END IF
!$OMP END PARALLEL

!$OMP PARALLEL PRIVATE(I)
      I = OMP_GET_THREAD_NUM()
      IF (I .EQ. 0) THEN
        PRINT *, 'A(2) = ', A(2)
      ELSE IF (I .EQ. 1) THEN
        PRINT *, 'A(1) = ', A(1)
        PRINT *, 'P => ', P
      END IF
!$OMP END PARALLEL

      END PROGRAM MVAR
```

If dynamic threads mechanism is disabled, the expected output is:

```
A(2) = 2
A(1) = 1
P => 5
or
A(1) = 1
P => 5
A(2) = 2
```

Related reference:

See COMMON in the Language Reference

“OMP_DYNAMIC” on page 79

“omp_set_dynamic(enable_expr)” on page 187

“PARALLEL / END PARALLEL” on page 111

“PARALLEL DO / END PARALLEL DO” on page 113

“PARALLEL SECTIONS / END PARALLEL SECTIONS” on page 117

TM_ATOMIC / END TM_ATOMIC

The **TM_ATOMIC** directive indicates a transactional atomic region.

Syntax

```
►►──!TM$──TM_ATOMIC──┬───────────┬──────────────────────────────────►◄
                     └─safe_mode─┘


►►──!TM$──END TM_ATOMIC─────────────────────────────────────────────►◄
```

safe_mode
: Using the **safe_mode** clause reduces overhead and increases performance.

However, if **safe_mode** is specified, irrevocable actions are not checked at runtime. The run result is undefined if an irrevocable action occurs during the execution.

Usage

The transactional memory directive is enabled with the -qtm compiler option. To compile your program with transactional memory, you must use thread safe invocation commands.

This directive must be placed immediately before the code block of the transactional atomic region.

A transactional atomic region must be one of the following structured blocks:

- A block of executable statements with a single entry at the top and a single exit at the bottom
- An OpenMP construct

You cannot branch into or break out from the middle of a transactional atomic region. For example, you cannot use a **GO TO** statement to transfer control out of a transactional atomic region.

When you use the **END=**, **ERR=**, or **EOR=** I/O statement specifier for branching, specify a statement that is within the same transactional atomic region as the I/O statement.

Transactional atomic regions can be nested and the maximum nesting level is 2^{22}. However, the atomicity, consistency, and isolation properties are provided at the outermost transactional atomic region.

- Stopping an inner transaction causes the corresponding outer transaction to stop.
- An inner transaction does not commit data at the end of its transactional atomic region. Instead, the data of the inner transaction is committed later when the data of the corresponding outer transaction is committed.
- If a conflict occurs in a nested transaction, the thread is rolled back to the beginning of the outermost transaction.

Example

```
  USE OMP_LIB

  INTEGER :: i
  INTEGER :: data_arr(100)

!$OMP PARALLEL DO
  DO i = 1, 100

! Use the TM_ATOMIC directive to indicate a transactional atomic region.
!TM$ TM_ATOMIC
  data_arr(i) = i + 1
!TM$ END TM_ATOMIC

  END DO
!$OMP END PARALLEL DO
```

Related information

- The "-qtm" compiler option
- Execution modes

- Routines for transactional memory
- Environment variables for transactional memory
- "STOP"

WORKSHARE / END WORKSHARE

Purpose

The **WORKSHARE** directive allows you to parallelize the execution of array operations. A **WORKSHARE** directive divides the tasks associated with an enclosed block of code into *units of work*. When a team of threads encounters a **WORKSHARE** directive, the threads in the team share the tasks, so that each *unit of work* executes exactly once.

The **WORKSHARE** directive only takes effect if you specify the **-qsmp** compiler option.

Syntax

```
►►──WORKSHARE────────────────────────────────────────────────────────►◄


►►──block────────────────────────────────────────────────────────────►◄


►►──END WORKSHARE──┬────────┬────────────────────────────────────────►◄
                   └─NOWAIT─┘
```

block is a structured block of statements that allows work sharing within the lexical extent of the **WORKSHARE** construct. The execution of statements are synchronized so that statements whose result is a dependent on another statement are evaluated before that result is required. The *block* can contain any of the following:

- Array assignment statements
- **ATOMIC** directives
- **CRITICAL** constructs
- **FORALL** constructs
- **FORALL** statements
- **PARALLEL** construct
- **PARALLEL DO** construct
- **PARALLEL SECTION** construct
- **PARALLEL WORKSHARE** construct
- Scalar assignment statements
- **WHERE** constructs
- **WHERE** statements

The transformational intrinsic functions you can use as part of an array operation are:

- **ALL**
- **ANY**
- **COUNT**
- **CSHIFT**
- **DOT_PRODUCT**
- **EOSHIFT**
- **MATMUL**
- **MAXLOC**
- **MAXVAL**
- **MINLOC**
- **MINVAL**
- **PACK**
- **PRODUCT**
- **RESHAPE**
- **SPREAD**
- **SUM**
- **TRANSPOSE**
- **UNPACK**

The *block* can also contain statements bound to lexically enclosed **PARALLEL** constructs. These statements are not restricted.

Any user–defined function calls within the *block* must be elemental.

Statements enclosed in a **WORKSHARE** directive are divided into *units of work*. The definition of a *unit of work* varies according to the statement evaluated. A *unit of work* is defined as follows:

- **Array expressions:** Evaluation of each element of an array expression is a *unit of work*. Any of the transformational intrinsic functions listed above may be divided into any number of *units of work*.
- **Assignment statements:** In an array assignment statement, the assignment of each element in the array is a *unit of work*. For scalar assignment statements, the assignment operation is a *unit of work*.
- **Constructs:** Evaluation of each **CRITICAL** construct is a *unit of work*. Each **PARALLEL** construct contained within a **WORKSHARE** construct is a single *unit of work*. New teams of threads execute the statements contained within the lexical extent of the enclosed **PARALLEL** constructs. In **FORALL** constructs or statements, the evaluation of the mask expression, expressions occurring in the specification of the iteration space, and the masked assignments are *units of work*. In **WHERE** constructs or statements, the evaluation of the mask expression and the masked assignments are *units of work*.
- **Directives:** The update of each scalar variable for an **ATOMIC** directive and its assignments is a *unit of work*.
- **ELEMENTAL functions:** If the argument to an **ELEMENTAL** function is an array, then the application of the function to each element of an array is a *unit of work*.

If none of the above definitions apply to a statement within the *block*, then that statement is a *unit of work*.

Rules

In order to ensure that the statements within a **WORKSHARE** construct execute in parallel, the construct must be enclosed within a parallel region. Threads encountering a **WORKSHARE** construct outside the dynamic extent of a parallel region will evaluate the statements within the construct serially.

A **WORKSHARE** directive binds to the closest enclosing **PARALLEL** region if one exists.

You must not nest work-sharing regions that bind to the same **PARALLEL** region.

You must not specify a **WORKSHARE** directive within the **CRITICAL**, **MASTER**, or **ORDERED** regions.

You must not specify **BARRIER**, **MASTER**, or **ORDERED** directives within a **WORKSHARE** region.

If an array assignment, scalar assignment, a masked array assignment or a **FORALL** assignment assigns to a private variable in the *block*, the result is undefined.

If an array expression in the *block* references the value, association status or allocation status of private variables, the value of the expression is undefined unless each thread computes the same value.

If you do not specify a **NO WAIT** clause at the end of a **WORKSHARE** construct, a **BARRIER** directive is implied.

A **WORKSHARE** construct must be encountered by all threads in the team or by none at all.

Examples

Example 1: In the following example, the **WORKSHARE** directive evaluates the masked expressions in parallel.

```
!$OMP WORKSHARE
      FORALL (I = 1 : N, AA(1, I) == 0) AA(1, I) = I
      BB = TRANSPOSE(AA)
      CC = MATMUL(AA, BB)
!$OMP ATOMIC
      S = S + SUM(CC)
!$OMP END WORKSHARE
```

Example 2: The following example includes a user defined **ELEMENTAL** as part of a **WORKSHARE** construct.

```
!$OMP WORKSHARE
      WHERE (AA(1, :) /= 0.0) AA(1, :) = 1 / AA(1, :)
      DD = TRANS(AA(1, :))
!$OMP END WORKSHARE

      ELEMENTAL REAL FUNCTION TRANS(ELM) RESULT(RES)
      REAL, INTENT(IN) :: ELM
      RES = ELM * ELM + 4
      END FUNCTION
```

Related reference:

"ATOMIC" on page 93

"BARRIER" on page 97

"CRITICAL / END CRITICAL" on page 99

"PARALLEL WORKSHARE / END PARALLEL WORKSHARE" on page 119

See -qsmp in the Compiler Reference

Directive clauses

You can use directive clauses to specify additional information to directives.

Global rules for directive clauses

You must not specify a variable or common block name more than once in a clause.

A variable, common block name, or variable name that is a member of a common block must not appear in more than one clause on the same directive, with the following exceptions:

- You can define a named common block or named variable as **FIRSTPRIVATE** and **LASTPRIVATE** for the same directive.
- A variable appearing in a **NUM_THREADS** clause can appear in another clause for the same directive.
- A variable appearing in a **IF** clause can appear in another clause for the same directive.

If you do not specify a clause that changes the scope of a variable, the default scope for variables affected by a directive is **SHARED**.

A local variable with the **SAVE** or **STATIC** attribute declared in a procedure referenced a parallel region has an implicit **SHARED** attribute. A local variable without the **SAVE** or **STATIC** attribute declared in a procedure referenced a parallel region has an implicit **PRIVATE** attribute.

Members of common blocks and variables of modules declared in a procedure referenced within the dynamic extent of a parallel region have an implicit **SHARED** attribute, unless they are **THREADLOCAL** or **THREADPRIVATE** common blocks and module variables.

While a parallel or work-sharing construct is running, a variable or variable subobject used in a **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE** or **REDUCTION** clause of the directive must not be referenced, become defined, become undefined, have its association status or allocation status changed, or appear as an actual argument:
- In a scoping unit other than the one in which the directive construct appears
- In a variable format expression

You can declare a variable as **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE**, or **REDUCTION**, even if that variable is already storage associated with other variables. Storage association may exist for variables declared in **EQUIVALENCE** statements or in **COMMON** blocks. If a variable is storage associated with a **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE**, or **REDUCTION** variable, then:
- The contents, allocation status and association status of the variable that is storage associated with the **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE** or **REDUCTION** variable are undefined on entry to the parallel construct.
- The allocation status, association status and the contents of the associated variable become undefined if you define the **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE** or **REDUCTION** variable or if you define that variable's allocation or association status.
- The allocation status, association status and the contents of the **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE** or **REDUCTION** variable become undefined if you define the associated variable or if you define the associated variable's allocation or association status.

Pointers and OpenMP API

OpenMP API allows a variable or variable subobject of a **PRIVATE** clause to have the **POINTER** or **ALLOCATABLE** attribute. The association status of the pointer is undefined at thread creation and when the thread is destroyed.

See the following topics for more information about the directive clauses:

COLLAPSE	**FIRSTPRIVATE**	**PRIVATE**
COPYIN	**LASTPRIVATE**	**REDUCTION**
COPYPRIVATE	**NUM_THREADS**	**SCHEDULE**
DEFAULT	**ORDERED**	**SHARED**
IF		**UNTIED**

COLLAPSE

Purpose

Specifying the **COLLAPSE** clause allows you to parallelize multiple loops in a nest without introducing nested parallelism.

Syntax

```
►►──COLLAPSE──(──n──)──────────────────────────────────────────────►◄
```

n is a positive constant integer expression

Rules

- Only one collapse clause is allowed on a worksharing **DO** or **PARALLEL DO** directive
- The specified number of loops must be present lexically. That is, none of the loops can be in a called subroutine.
- The loops must form a rectangular iteration space and the bounds and stride of each loop must be invariant over all the loops.
- If the loop indices are of different size, the index with the largest size will be used for the collapsed loop.
- The loops must be perfectly nested; that is, there is no intervening code nor any OpenMP directive between the loops which are collapsed.
- The associated do-loops must be structured blocks. Their execution must not be terminated by an **EXIT** statement.
- If multiple loops are associated to the loop construct, only an iteration of the innermost associated loop may be curtailed by a **CYCLE** statement. If multiple loops are associated to the loop construct, there must be no branches to any of the loop termination statements except for the innermost associated loop.

Ordered construct
During execution of an iteration of a loop or a loop nested within a loop region, the executing thread must not execute more than one ordered region which binds to the same loop region. As a consequence, if multiple loops are associated to the loop construct by a collapse clause, the ordered construct has to be located inside all associated loops.

LASTPRIVATE clause
When a **LASTPRIVATE** clause appears on the directive that identifies a work-sharing construct, the value of each new list item from the sequentially last iteration of the associated loops is assigned to the original list item even if a collapse clause is associated with the loop

Other SMP and performance directives
The **STREAM_UNROLL**, **UNROLL**, **UNROLL_AND_FUSE**, and **NOUNROLL_AND_FUSE** directives cannot be used for any of the loops

associated with the **COLLAPSE** clause loop nest. The **INDEPENDENT** directive can be used for any of the loops associated with the **COLLAPSE** clause.

Examples

In Example 1 and Example 2 the loops over k and j are collapsed and their iteration space is executed by all threads of the current team.

Example 1

```
    !$omp do collapse(2) private(i,j,k)
       do k = kl, ku, ks
         do j = jl, ju, js
           do i = il, iu, is
             call bar(a,i,j,k)
           enddo
         enddo
       enddo
    !$omp end do
```

Example 2

```
 program test
 !$omp parallel
 !$omp do private(j,k) collapse(2) lastprivate(jlast, klast)
       do k = 1,2
         do j = 1,3
            jlast=j
            klast=k
         enddo
       enddo
 !$omp end do
 !$omp single
     print *, klast, jlast
 !$omp end single
 !$omp end parallel
 end program test
```

Output:

```
2 3
```

Example 3

As both loops are collapsed into one, the ordered construct has to be inside all loops associated to the for construct. As an iteration may not execute more than one ordered region, this program would be incorrect without the `collapse(2)` clause.

```
program test
!$omp parallel num_threads(2)
 !$omp do collapse(2) ordered private(j,k) schedule(static,3)
       do k = 1,3
         do j = 1,2
 !$omp ordered
          print *, k, j
 !$omp end ordered
          enddo
       enddo
 !$omp end do
 !$omp end parallel
 end program test
```

Output:

```
1 1
1 2
2 1
2 2
3 1
3 2
```

Related reference:

ORDERED / END ORDERED

DO / END DO

PARALLEL DO / END PARALLEL DO

COPYIN

Purpose

If you specify the **COPYIN** clause, the master thread's copy of each variable, or common block declared in the *copyin_entity_list* is duplicated at the beginning of a parallel region. Each thread in the team that will execute within that parallel region receives a private copy of all entities in the *copyin_entity_list*. All variables declared in the *copyin_entity_list* must be **THREADPRIVATE** or members of a common block that appears in a **THREADPRIVATE** directive.

Syntax

```
►►──COPYIN──(──copyin_entity_list──)──────────────────────────────────►◄
```

copyin_entity

```
►►──┬─variable_name───────────┬───────────────────────────────────────►◄
    └─/──common_block_name──/─┘
```

variable
: is a **THREADPRIVATE** variable, or **THREADPRIVATE** variable in a common block.

common_block_name
: is a **THREADPRIVATE** common block name.

Rules

If you specify a **COPYIN** clause, you cannot:

- specify the same entity name more than once in a *copyin_entity_list*.
- specify the same entity name in separate **COPYIN** clauses on the same directive.
- specify both a common block name and any variable within that same named common block in a *copyin_entity_list*.
- specify both a common block name and any variable within that same named common block in different **COPYIN** clauses on the same directive.
- specify a variable that contains **ALLOCATABLE** components.

When the master thread of a team of threads reaches a directive containing the **COPYIN** clause, thread's private copy of a variable or common block specified in the **COPYIN** clause will have the same value as the master thread's copy.

On entry into any parallel region, a **THREADPRIVATE** variable, or a variable in a **THREADPRIVATE** common block is subject to the following criteria when declared in a **COPYIN** clause:

- If the variable has the **POINTER** attribute and the master thread's copy of the variable is associated with a target, then each copy of that variable is associated with the same target. If the master thread's pointer is disassociated, then each copy of that variable is disassociated. If the master thread's copy of the variable has an undefined association status, then each copy of that variable has an undefined association status.
- Each copy of a variable without the **POINTER** attribute becomes defined with the value of the master thread's copy as if by intrinsic assignment.

If an allocatable array is specified in a **COPYIN** clause and it is allocated on entry into a parallel region, each thread copy of that array must be allocated with the same bounds and rank.

Related reference:

PARALLEL / END PARALLEL

PARALLEL DO / END PARALLEL DO

PARALLEL SECTIONS / END PARALLEL SECTIONS

PARALLEL WORKSHARE / END PARALLEL WORKSHARE

COPYPRIVATE

Purpose

If you specify the **COPYPRIVATE** clause, the value of a private variable or pointer to a shared object from one thread in a team is copied into the corresponding variables of all other threads in that team. If the variable in *copyprivate_entity_list* is not a pointer, then the corresponding variables of all threads within that team are defined with the value of that variable. If the variable is a pointer, then the corresponding variables of all threads within that team are defined with the association status of the pointer. Integer pointers and assumed-size arrays must not appear in *copyprivate_entity_list*.

Syntax

```
►►──COPYPRIVATE──(──copyprivate_entity_list──)────────────────────────►◄
```

copyprivate_entity

```
►►──┬─variable──────────────────┬─────────────────────────────────────►◄
    └─/──common_block_name──/───┘
```

variable
is a private variable within the enclosing parallel region

common_block_name
is a **THREADPRIVATE** common block name

Rules

If a common block is part of the *copyprivate_entity_list*, then it must appear in a **THREADPRIVATE** directive. Furthermore, the **COPYPRIVATE** clause treats a common block as if all variables within its *object_list* were specified in the *copyprivate_entity_list*.

A **COPYPRIVATE** clause must occur on an **END SINGLE** directive at the end of a **SINGLE** construct. The compiler evaluates a **COPYPRIVATE** clause before any threads have passed the implied **BARRIER** directive at the end of that construct. The variables you specify in *copyprivate_entity_list* must not appear in a **PRIVATE** or **FIRSTPRIVATE** clause for the **SINGLE** construct. If the **END SINGLE** directive occurs within the dynamic extent of a parallel region, the variables you specify in *copyprivate_entity_list* must be private within that parallel region.

A **COPYPRIVATE** clause must not appear on the same **END SINGLE** directive as a **NOWAIT** clause.

A **THREADLOCAL** common block, or members of that common block, are not permitted as part of a **COPYPRIVATE** clause.

If an allocatable array appears on a **COPYPRIVATE** clause, it must have an allocation status of allocated with the same bounds and rank in all threads that are affected by the **COPYPRIVATE** clause.
Related reference:
SINGLE / END SINGLE

DEFAULT

Purpose

If you specify the **DEFAULT** clause, all variables in the lexical extent of the parallel construct will have a scope attribute of *default_scope_attr*.

If you specify **DEFAULT(NONE)**, there is no default scope attribute. Therefore, you must explicitly list each variable you use in the lexical extent of the parallel construct in a data scope attribute clause on the parallel construct, unless the variable is:
- **THREADPRIVATE**
- A member of a **THREADPRIVATE** common block.
- A pointee
- A loop iteration variable used only as a loop iteration variable for:
  - Sequential loops in the lexical extent of the parallel region, or,
  - Parallel do loops that bind to the parallel region
- A variable that is only used in work-sharing constructs that bind to the parallel region, and is specified in a data scope attribute clause for each of the work-sharing constructs.

The **DEFAULT** clause specifies that all variables in the parallel construct share the same default scope attribute of either **FIRSTPRIVATE**, **PRIVATE**, **SHARED**, or no default scope attribute.

Syntax

```
►►──DEFAULT──(──default_scope_attr──)─────────────────────────────────►◄
```

default_scope_attr
is one of **FIRSTPRIVATE**, **PRIVATE**, **SHARED**, or **NONE**

Rules

If you specify **DEFAULT(NONE)** on a directive you must specify all named variables and all the leftmost names of referenced array sections, array elements, structure components, or substrings in the lexical extent of the directive construct in a **FIRSTPRIVATE**, **LASTPRIVATE**, **PRIVATE**, **REDUCTION**, or **SHARED** clause.

If you specify **DEFAULT(FIRSTPRIVATE)** on a directive, all named variables and all leftmost names of referenced array sections, array elements, structure components, or substrings in the lexical extent of the directive construct, including common block and use associated variables, but excluding **POINTEE**s and **THREADLOCAL** common blocks, have a **FIRSTPRIVATE** attribute to a thread as if they were listed explicitly in a **FIRSTPRIVATE** clause.

If you specify **DEFAULT(PRIVATE)** on a directive, all named variables and all leftmost names of referenced array sections, array elements, structure components, or substrings in the lexical extent of the directive construct, including common block and use associated variables, but excluding **POINTEE**s and **THREADLOCAL** common blocks, have a **PRIVATE** attribute to a thread as if they were listed explicitly in a **PRIVATE** clause.

If you specify **DEFAULT(SHARED)** on a directive, all named variables and all leftmost names of referenced array sections, array elements, structure components, or substrings in the lexical extent of the directive construct, excluding **POINTEE**s have a **SHARED** attribute to a thread as if they were listed explicitly in a **SHARED** clause.

The default behavior will be **DEFAULT(SHARED)** if you do not explicitly indicate a **DEFAULT** clause on a directive.

Example for thread-level speculative execution

The following example demonstrates the use of **DEFAULT(PRIVATE)** in **SPECULATIVE** directives.

```
PROGRAM MAIN
 REAL(8) :: x
 REAL(8), DIMENSION(4) :: y

!SEP$ SPECULATIVE SECTIONS DEFAULT(PRIVATE), SHARED(y)
 x = 1.0
 y(1) = x
!SEP$ SPECULATIVE SECTION
 x = 2.25
 y(2) = x
!SEP$ SPECULATIVE SECTION
 x = 3.5
 y(3) = x
!SEP$ SPECULATIVE SECTION
 x = 4.75
 y(4) = x
!SEP$ END SPECULATIVE SECTIONS

  IF(ANY(y .NE. (/1.0, 2.25, 3.5, 4.75/))) STOP 11
END PROGRAM
```

Example for OpenMP

The following example demonstrates the use of **DEFAULT(NONE)** for OpenMP, and some of the rules for specifying the data scope attributes of variables in the parallel region.

```
PROGRAM MAIN
  COMMON /COMBLK/ abc(10), def

  ! The loop iteration variable, i, is not required to be
  ! in data scope attribute clause.
$OMP PARALLEL DEFAULT(NONE) SHARED(ABC)

  ! def is specified on the work-sharing DO, and is not required to be
  ! specified in a data scope attribute clause on the parallel region.
!$OMP DO FIRSTPRIVATE(def)
  DO i = 1,10
    ABC(i) = def
  END DO
!$OMP END PARALLEL
END PROGRAM
```

Related reference:

PARALLEL / END PARALLEL

PARALLEL DO / END PARALLEL DO

PARALLEL SECTIONS / END PARALLEL SECTIONS

PARALLEL WORKSHARE / END PARALLEL WORKSHARE

"TASK / END TASK" on page 133

FINAL

Purpose

The **FINAL** clause is used with the **TASK** directive. If you specify a **FINAL** clause and the *scalar_logical_expr* evaluates to `.TRUE.`, the generated task is a final task. All task constructs encountered inside a final task create final and included tasks.

Syntax

```
►►──FINAL──(──scalar_logical_expr──)─────────────────────────────────►◄
```

Rules

You can specify only one **FINAL** clause on the **TASK** directive.

Related reference

"TASK / END TASK" on page 133

FIRSTPRIVATE

Purpose

If you use the **FIRSTPRIVATE** clause, each thread has its own initialized local copy of the variables and common blocks in *data_scope_entity_list*.

The **FIRSTPRIVATE** clause can be specified for the same variables as the **PRIVATE** clause, and functions in a manner similar to the **PRIVATE** clause. The exception is the status of the variable upon entry into the directive construct; the **FIRSTPRIVATE** variable exists and is initialized for each thread entering the directive construct.

Syntax

```
►►──FIRSTPRIVATE──(──data_scope_entity_list──)─────────────────────────►◄
```

Rules

A variable in a **FIRSTPRIVATE** clause must not be any of the following elements:
- A pointee
- An assumed-size array
- A **THREADLOCAL** common block
- A **THREADPRIVATE** common block or its members
- A **THREADPRIVATE** variable
- An allocatable scalar object

You cannot specify a variable in a **FIRSTPRIVATE** clause of a parallel construct if both the following conditions are true:
- The variable appears in a namelist statement, variable format expression or in an expression for a statement function definition.
- You reference the statement function, the variable format expression through formatted I/O, or the namelist through namelist I/O, within the parallel construct.

For a variable specified in the **FIRSTPRIVATE** clause, the status of the private copies is determined as follows:
- If the variable has the **POINTER** attribute, the private copies of the **FIRSTPRIVATE** variable receive the same association status as the original copy as if by pointer assignment.
- If the variable does not have the **POINTER** attribute, the initialization of the private copies occurs as if by intrinsic assignment. However, if the original variable is not currently allocated, the private copies have the same allocation status as the original copy.

If an allocatable array appears on a **FIRSTPRIVATE** clause, it must have an allocation status of allocated upon entrance into the parallel construct that contains the **FIRSTPRIVATE** clause.

When individual members of a common block are privatized, the storage of the specified variable is no longer associated with the storage of the common block.

Any variable that is storage associated with a **FIRSTPRIVATE** variable is undefined on entrance into the parallel construct.

If a directive construct contains a **FIRSTPRIVATE** argument to a Message Passing Interface (MPI) routine performing non-blocking communication, the MPI communication must complete before the end of the construct.

Example for thread-level speculative execution

The following example demonstrates the use of **FIRSTPRIVATE** in **SPECULATIVE** directives.

```
PROGRAM MAIN
 INTEGER(8) :: x, i8
 REAL(8) :: rar(2), r8

 r8 = 2.0_8
 i8 = 9_8

!SEP$ SPECULATIVE SECTIONS FIRSTPRIVATE(r8, i8)
 rar(1) = r8 * 2.0
 x = i8
!SEP$ SPECULATIVE SECTION
 rar(2) = r8 + i8
!SEP$ END SPECULATIVE SECTIONS
END PROGRAM
```

Related reference:

DO / END DO

PARALLEL / END PARALLEL

PARALLEL DO / END PARALLEL DO

PARALLEL SECTIONS / END PARALLEL SECTIONS

PARALLEL WORKSHARE / END PARALLEL WORKSHARE

SECTIONS / END SECTIONS

SINGLE / END SINGLE

"TASK / END TASK" on page 133

IF

Purpose

If you specify the **IF** clause, the runtime environment evaluates whether the *scalar_logical_expression* is true or false. If *scalar_logical_expression* is:

- true, the block is run in parallel.
- false, the containing region is suspended and the generated task is immediately run as though it is in a distinct task region.

Note that for the **TASK** directive, if the IF clause is evaluated to true, the block is not required to run in parallel.

Syntax

```
►►──IF──(──scalar_logical_expression──)─────────────────────────────►◄
```

Rules

The **IF** clause can be used in the **PARALLEL**, **PARALLEL DO**, **PARALLEL SECTIONS**, **PARALLEL WORKSHARE**, and **TASK** directives.

The **IF** clause may appear at most once in any directive.

By default, a nested parallel loop is serialized, regardless of the setting of the **IF** clause. You can change this default by using the **-qsmp=nested_par** compiler option.

An **IF** expression is evaluated outside of the context of the parallel construct. Any function reference in the **IF** expression must not have side effects.

Related reference:
"PARALLEL / END PARALLEL" on page 111
"PARALLEL DO / END PARALLEL DO" on page 113
"PARALLEL SECTIONS / END PARALLEL SECTIONS" on page 117
"PARALLEL WORKSHARE / END PARALLEL WORKSHARE" on page 119
"TASK / END TASK" on page 133

LASTPRIVATE

Purpose

If you use the **LASTPRIVATE** clause, each variable and common block in *data_scope_entity_list* is **PRIVATE**, and the last value of each variable in *data_scope_entity_list* can be referred to outside of the construct of the directive. If you use the **LASTPRIVATE** clause with **DO** or **PARALLEL DO**, the last value is the value of the variable after the last sequential iteration of the loop. If you use the **LASTPRIVATE** clause with **SECTIONS** or **PARALLEL SECTIONS**, the last value is the value of the variable after the last **SECTION** of the construct. If the last iteration of the loop or last section of the construct does not define a **LASTPRIVATE** variable, the variable is undefined after the loop or construct.

The **LASTPRIVATE** clause functions in a manner similar to the **PRIVATE** clause and you should specify it for variables that match the same criteria. The exception is in the status of the variable on exit from the directive construct. The compiler determines the last value of the variable, and takes a copy of that value which it saves in the named variable for use after the construct. A **LASTPRIVATE** variable is undefined on entry to the construct if it is not a **FIRSTPRIVATE** variable.

Syntax

```
►►──LASTPRIVATE──(──data_scope_entity_list──)───────────────────────►◄
```

Rules

A variable in a **LASTPRIVATE** clause must not be any of the following elements:
- A pointee
- An allocatable scalar object
- An assumed-size array
- A **THREADLOCAL** common block
- A **THREADPRIVATE** common block or its members
- A **THREADPRIVATE** variable

You cannot specify a variable in a **LASTPRIVATE** clause of a parallel construct if both the following conditions are true:
- The variable appears in a namelist statement, variable format expression or in an expression for a statement function definition.
- You reference the statement function, the variable format expression through formatted I/O, or the namelist through namelist I/O, within the parallel construct.

A **LASTPRIVATE** variable must be definable.

For a variable specified in a **LASTPRIVATE** clause,
- If the variable has the **POINTER** attribute, the original variable is updated as if by pointer assignment.
- If the variable does not have the **POINTER** attribute, the original variable is updated as if by intrinsic assignment.

If an allocatable array appears on a **LASTPRIVATE** clause, its allocation status must be allocated when it enters into the parallel construct that contains the **LASTPRIVATE** clause. The private copies of the **LASTPRIVATE** variable in the sequentially last iteration or lexically last section must have an allocation status of allocated. They must have the same bounds and rank as the corresponding **LASTPRIVATE** variable when they exit from that iteration or section.

When individual members of a common block are privatized, the storage of the specified variable is no longer associated with the storage of the common block.

Any variable that is storage associated with a **LASTPRIVATE** variable is undefined on entrance into the parallel construct.

If you specify a variable as **LASTPRIVATE** on a work-sharing directive, and you have specified a **NOWAIT** clause on that directive, you cannot use that variable between the end of the work-sharing construct and a **BARRIER** directive.

Variables that you specify as **LASTPRIVATE** to a parallel construct become defined at the end of the construct. If you have concurrent definitions or uses of **LASTPRIVATE** variables on multiple threads, you must ensure that the threads are synchronized at the end of the construct when the variables become defined. For example, if multiple threads encounter a **PARALLEL** construct with a **LASTPRIVATE** variable, you must synchronize the threads when they reach the **END PARALLEL** directive, because the **LASTPRIVATE** variable becomes defined at **END PARALLEL**. Therefore the whole **PARALLEL** construct must be enclosed within a synchronization construct.

If a directive construct contains a **LASTPRIVATE** argument to a Message Passing Interface (MPI) routine performing non-blocking communication, the MPI communication must complete before the end of that construct.

Example for thread-level speculative execution

The following example demonstrates the use of **LASTPRIVATE** in **SPECULATIVE** directives.

```
PROGRAM MAIN
 INTEGER :: flag1, flag2
 INTEGER, ALLOCATABLE :: i2a(:)
 INTEGER :: n

  flag1 = 0
  flag2 = 0
 ALLOCATE(i2a(2))
 i2a = -99
  n = 80

!SEP$ SPECULATIVE SECTIONS LASTPRIVATE(i2a)
 IF(i2a(1) == -99) flag1 = 0
   i2a(1) = 8
!SEP$ SPECULATIVE SECTION
 IF(i2a(2) == -99) flag2 = 0
  i2a(2) = n
  i2a(2) = i2a(2) * 10
!SEP$ END SPECULATIVE SECTIONS
 IF(flag1 == 1) STOP 21
 IF(flag2 == 1) STOP 22
END PROGRAM
```

Example for OpenMP

The following example shows the proper use of a **LASTPRIVATE** variable after a **NOWAIT** clause.

```
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(k)
DO i = 1,10
  k = i + 1
END DO

!$OMP END DO NOWAIT
k = ... **ERROR**     ! The reference to k must occur after a barrier.
!$OMP BARRIER
k = ...               ! this reference to k is valid.
!$OMP END PARALLEL
END
```

Related reference:

DO / END DO

PARALLEL DO / END PARALLEL DO

PARALLEL SECTIONS / END PARALLEL SECTIONS

SECTIONS / END SECTIONS

MERGEABLE

Purpose

The **MERGEABLE** clause is used with the **TASK** directive. If you specify a **MERGEABLE** clause and the generated task is an undeferred task or an included task, a merged task might be generated.

Syntax

```
►►──MERGEABLE──────────────────────────────────────────────────────►◄
```

Related reference

"TASK / END TASK" on page 133

NUM_THREADS

Purpose

The **NUM_THREADS** clause allows you to specify the number of threads used in a parallel region. Subsequent parallel regions are not affected. The **NUM_THREADS** clause takes precedence over the number of threads specified using the **omp_set_num_threads** library routine or the environment variable **OMP_NUM_THREADS**.

Syntax

```
►►──NUM_THREADS──(──scalar_integer_expression──)───────────────────►◄
```

Rules

The value of *scalar_integer_expression* must be a positive. Evaluation of the expression occurs outside the context of the parallel region. Any function calls that appear in the expression and change the value of a variable referenced in the expression will have unspecified results.

If you are using the environment variable **OMP_DYNAMIC** to enable dynamic threads, *scalar_integer_expression* defines the maximum number of threads available in the parallel region.

You must specify the **omp_set_nested** library routine or set the **OMP_NESTED** environment variable when including the **NUM_THREADS** clause as part of a nested parallel regions otherwise, the execution of that parallel region is serialized.

Related reference:

PARALLEL / END PARALLEL

PARALLEL DO / END PARALLEL DO

PARALLEL SECTIONS / END PARALLEL SECTIONS

PARALLEL WORKSHARE / END PARALLEL WORKSHARE

ORDERED

Purpose

Specifying the **ORDERED** clause on a work–sharing construct allows you to specify the **ORDERED** directive within the dynamic extent of a parallel loop.

Syntax

```
►►──ORDERED──────────────────────────────────────────────────────────►◄
```

Rules

The **ORDERED** clause applies to the following directives:

Related reference:

"DO / END DO" on page 100

"PARALLEL DO / END PARALLEL DO" on page 113

PRIVATE

Purpose

If you specify the **PRIVATE** clause on one of the directives listed below, each thread in a team has its own uninitialized local copy of the variables and common blocks in *data_scope_entity_list*.

You should specify a variable in the **PRIVATE** clause if its value is calculated by a single thread and that value is not dependent on any other thread, if it is defined before it is used in the construct, and if its value is not used after the construct ends. Copies of the **PRIVATE** variable exist, locally, on each thread. Each thread receives its own uninitialized copy of the **PRIVATE** variable. All thread variables within the lexical extent of the directive construct have the **PRIVATE** attribute by default.

Syntax

```
►►──PRIVATE──(──data_scope_entity_list──)────────────────────────────►◄
```

Rules

A variable in the **PRIVATE** clause must not be any of the following elements:

- A pointee
- An assumed-size array
- A **THREADLOCAL** common block
- A **THREADPRIVATE** common block or its members
- A **THREADPRIVATE** variable or the variable equivalenced with a **THREADPRIVATE** variable

You cannot specify a variable in a **PRIVATE** clause of a parallel construct if:

- the variable appears in a namelist statement, variable format expression or in an expression for a statement function definition, and,
- you reference the statement function, the variable format expression through formatted I/O, or the namelist through namelist I/O, within the parallel construct.

When individual members of a common block are privatized, the storage of the specified variable is no longer associated with the storage of the common block.

A variable that appears in the **REDUCTION** clause of a parallel construct can also appear in a **PRIVATE** clause on a work-sharing construct.

If a directive construct contains a **PRIVATE** argument to a Message Passing Interface (MPI) routine performing non-blocking communication, the MPI communication must complete before the end of that construct.

A variable name in the *data_scope_entity_list* of the **PRIVATE** clause can be an allocatable array. If the allocatable array is allocated on entry to a parallel region, the private copies of the array has an allocation status of allocated and has the same rank and bounds as the **PRIVATE** variable. If the allocatable array is unallocated on entry to a parallel region, the private copies of the array has an allocation status of unallocated.

Local variables without the **SAVE** or **STATIC** attributes in referenced subprograms in the dynamic extent of a directive construct have an implicit **PRIVATE** attribute.

Example for thread-level speculative execution

The following example demonstrates the use of **PRIVATE** in **SPECULATIVE** directives.

```
PROGRAM MAIN
 INTEGER, TARGET :: e
 INTEGER, POINTER :: p
 INTEGER :: f(10), flag

 flag = 0
 e = -1
 p => e

!SEP$ SPECULATIVE DO PRIVATE(e)
 DO i = 1, 10
  e = i
  f(i) = e
  IF(p /= -1) flag = 1
 END DO
!SEP$ END SPECULATIVE DO

 IF(flag == 1) STOP 11
END PROGRAM
```

Examples for OpenMP

Example 1: The following example demonstrates the proper use of a **PRIVATE** variable that is used to define a statement function. A commented line shows the invalid use. Since *J* appears in a statement function, the statement function cannot be referenced within the parallel construct for which *J* is **PRIVATE**.

```
INTEGER :: arr(10), j = 17
ISTFNC() = j

!$OMP PARALLEL DO PRIVATE(j)
DO i = 1, 10
  j = i
  ! arr(i) = ISTFNC() **ERROR** A reference to ISTFNC would
  ! make the PRIVATE(J) clause invalid.
  ARR(i) = j
END DO
PRINT *, arr
END
```

Example 2: The following example demonstrates the use of allocatable arrays on a **PRIVATE** clause:

```
USE OMP_LIB
REAL, ALLOCATABLE :: temp(:,:)
REAL :: arr(4, 20, 20)
INTEGER :: thd

ALLOCATE(temp(20, 20))
!$OMP PARALLEL PRIVATE(thd, temp) NUM_THREADS(4)

! Private copies of "temp" are allocated with the same
! bounds and shape of the original "temp".
thd = OMP_GET_THREAD_NUM()
IF(MOD(thd, 2) .EQ. 0) THEN
  temp = RESHAPE((/(i, i=1, 400)/), (/20, 20/))
ELSE
  temp = RESHAPE((/(i, i=1, 800, 2)/), (/20, 20/))
ENDIF
arr(thd + 1, :, :) = temp

! Private copies of "temp" are deallocated.
!$OMP END PARALLEL
DEALLOCATE(temp)
END
```

Note: If the machine has less than 4 CPUs, you must set *OMP_THREAD_LIMIT=4*.

Example 3: The following example demonstrates the persistence of the original value of the **PRIVATE** variables after exit from a parallel region:

```
PROGRAM MAIN
  INTEGER :: i, j

  i = 1
  j = 2
!$OMP PARALLEL PRIVATE(i, j)
  i = 3
  j = j + 2
!$OMP END PARALLEL
  PRINT *, i, j                ! Output: 1 2
END PROGRAM
```

Related reference:
DO / END DO
PARALLEL / END PARALLEL
PARALLEL DO / END PARALLEL DO
PARALLEL SECTIONS / END PARALLEL SECTIONS
PARALLEL WORKSHARE / END PARALLEL WORKSHARE
SECTIONS / END SECTIONS
SINGLE / END SINGLE
"TASK / END TASK" on page 133

REDUCTION

Purpose

The **REDUCTION** clause updates named variables declared on the clause within the directive construct. Intermediate values of **REDUCTION** variables are not used within the parallel construct, other than in the updates themselves.

Syntax

```
►►──REDUCTION──(──┬─────────────┬──variable_name_list──)──────────────────────►◄
                  └─op_fnc──:───┘
```

op_fnc is a *reduction_op* or a *reduction_function* that appears in all **REDUCTION** statements involving this variable. You must not specify more than one **REDUCTION** operator or function for a variable in the directive construct. To maintain OpenMP API compliance, you must specify *op_fnc* for the **REDUCTION** clause.

A **REDUCTION** statement can have one of the following forms:

```
►►──reduction_var_ref──=──expr──reduction_op──reduction_var_ref──────────────────►◄
```

```
►►──reduction_var_ref──=──reduction_var_ref──reduction_op──expr──────────────────►◄
```

```
►►──reduction_var_ref =──reduction_function──(expr,──reduction_var_ref)──────────►◄
```

```
►►──reduction_var_ref =──reduction_function──(reduction_var_ref,──expr)──────────►◄
```

where:

reduction_var_ref
is a variable or subobject of a variable that appears in a **REDUCTION** clause

reduction_op
is one of the intrinsic operators: **+**, **-**, *, **.AND.**, **.OR.**, **.EQV.**, **.NEQV.**, or **.XOR.**

when *reduction_op* is an intrinsic operator, it should be the last operation performed on the right side.

reduction_function
: is one of the intrinsic procedures: **MAX**, **MIN**, **IAND**, **IOR**, or **IEOR**.

expr
: should not contain references to *reduction_var_ref*

The canonical initialization value of each of the operators and intrinsics are shown in the following table. The actual initialization value will be consistent with the data type of your corresponding **REDUCTION** variable.

Intrinsic Operator	**Initialization**
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
.XOR.	.FALSE.
Intrinsic Procedure	**Initialization**
MAX	Smallest representable number
MIN	Largest representable number
IAND	All bits on
IOR	0
IEOR	0

Rules

The following rules apply to **REDUCTION** statements:

- A variable in the **REDUCTION** clause must only occur in a **REDUCTION** statement within the directive construct on which the **REDUCTION** clause appears.
- The two *reduction_var_ref*s that appear in a **REDUCTION** statement must be lexically identical.
- You cannot use the following form of the **REDUCTION** statement: *reduction_var_ref = expr operator reduction_var_ref*, where *operator* is any operator other than *reduction_op*.

When you specify individual members of a common block in a **REDUCTION** clause, the storage of the specified variable is no longer associated with the storage of the common block.

Any variable you specify in a **REDUCTION** clause of a work-sharing construct must be shared in the enclosing **PARALLEL** construct.

A variable that appears in the **REDUCTION** clause of a parallel construct can also appear in a **PRIVATE** clause on a work-sharing construct.

If you use a **REDUCTION** clause on a construct that has a **NOWAIT** clause, the **REDUCTION** variable remains undefined until a barrier synchronization has been performed to ensure that all threads have completed the **REDUCTION** clause.

A **REDUCTION** variable must not appear in a **FIRSTPRIVATE**, **PRIVATE**, or **LASTPRIVATE** clause of another construct within the dynamic extent of the construct in which it appeared as a **REDUCTION** variable.

If you specify *op_fnc* for the **REDUCTION** clause, each variable in the *variable_name_list* must be of intrinsic type. The variable can only appear in a **REDUCTION** statement within the lexical extent of the directive construct. You must specify *op_fnc* if the directive uses the *trigger_constant* **$OMP**.

The **REDUCTION** clause specifies named variables that appear in reduction operations. The compiler will maintain local copies of such variables, but will combine them upon exit from the construct. The intermediate values of the **REDUCTION** variables are combined in random order, dependent on which threads finish their calculations first. Therefore, there is no guarantee that bit-identical results will be obtained from one parallel run to another. This is true even if the parallel runs use the same number of threads, scheduling type, and chunk size.

Variables that you specify as **REDUCTION** or **LASTPRIVATE** to a parallel construct become defined at the end of the construct. If you have concurrent definitions or uses of **REDUCTION** or **LASTPRIVATE** variables on multiple threads, you must ensure that the threads are synchronized at the end of the construct when the variables become defined. For example, if multiple threads encounter a **PARALLEL** construct with a **REDUCTION** variable, you must synchronize the threads when they reach the **END PARALLEL** directive, because the **REDUCTION** variable becomes defined at **END PARALLEL**. Therefore the whole **PARALLEL** construct must be enclosed within a synchronization construct.

If an allocatable array appears on a **REDUCTION** clause, it must have an allocation status of allocated upon entrance into the construct that contains the **REDUCTION** clause. Additionally, the private copies of the **REDUCTION** variable must not be deallocated or allocated within the region.

A variable in the **REDUCTION** clause must be of intrinsic type. A variable in the **REDUCTION** clause, or any element thereof, must not be any of the following:

- A pointee
- An assumed-size array
- A **THREADLOCAL** common block
- A **THREADPRIVATE** common block or its members
- A **THREADPRIVATE** variable
- An allocatable scalar object
- A Fortran 90 pointer

These rules describe the use of **REDUCTION** on OpenMP directives. If you are using the **REDUCTION** clause on the **INDEPENDENT** directive, see the **INDEPENDENT** directive in the *XL Fortran Language Reference* directive.

Example for thread-level speculative execution

The following example demonstrates the use of **REDUCTION** in **SPECULATIVE** directives.

```
PROGRAM MAIN
 INTEGER(2), ALLOCATABLE :: i2a(:)
 INTEGER(2) :: i2b

 ALLOCATE(i2a(5))
 i2a = [5, 7, -5, 13, 9]
 i2b = 6

!SEP$ SPECULATIVE DO REDUCTION(MAX: i2b), NUM_THREADS(4)
 DO i = 1, 5
   IF(i2b == 6) STOP 11
  i2b = MAX(i2b, i2a(i))
 END DO
!SEP$ END SPECULATIVE DO

 IF(i2b /= 13) STOP 12
 DEALLOCATE(i2a)
END PROGRAM
```

Related reference:

DO / END DO

PARALLEL / END PARALLEL

PARALLEL DO / END PARALLEL DO

PARALLEL SECTIONS / END PARALLEL SECTIONS

PARALLEL WORKSHARE / END PARALLEL WORKSHARE

SECTIONS / END SECTIONS

SCHEDULE

Purpose

You can use the **SCHEDULE** clause to specify the chunking method for parallelization. Work is assigned to threads in different manners depending on the scheduling type or chunk size used.

Syntax

```
►►──SCHEDULE──(──sched_type──┬────┬──)───────────────────────────────►◄
                             └─,n─┘
```

sched_type
: is one of **AFFINITY**, **AUTO**, **DYNAMIC**, **GUIDED**, **RUNTIME**, or **STATIC**.

  Note: Thread-level speculative execution supports the **STATIC** scheduling type. You can also specify **DYNAMIC**, **GUIDED**, or other types, but the runtime treats all these values as static scheduling.

n
: must be a positive scalar integer expression; do not specify *n* for the **AUTO** and **RUNTIME** schedule type. If you are using the *trigger_constant* **$OMP**, do not specify the scheduling type **AFFINITY**.

AFFINITY

The iterations of a loop are initially divided into *number_of_threads* partitions, containing `CEILING(number_of_iterations / number_of_threads)` iterations. Each partition is initially assigned to a thread, and is then further subdivided into chunks containing *n* iterations, if *n* has been specified. If *n* has not been specified, then the chunks consist of `CEILING(number_of_iterations_remaining_in_partition / 2)` loop iterations.

When a thread becomes free, it takes the next chunk from its initially assigned partition. If there are no more chunks in that partition, then the thread takes the next available chunk from a partition that is initially assigned to another thread.

Threads that are active will complete the work in a partition that is initially assigned to a sleeping thread.

AUTO

The compiler and runtime system choose the most appropriate mapping of iteration to threads for each loop.

DYNAMIC

If *n* has been specified, the iterations of a loop are divided into chunks containing *n* iterations each. If *n* has not been specified, then the default chunk size is 1 iteration.

Threads are assigned these chunks on a "first-come, first-do" basis. Chunks of the remaining work are assigned to available threads, until all work has been assigned.

If a thread is asleep, its assigned work will be taken over by an active thread, once that other thread becomes available.

GUIDED

If you specify a value for *n*, the iterations of a loop are divided into chunks such that the size of each successive chunk is exponentially decreasing. *n* specifies the size of the smallest chunk, except possibly the last. If you do not specify a value for *n*, the default value is 1.

The size of the initial chunk is proportional to `CEILING(number_of_iterations / number_of_threads)` iterations. Subsequent chunks are proportional to `CEILING(number_of_iterations_remaining / number_of_threads)` iterations. If *n* is greater than 1, each chunk should contain fewer than n iterations (except for the last chunk to be assigned, which can have fewer than *n* iterations. As each thread finishes a chunk, it dynamically obtains the next available chunk.

You can use guided scheduling in a situation in which multiple threads in a team might arrive at a **DO** work-sharing construct at varying times, and each iteration requires roughly the same amount of work. For example, if you have a **DO** loop preceded by one or more work-sharing **SECTIONS** or **DO** constructs with **NOWAIT** clauses, you can guarantee that no thread waits at the barrier longer than it takes another thread to execute its final iteration, or final k iterations if a chunk size of k is specified. The **GUIDED** schedule requires the fewest synchronizations of all the scheduling methods.

An *n* expression is evaluated outside of the context of the **DO** construct. Any function reference in the *n* expression must not have side effects.

The value of the *n* parameter on the **SCHEDULE** clause must be the same for all of the threads in the team.

RUNTIME
Determine the scheduling type at run time.

At run time, the scheduling type can be specified using the environment variable **OMP_SCHEDULE**. If no scheduling type is specified using that variable, the default scheduling type used is **AUTO**.

STATIC
If *n* has been specified, the iterations of a loop are divided into chunks that contain *n* iterations. Each thread is assigned chunks in a "round robin" fashion. This is known as block cyclic scheduling. If the value of *n* is 1, then the scheduling type is specifically referred to as cyclic scheduling.

If *n* has not been specified, the chunks will contain `CEILING(number_of_iterations / number_of_threads)` iterations. Each thread is assigned one of these chunks. This is known as block cyclic scheduling.

If a thread is asleep and it has been assigned work, it will be awakened so that it may complete its work.

The **STATIC** schedule ensures that the same logical iteration numbers are assigned to threads in two work-sharing loop regions if the following conditions are satisfied:
- Both loop regions have the same number of loop iterations
- Both loop regions either have the same value of *n* specified, or have no *n* specified
- Both loop regions bind to the same parallel region

A data dependence between the same logical iterations in two such loops is guaranteed to be satisfied to allow the safe use of the **NOWAIT** clause. In addition, you must make sure that all three conditions mentioned above are satisfied to get the correct result.

Consecutive loop constructs with **STATIC** schedule with **NOWAIT** clause now guarantee the same iterations are being assigned to the same thread in the constructs.

For an example of the loop constructs that satisfy all three conditions, see "Example for OpenMP" on page 170.

Rules

You must not specify the **SCHEDULE** clause more than once for a particular **DO** directive.

Example for thread-level speculative execution

The following example demonstrates the use of **SCHEDULE(STATIC, n)** in **SPECULATIVE** directives.

```
PROGRAM MAIN
 REAL(4), DIMENSION(200) :: x, y, z
 REAL(4), PARAMETER :: c1 = 1.2345_4, c2 = 5.6789_4
 REAL(4), PARAMETER :: e = 2.718281828_4, pi = 3.141592654_4
 INTEGER, PARAMETER :: n = 6
 INTEGER :: num, a, b

 TYPE schedinfo
```

```
  CHARACTER(10) :: name
  INTEGER :: chunk(n)
 END TYPE
 TYPE(schedinfo) :: sc

 sc = schedinfo('static', (/0, 10, 50, 100, 200, 0/))
 sc%chunk(n) = MAX_INT

 DO i = 1, n
  num = sc%chunk(i)
 !SEP$ SPECULATIVE DO SCHEDULE(STATIC, num)
  DO j = 1, 200
    x(j) = (c1 + j/10) * pi
    y(j) = (c2 + j/100) * e
  END DO
 !SEP$ END SPECULATIVE DO
 END DO
END PROGRAM
```

Example for OpenMP

The following example illustrates loop constructs that satisfy all three conditions listed in the **STATIC** section.

```
!$OMP PARALLEL
!$OMP DO SCHEDULE(STATIC)
  DO i = 1, n
    c(i) = (a(i) + b(i)) / 2.0;
  ENDDO
!$OMP END DO NOWAIT

!$OMP DO SCHEDULE(STATIC)
  DO i = 1, n
    z(i) = sqrt(c(i))
  ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

Related reference:

"DO / END DO" on page 100

"PARALLEL DO / END PARALLEL DO" on page 113

SHARED

Purpose

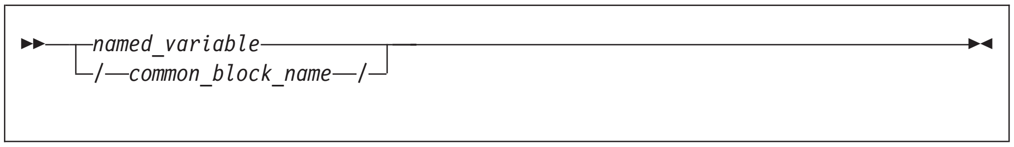
All sections use the same copy of the variables and common blocks you specify in *data_scope_entity_list*.

The **SHARED** clause specifies variables that must be available to all threads. If you specify a variable as **SHARED**, you are stating that all threads can safely share a single copy of the variable.

Syntax

```
►►──SHARED──(──data_scope_entity_list──)──────────────────────────────►◄
```

data_scope_entity



named_variable
is a named variable that is accessible in the directive construct

common_block_name
is a common block name that is accessible in the directive construct

Rules

A variable in the **SHARED** clause must not be either:
- A pointee
- A **THREADLOCAL** common block.
- A **THREADPRIVATE** common block or its members.
- A **THREADPRIVATE** variable.

If a **SHARED** variable, a subobject of a **SHARED** variable, or an object associated with a **SHARED** variable or subobject of a **SHARED** variable appears as an actual argument in a reference to a non-intrinsic procedure and:
- The actual argument is an array section with a vector subscript; or
- The actual argument is
  - An array section,
  - An assumed-shape array, or,
  - A pointer array

  and the associated dummy argument is an explicit-shape or assumed-size array;

then any references to or definitions of the shared storage that is associated with the dummy argument by any other thread must be synchronized with the procedure reference. In other words, you must structure your code in such a way that if a thread encounters a procedure reference, then the procedure call by that thread and any reference to or definition of the shared storage by any other thread will always occur in the same sequence. You can do this, for example, by placing the procedure reference after a **BARRIER**.

Example for thread-level speculative execution

The following example demonstrates the use of **SHARED** in **SPECULATIVE** directives.

```
PROGRAM MAIN
 REAL(8) :: rr8
 INTEGER(8) :: i8

 i8 = 1
 rr8 = 0

!SEP$ SPECULATIVE DO SHARED(i8) SHARED(rr8)
 DO i = 1, 100
  rr8 = rr8 + 1.55_8
  i8 = i8 * 2_8
 END DO
!SEP$ END SPECULATIVE DO
END PROGRAM
```

Example for OpenMP

In the following example, the procedure reference with an array section actual argument is required to be synchronized with references to the dummy argument by placing the procedure reference in a critical section, because the associated dummy argument is an explicit-shape array.

```
INTEGER :: abc(10)

i = 2
j = 5

!$OMP PARALLEL DEFAULT(NONE), SHARED(abc, i, j)
!$OMP CRITICAL
! Actual argument is an array section.
! The procedure reference must be in a critical section.
CALL sub1(abc(i: j))
!$OMP END CRITICAL
!$OMP END PARALLEL

  CONTAINS
    SUBROUTINE sub1(arr)
      INTEGER:: arr(1: 4)
      DO i = 1, 4
        arr(i) = i
      END DO
    END SUBROUTINE
END
```

Related reference:

PARALLEL / END PARALLEL

PARALLEL DO / END PARALLEL DO

PARALLEL SECTIONS / END PARALLEL SECTIONS

PARALLEL WORKSHARE / END PARALLEL WORKSHARE

"TASK / END TASK" on page 133

UNTIED

Purpose

The **UNTIED** clause is used with the **TASK** directive. When a task region is suspended, untied tasks can be resumed by any thread in a team.

Syntax

```
>>──UNTIED──────────────────────────────────────────────────><
```

Rules

The **UNTIED** clause is ignored if either of the following conditions is true:

- A **FINAL** clause is specified on the same task construct and the **FINAL** clause expression evaluates to `.TRUE.`.
- The task is an included task.

Related reference:
"TASK / END TASK" on page 133

Routines for parallel programming

This section describes routines that are supported by OpenMP, POMP, thread-level speculative execution, and transactional memory on Blue Gene/Q platforms.

Routines for OpenMP

The OpenMP specification provides a number of routines that you can use to control and query the parallel execution environment, timing, and lock.

Parallel threads created by the runtime environment through the OpenMP interface are considered independent of the threads you create and control using calls to the **Fortran Pthreads library module**. References within the following descriptions to "serial portions of the program" refer to portions of the program that are executed by only one of the threads that have been created by the runtime environment. For example, you can create multiple threads by using **f_pthread_create**. However, if you then call **omp_get_num_threads** from outside of an OpenMP parallel block, or from within a serialized nested parallel region, the function will return **1**, regardless of the number of threads that are currently executing.

OpenMP runtime library calls must not appear in **PURE** and **ELEMENTAL** procedures.

Table 19. OpenMP execution environment routines

omp_get_active_level	**omp_get_thread_num**
omp_get_ancestor_thread_num	**omp_get_schedule**
omp_get_dynamic	**omp_get_team_size**
omp_get_level	**omp_get_thread_limit**
omp_get_max_active_levels	**omp_in_final**
omp_get_max_threads	**omp_in_parallel**
omp_get_nested	**omp_set_dynamic**
omp_get_num_procs	**omp_set_max_active_levels**
omp_get_num_threads	**omp_set_nested**
	omp_set_num_threads
	omp_set_schedule

Included in the OpenMP runtime library are two routines that support a portable wall-clock timer.

Table 20. OpenMP timing routines

omp_get_wtick	**omp_get_wtime**

The OpenMP runtime library also supports a set of simple and nestable lock routines. You must only lock variables through these routines. Simple locks may not be locked if they are already in a locked state. Simple lock variables are associated with simple locks and may only be passed to simple lock routines. Nestable locks may be locked multiple times by the same thread. Nestable lock variables are associated with nestable locks and may only be passed to nestable lock routines. Note that locks are now associated with task regions, and no longer with threads as such, in accordance with changes in the OMP standard.

For all the routines listed below, the lock variable is an integer whose **KIND** type parameter is denoted either by the symbolic constant **omp_lock_kind**, or by **omp_nest_lock_kind**.

This variable is set to '8' for 64-bit applications.

Table 21. OpenMP simple lock routines

omp_destroy_lock	**omp_test_lock**
omp_init_lock	**omp_unset_lock**
omp_set_lock	

Table 22. OpenMP nestable lock routines

omp_destroy_nest_lock	**omp_test_nest_lock**
omp_init_nest_lock	**omp_unset_nest_lock**
omp_set_nest_lock	

Note: You can define and implement your own versions of the OpenMP routines. However, by default, the compiler will substitute the XL Fortran versions of the OpenMP routines regardless of the existence of other implementations, unless you specify the **-qnoswapomp** compiler option. For more information, see *XL Fortran Compiler Reference*.

omp_destroy_lock(svar)

Purpose

The **omp_destroy_lock** subroutine disassociates a given lock variable from all locks. You must use **omp_init_lock** to reinitialize a lock variable that was destroyed with a call to **omp_destroy_lock** before using it again as a lock variable.

If you call **omp_destroy_lock** with an uninitialized lock variable, the result of the call is undefined.

Class

Subroutine.

Argument Type and Attributes

svar	Type integer with kind **omp_lock_kind**.

Result Type and Attributes

None.

Result Value

None.

Examples

In the following example, threads and their associated tasks are generated by the parallel region, and one at a time, each task gains ownership of the lock associated with the lock variable LCK, prints the thread ID, and releases ownership of the lock.

```
      USE omp_lib
      INTEGER(kind=omp_lock_kind) LCK
      INTEGER ID
      CALL omp_init_lock(LCK)
!$OMP PARALLEL SHARED(LCK), PRIVATE(ID)
      ID = omp_get_thread_num()
      CALL omp_set_lock(LCK)
      PRINT *,'MY THREAD ID IS', ID
      CALL omp_unset_lock(LCK)
!$OMP END PARALLEL
      CALL omp_destroy_lock(LCK)
      END
```

omp_destroy_nest_lock(nvar)

Purpose

The **omp_destroy_nest_lock** subroutine initializes a nestable lock variable, causing the lock variable to become undefined. The variable *nvar* must be an unlocked and initialized nestable lock variable.

If you call **omp_destroy_nest_lock** using an uninitialized variable, the result is undefined.

Class

Subroutine.

Argument Type and Attributes

nvar Type integer with kind **omp_nest_lock_kind**.

Result Type and Attributes

None.

Result Value

None.

omp_get_active_level()

Purpose

The **omp_get_active_level** function returns the number of nested, active parallel regions.

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Default integer.

Result Value

An integer that indicates the number of nested, active parallel regions.

omp_get_ancestor_thread_num(level)

Purpose

The **omp_get_ancestor_thread_num** function returns the thread number of the ancestor at a given nested level of the current thread.

Class

Function.

Argument Type and Attributes

level
: Default integer.

Result Type and Attributes

Default integer.

Result Value

The thread number of the ancestor at a given nested level (***level***) of the current thread. If ***level*** is outside the range of 0 and the nested level of the current thread, as returned by the **omp_get_level** routine, the function returns -1.

omp_get_dynamic()

Purpose

The **omp_get_dynamic** function returns **.TRUE.** if dynamic thread adjustment by the runtime environment is enabled. Otherwise, the **omp_get_dynamic** function returns **.FALSE.**

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Default logical.

Result Value

.TRUE. if dynamic thread adjustment by the runtime environment is enabled; **.FALSE.** otherwise.

omp_get_level()

Purpose

The **omp_get_level** function returns the number of nested parallel regions (both active and inactive).

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Default integer.

Result Value

The number of nested parallel regions (both active and inactive) in which the generating task is executing, not including the implicit parallel region.

omp_get_max_active_levels()

Purpose

The **omp_get_max_active_levels** function returns the maximum number of nested, active parallel regions.

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Default integer.

Result Value

The maximum number of nested, active parallel regions that is allowed.

Note: XL Fortran does not support OpenMP nested parallelism. This function always returns 1.

omp_get_max_threads()

Purpose

The **omp_get_max_threads** routine returns the first value of *num_list* for the **OMP_NUM_THREADS** environment variable. This value is the maximum number of threads that can be used to form a new team if a parallel region without a **num_threads** clause is encountered.

If you use **omp_set_num_threads** to change the number of threads, subsequent calls to **omp_get_max_threads** will return the new value.

The routine has global scope, which means that the maximum value it returns applies to all routines, subroutines, and compilation units in the program. It returns the same value whether executing from a serial or parallel region.

You can use **omp_get_max_threads** to allocate maximum-sized data structures for each thread when you have enabled dynamic thread adjustment by passing **omp_set_dynamic** an argument which evaluates to **.TRUE.**

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Default integer.

Result Value

The maximum number of threads that can execute concurrently in a single parallel region.

omp_get_nested()

Purpose

The **omp_get_nested** function returns **.TRUE.** if nested parallelism is enabled and **.FALSE.** if nested parallelism is disabled.

Currently, XL Fortran does not support OpenMP nested parallelism.

Class

Function

Argument Type and Attributes

None.

Result Type and Attributes

Default logical.

Result Value

.TRUE. if nested parallelism is enabled. **.FALSE.** otherwise.

omp_get_num_procs()

Purpose

The **omp_get_num_procs** function returns the number of online processors on the machine.

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Default integer.

Result Value

The number of online processors on the machine.

omp_get_num_threads()

Purpose

The **omp_get_num_threads** function returns the number of threads in the team currently executing the parallel region from which it is called. The function binds to the closest enclosing **PARALLEL** directive.

The **omp_set_num_threads** subroutine and the **OMP_NUM_THREADS** environment variable control the number of threads in a team. If you do not explicitly set the number of threads, the runtime environment will use the number of online processors on the machine by default. The number of online processors is less than or equal to the number of physical processors actually installed in a machine.

If you call **omp_get_num_threads** from a serial portion of your program or from a nested parallel region that is serialized, the function returns 1.

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Default integer.

Result Value

The number of threads in the team currently executing the parallel region from which the function is called.

Examples

```
      USE omp_lib
      INTEGER N1, N2

      N1 = omp_get_num_threads()
      PRINT *, N1
!$OMP PARALLEL PRIVATE(N2)
      N2 = omp_get_num_threads()
      PRINT *, N2
!$OMP END PARALLEL
      END
```

The **omp_get_num_threads** call returns 1 in the serial section of the code, so N1 is assigned the value 1. N2 is assigned the number of threads in the team executing the parallel region, so the output of the second print statement will be an arbitrary number less than or equal to the value returned by **omp_get_max_threads**.

omp_get_schedule(kind, modifier)

Purpose

The **omp_get_schedule** subroutine returns the scheduling type that is applied when using the **runtime** schedule. The argument *kind* returns the type of schedule that is used. *modifier* represents the chunk size that is set for applicable schedule types.

Class

Subroutine.

Argument Type and Attributes

kind
  Integer of kind **omp_sched_kind**. The value returned for *kind* is one of the following constants that are defined in **omp_lib** module:
  - **omp_sched_static**
  - **omp_sched_dynamic**
  - **omp_sched_guided**
  - **omp_sched_auto**
  - **omp_sched_affinity**

  where **omp_sched_affinity** is not part of the OpenMP specification.

modifier
  Default integer. For the schedule type **dynamic**, **guided**, or **static**, *modifier* is the chunk size that is set. For the schedule type **auto**, *modifier* has no meaning.

Result Type and Attributes

None.

Result Value

None.

omp_get_team_size(level)

Purpose

The **omp_get_team_size** function returns the size of the thread team that the ancestor belongs to.

Class

Function.

Argument Type and Attributes

`level`
: Default integer. ***level*** is the nested level of the current thread.

Result Type and Attributes

Default integer.

Result Value

The size of the thread team that the ancestor belongs to. If *level* is outside of the range of 0 and the nested level of the current thread, as returned by the **omp_get_level** function, the function returns -1.

omp_get_thread_limit()

Purpose

The **omp_get_thread_limit** function returns the maximum number of OpenMP threads that are available to the program.

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Default integer.

Result Value

The maximum number of OpenMP threads that are available to the program.

omp_get_thread_num()

Purpose

The **omp_get_thread_num** function returns the number of the currently executing thread within the team. The number returned will always be between 0 and *NUM_PARTHDS* - 1. *NUM_PARTHDS* is the number of currently executing threads within the team. The master thread of the team returns a value of 0.

If you call **omp_get_thread_num** from within a serial region, from within a serialized nested parallel region, or from outside the dynamic extent of any parallel region, this function will return a value of 0.

This function binds to the closest parallel region.

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Default integer.

Result Value

The value of the currently executing thread within the team between 0 and *NUM_PARTHDS* - 1. *NUM_PARTHDS* is the number of currently executing threads within the team. A call to **omp_get_thread_num** from a serialized nested parallel region, or from outside the dynamic extent of any parallel region returns 0.

Examples

The following example illustrates the return value of the **omp_get_thread_num** routine in a PARALLEL region and a MASTER construct.

```
      USE omp_lib
      INTEGER NP
      call omp_set_num_threads(4)  ! 4 threads are used in the
                                   ! parallel region

!$OMP PARALLEL PRIVATE(NP)
      NP = omp_get_thread_num()
      CALL WORK('in parallel', NP)

!$OMP MASTER
      NP = omp_get_thread_num()
      CALL WORK('in master', NP)
!$OMP END MASTER
!$OMP END PARALLEL
      END
      SUBROUTINE WORK(msg, THD_NUM)
      INTEGER THD_NUM
      character(*) msg
      PRINT *, msg, THD_NUM
      END
```

Output:

```
in parallel 1
in parallel 3
in parallel 2
in parallel 0
in master 0
```

(The order may be different.)

omp_get_wtick()

Purpose

The **omp_get_wtick** function returns a double precision value equal to the number of seconds between consecutive clock ticks.

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Double precision real.

Result Value

The number of seconds between consecutive ticks of the operating system real-time clock.

Examples

```
USE omp_lib
DOUBLE PRECISION WTICKS
WTICKS = omp_get_wtick()
PRINT *, 'The clock ticks ', 10 / WTICKS, &
' times in 10 seconds.'
END
```

omp_get_wtime()

Purpose

The **omp_get_wtime** function returns a double precision value equal to the number of seconds since the initial value of the operating system real-time clock. The initial value is guaranteed not to change during execution of the program.

The value returned by the **omp_get_wtime** function is not consistent across all threads in the team.

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Double precision real.

Result Value

The number of seconds since the initial value of the operating system real-time clock.

Examples

```
      USE omp_lib
      DOUBLE PRECISION START, END
      START = omp_get_wtime()
!     Work to be timed
      END = omp_get_wtime()
      PRINT *, 'Stuff took ', END - START, ' seconds.'
      END
```

omp_in_final()

Purpose

The **omp_in_final** routine returns `.TRUE.` if the routine is called in a final task region. Otherwise, the routine returns `.FALSE.`.

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Default logical.

Result Value

If the routine is called in a final task region, the result value is `.TRUE.`; otherwise, the result value is `.FALSE.`.

omp_in_parallel()

Purpose

The **omp_in_parallel** function returns **.TRUE.** if you call it from the dynamic extent of a region executing in parallel and returns **.FALSE.** otherwise. If you call **omp_in_parallel** from a region that is serialized but nested within the dynamic extent of a region executing in parallel, the function will still return **.TRUE.**. (Nested parallel regions are serialized by default. See "omp_set_nested(enable_expr)" on page 189 and the OMP_NESTED environment variable for more information.)

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Default logical.

Result Value

.TRUE. if called from the dynamic extent of a region executing in parallel. **.FALSE.** otherwise.

Examples

In the following example, the first call to **omp_in_parallel** returns **.FALSE.** because the call is outside the dynamic extent of any parallel region. The second call returns **.TRUE.**, even if the nested **PARALLEL DO** loop is serialized, because the call is still inside the dynamic extent of the outer **PARALLEL DO** loop.

```
      USE omp_lib
      INTEGER N, M
      N = 4
      M = 3
      PRINT*, omp_in_parallel()
!$OMP PARALLEL DO
      DO I = 1,N
!$OMP   PARALLEL DO
        DO J=1, M
          PRINT *, omp_in_parallel()
        END DO
!$OMP   END PARALLEL DO
      END DO
!$OMP END PARALLEL DO
      END
```

omp_init_lock(svar)

Purpose

The **omp_init_lock** subroutine initializes a lock and associates it with the lock variable passed in as a parameter. After the call to **omp_init_lock**, the initial state of the lock variable is unlocked.

If you call this routine with a lock variable that you have already initialized, the result of the call is undefined.

Class

Subroutine.

Argument Type and Attributes

svar	Integer of kind **omp_lock_kind**.

Result Type and Attributes

None.

Result Value

None.

Examples

In the following example, threads and their associated tasks are generated by the parallel region, and one at a time, each task gains ownership of the lock associated with the lock variable LCK, prints the thread ID, and releases ownership of the lock.

```
      USE omp_lib
      INTEGER(kind=omp_lock_kind) LCK
      INTEGER ID
      CALL omp_init_lock(LCK)
!$OMP PARALLEL SHARED(LCK), PRIVATE(ID)
      ID = omp_get_thread_num()
      CALL omp_set_lock(LCK)
      PRINT *,'MY THREAD ID IS', ID
      CALL omp_unset_lock(LCK)
!$OMP END PARALLEL
      CALL omp_destroy_lock(LCK)
      END
```

omp_init_nest_lock(nvar)

Purpose

The **omp_init_nest_lock** subroutine allows you to initialize a nestable lock and associate it with the lock variable you specify. The initial state of the lock variable is unlocked, and the initial nesting count is zero. The value of *nvar* must be an unitialized nestable lock variable.

If you call **omp_init_nest_lock** using a variable that is already initialized, the result is undefined.

Class

Subroutine.

Argument Type and Attributes

nvar	Integer of kind **omp_nest_lock_kind**.

Result Type and Attributes

None.

Result Value

None.

Examples

The following example illustrates the use of a nestable lock for updating variable P in the PARALLEL SECTIONS construct.

```
      USE omp_lib
      INTEGER P
      INTEGER A
      INTEGER B
      INTEGER ( kind=omp_nest_lock_kind ) LCK
      CALL omp_init_nest_lock ( LCK )   ! initialize the nestable lock
!$OMP PARALLEL SECTIONS
!$OMP SECTION
      CALL omp_set_nest_lock ( LCK )
      P = P + A
      CALL omp_set_nest_lock ( LCK )
      P = P + B
      CALL omp_unset_nest_lock ( LCK )
      CALL omp_unset_nest_lock ( LCK )
!$OMP SECTION
      CALL omp_set_nest_lock ( LCK )
      P = P + B
      CALL omp_unset_nest_lock ( LCK )
!$OMP END PARALLEL SECTIONS

      CALL omp_destroy_nest_lock ( LCK )
      END
```

omp_set_dynamic(enable_expr)

Purpose

The **omp_set_dynamic** subroutine enables or disables dynamic adjustment, by the runtime environment, of the number of threads available to execute parallel regions.

If **enable_expr** is evaluated to **.TRUE.**, the runtime environment can automatically adjust the number of threads that are used to execute subsequent parallel regions to obtain the best use of system resources. The number of threads you specify using **omp_set_num_threads** becomes the maximum, not exact, thread count.

If **enable_expr** is evaluated to **.FALSE.**, dynamic adjustment of the number of threads is disabled. The runtime environment cannot automatically adjust the number of threads used to execute subsequent parallel regions. The value you pass to **omp_set_num_threads** becomes the exact thread count.

By default, dynamic thread adjustment is disabled. If your code depends on a specific number of threads for correct execution, you should explicitly disable dynamic threads.

If the routine is called from a portion of the program where the **omp_in_parallel** routine returns **.TRUE.**, the routine has no effect.

This subroutine has precedence over the **OMP_DYNAMIC** environment variable.

Class

Subroutine.

Argument Type and Attributes

enable_expr
    Logical.

Result Type and Attributes

None.

Result Value

None.

omp_set_lock(svar)

Purpose

The **omp_set_lock** subroutine forces the calling task region to wait until the specified lock is available before executing subsequent instructions. The calling task region is given ownership of the lock when it becomes available.

If you call this routine with an uninitialized lock variable, the result of the call is undefined. If a task region that owns a lock tries to lock it again by issuing a call to **omp_set_lock**, the call produces a deadlock.

Class

Subroutine.

Argument Type and Attributes

svar Integer of kind **omp_lock_kind**.

Result Type and Attributes

None.

Result Value

None.

Examples

In the following example, the lock variable LCK_X is used to avoid race conditions when updating the shared variable X. By setting the lock before each update to X and unsetting it after the update, you ensure that only one task region updates X at a given time.

```
      USE omp_lib
      INTEGER A(100), X
      INTEGER(kind=omp_lock_kind) LCK_X
      X=1
      CALL omp_init_lock (LCK_X)
!$OMP PARALLEL PRIVATE (I), SHARED (A, X)
!$OMP DO
      DO I = 3, 100
        A(I) = I * 10
        CALL omp_set_lock (LCK_X)
        X = X + A(I)
        CALL omp_unset_lock (LCK_X)
      END DO
!$OMP END DO
!$OMP END PARALLEL
      CALL omp_destroy_lock (LCK_X)
      END
```

omp_set_max_active_levels(max_levels)

Purpose

The **omp_set_max_active_levels** subroutine limits the number of nested, active parallel regions.

Class

Subroutine.

Argument Type and Attributes

`max_levels`
: Default integer.

Result Type and Attributes

None.

Result Value

None.

omp_set_nested(enable_expr)

Purpose

The **omp_set_nested** subroutine enables or disables nested parallelism.

If **enable_expr** is evaluated to **.FALSE.**, nested parallelism is disabled. Nested parallel regions are serialized, and they are executed by the current thread. This is the default setting.

If **enable_expr** is evaluated to **.TRUE.**, nested parallelism is enabled. Parallel regions that are nested can deploy additional threads to the team. It is up to the runtime environment to determine whether additional threads should be deployed. Therefore, the number of threads used to execute parallel regions may vary from one nested region to the next.

If the routine is called from a portion of the program where the **omp_in_parallel** routine returns true, the routine has no effect.

This subroutine takes precedence over the **OMP_NESTED** environment variable.

Currently, XL Fortran does not support OpenMP nested parallelism.

Class

Subroutine.

Argument Type and Attributes

enable_expr
Logical.

Result Type and Attributes

Default logical.

Result Value

None.

omp_set_nest_lock(nvar)

Purpose

The **omp_set_nest_lock** subroutine allows you to set a nestable lock. The task region executing the subroutine will wait until the lock becomes available and then set that lock, incrementing the nesting count. A nestable lock is available if it is owned by the task region executing the subroutine, or is unlocked.

Class

Subroutine.

Argument Type and Attributes

nvar Integer of kind **omp_nest_lock_kind**.

Result Type and Attributes

None.

Result Value

None.

Examples

```
USE omp_lib
INTEGER P
INTEGER A
INTEGER B
INTEGER ( kind=omp_nest_lock_kind ) LCK

CALL omp_init_nest_lock ( LCK )

!$OMP PARALLEL SECTIONS
!$OMP SECTION
CALL omp_set_nest_lock ( LCK )
P = P + A
CALL omp_set_nest_lock ( LCK )
P = P + B
CALL omp_unset_nest_lock ( LCK )
CALL omp_unset_nest_lock ( LCK )
!$OMP SECTION
CALL omp_set_nest_lock ( LCK )
P = P + B
CALL omp_unset_nest_lock ( LCK )
!$OMP END PARALLEL SECTIONS

CALL omp_destroy_nest_lock ( LCK )
END
```

omp_set_num_threads(number_of_threads_expr)

Purpose

The **omp_set_num_threads** routine specifies the number of threads to use for the next parallel region by setting the first value of *num_list* for the **OMP_NUM_THREADS** environment variable.

The **number_of_threads_expr** argument is evaluated, and its value is used as the number of threads. If you have enabled dynamic adjustment of the number of threads (see "omp_set_dynamic(enable_expr)" on page 187), **omp_set_num_threads** sets the maximum number of threads to use for the next parallel region. The runtime environment then determines the exact number of threads to use. However, when dynamic adjustment of the number of threads is disabled, **omp_set_num_threads** sets the exact number of threads to use in the next parallel region. If the number of threads you request exceeds the number your execution environment can support, your application will terminate.

This subroutine takes precedence over the **OMP_NUM_THREADS** environment variable.

If you call this subroutine from the dynamic extent of a region executing in parallel, the behavior of the subroutine is undefined.

Class

Subroutine.

Argument Type and Attributes

number_of_threads_expr
: integer

Result Type and Attributes

None.

Result Value

None.

omp_set_schedule(kind, modifier)

Purpose

The **omp_set_schedule** routine affects the schedule that is applied when **runtime** is used as schedule kind. Use **omp_set_schedule** if you want to set the schedule type separately from the *OMP_SCHEDULE* environment variable.

Note: You can use the **omp_get_schedule** to return scheduling type. For details, see omp_get_schedule .

Class

Subroutine.

Argument Type and Attributes

kind
: Type integer with kind **omp_sched_kind**. Must be one of the schedule types as represented by the following constants:

- **omp_sched_static**
- **omp_sched_dynamic**
- **omp_sched_guided**
- **omp_sched_auto**
- **omp_sched_affinity**

where **omp_sched_affinity** is not part of the OpenMP specification.

modifier
: Default integer. For the schedule type **dynamic**, **guided**, or **static**, *modifier* is the chunk size that you want to set. Typically, it is a positive integer. If the value is less than one, the default is used. For the schedule type **auto**, *modifier* has no meaning. For the default setting of each schedule type, see -qsmp in the *XL Fortran Compiler Reference*.

Result Type and Attributes

None.

Result Value

None.

omp_test_lock(svar)

Purpose

The **omp_test_lock** function attempts to set the lock associated with the specified lock variable. It returns **.TRUE.** if it was able to set the lock and **.FALSE.** otherwise. In either case, the calling task region will continue to execute subsequent instructions in the program.

If you call **omp_test_lock** with an uninitialized lock variable, the result of the call is undefined.

Class

Function.

Argument Type and Attributes

svar	Integer of kind **omp_lock_kind**.

Result Type and Attributes

Default logical.

Result Value

.TRUE. if the function was able to set the lock. **.FALSE.** otherwise.

Examples

In the following example, a task region repeatedly executes `WORK_A` until it can set the lock variable, `LCK`. When the lock is set, the task region executes `WORK_B`.

```
      USE omp_lib
      INTEGER LCK
      INTEGER ID
      CALL omp_init_lock (LCK)
!$OMP PARALLEL SHARED(LCK), PRIVATE(ID)
      ID = omp_get_thread_num()
      DO WHILE (.NOT. omp_test_lock(LCK))
        CALL WORK_A (ID)
      END DO
      CALL WORK_B (ID)
      CALL omp_unset_lock (LCK)
!$OMP END PARALLEL
      CALL omp_destroy_lock (LCK)
      END
```

omp_test_nest_lock(nvar)

Purpose

The **omp_test_nest_lock** subroutine allows you to attempt to set a lock using the same method as **omp_set_nest_lock**, but the execution task region does not wait for confirmation that the lock is available. If the lock is successfully set, the function will increment the nesting count and return the new nesting count. If the lock is unavailable the function returns a value of zero. Also, a child task sees a value of zero if the parent task has already set the same lock. The result value is always a default integer.

Class

Function.

Argument Type and Attributes

nvar Integer of kind **omp_nest_lock_kind**.

Result Type and Attributes

Default integer.

Result Value

The new nesting count if the lock is successfully set; otherwise, it returns zero.

omp_unset_lock(svar)

Purpose

The **omp_unset_lock** subroutine causes the executing task region to release ownership of the specified lock. The lock can then be set by another task region as required. The behavior of the **omp_unset_lock** subroutine is undefined if either of the following conditions occur:

- The calling task region does not own the lock specified.
- The routine is called with an uninitialized lock variable.

Class

Subroutine.

Argument Type and Attributes

svar Integer of kind **omp_lock_kind**.

Result Type and Attributes

None.

Result Value

None.

Examples

```
      USE omp_lib
      INTEGER A(100)
      INTEGER(kind=omp_lock_kind) LCK_X
      CALL omp_init_lock (LCK_X)
!$OMP PARALLEL PRIVATE (I), SHARED (A, X)
!$OMP DO
      DO I = 3, 100
        A(I) = I * 10
        CALL omp_set_lock (LCK_X)
        X = X + A(I)
        CALL omp_unset_lock (LCK_X)
      END DO
!$OMP END DO
!$OMP END PARALLEL
      CALL omp_destroy_lock (LCK_X)
      END
```

In this example, the lock variable LCK_X is used to avoid race conditions when updating the shared variable X. By setting the lock before each update to X and unsetting it after the update, you ensure that only one task region is updating X at a given time.

omp_unset_nest_lock(nvar)

Purpose

The **omp_unset_nest_lock** subroutine allows you to release ownership of a nestable lock. The subroutine decrements the nesting count and releases the associated task region from ownership of the nestable lock.

Class

Subroutine.

Argument Type and Attributes

nvar Integer of kind **omp_lock_kind**.

Result Type and Attributes

None.

Result Value

None.

Examples

```
USE omp_lib
INTEGER P
INTEGER A
INTEGER B
INTEGER ( kind=omp_nest_lock_kind ) LCK

CALL omp_init_nest_lock ( LCK )

!$OMP PARALLEL SECTIONS
!$OMP SECTION
CALL omp_set_nest_lock ( LCK )
P = P + A
CALL omp_set_nest_lock ( LCK )
P = P + B
CALL omp_unset_nest_lock ( LCK )
CALL omp_unset_nest_lock ( LCK )
!$OMP SECTION
CALL omp_set_nest_lock ( LCK )
P = P + B
CALL omp_unset_nest_lock ( LCK )
!$OMP END PARALLEL SECTIONS

CALL omp_destroy_nest_lock ( LCK )
END
```

Routines for thread-level speculative execution

To call the routines that are specific to thread-level speculative execution, you must use the `speculation_util` intrinsic module.

The routines for thread-level speculative execution have no effect unless the thread-level speculative execution is enabled with the "-qsmp=speculative" compiler option.

Structure of statistic counters

The following structure that contains the statistic counters can be used with the thread-level speculative execution routines. It is declared in the `speculation_util` intrinsic module.

```
TYPE SeReport

  SEQUENCE

  ! Total number of chunks that are committed by nonespeculative threads
  INTEGER(8) totalNONSpecCommitted

  ! Total number of chunks that are committed by speculative threads
  INTEGER(8) totalSpecCommitted

  ! Total number of rollbacks for speculative threads
  INTEGER(8) totalRollbacks

  ! Total number of serialization caused by JMV conflicts
  INTEGER(8) totalSerializedJMV

  ! Total number of serialization caused by SE_MAX_NUM_ROLLBACK reached
  INTEGER(8) totalSerializedMAXRB

  ! Total number of serialization caused by incorrect hardware
  ! speculative mode or nesting
  INTEGER(8) totalSerializedOTHER

END TYPE
```

se_get_all_stats(stats)

Purpose

With the **se_get_all_stats** routine, you can retrieve statistical information for the thread-level speculative execution of all threads in a program.

Class

Subroutine.

Argument Type and Attributes

`stats`
An `INTENT(OUT) SeReport` variable.

Result Type and Attributes

None.

Result Value

None.

Usage

When the **se_get_all_stats** routine is called, it updates the argument with cumulative statistics of all the regions of thread-level speculative execution that all hardware threads have run.

To use the **se_get_all_stats** routine, you must set the SE_REPORT_STAT_ENABLE environment variable to YES.

You must use this routine outside of parallel regions.

Related information

- Environment variables for thread-level speculative execution

se_print_stats()

Purpose

With the **se_print_stats** routine, you can write statistics about thread-level speculative execution to a log file.

Class

Subroutine.

Example

```
...
!SEP$ SPECULATIVE DO
  DO-LOOP

  CALL SE_PRINT_STATS()
...
```

Usage

After a region of thread-level speculative execution, you can call the **se_print_stats** routine to write cumulative statistics of all the regions of thread-level speculative execution that each hardware thread has run to a log file.

To enable the statistics log file to be generated at each call to the **se_print_stats** routine, you must set the SE_REPORT_LOG environment variable to FUNC, ALL, or VERBOSE.

By default, the log file is named `se_report.log.`*rank* where *rank* is the MPI rank of the process that called the **se_print_stats** routine. The file is saved in the current working directory of the program that uses thread-level speculative execution. To override the defaults, you can specify the SE_REPORT_NAME environment variable.

You must use this routine outside of parallel regions.

Note: If you use this function in a region of thread-level speculative execution, the code of that region is run in irrevocable mode.

Related information

- Environment variables for thread-level speculative execution

reset_speculation_mode()

Purpose

With the **reset_speculation_mode** routine, you can switch mode in between thread-level speculative execution (SE) and transactional memory (TM) at runtime.

Class

Subroutine.

Example

```
...
!SEP$ SPECULATIVE DO
  do-loop
!SEP$ END SPECULATIVE DO

  CALL RESET_SPECULATION_MODE()

!TM$ TM_ATOMIC
  code-block
!TM$ END TM_ATOMIC
...
```

Usage

The call to the **reset_speculation_mode** routine is needed only when you are switching mode between TM and SE.

If your program consists of both TM and SE regions, it is recommended that you make a call to the **reset_speculation_mode** routine; otherwise, undefined behavior might be caused.

When you call the **reset_speculation_mode** routine, make sure all the TM or SE executions before the call are completed.

Related information

- "Nesting OpenMP, transactional memory, and thread-level speculative execution" on page 90

Routines for transactional memory

To call the routines that are specific to transactional memory, you must use the `speculation_util` intrinsic module.

The routines for transactional memory have no effect unless transactional memory is enabled with the "-qtm" compiler option.

Structure of statistic counters

The following structure that contains the statistic counters can be used with the transactional memory routines. It is declared in the `speculation_util` intrinsic module.

```
TYPE TmReport

  SEQUENCE

  ! Thread ID
  INTEGER(8) hwThreadId

  ! Total number of transactions
  INTEGER(8) totalTransactions

  ! Total number of rollbacks for transactional memory threads
  INTEGER(8) totalRollbacks

  ! Total number of serialization caused by JMV conflicts
  INTEGER(8) totalSerializedJMV

  ! Total number of serialization caused by TM_MAX_NUM_ROLLBACK reached
  INTEGER(8) totalSerializedMAXRB

  ! Total number of serialization caused by incorrect hardware
  ! speculative mode or nesting
  INTEGER(8) totalSerializedOTHER

END TYPE
```

tm_get_stats(stats)

Purpose

With the **tm_get_stats** routine, you can retrieve statistical information for the transactional memory of a particular hardware thread in your program.

Class

Subroutine.

Argument Type and Attributes

`stats`
An `INTENT(OUT) TmReport` variable.

Result Type and Attributes

None.

Result Value

None.

Usage

When the **tm_get_stats** routine is called, it updates the argument with cumulative statistics of all the transactional atomic regions that a particular hardware thread has run.

To use the **tm_get_stats** routine, you must set the TM_REPORT_STAT_ENABLE environment variable to YES.

Related information

- Environment variables for transactional memory

tm_get_all_stats(stats)

Purpose

With the **tm_get_all_stats** routine, you can retrieve statistical information for the transactional memory of all hardware threads in a program.

Class

Subroutine.

Argument Type and Attributes

`stats`
An `INTENT(OUT)` `TmReport` variable.

Result Type and Attributes

None.

Result Value

None.

Usage

When the **tm_get_all_stats** routine is called, it updates the argument with cumulative statistics of all the transactional atomic regions that all hardware threads have run.

To use the **tm_get_all_stats** routine, you must set the TM_REPORT_STAT_ENABLE environment variable to YES.

You must use this routine outside of parallel regions.

Related information

- Environment variables for transactional memory

tm_print_stats()

Purpose

With the **tm_print_stats** routine, you can write statistics for the transactional memory of a particular hardware thread to a log file.

Class

Subroutine.

Example

```
  ...
!TM$ TM_ATOMIC
  code-block
!TM$ END TM_ATOMIC

  CALL TM_PRINT_STATS()
  ...
```

Usage

After a transactional atomic region, you can call the **tm_print_stats** routine to write cumulative statistics of all the transactional atomic regions that a particular hardware thread has run to a log file.

To enable the statistics log file to be generated at each call to the **tm_print_stats** routine, you must set the TM_REPORT_LOG variable to FUNC, ALL, or VERBOSE.

By default, the log file is named `tm_report.log.`*`rank`*, where *rank* in the extension is the MPI rank of the process that called **tm_print_stats**. The log file is saved in the current working directory of the program that uses transactional memory. To override the defaults, you can specify the TM_REPORT_NAME environment variable.

Note: If you use this routine in a transactional atomic region, it causes the transaction running in irrevocable mode.

Related information

- Environment variables for transactional memory

tm_print_all_stats()

Purpose

With the **tm_print_all_stats** routine, you can write statistical information for the transactional memory of all hardware threads in a program to a log file.

Class

Subroutine.

Example

```
  ...
!TM$ TM_ATOMIC
  code-block
!TM$ END TM_ATOMIC

  CALL TM_PRINT_ALL_STATS()
  ...
```

Usage

When the **tm_print_all_stats** routine is called, it writes the cumulative statistics of all the transactional atomic regions that all hardware threads have run to a log file.

By default, the log file is named `tm_report.log.`*`rank`*, where *rank* in the extension is the MPI rank of the process that called **tm_print_all_stats**. The log file is saved in the current working directory of the program that uses transactional memory. To override the defaults, you can specify the TM_REPORT_NAME environment variable.

To enable the statistics log file to be generated at each call to the **tm_print_all_stats** routine, you must set the TM_REPORT_LOG variable to FUNC, ALL, or VERBOSE.

You must use this routine outside of parallel regions.

Related information

- Environment variables for transactional memory

reset_speculation_mode()

Purpose

With the **reset_speculation_mode** routine, you can switch mode in between thread-level speculative execution (SE) and transactional memory (TM) at runtime.

Class

Subroutine.

Example

```
...
!SEP$ SPECULATIVE DO
  do-loop
!SEP$ END SPECULATIVE DO

  CALL RESET_SPECULATION_MODE()

!TM$ TM_ATOMIC
  code-block
!TM$ END TM_ATOMIC
...
```

Usage

The call to the **reset_speculation_mode** routine is needed only when you are switching mode between TM and SE.

If your program consists of both TM and SE regions, it is recommended that you make a call to the **reset_speculation_mode** routine; otherwise, undefined behavior might be caused.

When you call the **reset_speculation_mode** routine, make sure all the TM or SE executions before the call are completed.

Related information

- "Nesting OpenMP, transactional memory, and thread-level speculative execution" on page 90

Pthreads library module

The Pthreads Library Module (**f_pthread**) is a Fortran 90 module that defines data types and routines to make it easier to interface with the Blue Gene/Q pthreads library. The Blue Gene/Q pthreads library is used to parallelize and to make your code thread-safe.

The **f_pthread** library module naming convention is the use of the prefix **f_** before the corresponding Blue Gene/Q pthreads library routine name or type definition name.

In general, there is a one-to-one corresponding relationship between the procedures in the Fortran 90 module **f_pthread** and the library routines contained in the Blue Gene/Q pthreads library. However, some of the pthread routines have no corresponding procedures in this module because they are not supported on Blue Gene/Q. One example of these routines is the thread stack address option. There are also some non-pthread interfacing routines contained in the **f_pthread** library module. The **f_maketime** routine is one example and is included to return an absolute time in a **f_timespec** derived type variable.

Most of the routines return an integer value. A return value of **0** will always indicate that the routine call did not result in any error. Any non-zero return value indicates an error. Each error code has a corresponding definition of a system error code in Fortran. These error codes are available as Fortran integer constants. The naming of these error codes in Fortran is consistent with the corresponding Blue Gene/Q error code names. For example, **EINVAL** is the Fortran constant name of the error code **EINVAL** on the system. For a complete list of these error codes, refer to the file **/usr/include/errno.h**.

Note: The pthread module in XL Fortran is an extension to the standard Fortran language.

Pthreads data structures, functions, and subroutines

Pthreads Data Types

- f_pthread_attr_t
- f_pthread_cond_t
- f_pthread_condattr_t
- f_pthread_key_t
- f_pthread_mutex_t
- f_pthread_mutexattr_t
- f_pthread_once_t
- f_pthread_rwlock_t
- f_pthread_rwlockattr_t
- f_pthread_t
- f_sched_param
- f_timespec

Functions that perform operations on thread attribute objects

- f_pthread_attr_destroy(attr)
- f_pthread_attr_getdetachstate(attr, detach)
- f_pthread_attr_getguardsize(attr, guardsize)
- f_pthread_attr_getinheritsched(attr, inherit)
- f_pthread_attr_getschedparam(attr, param)
- f_pthread_attr_getschedpolicy(attr, policy)
- f_pthread_attr_getscope(attr, scope)
- f_pthread_attr_getstack(attr, stackaddr, ssize)
- f_pthread_attr_init(attr)
- f_pthread_attr_setdetachstate(attr, detach)
- f_pthread_attr_setguardsize(attr, guardsize)
- f_pthread_attr_setinheritsched(attr, inherit)
- f_pthread_attr_setschedparam(attr, param)
- f_pthread_attr_setschedpolicy(attr, policy)
- f_pthread_attr_setscope(attr, scope)

- f_pthread_attr_setstack(attr, stackaddr, ssize)

Functions and Subroutines That Perform Operations on Threads

- f_pthread_cancel(thread)
- f_pthread_cleanup_pop(exec)
- f_pthread_cleanup_push(cleanup, flag, arg)
- f_pthread_create(thread, attr, flag, ent, arg)
- f_pthread_detach(thread)
- f_pthread_equal(thread1, thread2)
- f_pthread_exit(ret)
- f_pthread_getconcurrency()
- f_pthread_getschedparam(thread, policy, param)
- f_pthread_join(thread, ret)
- f_pthread_kill(thread, sig)
- f_pthread_self()
- f_pthread_setconcurrency(new_level)
- f_pthread_setschedparam(thread, policy, param)

Functions that perform operations on mutex attribute objects

- f_pthread_mutexattr_destroy(mattr)
- f_pthread_mutexattr_getpshared(mattr, pshared)
- f_pthread_mutexattr_gettype(mattr, type)
- f_pthread_mutexattr_init(mattr)
- f_pthread_mutexattr_setpshared(mattr, pshared)
- f_pthread_mutexattr_settype(mattr, type)

Functions that perform operations on mutex objects

- f_pthread_mutex_destroy(mutex)
- f_pthread_mutex_init(mutex, mattr)
- f_pthread_mutex_lock(mutex)
- f_pthread_mutex_trylock(mutex)
- f_pthread_mutex_unlock(mutex)

Functions that perform operations on attribute objects of condition variables

- f_pthread_condattr_destroy(cattr)
- f_pthread_condattr_getpshared(cattr, pshared)
- f_pthread_condattr_init(cattr)
- f_pthread_condattr_setpshared(cattr, pshared)

Functions that perform operations on condition variable objects

- f_maketime(delay)
- f_pthread_cond_broadcast(cond)
- f_pthread_cond_destroy(cond)
- f_pthread_cond_init(cond, cattr)
- f_pthread_cond_signal(cond)
- f_pthread_cond_timedwait(cond, mutex, timeout)

- f_pthread_cond_wait(cond, mutex)

Functions that perform operations on thread-specific data

- f_pthread_getspecific(key, arg)
- f_pthread_key_create(key, dtr)
- f_pthread_key_delete(key)
- f_pthread_setspecific(key, arg)

Functions and subroutines that perform operations to control thread cancelability

- f_pthread_setcancelstate(state, oldstate)
- f_pthread_setcanceltype(type, oldtype)
- f_pthread_testcancel()

Functions that perform operations on read-write lock attribute objects

- f_pthread_rwlockattr_destroy(rwattr)
- f_pthread_rwlockattr_getpshared(rwattr, pshared)
- f_pthread_rwlockattr_init(rwattr)
- f_pthread_rwlockattr_setpshared(rwattr, pshared)

Functions that perform operations on read-write lock objects

- f_pthread_rwlock_destroy(rwlock)
- f_pthread_rwlock_init(rwlock, rwattr)
- f_pthread_rwlock_rdlock(rwlock)
- f_pthread_rwlock_tryrdlock(rwlock)
- f_pthread_rwlock_trywrlock(rwlock)
- f_pthread_rwlock_unlock(rwlock)
- f_pthread_rwlock_wrlock(rwlock)

Functions that perform operations for one-time initialization

- f_pthread_once(once, initr)

f_maketime(delay)

Purpose

This function accepts an integer value specifying a delay in seconds and returns an **f_timespec** type object containing the absolute time, which is **delay** seconds from the calling moment.

Class

Function

Argument Type and Attributes

delay INTEGER(4), INTENT(IN)

Result Type and Attributes

TYPE (f_timespec)

Result Value

The absolute time, which is **delay** seconds from the calling moment, is returned.

f_pthread_attr_destroy(attr)

Purpose

This function must be called to destroy any previously initialized thread attribute objects when they will no longer be used. Threads that were created with this attribute object will not be affected in any way by this action. Memory that was allocated when it was initialized will be recollected by the system.

Class

Function

Argument Type and Attributes

attr TYPE(f_pthread_attr_t), INTENT(IN)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument **attr** is invalid.

f_pthread_attr_getdetachstate(attr, detach)

Purpose

This function can be used to query the setting of the detach state attribute in the thread attribute object **attr**. The current setting will be returned through argument **detach**.

Class

Function

Argument Type and Attributes

attr TYPE(f_pthread_attr_t), INTENT(IN)

detach INTEGER(4), INTENT(OUT)

Contains one of the following values:

PTHREAD_CREATE_DETACHED:
when a thread attribute object of this attribute setting is used to create a new thread, the newly created thread will be in detached state.

PTHREAD_CREATE_JOINABLE:
when a thread attribute object of this attribute setting is used to create a new thread, the newly created thread will be in undetached state.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error:

EINVAL
The argument **attr** is invalid.

f_pthread_attr_getguardsize(attr, guardsize)

Purpose

This function is used to get the *guardsize* attribute in the thread attribute object *attr*. The current setting of the attribute will be returned through the argument *guardsize*.

Class

Function

Argument Type and Attributes

attr TYPE(f_pthread_attr_t), INTENT(IN)

guardsize
INTEGER(KIND=register_size), INTENT(IN)

where *register_size* is 8 in 64-bit mode.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error:

EINVAL
The argument attr is invalid.

f_pthread_attr_getinheritsched(attr, inherit)

Purpose

This function can be used to query the inheritance scheduling attribute in the thread attribute object **attr**. The current setting will be returned through the argument **inherit**.

Class

Function

Argument Type and Attributes

attr TYPE(f_pthread_attr_t), INTENT(OUT)

inherit
INTEGER(4)

On return from the function, **inherit** contains one of the following values:

PTHREAD_INHERIT_SCHED:
indicating that newly created threads will inherit the scheduling property of the parent thread and ignore the scheduling property of the thread attribute object used to create them.

PTHREAD_EXPLICIT_SCHED:
the scheduling property in the thread attribute object will be assigned to the newly created threads when it is used to create them.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise this function returns the following error.

EINVAL
The argument **attr** is invalid.

f_pthread_attr_getschedparam(attr, param)

Purpose

This function can be used to query the scheduling property setting in the thread attribute object **attr**. The current setting will be returned in the argument **param**.

Class

Function

Argument Type and Attributes

attr TYPE(f_pthread_attr_t), INTENT(IN)

param TYPE(f_sched_param), INTENT(OUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument **attr** is invalid.

f_pthread_attr_getschedpolicy(attr, policy)

Purpose

This function can be used to query the scheduling policy attribute setting in the attribute object **attr**. The current setting of the scheduling policy will be returned in the argument **policy**.

Class

Function

Argument Type and Attributes

attr	TYPE(f_pthread_attr_t), INTENT(IN)
policy	INTEGER(4), INTENT(OUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument **attr** is invalid.

f_pthread_attr_getscope(attr, scope)

Purpose

This function can be used to query the current setting of the scheduling scope attribute in the thread attribute object **attr**. The current setting will be returned through the argument **scope**.

Class

Function

Argument Type and Attributes

attr TYPE(f_pthread_attr_t), INTENT(IN)

scope INTEGER(4), INTENT(OUT)

On return from the function, **scope** will contain one of the following values:

PTHREAD_SCOPE_SYSTEM:
the thread will compete for system resources on a system wide scope.

PTHREAD_SCOPE_PROCESS:
the thread will compete for system resources locally within the owning process.

scope Contains the following value:

PTHREAD_SCOPE_SYSTEM:
the thread will compete for system resources on a system wide scope.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument **attr** is invalid.

f_pthread_attr_getstack(attr, stackaddr, ssize)

Purpose

Retrieves the values of the **stackaddr** and **stacksize** arguments from the thread attribute object **attr**.

Class

Function

Argument Type and Attributes

attr TYPE(f_pthread_attr_t), INTENT(IN)

stackaddr
Integer pointer, INTENT(OUT)

ssize INTEGER(KIND=register_size), INTENT(OUT)

where *register_size* is 8 in 64-bit mode.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
One or more of the supplied arguments is invalid.

f_pthread_attr_init(attr)

Purpose

This function must be called to create and initialize the pthread attribute object **attr** before it can be used in any way. It will be filled with system default thread attribute values. After it is initialized, certain pthread attributes can be changed and/or set through attribute access procedures. Once initialized, this attribute object can be used to create a thread with the intended attributes.

Class

Function

Argument Type and Attributes

attr TYPE(f_pthread_attr_t), INTENT(OUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument **attr** is invalid.

f_pthread_attr_setdetachstate(attr, detach)

Purpose

This function can be used to set the detach state attribute in the thread attribute object **attr**.

Class

Function

Argument Type and Attributes

attr TYPE(f_pthread_attr_t), INTENT(OUT)

detach INTEGER(4), INTENT(IN)

Must contain one of the following values:

PTHREAD_CREATE_DETACHED:
when a thread attribute object of this attribute setting is used to create a new thread, the newly created thread will be in detached state. This is the system default setting.

PTHREAD_CREATE_JOINABLE:
when a thread attribute object of this attribute setting is used to create a new thread, the newly created thread will be in undetached state.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument **detach** is invalid.

f_pthread_attr_setguardsize(attr, guardsize)

Purpose

This function is used to set the **guardsize**attribute in the thread attributes object **attr**. The new value of this attribute is obtained from the argument **guardsize**. If **guardsize** is zero, a guard area will not be provided for threads created with **attr**. If **guardsize** is greater than zero, a guard area of at least size**guardsize** bytes is provided for each thread created with **attr**.

Class

Function

Argument Type and Attributes

attr TYPE(f_pthread_attr_t), INTENT(INOUT)

guardsize
INTEGER(KIND=register_size), INTENT(IN)

where *register_size* is 8 in 64-bit mode.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument **attr** or the argument **guardsize** is invalid.

f_pthread_attr_setinheritsched(attr, inherit)

Purpose

This function can be used to set the inheritance attribute of the thread scheduling property in the thread attribute object **attr**.

Class

Function

Argument Type and Attributes

attr TYPE(f_pthread_attr_t), INTENT(OUT)

inherit
INTEGER(4), INTENT(IN)

Must contain one of the following values:

PTHREAD_INHERIT_SCHED:
indicating that newly created threads will inherit the scheduling

property of the parent thread and ignore the scheduling property of the thread attribute object used to create them.

PTHREAD_EXPLICIT_SCHED:
: the scheduling property in the thread attribute object will be assigned to the newly created threads when it is used to create them.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
: The argument **inherit** is invalid.

f_pthread_attr_setschedparam(attr, param)

Purpose

This function can be used to set the scheduling property attribute in the thread attribute object **attr**. Threads created with this new attribute object will assume the scheduling property of argument **param** if they are not inherited from the creating thread. The sched_priority field in argument **param** indicates the thread's scheduling priority. The priority field must assume a value in the range of 1-127, where *127* is the most favored scheduling priority while *1* is the least.

Class

Function

Argument Type and Attributes

attr	TYPE(f_pthread_attr_t), INTENT(INOUT)
param	TYPE(f_sched_param), INTENT(IN)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
: The argument **param** is invalid.

f_pthread_attr_setschedpolicy(attr, policy)

Purpose

After the attribute object is set by this function, threads created with this attribute object will assume the set scheduling policy if the scheduling property is not inherited from the creating thread.

Class

Function

Argument Type and Attributes

attr TYPE(f_pthread_attr_t), INTENT(INOUT)

policy INTEGER(4), INTENT(IN)

Must contain one of the following values:

SCHED_FIFO:
indicating a first-in first-out thread scheduling policy.

SCHED_RR:
indicating a round-robin scheduling policy.

SCHED_OTHER:
the default scheduling policy.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following error.

EINVAL
The argument **policy** is invalid.

f_pthread_attr_setscope(attr, scope)

Purpose

This function can be used to set the contention scope attribute in the thread attribute object **attr**.

Class

Function

Argument Type and Attributes

attr TYPE(f_pthread_attr_t), INTENT(INOUT)

scope INTEGER(4), INTENT(IN)

Must contain one of the following values:

PTHREAD_SCOPE_SYSTEM:
the thread will compete for system resources on a system wide scope.

PTHREAD_SCOPE_PROCESS:
the thread will compete for system resources locally within the owning process.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument **scope** is invalid.

f_pthread_attr_setstack(attr, stackaddr, ssize)

Purpose

Use this function to set the stack address and stack size attributes in the pthread attribute object **attr**. The **stackaddr** argument represents the stack address as an Integer pointer. The **stacksize** argument is an integer that represents the size of the stack in bytes. When creating a thread using the attribute object **attr**, the system allocates a minimum stack size of **stacksize** bytes.

Class

Function

Argument Type and Attributes

attr TYPE(f_pthread_attr_t), INTENT(INOUT)

stackaddr
Integer pointer, INTENT(IN)

ssize INTEGER(KIND=register_size)

where *register_size* is 8 in 64-bit mode.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EINVAL
The value of one or both of the supplied arguments is invalid.

EACCES
The stack pages specified are not readable by the thread.

f_pthread_attr_t

Purpose

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the POSIX **pthread_attr_t**, which is the type of thread attribute object.

Class

Data Type.

f_pthread_cancel(thread)

Purpose

This function can be used to cancel a target thread. How this cancelation request will be processed depends on the state of the cancelability of the target thread. The target thread is identified by argument **thread**. If the target thread is in deferred-cancel state, this cancelation request will be put on hold until the target thread reaches its next cancelation point. If the target thread disables its cancelability, this request will be put on hold until it is enabled again. If the target thread is in async-cancel state, this request will be acted upon immediately.

Class

Function

Argument Type and Attributes

thread TYPE(f_pthread_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

ESRCH
The argument **thread** is invalid.

f_pthread_cleanup_pop(exec)

Purpose

This subroutine should be paired with **f_pthread_cleanup_push** in using the cleanup stack for thread safety. If the supplied argument **exec** contains a non-zero value, the last pushed cleanup function will be popped from the cleanup stack and executed, with the argument **arg** (from the last **f_pthread_cleanup_push**) passed to the cleanup function.

If **exec** contains a zero value, the last pushed cleanup function will be popped from the cleanup stack, but will not be executed.

Class

Subroutine

Argument Type and Attributes

exec	INTEGER(4), INTENT(IN)

Result Type and Attributes

None.

Result Value

None.

f_pthread_cleanup_push(cleanup, flag, arg)

Purpose

This function can be used to register a cleanup subroutine for the calling thread. In case of an unexpected termination of the calling thread, the system will automatically execute the cleanup subroutine in order for the calling thread to terminate safely. The argument **cleanup** must be a subroutine expecting exactly one argument. If it is executed, the argument **arg** will be passed to it as the actual argument.

The argument **arg** is a generic argument that can be of any type and any rank. The actual argument arg must be a variable, and consequently eligible as a left-value in an assignment statement. If you pass an array section with vector subscripts to the argument **arg**, the result is unpredictable.

If the actual argument **arg** is an array section, the corresponding dummy argument in subroutine **cleanup** must be an assumed-shape array. Otherwise, the result is unpredictable.

If the actual argument **arg** has the pointer attribute that points to an array or array section, the corresponding dummy argument in subroutine **cleanup** must have a pointer attribute or be an assumed-shape array. Otherwise, the result is unpredictable.

For a normal execution path, this function must be paired with a call to **f_pthread_cleanup_pop**.

The argument **flag** must be used to convey the property of argument **arg** exactly to the system.

Class

Function

Argument Type and Attributes

cleanup
: A subroutine that has one dummy argument.

flag
: Flag is an INTEGER(4), INTENT(IN) argument that can contain one of, or a combination of, the following constants:

  FLAG_CHARACTER:
  : if the entry subroutine **cleanup** expects an argument of type **CHARACTER** in any way or any form, this flag value must be included to indicate this fact. However, if the subroutine expects a Fortran 90 pointer pointing to an argument of type **CHARACTER**, the **FLAG_DEFAULT** value should be included instead.

  FLAG_ASSUMED_SHAPE:
  : if the entry subroutine **cleanup** has a dummy argument that is an assumed-shape array of any rank, this flag value must be included to indicate this fact.

  FLAG_DEFAULT:
  : otherwise, this flag value is needed.

arg
: A generic argument that can be of any type, kind, and rank.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

ENOMEM
: The system cannot allocate memory to push this routine.

EAGAIN
: The system cannot allocate resources to push this routine.

EINVAL
: The argument **flag** is invalid.

f_pthread_cond_broadcast(cond)

Purpose

This function will unblock all threads waiting on the condition variable **cond**. If there is no thread waiting on this condition variable, the function will still succeed, but the next caller to **f_pthread_cond_wait** will be blocked, and will wait on the condition variable **cond**.

Class

Function

Argument Type and Attributes

cond
: TYPE(f_pthread_cond_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns following error.

EINVAL
: The argument **cond** is invalid.

f_pthread_cond_destroy(cond)

Purpose

This function can be used to destroy those condition variables that are no longer required. The target condition variable is identified by the argument **cond**. System resources allocated during initialization will be recollected by the system.

Class

Function

Argument Type and Attributes

cond TYPE(f_pthread_cond_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EBUSY
: The condition variable **cond** is being used by another thread.

f_pthread_cond_init(cond, cattr)

Purpose

This function can be used to dynamically initialize a condition variable **cond**. Its attributes will be set according to the attribute object **cattr**, if it is provided; otherwise, its attributes will be set to the system default. After the condition variable is initialized successfully, it can be used to synchronize threads.

Another method of initializing a condition variable is to initialize it statically using the Fortran constant **PTHREAD_COND_INITIALIZER**.

Class

Function

Argument Type and Attributes

cond TYPE(f_pthread_cond_t), INTENT(INOUT)

cattr TYPE(f_pthread_condattr_t), INTENT(IN), OPTIONAL

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EBUSY
The condition variable is already in use. It is initialized and not destroyed.

EINVAL
The argument **cond** or **cattr** is invalid.

f_pthread_cond_signal(cond)

Purpose

This function will unblock at least one thread waiting on the condition variable **cond**. If there is no thread waiting on this condition variable, the function will still succeed, but the next caller to **f_pthread_cond_wait** will be blocked, and will wait on the condition variable **cond**.

Class

Function

Argument Type and Attributes

cond TYPE(f_pthread_cond_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument **cond** is invalid.

f_pthread_cond_t

Purpose

A derived data type whose components are all private. Any object of this type should be manipulated through the appropriate interfaces provided in this module. In addition, objects of this type can be initialized at compile time using the Fortran constant **PTHREAD_COND_INITIALIZER**.

This data type corresponds to the POSIX **pthread_cond_t**, which is the type of condition variable object.

Class

Data Type.

f_pthread_cond_timedwait(cond, mutex, timeout)

Purpose

This function can be used to wait for a certain condition to occur. The argument **mutex** must be locked before calling this function. The mutex is unlocked atomically and the calling thread waits for the condition to occur. The argument **timeout** specifies a deadline before which the condition must occur. If the deadline is reached before the condition occurs, the function will return an error code. This function provides a cancelation point in that the calling thread can be canceled if it is in the enabled state.

The argument **timeout** will specify an absolute date of the form: Oct. 31 10:00:53, 1998. For related information, see **f_maketime** and **f_timespec**.

Class

Function

Argument Type and Attributes

cond	TYPE(f_pthread_cond_t), INTENT(INOUT)
mutex	TYPE(f_pthread_mutex_t), INTENT(INOUT)
timeout	TYPE(f_timespec), INTENT(IN)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise this function returns one of the following errors:

EINVAL	The argument **cond**, **mutex**, or **timeout** is invalid.
ETIMEDOUT	The waiting deadline was reached before the condition occurred.

f_pthread_cond_wait(cond, mutex)

Purpose

This function can be used to wait for a certain condition to occur. The argument **mutex** must be locked before calling this function. The mutex is unlocked atomically, and the calling thread waits for the condition to occur. If the condition does not occur, the function will wait until the calling thread is terminated in another way. This function provides a cancelation point in that the calling thread can be canceled if it is in the enabled state.

Class

Function

Argument Type and Attributes

cond	TYPE(f_pthread_cond_t), INTENT(INOUT)
mutex	TYPE(f_pthread_mutex_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

This function returns 0.

f_pthread_condattr_destroy(cattr)

Purpose

This function can be called to destroy the condition variable attribute objects that are no longer required. The target object is identified by the argument **cattr**. The system resources allocated when it is initialized will be recollected.

Class

Function

Argument Type and Attributes

cattr	TYPE(f_pthread_condattr_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns on of the following errors.

EINVAL
The argument **cattr** is invalid.

EBUSY
Returns **EBUSY** if threads are waiting on the for the condition to occur.

f_pthread_condattr_getpshared(cattr, pshared)

Purpose

This function can be used to query the process-shared attribute of the condition variable attributes object identified by the argument **cattr**. The current setting of this attribute will be returned in the argument **pshared**.

Class

Function

Argument Type and Attributes

cattr
TYPE(f_pthread_condattr_t), INTENT(IN)

pshared
INTEGER(4), INTENT(OUT)

On successful completion, **pshared** contains one of the following values:

PTHREAD_PROCESS_SHARED
The condition variable can be used by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.

PTHREAD_PROCESS_PRIVATE
The condition variable shall only be used by threads within the same process as the thread that created it.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument **cattr** is invalid.

f_pthread_condattr_init(cattr)

Purpose

Use this function to initialize a condition variable attributes object cattr with the default value for all of the attributes defined by the implementation. Attempting to initialize an already initialized condition variable attributes object results in undefined behavior. After a condition variable attributes object has been used to initialize one or more condition variables, any function affecting the attributes object (including destruction) does not affect any previously initialized condition variables.

Class

Function

Argument Type and Attributes

cattr
TYPE(f_pthread_condattr_t), INTENT(OUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

ENOMEM
: There is insufficient memory to initialize the condition variable attributes object.

f_pthread_condattr_setpshared(cattr, pshared)

Purpose

This function is used to set the process-shared attribute of the condition variable attributes object identified by the argument **cattr**. Its process-shared attribute will be set according to the argument **pshared**.

Class

Function

Argument Type and Attributes

cattr TYPE(f_pthread_condattr_t), INTENT(INOUT)

pshared
: is an INTEGER(4), INTENT(IN) argument that must contain one of the following values:

  PTHREAD_PROCESS_SHARED
  : Specifies that the condition variable can be used by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.

  PTHREAD_PROCESS_PRIVATE
  : Specifies that the condition variable shall only be used by threads within the same process as the thread that created it. This is the default setting of the attribute.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
: The value specified by the argument **cattr** or **pshared** is invalid.

f_pthread_condattr_t

Purpose

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the POSIX **pthread_condattr_t**, which is the type of condition variable attribute object.

Class

Data Type

f_pthread_create(thread, attr, flag, ent, arg)

Purpose

This function is used to create a new thread in the current process. The newly created thread will assume the attributes defined in the thread attribute object **attr**, if it is provided. Otherwise, the new thread will have system default attributes. The new thread will begin execution at the subroutine **ent**, which is required to have one dummy argument. The system will pass the argument **arg** to the thread entry subroutine **ent** as its actual argument. The argument **flag** is used to inform the system of the property of the argument **arg**. When the execution returns from the entry subroutine **ent**, the new thread will terminate automatically.

If subroutine **ent** was declared such that an explicit interface would be required if it was called directly, then an explicit interface is also required when it is passed as an argument to this function.

The argument **arg** is a generic argument that can be of any type and any rank. The actual argument **arg** must be a variable, and consequently eligible as a left- value in an assignment statement. If you pass an array section with vector subscripts to the argument **arg**, the result is unpredictable.

If the actual argument **arg** is an array section, the corresponding dummy argument in subroutine **ent** must be an assumed-shape array. Otherwise, the result is unpredictable.

If the actual argument **arg** has the pointer attribute that points to an array or array section, the corresponding dummy argument in subroutine **ent** must have a pointer attribute or be an assumed-shape array. Otherwise, the result is unpredictable.

Class

Function

Argument Type and Attributes

thread TYPE(f_pthread_t), INTENT(OUT)

On successful completion of the function, **f_pthread_create** stores the ID of the created thread in the **thread**.

attr TYPE(f_pthread_attr_t), INTENT(IN)

flag INTEGER(4), INTENT(IN)

The argument **flag** must convey the property of the argument **arg** exactly to the system. The argument **flag** can be one of, or a combination of, the following constants:

FLAG_CHARACTER:
if the entry subroutine **ent** expects an argument of type

CHARACTER in any way or any form, this flag value must be included to indicate this fact. However, if the subroutine expects a Fortran 90 pointer pointing to an argument of type **CHARACTER**, the **FLAG_DEFAULT** value should be included instead.

FLAG_ASSUMED_SHAPE:
: if the entry subroutine **ent** has a dummy argument which is an assumed-shape array of any rank, this flag value must be included to indicate this fact.

FLAG_DEFAULT:
: otherwise, this flag value is needed.

ent A subroutine that has one dummy argument of any type, kind and rank.

arg A generic argument of any type, kind, and rank. It is passed to **ent** as the only actual argument.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EAGAIN
The system does not have enough resources to create a new thread.

EINVAL
The argument **thread**, **attr**, or **flag** is invalid.

ENOMEM
The system does not have sufficient memory to create a new thread.

f_pthread_detach(thread)

Purpose

This function is used to indicate to the pthreads library implementation that storage for the thread whose thread ID is specified by the argument thread can be claimed when this thread terminates. If the thread has not yet terminated, **f_pthread_detach** shall not cause it to terminate. Multiple **f_pthread_detach** calls on the same target thread cause an error.

Class

Function

Argument Type and Attributes

thread TYPE(f_pthread_t), INTENT(IN)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

ESRCH
: The argument thread is invalid.

f_pthread_equal(thread1, thread2)

Purpose

This function can be used to compare whether two thread ID's identify the same thread or not.

Class

Function

Argument Type and Attributes

thread1
: TYPE(f_pthread_t), INTENT(IN)

thread2
: TYPE(f_pthread_t), INTENT(IN)

Result Type and Attributes

LOGICAL(4)

Result Value

TRUE The two thread ID's identify the same thread.

FALSE
: The two thread ID's do not identify the same thread.

f_pthread_exit(ret)

Purpose

This subroutine can be called explicitly to terminate the calling thread before it returns from the entry subroutine. The actions taken depend on the state of the calling thread. If it is in non-detached state, the calling thread will wait to be joined. If the thread is in detached state, or when it is joined by another thread, the calling thread will terminate safely. First, the cleanup stack will be popped and executed, and then any thread-specific data will be destructed by the destructors. Finally, the thread resources are freed and the argument **ret** will be returned to the joining threads. The argument **ret** of this subroutine is optional. Currently, argument **ret** is limited to be an Integer pointer. If it is not an Integer pointer, the behavior is undefined. Calling **f_pthread_exit** will not automatically free all of the memory allocated to a thread. To avoid memory leaks, finalization must be handled separately from **f_pthread_exit**.

This subroutine never returns. If argument **ret** is not provided, NULL will be provided as this thread's exit status.

Class

Subroutine

Argument Type and Attributes

ret Integer pointer, OPTIONAL, INTENT(IN)

Result Type and Attributes

None

Result Value

None

f_pthread_getconcurrency()

Purpose

This function returns the value of the concurrency level set by a previous call to the **f_pthread_setconcurrency** function. If the **f_pthread_setconcurrency** function was not previously called, this function returns zero to indicate that the system is maintaining the concurrency level.

Class

Function

Argument Type and Attributes

None

Result Type and Attributes

INTEGER(4)

Result Value

This function returns the value of the concurrency level set by a previous call to the **f_pthread_setconcurrency** function. If the **f_pthread_setconcurrency** function was not previously called, this function returns 0.

f_pthread_getschedparam(thread, policy, param)

Purpose

This function can be used to query the current setting of the scheduling property of the target thread. The target thread is identified by argument **thread**. Its scheduling policy will be returned through argument **policy** and its scheduling property through argument **param**. The sched_priority field in **param** defines the scheduling priority. The priority field will assume a value in the range of 1-127, where *127* is the most favored scheduling priority while *1* is the least.

Class

Function

Argument Type and Attributes

thread	TYPE(f_pthread_t), INTENT(IN)
policy	INTEGER(4), INTENT(OUT)
param	TYPE(f_sched_param), INTENT(OUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

ESRCH
The target thread is invalid or has already terminated.

EFAULT
The **policy** or **param** points are outside the process memory space

f_pthread_getspecific(key, arg)

Purpose

This function can be used to retrieve the thread-specific data associated with **key**. Note that the argument **arg** is not optional in this function as it will return the thread-specific data. After execution of the procedure, the argument **arg** holds a pointer to the data, or **NULL** if there is no data to retrieve. The argument **arg** must be an Integer pointer, or the result is undefined.

The actual argument **arg** must be a variable, and consequently eligible as a left-value in an assignment statement. If you pass an array section with vector subscripts to the argument **arg**, the result is unpredictable.

Class

Function

Argument Type and Attributes

key	TYPE(f_pthread_key_t), INTENT(IN)
arg	Integer pointer, INTENT(OUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument **key** is invalid.

f_pthread_join(thread, ret)

Purpose

This function can be called to join a particular thread designated by the argument **thread**. If the target thread is in non-detached state and is already terminated, this call will return immediately with the target thread's status returned in argument **ret** if it is provided. The argument **ret** is optional. Currently, **ret** must be an Integer pointer if it is provided.

If the target thread is in detached state, it is an error to join it.

Class

Function

Argument Type and Attributes

thread	TYPE(f_pthread_t), INTENT(IN)
ret	Integer pointer, INTENT(OUT), OPTIONAL

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EDEADLK
This call will cause a deadlock, or the calling thread is trying to join itself.

EINVAL
The argument **thread** is invalid.

ESRCH
The argument **thread** designates a thread which does not exist or is in detached state.

f_pthread_key_create(key, dtr)

Purpose

This function can be used to acquire a thread-specific data key. The key will be returned in the argument **key**. The argument **dtr** is a subroutine that will be used to destruct the thread-specific data associated with this key when any thread terminates after this calling point. The destructor will receive the thread-specific data as its argument. The destructor itself is optional. If it is not provided, the system will not invoke any destructor on the thread-specific data associated with this key. Note that the number of thread-specific data keys is limited in each process. It is the user's responsibility to manage the usage of the keys. The per-process limit can be checked by the Fortran constant **PTHREAD_DATAKEYS_MAX**.

Class

Function

Argument Type and Attributes

key TYPE(f_pthread_key_t), INTENT(OUT)

dtr External, optional subroutine

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EAGAIN
: The maximum number of keys has been exceeded.

EINVAL
: The argument **key** is invalid.

ENOMEM
: There is insufficient memory to create this key.

f_pthread_key_delete(key)

Purpose

This function will destroy the thread-specific data key identified by the argument **key**. It is the user's responsibility to ensure that there is no thread-specific data associated with this key. This function does not call any destructor on the thread's behalf. After the key is destroyed, it can be reused by the system for **f_pthread_key_create** requests.

Class

Function

Argument Type and Attributes

key TYPE(f_pthread_key_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EINVAL
: The argument **key** is invalid.

EBUSY
: There is still data associated with this key.

f_pthread_key_t

Purpose

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the POSIX **pthread_key_t**, which is the type of key object for accessing thread-specific data.

Class

Data Type

f_pthread_kill(thread, sig)

Purpose

This function can be used to send a signal to a target thread. The target thread is identified by argument **thread**. The signal which will be sent to the target thread is identified in argument **sig**. If **sig** contains value zero, error checking will be done by the system but no signal will be sent.

Class

Function

Argument Type and Attributes

thread	TYPE(f_pthread_t), INTENT(INOUT)
sig	INTEGER(4), INTENT(IN)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EINVAL
    The argument **thread** or **sig** is invalid.

ESRCH
    The target thread does not exist.

f_pthread_mutex_destroy(mutex)

Purpose

This function should be called to destroy those mutex objects that are no longer required. In this way, the system can recollect the memory resources. The target mutex object is identified by the argument **mutex**.

Class

Function

Argument Type and Attributes

mutex TYPE(f_pthread_mutex_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EBUSY
The target mutex is locked or referenced by another thread.

EINVAL
The argument **mutex** is invalid.

f_pthread_mutex_init(mutex, mattr)

Purpose

This function can be used to initialize the mutex object identified by argument **mutex**. The initialized mutex will assume attributes set in the mutex attribute object **mattr**, if it is provided. If **mattr** is not provided, the system will initialize the mutex to have default attributes. After it is initialized, the mutex object can be used to synchronize accesses to critical data or code. It can also be used to build more complicated thread synchronization objects.

Another method to initialize mutex objects is to statically initialize them through the Fortran constant **PTHREAD_MUTEX_INITIALIZER**. If this method of initialization is used it is not necessary to call the function before using the mutex objects.

Class

Function

Argument Type and Attributes

mutex TYPE(f_pthread_mutex_t), INTENT(OUT)

mattr TYPE(f_pthread_mutexattr_t), INTENT(IN), OPTIONAL

Result Type and Attributes

INTEGER(4)

Result Value

This function always returns 0.

f_pthread_mutex_lock(mutex)

Purpose

This function can be used to acquire ownership of the mutex object. (In other words, the function will lock the mutex.) If the mutex has already been locked by another thread, the caller will wait until the mutex is unlocked. If the mutex is already locked by the caller itself, an error will be returned to prevent recursive locking.

Class

Function

Argument Type and Attributes

mutex TYPE(f_pthread_mutex_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EDEADLK
The mutex is locked by the calling thread already.

EINVAL
The argument **mutex** is invalid.

f_pthread_mutex_t

Purpose

A derived data type whose components are all private. Any object of this type should be manipulated through the appropriate interfaces provided in this module. In addition, objects of this type can be initialized statically through the Fortran constant **PTHREAD_MUTEX_INITIALIZER**.

This data type corresponds to the POSIX **pthread_mutex_t**, which is the type of mutex object.

Class

Data Type

f_pthread_mutex_trylock(mutex)

Purpose

This function can be used to acquire ownership of the mutex object. (In other words, the function will lock the mutex.) If the mutex has already been locked by another thread, the function returns the error code **EBUSY**. The calling thread can check the return code to take further actions. If the mutex is already locked by the caller itself, an error will be returned to prevent recursive locking.

Class

Function

Argument Type and Attributes

mutex TYPE(f_pthread_mutex_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EBUSY
The target mutex is locked or referenced by another thread.

EINVAL
The argument **mutex** is invalid.

f_pthread_mutex_unlock(mutex)

Purpose

This function releases the mutex object's ownership in order to allow other threads to lock the mutex.

Class

Function

Argument Type and Attributes

mutex TYPE(f_pthread_mutex_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EINVAL
The argument **mutex** is invalid.

EPERM
The mutex is not locked by the calling thread.

f_pthread_mutexattr_destroy(mattr)

Purpose

This function can be used to destroy a mutex attribute object that has been initialized previously. Allocated memory will then be recollected. A mutex created with this attribute will not be affected by this action.

Class

Function

Argument Type and Attributes

mattr TYPE(f_pthread_mutexattr_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

This function always returns 0.

f_pthread_mutexattr_getpshared(mattr, pshared)

Purpose

This function is used to query the process-shared attribute in the mutex attributes object identified by the argument **mattr**. The current setting of the attribute will be returned through the argument **pshared**.

Class

Function

Argument Type and Attributes

mattr TYPE(f_pthread_mutexattr_t), INTENT(IN)

pshared
INTEGER(4), INTENT(IN)

On return from this function, **pshared** contains one of the following values:

PTHREAD_PROCESS_SHARED
The mutex can be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes.

PTHREAD_PROCESS_PRIVATE
The mutex will only be operated upon by threads created within the same process as the thread that initialized the mutex.

Result Type and Attributes

INTEGER(4)

Result Value

If this function completes successfully, value 0 is returned and the value of the process-shared attribute is returned through the argument **pshared**. Otherwise, the following error will be returned:

EINVAL
The argument **mattr** is invalid.

f_pthread_mutexattr_gettype(mattr, type)

Purpose

This function is used to query the mutex type attribute in the mutex attributes object identified by the argument **mattr**.

If this function completes successfully, value 0 is returned and the type attribute will be returned through the argument type.

Class

Function

Argument Type and Attributes

mattr TYPE(f_pthread_mutexattr_t), INTENT(IN)

type INTEGER(4), INTENT(OUT)

On return from this function, **type** contains one of the following values:

PTHREAD_MUTEX_NORMAL
This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behavior.

PTHREAD_MUTEX_ERRORCHECK
This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it will return with an error. A thread attempting to unlock a mutex which another thread has locked will return an error. A thread attempting to unlock an unlocked mutex will return with an error.

PTHREAD_MUTEX_RECURSIVE
A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex. The relocking deadlock that can occur with mutexes of type **PTHREAD_MUTEX_NORMAL** cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument is invalid.

f_pthread_mutexattr_init(mattr)

Purpose

This function can be used to initialize a mutex attribute object before it can be used in any other way. The mutex attribute object will be returned through argument **mattr**.

Class

Function

Argument Type and Attributes

mattr TYPE(f_pthread_mutexattr_t), INTENT(OUT)

Result Type and Attributes

INTEGER(4)

Result Value

This function returns 0.

f_pthread_mutexattr_setpshared(mattr, pshared)

Purpose

This function is used to set the process-shared attribute of the mutex attributes object identified by the argument **mattr**.

Class

Function

Argument Type and Attributes

mattr TYPE(f_pthread_mutexattr_t), INTENT(INOUT)

pshared
INTEGER(4), INTENT(IN)

Must contain one of the following values:

PTHREAD_PROCESS_SHARED
Specifies the mutex can be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes.

PTHREAD_PROCESS_PRIVATE
Specifies the mutex will only be operated upon by threads created within the same process as the thread that initialized the mutex. This is the default setting of the attribute.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument is invalid.

f_pthread_mutexattr_settype(mattr, type)

Purpose

This function is used to set the mutex type attribute in the mutex attributes object identified by the argument **mattr** The argument type identifies the mutex type attribute to be set.

Class

Function

Argument Type and Attributes

mattr TYPE(f_pthread_mutexattr_t), INTENT(INOUT)

type INTEGER(4), INTENT(IN)

Must contain one of the following values:

PTHREAD_MUTEX_NORMAL
This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behavior.

PTHREAD_MUTEX_ERRORCHECK
This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it will return with an error. A thread attempting to unlock a mutex which another thread has locked will return an error. A thread attempting to unlock an unlocked mutex will return with an error.

PTHREAD_MUTEX_RECURSIVE
A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex. The relocking deadlock that can occur with mutexes of type **PTHREAD_MUTEX_NORMAL** cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex.

PTHREAD_MUTEX_DEFAULT
The same as **PTHREAD_MUTEX_NORMAL.**

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
: One of the arguments is invalid.

f_pthread_mutexattr_t

Purpose

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the POSIX **pthread_mutexattr_t**, which is the type of mutex attribute object.

Class

Data Type

f_pthread_once(once, initr)

Purpose

This function can be used to initialize those data required to be initialized only once. The first thread calling this function will call **initr** to do the initialization. Other threads calling this function afterwards will have no effect. Argument **initr** must be a subroutine without dummy arguments.

Class

Function

Argument Type and Attributes

once	TYPE(f_pthread_once_t), INTENT(INOUT)
initr	A subroutine that has no dummy arguments.

Result Type and Attributes

INTEGER(4)

Result Value

This function returns 0.

f_pthread_once_t

Purpose

A derived data type whose components are all private. Any object of this type should be manipulated through the appropriate interfaces provided in this module. However, objects of this type can *only* be initialized through the Fortran constant **PTHREAD_ONCE_INIT**.

Class

Data Type

f_pthread_rwlock_destroy(rwlock)

Purpose

This function destroys the read-write lock object specified by the argument **rwlock** and releases any resources used by the lock.

Class

Function

Argument Type and Attributes

rwlock
TYPE(f_pthread_rwlock_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EBUSY
The target read-write object is locked.

f_pthread_rwlock_init(rwlock, rwattr)

Purpose

This function initializes the read-write lock object specified by **rwlock** with the attribute specified by the argument **rwattr**. If the optional argument **rwattr** is not provided, the system will initialize the read-write lock object with the default attributes. After it is initialized, the lock can be used to synchronize access to critical data. With a read-write lock, many threads can have simultaneous read-only access to data, while only one thread can have write access at any given time and no other readers or writers are allowed.

Another method to initialize read-write lock objects is to statically initialize them through the Fortran constant **PTHREAD_RWLOCK_INITIALIZER.** If this method of initialization is used, it is not necessary to call this function before using the read-write lock objects.

Class

Function

Argument Type and Attributes

rwlock
TYPE(f_pthread_rwlock_t), INTENT(OUT)

rwattr TYPE(f_pthread_rwlockattr_t), INTENT(IN), OPTIONAL

Result Type and Attributes

INTEGER(4)

Result Value

This function returns 0.

f_pthread_rwlock_rdlock(rwlock)

Purpose

This function applies a read lock to the read-write lock specified by the argument **rwlock**. The calling thread acquires the read lock if a writer does not hold the lock and there are no writes blocked on the lock. Otherwise, the calling thread will not acquire the read lock. If the read lock is not acquired, the calling thread blocks (that is, it does not return from the **f_pthread_rwlock_rdlock** call) until it can acquire the lock. Results are undefined if the calling thread holds a write lock on **rwlock** at the time the call is made. A thread may hold multiple concurrent read locks on **rwlock** (that is, successfully call the **f_pthread_rwlock_rdlock** function *n* times). If so, the thread must perform matching unlocks (that is, it must call the **f_pthread_rwlock_unlock** function *n* times).

Class

Function

Argument Type and Attributes

rwlock
: TYPE(f_pthread_rwlock_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EAGAIN
: The read-write lock could not be acquired because the maximum number of read locks for rwlock has been exceeded.

EINVAL
: The argument rwlock does not refer to an initialized read-write lock object.

f_pthread_rwlock_t

Purpose

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module. In addition, objects of this type can be initialized statically through the Fortran constant **PTHREAD_RWLOCK_INITIALIZER**.

Class

Data Type

f_pthread_rwlock_tryrdlock(rwlock)

Purpose

This function applies a read lock like the **f_pthread_rwlock_rdlock** function with the exception that the function fails if any thread holds a write lock on **rwlock** or there are writers blocked on **rwlock**. In that case, the function returns **EBUSY**. The calling thread can check the return code to take further actions.

Class

Function

Argument Type and Attributes

rwlock
: TYPE(f_pthread_rwlock_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

This function returns zero if the lock for reading on the read-write lock object specified by **rwlock** is acquired. Otherwise, the following error will be returned:

EBUSY
: The read-write lock could not be acquired for reading because a writer holds the lock or was blocked on it.

f_pthread_rwlock_trywrlock(rwlock)

Purpose

This function applies a write lock like the **f_pthread_rwlock_wrlock** function with the exception that the function fails if any thread currently holds **rwlock** (for reading or writing). In that case, the function returns **EBUSY**. The calling thread can check the return code to take further actions.

Class

Function

Argument Type and Attributes

rwlock
: TYPE(f_pthread_rwlock_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

This function returns zero if the lock for writing on the read-write lock object specified by **rwlock** is acquired. Otherwise, the following error will be returned:

EBUSY
: The read-write lock could not be acquired for writing because it is already locked for reading or writing.

f_pthread_rwlock_unlock(rwlock)

Purpose

This function is used to release a lock held on the read-write lock object specified by the argument *rwlock*. If this function is called to release a read lock from the read-write lock object and there are other read locks currently held on this read-write lock object, the read-write lock object remains in the read locked state. If this function releases the calling thread's last read lock on this read-write lock object, then the calling thread is no longer one of the owners of the object. If this function releases the last read lock for this read-write lock object, the read-write lock object will be put in the unlocked state with no owners.

Class

Function

Argument Type and Attributes

rwlock
: TYPE(f_pthread_rwlock_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EPERM
: The current thread does not own the read-write lock.

f_pthread_rwlock_wrlock(rwlock)

Purpose

This function applies a write lock to the read-write lock specified by the argument *rwlock*. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock *rwlock*. Otherwise, the thread blocks (that is, does not return from the **f_pthread_rwlock_wrlock** call) until it acquires the lock. Results are undefined if the calling thread holds the read-write lock (whether a read or write lock) at the time the call is made.

Class

Function

Argument Type and Attributes

rwlock
: TYPE(f_pthread_rwlock_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument rwlock does not refer to an initialized read-write lock object.

f_pthread_rwlockattr_destroy(rwattr)

Purpose

This function destroys a read-write lock attributes object specified by the argument **rwattr** which has been initialized previously. A read-write lock created with this attribute will not be affected by the action.

Class

Function

Argument Type and Attributes

rwattr TYPE(f_pthread_rwlockattr_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument **rwattr** is invalid.

f_pthread_rwlockattr_getpshared(rwattr, pshared)

Purpose

This function is used to obtain the value of the process-shared attribute from the initialized read-write lock attributes object specified by the argument **rwattr**. The current setting of this attribute will be returned in the argument **pshared**.

Class

Function

Argument Type and Attributes

rwattr TYPE(f_pthread_rwlockattr_t), INTENT(IN)

pshared
INTEGER(4), INTENT(OUT)

On return from this function, the value of **pshared** will be one of the following:

PTHREAD_PROCESS_SHARED
: The read-write lock can be operated upon by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.

PTHREAD_PROCESS_PRIVATE
: The read-write lock shall only be used by threads within the same process as the thread that created it.

Result Type and Attributes

INTEGER(4)

Result Value

If this function completes successfully, value 0 is returned and the value of the process-shared attribute of **rwattr** is stored into the object specified by the argument **pshared**. Otherwise, the following error will be returned:

EINVAL
: The argument **rwattr** is invalid.

f_pthread_rwlockattr_init(rwattr)

Purpose

This function initializes a read-write lock attributes object specified by *rwattr* with the default value for all of the attributes.

Class

Function

Argument Type and Attributes

rwattr TYPE(f_pthread_rwlockattr_t), INTENT(OUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

ENOMEM
: There is insufficient memory to initialize the read-write lock attributes object.

f_pthread_rwlockattr_setpshared(rwattr, pshared)

Purpose

This function is used to set the process-shared attribute in an initialized read-write lock attributes object specified by the argument **rwattr**, based on the value provided by the argument **pshared**.

Class

Function

Argument Type and Attributes

rwattr TYPE(f_pthread_rwlockattr_t), INTENT(INOUT)

pshared
INTEGER(4), INTENT(IN)

Must be one of the following:

PTHREAD_PROCESS_SHARED
Specifies the read-write lock can be operated upon by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.

PTHREAD_PROCESS_PRIVATE
Specifies the read-write lock shall only be used by threads within the same process as the thread that created it. This is the default setting of the attribute.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors:

EINVAL
The argument **rwattr** is invalid.

ENOSYS
The value of **pshared** is equal to **pthread_process_shared**.

f_pthread_rwlockattr_t

Purpose

This is a derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

Class

Data Type

f_pthread_self()

Purpose

This function can be used to return the thread ID of the calling thread.

Class

Function

Argument Type and Attributes

None

Result Type and Attributes

TYPE(f_pthread_t)

Result Value

The calling thread's ID is returned.

f_pthread_setcancelstate(state, oldstate)

Purpose

This function can be used to set the thread's cancelability state. The new state will be set according to the argument **state**. The old state will be returned in the argument **oldstate**.

Class

Function

Argument Type and Attributes

state INTEGER(4), INTENT(IN)

Must contain one of the following:

PTHREAD_CANCEL_DISABLE:
The thread's cancelability is disabled.

PTHREAD_CANCEL_ENABLE:
The thread's cancelability is enabled.

oldstate
INTEGER(4), INTENT(OUT)

On return from this function, **oldstate** will contain one of the following values:

PTHREAD_CANCEL_DISABLE:
The thread's cancelability is disabled.

PTHREAD_CANCEL_ENABLE:
The thread's cancelability is enabled.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
: The argument **state** is invalid.

f_pthread_setcanceltype(type, oldtype)

Purpose

This function can be used to set the thread's cancelability type. The new type will be set according to the argument **type**. The old type will be returned in argument **oldtype**.

Class

Function

Argument Type and Attributes

type INTEGER(4), INTENT(IN)

Must contain one of the following values:

PTHREAD_CANCEL_DEFERRED:
Cancelation request will be delayed until a cancelation point.

PTHREAD_CANCEL_ASYNCHRONOUS:
Cancelation request will be acted upon immediately. This may cause unexpected results.

oldtype
INTEGER(4), INTENT(OUT)

On return from this procedure, **oldtype** will contain one of the following values:

PTHREAD_CANCEL_DEFERRED:
Cancelation request will be delayed until a cancelation point.

PTHREAD_CANCEL_ASYNCHRONOUS:
Cancelation request will be acted upon immediately. This may cause unexpected results.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument **type** is invalid.

f_pthread_setconcurrency(new_level)

Purpose

This function is used to inform the pthreads library implementation of desired concurrency level as specified by the argument *new_level*. The actual level of concurrency provided by the implementation as a result of this function call is unspecified.

Class

Function

Argument Type and Attributes

new_level
INTEGER(4), INTENT(IN)

Result Type and Attributes

INTEGER(4)

Result Value

This function returns 0.

f_pthread_setschedparam(thread, policy, param)

Purpose

This function can be used to dynamically set the scheduling policy and the scheduling property of a thread. The target thread is identified by argument **thread**. The new scheduling policy for the target thread is provided through argument **policy**. The new scheduling property of the target thread will be set to the value provided by argument **param**. The sched_priority field in **param** defines the scheduling priority. Its range is 1-127.

Class

Function

Argument Type and Attributes

thread	TYPE(f_pthread_t), INTENT(INOUT)
policy	INTEGER(4), INTENT(IN)
param	TYPE(f_sched_param), INTENT(IN)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors

ENOSYS
The POSIX priority scheduling option is not implemented on Blue Gene/Q.

ENOTSUP
The value of argument **policy** or **param** is not supported.

EPERM
The target thread is not permitted to perform the operation or is in a mutex protocol already.

ESRCH
The target thread does not exist or is invalid.

f_pthread_setspecific(key, arg)

Purpose

This function can be used to set the calling thread's specific data associated with the key identified by argument **key**. The argument **arg**, which is optional, identifies the thread-specific data to be set. If **arg** is not provided, the thread-specific data will be set to NULL, which is the initial value for each thread. Only an Integer pointer can be passed as the **arg** argument. If **arg** is not an Integer pointer, the result is undefined.

The actual argument **arg** must be a variable, and consequently eligible as a left-value in an assignment statement. If you pass an array section with vector subscripts to the argument **arg**, the result is unpredictable.

Class

Function

Argument Type and Attributes

key	TYPE(f_pthread_key_t), INTENT(IN)
arg	Integer pointer, INTENT(IN), OPTIONAL

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors

EINVAL
The argument **key** is invalid.

ENOMEM
There is insufficient memory to associate the data with the key.

f_pthread_t

Purpose

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the POSIX **pthread_t**, which is the type of thread object.

Class

Data Type

f_pthread_testcancel()

Purpose

This subroutine provides a cancelation point in a thread. When it is called, any pending cancelation request will be acted upon immediately if it is in the enabled state.

Class

Subroutine

Argument Type and Attributes

None

Result Type and Attributes

None

f_sched_param

Purpose

This data type corresponds to the Blue Gene/Q system data structure **sched_param**, which is a system data type.

This is a public data structure defined as:

```
type f_sched_param
  sequence
  integer sched_priority
end type f_sched_param
```

Class

Data Type

f_sched_yield()

Purpose

This function is used to force the calling thread to relinquish the processor until it again becomes the head of its thread list.

Class

Function

Argument Type and Attributes

None.

Result Type and Attributes

INTEGER(4)

Result Value

If this function completes successfully, value 0 is returned. Otherwise, a value of -1 will be returned.

f_timespec

Purpose

This is a Fortran definition of the Blue Gene/Q system data structure **timespec**. Within the Fortran Pthreads module, objects of this type are used to specify an absolute date and time. This *deadline absolute date* is used when waiting on a POSIX condition variable.

In 64–bit mode, **f_timespec** is defined as:

```
TYPE F_Timespec
    SEQUENCE
    INTEGER(4) tv_sec
    INTEGER(4) pad
    INTEGER(KIND=REGISTER_SIZE) tv_nsec
END TYPE F_Timespec
```

Class

Data Type

Chapter 8. Interlanguage calls

Your Fortran application can perform interlanguage calls to routines written in a language other than Fortran.

The guidelines assume that you are familiar with the syntax of all applicable languages.

Conventions for XL Fortran external names

To assist you in writing mixed-language programs, XL Fortran follows a consistent set of rules when translating the name of a global entity into an external name that the linker can resolve.

The rules are:

- Both the underscore (_) and the dollar sign ($) are valid characters anywhere in names.

  Because names that begin with an underscore are reserved for the names of library routines, do not use an underscore as the first character of a Fortran external name.

  To avoid conflicts between Fortran and non-Fortran function names, you can compile the Fortran program with the **-qextname** option. This option adds an underscore to the end of the Fortran names. Then use an underscore as the last character of any non-Fortran procedures that you want to call from Fortran.
- Names can be up to 250 characters long.
- Program and symbolic names are interpreted as all lowercase by default. If you are writing new non-Fortran code, use all-lowercase procedure names to simplify calling the procedures from Fortran.

  You can use the **-U** option or the **@PROCESS MIXED** directive if you want the names to use both uppercase and lowercase:

  ```
  @process mixed
         external C_Func    ! With MIXED, we can call C_Func, not just c_func.
         integer aBc, ABC   ! With MIXED, these are different variables.
         common /xYz/ aBc   ! The same applies to the common block names.
         common /XYZ/ ABC   ! xYz and XYZ are external names that are
                            ! visible during linking.
         end
  ```
- Names for module procedures are formed by concatenating __ (two underscores), the module name, `_IMOD_` (for intrinsic modules) or `_NMOD_` (for non-intrinsic modules), and the name of the module procedure. For example, module procedure `MYPROC` in module `MYMOD` has the external name `__mymod_NMOD_myproc`.

  Note: Symbolic debuggers and other tools should account for this naming scheme when debugging XL Fortran programs that contain module procedures. For example, some debuggers default to lowercase for program and symbolic names. This behavior should be changed to use mixed case when debugging XL Fortran programs with module procedures.
- The XL compilers generate code that uses `main` as an external entry point name. You can only use `main` as an external name in these contexts:

- A Fortran program or local-variable name. (This restriction means that you cannot use `main` as a binding label, or for the name of an external function, external subroutine, block data program unit, or common block. References to such an object use the compiler-generated `main` instead of your own.)
- The name of the top-level main function in a C program.

- Some other potential naming conflicts may occur when linking a program. For instructions on avoiding them, see Avoiding naming conflicts during linking in the *XL Fortran Compiler Reference*.

If you are porting your application from another system and your application does encounter naming conflicts like these, you may need to use the **-qextname** option.

Mixed-language input and output

To improve performance, the XL Fortran runtime library has its own buffers and its own handling of these buffers. This means that mixed-language programs cannot freely mix I/O operations on the same file from the different languages.

Mixing code compiled by multiple Fortran compilers, for example **xlf** and **gfortran**, could face similar problems. The safest approach is to treat the code compiled by another Fortran compiler as non-Fortran code. To maintain data integrity in such cases:

- If the file position is not important, open and explicitly close the file within the Fortran part of the program before performing any I/O operations on that file from subprograms written in another language.
- To open a file in Fortran and manipulate the open file from another language, call the **flush_** procedure to save any buffer for that file, and then use the **getfd** procedure to find the corresponding file descriptor and pass it to the non-Fortran subprogram.

  As an alternative to calling the **flush_** procedure, you can use the **buffering** runtime option to disable the buffering for I/O operations. When you specify **buffering=disable_preconn**, XL Fortran disables the buffering for preconnected units. When you specify **buffering=disable_all**, XL Fortran disables the buffering for all logical units.

  Note: After you call **flush_** to flush the buffer for a file, do not do anything to the file from the Fortran part of the program except to close it when the non-Fortran processing is finished.
- If any XL Fortran subprograms containing **WRITE** statements are called from a non-Fortran main program, explicitly **CLOSE** the data file, or use the **flush_** subroutine in the XL Fortran subprograms to ensure that the buffers are flushed. Alternatively, you can use the **buffering** runtime option to disable buffering for I/O operations.

For more information on the **flush_** and **getfd** procedures, see the *Service and utility procedures* topic in the *XL Fortran Language Reference*. For more information on the **buffering** runtime option, see *Setting runtime options* in the *XL Fortran Compiler Reference*.

Mixing Fortran and C++

When mixing Fortran and C++ in the same program, you need to invoke the C++ compiler to correctly link the final program.

Most of the information in this section applies to Fortran and languages with similar data types and naming schemes. However, to mix Fortran and C++ in the same program, you must add an extra level of indirection and pass the interlanguage calls through C++ wrapper functions.

Because the C++ compiler mangles the names of some C++ objects, you must use your C++ compiler's invocation command, like **bgxlC** or **g++**, to link the final program and include **-L** and **-l** options for the XL Fortran library directories and libraries.

```
      program main

      integer idim,idim1

      idim = 35
      idim1= 45

      write(6,*) 'Inside Fortran calling first C function'
      call cfun(idim)
      write(6,*) 'Inside Fortran calling second C function'
      call cfun1(idim1)
      write(6,*) 'Exiting the Fortran program'
      end
```

Figure 3. Main Fortran program that calls C++ (main1.f)

```
#include <stdio.h>
#include "cplus.h"

extern "C" void cfun(int *idim){
  printf("%%Inside C function before creating C++ Object\n");
  int i = *idim;
  junk<int>* jj= new junk<int>(10,30);
  jj->store(idim);
  jj->print();
  printf("%%Inside C function after creating C++ Object\n");
  delete jj;
  return;
}

extern "C" void cfun1(int *idim1) {
  printf("%%Inside C function cfun1 before creating C++ Object\n");
  int i = *idim1;
  temp<double> *tmp = new temp<double>(40, 50.54);
  tmp->print();
  printf("%%Inside C function after creating C++ temp object\n");
  delete tmp;
  return;
}
```

Figure 4. C++ wrapper functions for calling C++ (cfun.C)

```
#include <iostream.h>

template<class T> class junk {

  private:
   int inter;
   T   templ_mem;
   T   stor_val;

  public:
   junk(int i,T j): inter(i),templ_mem(j)
                   {cout <<"***Inside C++ constructor" << endl;}

    ~junk()         {cout <<"***Inside C++ Destructor"  << endl;}

   void store(T *val){ stor_val = *val;}

   void print(void) {cout << inter << "\t" << templ_mem ;
                    cout <<"\t" << stor_val << endl; }};

template<class T> class temp {

  private:
    int internal;
    T temp_var;

  public:
    temp(int i, T j): internal(i),temp_var(j)
                    {cout <<"***Inside C++ temp Constructor" <<endl;}

     ~temp()         {cout <<"***Inside C++ temp destructor"  <<endl;}

    void print(void) {cout << internal << "\t" << temp_var << endl;}};
```

Figure 5. C++ code called from Fortran (cplus.h)

Compiling this program, linking it with the **bgxlC** command, and running it produces the following output:

```
 Inside Fortran calling first C function
%Inside C function before creating C++ Object
***Inside C++ constructor
10      30      35
%Inside C function after creating C++ Object
***Inside C++ Destructor
 Inside Fortran calling second C function
%Inside C function cfun1 before creating C++ Object
***Inside C++ temp Constructor
40      50.54
%Inside C function after creating C++ temp object
***Inside C++ temp destructor
 Exiting the Fortran program
```

Making calls to C functions work

When you pass an argument to a subprogram call, the usual Fortran convention is to pass the address of the argument. Many C functions expect arguments to be passed as values, however, not as addresses.

For these arguments, specify them as **%VAL**(*argument*) in the call to C, or make use of the standards-compliant **VALUE** attribute. For example:

```
MEMBLK = MALLOC(1024)    ! Wrong, passes the address of the constant
MEMBLK = MALLOC(N)       ! Wrong, passes the address of the variable

MEMBLK = MALLOC(%VAL(1024)) ! Right, passes the value 1024
MEMBLK = MALLOC(%VAL(N))    ! Right, passes the value of the variable
```

See “Passing arguments by reference or by value” on page 261 and *%VAL and %REF* in the *XL Fortran Language Reference* for more details.

Passing data from one language to another

You need to account for corresponding data types in Fortran and C when passing data from one language to another.

The Corresponding data types in Fortran and C table shows the data types available in the XL Fortran and C languages. Further topics detail how Fortran arguments can be passed by reference to C programs. To use the Fortran 2003 Standard interoperability features, see the BIND attribute and ISO_C_BINDING module in the *XL Fortran Language Reference*.

Passing arguments between languages

When calling Fortran procedures, the C routines must pass arguments as pointers to the types listed in the following table.

Table 23. Corresponding data types in Fortran and C

XL Fortran Data Types	XL C/C++ Data Types
INTEGER(1), BYTE	signed char
INTEGER(2)	signed short
INTEGER(4)	signed int
INTEGER(8)	signed long long
REAL, REAL(4)	float
REAL(8), DOUBLE PRECISION	double
REAL(16)	long double (see note 1)
COMPLEX, COMPLEX(4)	float _Complex
COMPLEX(8), DOUBLE COMPLEX	double _Complex
COMPLEX(16)	long double _Complex (see note 1)
LOGICAL(1)	unsigned char
LOGICAL(2)	unsigned short
LOGICAL(4)	unsigned int
LOGICAL(8)	unsigned long long
CHARACTER	char
CHARACTER(n)	char[n]
Integer POINTER	void *
Array	array
Sequence-derived type	structure (with C/C++ -qalign=packed option)
Note: 1. Requires C/C++ compiler -qlongdbl option.	

Notes:

1. In interlanguage communication, it is often necessary to use the **%VAL** built-in function, or the standards-compliant **VALUE** attribute, and the **%REF** built-in function that are defined in “Passing arguments by reference or by value” on page 261.
2. C programs automatically convert float values to double and short integer values to integer when calling an unprototyped C function. Because XL Fortran does not perform a conversion on **REAL(4)** quantities passed by value, you should not pass **REAL(4)** and **INTEGER(2)** by value to a C function that you have not declared with an explicit interface.
3. The Fortran-derived type and the C structure must match in the number, data type, and length of subobjects to be compatible data types.

One or more sample programs under the directory `/opt/ibmcmp/xlf/bg/14.1/samples` illustrate how to call from Fortran to C.

To use the Fortran 2003 Standard interoperability features provided by XL Fortran, see the Language interoperability features section in the *XL Fortran Language Reference*.

Passing global variables between languages

To access a C data structure from within a Fortran program or to access a common block from within a C program, follow these steps:

1. Create a named common block that provides a one-to-one mapping of the C structure members. If you have an unnamed common block, change it to a named one. Name the common block with the name of the C structure.
2. Declare the C structure as a global variable by putting its declaration outside any function or inside a function with the **extern** qualifier.
3. Compile the C source file to get packed structures.

```
program cstruct                      struct mystuff {
real(8) a,d                                 double a;
integer b,c                                 int b,c;
.                                           double d;
.                                    };
common /mystuff/ a,b,c,d
.                                    main() {
.
end                                  }
```

If you do not have a specific need for a named common block, you can create a sequence-derived type with the same one-to-one mapping as a C structure and pass it as an argument to a C function. You must compile the C source file to get packed structures or put #pragmas into the struct.

Common blocks that are declared **THREADLOCAL** are thread-specific data areas that are dynamically allocated by compiler-generated code. A static block is still reserved for a **THREADLOCAL** common block, but the compiler and the compiler's runtime environment use it for control information. If you need to share **THREADLOCAL** common blocks between Fortran and C procedures, your C source must be aware of the implementation of the **THREADLOCAL** common block. For more information, see the *Directives* section in the *XL Fortran Language Reference*, and Chapter 12, “Sample Fortran programs,” on page 305.

Common blocks that are declared **THREADPRIVATE** can be accessed using a C global variable that is declared as **THREADPRIVATE**.

Passing character types between languages

One difficult aspect of interlanguage calls is passing character strings between languages. The difficulty is due to the following underlying differences in the way that different languages represent such entities:

- The only character type in Fortran is **CHARACTER**, which is stored as a set of contiguous bytes, one character per byte. The length is not stored as part of the entity. Instead, it is passed by value as an extra argument at the end of the declared argument list when the entity is passed as an argument. For the 64-bit complication mode, the size of the argument is 8 bytes.
- Character strings in C are stored as arrays of the type **char**. A null character indicates the end of the string.

  Note: To have the compiler automatically add the null character to certain character arguments, you can use the **-qnullterm** option (described in the *XL Fortran Compiler Reference*.

If you are writing both parts of the mixed-language program, you can make the C routines deal with the extra Fortran length argument, or you can suppress this extra argument by passing the string using the **%REF** function. If you use **%REF**, which you typically would for pre-existing C routines, you need to indicate where the string ends by concatenating a null character to the end of each character string that is passed to a C routine:

```
! Initialize a character string to pass to C.
              character*6 message1 /'Hello\0'/
! Initialize a character string as usual, and append the null later.
              character*5 message2 /'world'/

! Pass both strings to a C function that takes 2 (char *) arguments.
              call cfunc(%ref(message1), %ref(message2 // '\0'))
              end
```

For compatibility with C language usage, you can encode the following escape sequences in XL Fortran character strings:

Table 24. Escape sequences for character strings

Escape	Meaning
\b	Backspace
\f	Form feed
\n	New-line
\r	Carriage return
\t	Tab
\0	Null
\'	Apostrophe (does not terminate a string)
\"	Double quotation mark (does not terminate a string)
\ \	Backslash
\x	*x*, where *x* is any other character (the backslash is ignored)

If you do not want the backslash interpreted as an escape character within strings, you can compile with the **-qnoescape** option.

Passing arrays between languages

Fortran stores array elements in ascending storage units in column-major order. C stores array elements in row-major order. Fortran array indexes start at 1, while C array indexes start at 0.

The following example shows how a two-dimensional array that is declared by A(3,2) is stored in Fortran and C.

Table 25. Corresponding array layouts for Fortran and C. The Fortran array reference `A(X,Y,Z)` can be expressed in C as `a[Z-1][Y-1][X-1]`. Keep in mind that although C passes individual scalar array elements by value, it passes arrays by reference.

	Fortran Element Name	**C Element Name**
Lowest storage unit	A(1,1)	A[0][0]
	A(2,1)	A[0][1]
	A(3,1)	A[1][0]
	A(1,2)	A[1][1]
	A(2,2)	A[2][0]
Highest storage unit	A(3,2)	A[2][1]

To pass all or part of a Fortran array to another language, you can use Fortran 90/Fortran 95 array notation:

```
REAL, DIMENSION(4,8) :: A, B(10)

! Pass an entire 4 x 8 array.
CALL CFUNC( A )
! Pass only the upper-left quadrant of the array.
CALL CFUNC( A(1:2,1:4) )
! Pass an array consisting of every third element of A.
CALL CFUNC( A(1:4:3,1:8) )
! Pass a 1-dimensional array consisting of elements 1, 2, and 4 of B.
CALL CFUNC( B( (/1,2,4/) ) )
```

Where necessary, the Fortran program constructs a temporary array and copies all the elements into contiguous storage. In all cases, the C routine needs to account for the column-major layout of the array.

Any array section or noncontiguous array is passed as the address of a contiguous temporary unless an explicit interface exists where the corresponding dummy argument is declared as an assumed-shape array or a pointer. To avoid the creation of array descriptors (which are not supported for interlanguage calls) when calling non-Fortran procedures with array arguments, either do not give the non-Fortran procedures any explicit interface, or do not declare the corresponding dummy arguments as assumed-shape or pointers in the interface:

```
! This explicit interface must be changed before the C function
! can be called.
INTERFACE
  FUNCTION CFUNC (ARRAY, PTR1, PTR2)
    INTEGER, DIMENSION (:) :: ARRAY         ! Change this : to *.
    INTEGER, POINTER, DIMENSION (:) :: PTR1  ! Change this : to *
                                            ! and remove the POINTER
                                            ! attribute.
    REAL, POINTER :: PTR2                   ! Remove this POINTER
                                            ! attribute or change to TARGET.
  END FUNCTION
END INTERFACE
```

Passing pointers between languages

Integer **POINTER**s always represent the address of the pointee object and must be passed by value:

```
CALL CFUNC(%VAL(INTPTR))
```

Fortran 90 **POINTER**s can also be passed back and forth between languages but only if there is no explicit interface for the called procedure or if the argument in the explicit interface does not have a **POINTER** attribute or assumed-shape declarator. You can remove any **POINTER** attribute or change it to **TARGET** and can change any deferred-shape array declarator to be explicit-shape or assumed-size.

Because of XL Fortran's call-by-reference conventions, you must pass even scalar values from another language as the address of the value, rather than the value itself. For example, a C function passing an integer value `x` to Fortran must pass `&x`. Also, a C function passing a pointer value `p` to Fortran so that Fortran can use it as an integer **POINTER** must declare it as `void **p`. A C array is an exception: you can pass it to Fortran without the `&` operator.

Passing arguments by reference or by value

To call subprograms written in languages other than Fortran (for example, user-written C programs, or operating system routines), the actual arguments may need to be passed by a method different from the default method used by Fortran. C routines, including those in system libraries, such as **libc.so**, require you to pass arguments by value instead of by reference. (Although C passes individual scalar array elements by value, it passes arrays by reference.)

You can change the default passing method by using the **%VAL** built-in function or **VALUE** attribute and the **%REF** built-in function in the argument list of a **CALL** statement or function reference. You cannot use them in the argument lists of Fortran procedure references or with alternate return specifiers.

%REF Passes an argument by reference (that is, the called subprogram receives the address of the argument). It is the same as the default calling method for Fortran except that it also suppresses the extra length argument for character strings.

%VAL Passes an argument by value (that is, the called subprogram receives an argument that has the same value as the actual argument, but any change to this argument does not affect the actual argument).

You can use this built-in function with actual arguments that are **CHARACTER(1)**, **BYTE**, logical, integer, real, or complex expressions or that are sequence-derived type. Objects of derived type cannot contain pointers, arrays, or character structure components whose lengths are greater than one byte.

You cannot use **%VAL** with actual arguments that are array entities, procedure names, or character expressions of length greater than one byte.

%VAL causes XL Fortran to pass the actual argument as intermediate values.

64-bit intermediate values

If the actual argument is one of the following:

- An integer or a logical that is shorter than 64 bits, it is sign-extended to a 64-bit value.
- Of type real or complex, it is passed as multiple 64-bit intermediate values.
- Of sequence-derived type, it is passed as multiple 64-bit intermediate values.

Byte-named constants and variables are passed as if they were **INTEGER(1)**. If the actual argument is a **CHARACTER(1)**, the compiler pads it on the left with zeros to a 64-bit value, regardless of whether you specified the **-qctyplss** compiler option.

If you specified the **-qautodbl** compiler option, any padded storage space is not passed except for objects of derived type.

VALUE attribute

Specifies an argument association between a dummy and an actual argument that allows you to pass the dummy argument with the value of the actual argument. Changes to the value or definition status of the dummy argument do not affect the actual argument.

You must specify the **VALUE** attribute for dummy arguments only.

You must not use the **%VAL** or **%REF** built-in functions to reference a dummy argument with the **VALUE** attribute, or the associated actual argument.

A referenced procedure that has a dummy argument with the **VALUE** attribute must have an explicit interface.

You must not specify the **VALUE** attribute with the following:

- Arrays
- Derived types with **ALLOCATABLE** components
- Dummy procedures

```
  EXTERNAL FUNC
  COMPLEX XVAR
  IVARB=6

  CALL RIGHT2(%REF(FUNC))       ! procedure name passed by reference
  CALL RIGHT3(%VAL(XVAR))       ! complex argument passed by value
  CALL TPROG(%VAL(IVARB))       ! integer argument passed by value
  END
```

Explicit interface for %VAL and %REF

You can specify an explicit interface for non-Fortran procedures to avoid coding calls to **%VAL** and **%REF** in each argument list, as follows:

```
INTERFACE
    FUNCTION C_FUNC(%VAL(A),%VAL(B)) ! Now you can code "c_func(a,b)"
        INTEGER A,B                  ! instead of
    END FUNCTION C_FUNC              ! "c_func(%val(a),%val(b))".
END INTERFACE
```

Example with VALUE attribute

```
Program validexm1
  integer :: x = 10, y = 20
  print *, 'before calling: ', x, y
  call intersub(x, y)
  print *, 'after calling: ', x, y

  contains
  subroutine intersub(x,y)
```

```
    integer, value ::  x
    integer y
    x = x + y
    y = x*y
    print *, 'in subroutine after changing: ', x, y
  end subroutine
end program validexm1
```

Expected output:

```
before calling: 10 20
in subroutine after changing: 30 600
after calling: 10 600
```

Passing COMPLEX values to/from gcc

Passing COMPLEX values between Fortran and GCC depends on what is specified for the **-qfloat=[no]complexgcc** suboption. If **-qfloat=complexgcc** is specified, the compiler uses Blue Gene/Q conventions when passing or returning complex numbers. **-qfloat=nocomplexgcc** is the default.

For **-qfloat=complexgcc** in 64-bit mode, **COMPLEX***8 values are passed in 1 GPR, and **COMPLEX***16 in 2 GPRs. For **-qfloat=nocomplexgcc**, **COMPLEX***8 and **COMPLEX***16 values are passed in 2 floating-point registers (FPRs). **COMPLEX***32 values are always passed in 4 FPRs for both **-qfloat=complexgcc** and **-qfloat=nocomplexgcc** (since **gcc** does not support **COMPLEX***32).

For **-qfloat=complexgcc** in 64-bit mode, **COMPLEX***8 values are returned in GPR3, and **COMPLEX***16 in GPR 3-GPR4. For **-qfloat=nocomplexgcc**, **COMPLEX***8 and **COMPLEX***16 values are returned in FPR1-FPR2. For both **-qfloat=complexgcc** and **-qfloat=nocomplexgcc**, **COMPLEX***32 is always returned in FPR1-FPR4.

Returning values from Fortran functions

XL Fortran does not support calling certain types of Fortran functions from non-Fortran procedures. If a Fortran function returns a pointer, array, or character of nonconstant length, do not call it from outside Fortran.

You can call such a function indirectly:

```
SUBROUTINE MAT2(A,B,C)    ! You can call this subroutine from C, and the
                          ! result is stored in C.
INTEGER, DIMENSION(10,10) :: A,B,C
C = ARRAY_FUNC(A,B)       ! But you could not call ARRAY_FUNC directly.
END
```

Arguments with the OPTIONAL attribute

When you pass an optional argument by reference, the address in the argument list is zero if the argument is not present.

When you pass an optional argument by value, the value is zero if the argument is not present. The compiler uses an extra register argument to differentiate that value from a regular zero value. If the register has the value 1, the optional argument is present; if it has the value 0, the optional argument is not present.

Related information:
"Order of arguments in argument list" on page 269

Assembler-level subroutine linkage conventions

The subroutine linkage convention specifies the machine state at subroutine entry and exit, allowing routines that are compiled separately in the same or different languages to be linked.

The information on subroutine linkage and system calls in the *System V Application Binary Interface: PowerPC Processor Supplement* and *64–bit PowerPC ELF Application Binary Interface Supplement* are the base references on this topic. You should consult these for full details. This section summarizes the information needed to write mixed-language Fortran and assembler programs or to debug at the assembler level, where you need to be concerned with these kinds of low-level details.

The system linkage convention passes arguments in registers, taking full advantage of the large number of floating-point registers (FPRs), general-purpose registers (GPRs), vector registers (VPRs) and minimizing the saving and restoring of registers on subroutine entry and exit. The linkage convention allows for argument passing and return values to be in FPRs, GPRs, or both.

The following table lists floating-point registers and their functions. The floating-point registers are double precision (64 bits).

Table 26. Floating-point register usage across calls

Register	Preserved Across Calls	Use
0	no	
1	no	FP parameter 1, function return 1.
2	no	FP parameter 2, function return 2.
3	no	FP parameter 3, function return complex *32.
4	no	FP parameter 4, function return complex *32.
⋮	⋮	⋮
8	no	FP parameter 8
9-13	no	
14-31	yes	local variables

The following table lists general-purpose registers and their functions.

Table 27. General-purpose register usage across calls

Register	Preserved Across Calls	Use
0	no	
1	yes	Stack pointer.
2	yes	System-reserved.
3	no	1st word of arg list; return value 1.
4	no	2nd word of arg list; return value 2.

Table 27. General-purpose register usage across calls (continued)

Register	Preserved Across Calls	Use
5	no	3rd word of arg list
:	:	:
10	no	8th word of arg list.
11-12	no	
13	yes	SDA pointer.
14-30	no	Local variables.
31	yes	Local variables or "environment pointers".
If a register is not designated as preserved, its contents may be changed during the call, and the caller is responsible for saving any registers whose values are needed later. Conversely, if a register is supposed to be preserved, the callee is responsible for preserving its contents across the call, and the caller does not need any special action.		

The following table lists special-purpose register conventions.

Table 28. Special-purpose register usage across calls

Register	Preserved Across Calls
Condition register	
Bits 0-7 (CR0,CR1)	no
Bits 8-22 (CR2,CR3,CR4)	yes
Bits 23-31 (CR5,CR6,CR7)	no
Link register	no
Count register	no
XER register	no
FPSCR register	no

The stack

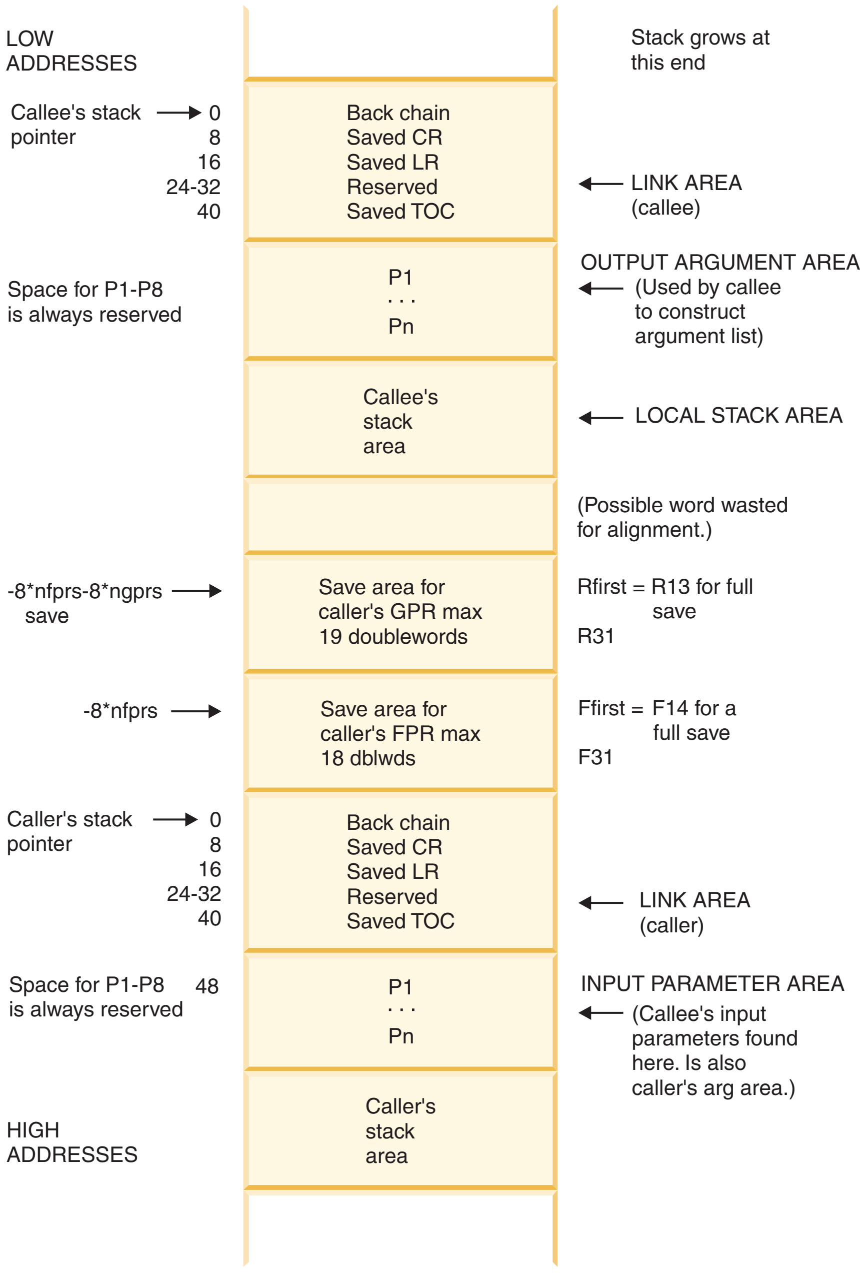
The stack is a portion of storage that is used to hold local storage, register save areas, parameter lists, and call-chain data. The stack grows from higher addresses to lower addresses. A stack pointer register (register 1) is used to mark the current "top" of the stack.

A stack frame is the portion of the stack that is used by a single procedure. The input parameters are considered part of the current stack frame. In a sense, each output argument belongs to both the caller's and the callee's stack frames. In either case, the stack frame size is best defined as the difference between the caller's stack pointer and the callee's.

The following diagram shows the storage map of typical stack frames for 64-bit environments.

In this diagram, the current routine has acquired a stack frame that allows it to call other functions. If the routine does not make any calls and there are no local variables or temporaries, and it does not need to save any non-volatile registers, the function need not allocate a stack frame. It can still use the register save area at the top of the caller's stack frame, if needed.

The stack frame is double-word aligned.



LOW ADDRESSES
Stack grows at this end
Callee's stack pointer
0
8
16
24-32
40
Back chain
Saved CR
Saved LR
Reserved
Saved TOC
LINK AREA (callee)
Space for P1-P8 is always reserved
P1
. . .
Pn
OUTPUT ARGUMENT AREA (Used by callee to construct argument list)
Callee's stack area
LOCAL STACK AREA
(Possible word wasted for alignment.)
-8*nfprs-8*ngprs save
Save area for caller's GPR max 19 doublewords
Rfirst = R13 for full save
R31
-8*nfprs
Save area for caller's FPR max 18 dblwds
Ffirst = F14 for a full save
F31
Caller's stack pointer
0
8
16
24-32
40
Back chain
Saved CR
Saved LR
Reserved
Saved TOC
LINK AREA (caller)
Space for P1-P8 is always reserved
48
P1
. . .
Pn
INPUT PARAMETER AREA (Callee's input parameters found here. Is also caller's arg area.)
HIGH ADDRESSES
Caller's stack area

Figure 6. Runtime Stack for 64-bit Environment

The Linkage Area and Minimum Stack Frame

In a 64-bit environment, the linkage area consists of six doublewords at offset zero from the caller's stack pointer on entry to a procedure. The first doubleword contains the caller's back chain (stack pointer). The second doubleword is the location where the callee saves the Condition Register (CR) if it is needed. The third doubleword is the location where the callee's prolog code saves the Link Register if it is needed. The fourth doubleword is reserved for C **SETJMP** and **LONGJMP** processing, and the fifth doubleword is reserved for future use. The last doubleword (doubleword 6) is reserved for use by the global linkage routines that are used when calling routines in other object modules (for example, in shared libraries).

The input parameter area

In a 64-bit environment, the input parameter area is a contiguous piece of storage reserved by the calling program to represent the register image of the input parameters of the callee. The input parameter area is double-word aligned and is located on the stack directly following the caller's link area. This area is at least 8 doublewords in size. If more than 8 doublewords of parameters are expected, they are stored as register images that start at positive offset 112 from the incoming stack pointer.

The first 8 doublewords only appear in registers at the call point, never in the stack. Remaining words are always in the stack, and they can also be in registers.

The register save area

In a 64-bit environment, the register save area is double-word aligned. It provides the space that is needed to save all nonvolatile FPRs and GPRs used by the callee program. The FPRs are saved next to the link area. The GPRs are saved below the FPRs (in lower addresses). The called function may save the registers here even if it does not need to allocate a new stack frame. The system-defined stack floor includes the maximum possible save area:

```
18*8 for FPRs + 19*4 for GPRs

64-bit platforms:  18*8 for FPRs + 19*8 for GPRs
```

A callee needs only to save the nonvolatile registers that it actually uses.

The local stack area

The local stack area is the space that is allocated by the callee procedure for local variables and temporaries.

The output parameter area

In a 64-bit environment, the output parameter area (P1...Pn) must be large enough to hold the largest parameter list of all procedures that the procedure that owns this stack frame calls. This area is at least 8 doublewords long, regardless of the length or existence of any argument list. If more than 8 doublewords are being passed, an extension list is constructed, which begins at offset 112 from the current stack pointer.

The first 8 doublewords only appear in registers at the call point, never in the stack. Remaining doublewords are always in the stack, and they can also be in registers.

Linkage convention for argument passing

The system linkage convention takes advantage of the large number of registers available.

The linkage convention passes arguments in both GPRs and FPRs. Two fixed lists, R3-R10 and FP1-FP13, specify the GPRs and FPRs available for argument passing.

When there are more argument words than available argument GPRs and FPRs, the remaining words are passed in storage on the stack. The values in storage are the same as if they were in registers.

In a 64-bit environment, the size of the parameter area is sufficient to contain all the arguments passed on any call statement from a procedure that is associated with the stack frame. Although not all the arguments for a particular call actually appear in storage, it is convenient to consider them as forming a list in this area, each one occupying one or more words.

For call by reference (as is the default for Fortran), the address of the argument is passed in a register. The following information refers to call by value, as in C or as in Fortran when **%VAL** is used. For purposes of their appearance in the list, arguments are classified as floating-point values or non-floating-point values:

- All non-floating-point values require one doubleword that is doubleword aligned.
- Each single-precision (**REAL(4)**) value and each double-precision (**REAL(8)**) value occupies one doubleword in the list. Each extended-precision (**REAL(16)**) value occupies two successive doublewords in the list.
- A **COMPLEX** value occupies twice as many doublewords as a **REAL** value with the same kind type parameter.
- In Fortran and C, structure values appear in successive words as they would anywhere in storage, satisfying all appropriate alignment requirements. Structures are aligned to a doubleword and occupy `(sizeof(struct X)+7)/8` doublewords, with any padding at the end. A structure that is smaller than a doubleword is left-justified within its doubleword or register. Larger structures can occupy multiple registers and may be passed partly in storage and partly in registers.
- Other aggregate values are passed "val-by-ref". That is, the compiler actually passes their address and arranges for a copy to be made in the invoked program.
- A procedure or function pointer is passed as a pointer to the routine's function descriptor; its first word contains its entry point address. (See "Pointers to functions" on page 270 for more information.)

Argument passing rules (by value)

In a 64-bit environment, from the following illustration, we state these rules:

- If the called procedure treats the parameter list as a contiguous piece of storage (for example, if the address of a parameter is taken in C), the parameter registers are stored in the space reserved for them in the stack.
- A register image is stored on the stack.
- The argument area ($P_1 ... P_n$) must be large enough to hold the largest parameter list.

Here is an example of a call to a function :

```
f(%val(l1), %val(l2), %val(l3),  %val(l4),  %val(l5),  %val(l6), %val(l7),
         %val(d1), %val(f1), %val(c1), %val(d2), %val(s1), %val(cx2))
```

where:

l denotes integer(4) (fullword integer)

d denotes real(8) (double precision)

f denotes real(4) (real)

s denotes integer(2) (halfword integer)

c denotes character (one character)

cx denotes complex(8) (double complex)

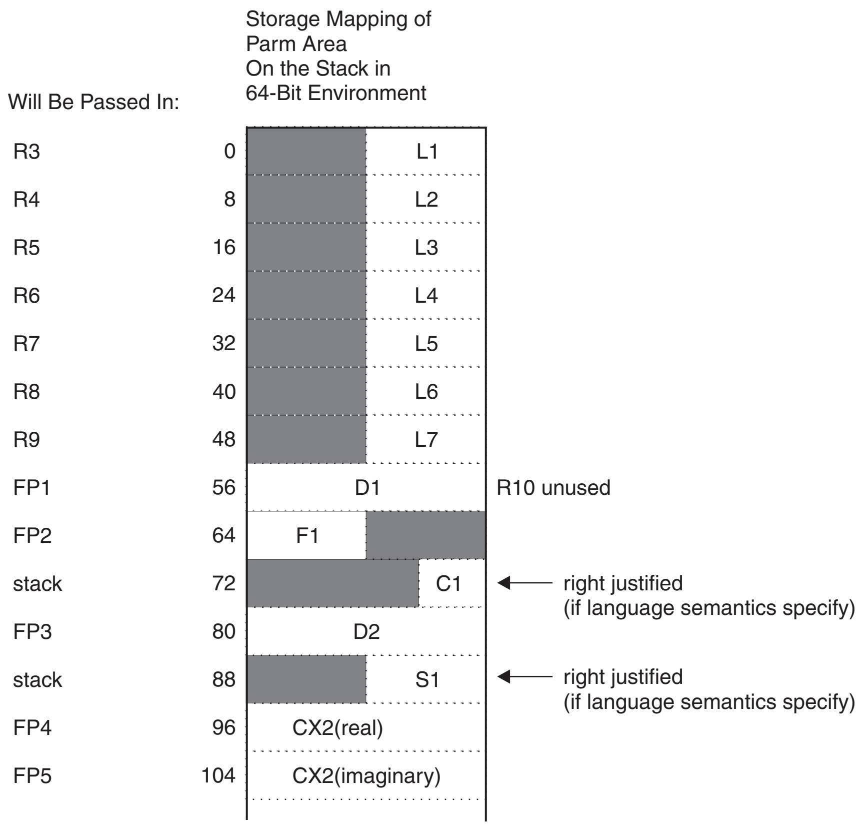


Figure 7. Storage mapping of parm area on the stack in 64-bit environment

Order of arguments in argument list

The argument list is constructed in the following order. Items in the same bullet appear in the same order as in the procedure declaration, whether or not argument keywords are used in the call.

- All addresses or values (or both) of actual arguments [1]
- "Present" indicators for optional arguments
- Length arguments for strings [1]

Linkage convention for function calls

Function calls to a routine make use of its function descriptor and entry point symbols.

In 64-bit mode, a routine has two symbols associated with it: a function descriptor (*name*) and an entry point (*.name*). When a call is made to a routine, the program branches to the entry point directly. Excluding the loading of parameters (if any) in the proper registers, compilers expand calls to functions to the following two-instruction sequence:

```
BL    .foo              # Branch to foo
ORI R0,R0,0x0000        # Special NOP
```

The linker does one of two things when it encounters a **BL** instruction:

1. There may be other items in this list during Fortran-Fortran calls. However, they will not be visible to non-Fortran procedures that follow the calling rules in this section.

1. If `foo` is imported (not in the same object module), the linker changes the **BL** to `.foo` to a **BL** to `.glink` (global linkage routine) of `foo` and inserts the `.glink` into the object module. Also, if a **NOP** instruction (`ORI R0,R0,0x0000`) immediately follows the **BL** instruction, the linker replaces the **NOP** instruction with the **LOAD** instruction `L R2, 20(R1)`.
2. If `foo` is bound in the same object module as its caller and a **LOAD** instruction `L R2,40(R1)` for 64-bit, or `ORI R0,R0,0` immediately follows the **BL** instruction, the linker replaces the **LOAD** instruction with a **NOP** (`ORI R0,R0,0`).

Note: For any export, the linker inserts the procedure's descriptor into the object module.

Pointers to functions

In 64–bit mode, a function pointer is a data type whose values range over procedure names. Variables of this type appear in several programming languages, such as C and Fortran. In Fortran, a dummy argument that appears in an **EXTERNAL** statement is a function pointer. Fortran provides support for the use of function pointers in contexts such as the target of a call statement or an actual argument of such a statement.

A function pointer is a fullword quantity that is the address of a function descriptor. The function descriptor is a 3-word object. The first word contains the address of the entry point of the procedure. The second has the address of the TOC of the object module in which the procedure is bound. The third is the environment pointer for some non-Fortran languages. There is only one function descriptor per entry point. It is bound into the same object module as the function it identifies if the function is external. The descriptor has an external name, which is the same as the function name but with a different storage class that uniquely identifies it. This descriptor name is used in all import or export operations.

In 64–bit mode, function pointers are 8 bytes long and contain a 64-bit address. For pointers to local functions, the address contained is the address of the function in the text section. For imported functions, the address is that of the function's stub. Every unique, imported function will have a stub in the object. The function stub is in the non-lazy symbol pointer section.

Function values

Functions return their values according to type:

- **INTEGER** and **LOGICAL** of kind 1, 2, and 4 are returned (right justified) in R3.
- **REAL(4)** or **REAL(8)** are returned in FP1. **REAL(16)** are returned in FP1 and FP2.
- **COMPLEX(4)** or **COMPLEX(8)** are returned in FP1 and FP2. **COMPLEX(16)** are returned in FP1-FP4.
-
- Vector results are returned in VPR2
- Character strings are returned in a buffer allocated by the caller. The address and the length of this buffer are passed in R3 and R4 as hidden parameters. The first explicit parameter word is in R5, and all subsequent parameters are moved to the next word.
- Structures are returned in a buffer that is allocated by the caller. The address is passed in R3; there is no length. The first explicit parameter is in R4.

The stack floor

In 64–bit mode, the stack floor is a system-defined address below which the stack cannot grow. All programs in the system must avoid accessing locations in the stack segment that are below the stack floor.

All programs must maintain other system invariants that are related to the stack:

- No data is saved or accessed from an address lower than the stack floor.
- The stack pointer is always valid. When the stack frame size is more than 32 767 bytes, you must take care to ensure that its value is changed in a single instruction. This step ensures that there is no timing window where a signal handler would either overlay the stack data or erroneously appear to overflow the stack segment.

Stack overflow

The linkage convention requires no explicit inline check for overflow. The operating system uses a storage protection mechanism to detect stores past the end of the stack segment.

Prolog and epilog

You need to consider a number of steps when entering a procedure and when exiting a procedure.

On entry to a procedure, you might have to do some or all of the following steps:

1. Save the link register.
2. If you use any of the CR bits 8-2319 (CR2, CR3, CR4, CR5), save the CR.
3. Save any nonvolatile FPRs that are used by this procedure in the FPR save area.
4. Save all nonvolatile VPRs that are used by this procedure in the callers VPR save area.
5. Save the VRSAVE register
6. Save all nonvolatile GPRs that are used by this procedure in the GPR save area.
7. Store back chain and decrement stack pointer by the size of the stack frame. Note that if a stack overflow occurs, it will be known immediately when the store of the back chain is done.

On exit from a procedure, you might have to perform some or all of the following steps:

1. Restore all GPRs saved.
2. Restore all VPRs saved
3. Restore the VRSAVE register
4. Restore stack pointer to the value it had on entry.
5. Restore link register if necessary.
6. Restore bits 8-2319 of the CR if necessary.
7. If you saved any FPRs, restore them.
8. Return to caller.

Traceback

In 64–bit mode, the compiler supports the traceback mechanism, which symbolic debuggers need to unravel the call or return stack. Each object module has a traceback table in the text segment at the end of its code. This table contains information about the object module, including the type of object module, as well as stack frame and register information.

Note: You can make the traceback table smaller or remove it entirely with the **-qtbtable** option.

Chapter 9. Implementation details of XL Fortran Input/Output (I/O)

This topic describes XL Fortran support (through extensions and platform-specific details) for the Blue Gene/Q file system.

See "Mixed-language input and output" on page 254 for further considerations related to input and output operations.

Implementation details of file formats

The manner in which XL Fortran implements files is based on their file format.

Sequential-access unformatted files:
An integer that contains the length of the record precedes and follows each record. For 64-bit applications, the length of the integer is 4 bytes if you set the **uwidth** runtime option to 32 (the default), and 8 bytes if you set the **uwidth** runtime option to 64.

Sequential-access formatted files:
XL Fortran programs break these files into records while reading, by using each newline character (X'0A') as a record separator.

On output, the input/output system writes a newline character at the end of each record. Programs can also write newline characters for themselves. This practice is not recommended because the effect is that the single record that appears to be written is treated as more than one record when being read or backspaced over.

Direct access files:
XL Fortran simulates direct-access files with operating system files whose length is a multiple of the record length of the XL Fortran file. You must specify, in an **OPEN** statement, the record length (**RECL**) of the direct-access file. XL Fortran uses this record length to distinguish records from each other.

For example, the third record of a direct-access file of record length 100 bytes would start at the 201st byte of the single record of a Blue Gene/Q file and end at the 300th byte.

If the length of the record of a direct-access file is greater than the total amount of data you want to write to the record, XL Fortran pads the record on the right with blanks (X'20').

Stream-access unformatted files:
Unformatted stream files are viewed as a collection of file storage units. In XL Fortran, a file storage unit is one byte.

A file connected for unformatted stream access has the following properties:

- The first file storage unit has position 1. Each subsequent file storage unit has a position that is one greater than that of the preceding one.
- For a file that can be positioned, file storage units need not be read or written in the order of their position. Any file storage unit may be read from the file while it is connected to a unit, provided that the file

storage unit has been written since the file was created, and if a READ statement for the connection is permitted.

Stream-access formatted files:

A record file connected for formatted stream access has the following properties:

- Some file storage units may represent record markers. The record marker is the newline character (X'0A').
- The file will have a record structure in addition to the stream structure.
- The record structure is inferred from the record markers that are stored in the file.
- Records can have any length up to the internal limit allowed by XL Fortran (See *XL Fortran Internal limits* in the *XL Fortran Compiler Reference*.)
- There may or may not be a record marker at the end of the file. If there is no record marker at the end of the file, the final record is incomplete, but not empty.

A file connected for formatted stream access has the following properties:

- The first file storage unit has position 1. Each subsequent file storage unit has a position that is greater than that of the preceding one. Unlike unformatted stream access, the positions of successive file storage units are not always consecutive.
- The position of a file connected for formatted stream access can be determined by the **POS=** specifier in an **INQUIRE** statement.
- For a file that can be positioned, the file position can be set to a value that was previously identified by the **POS=** specifier in **INQUIRE**.

File names

There are a number of considerations to be aware of when working with file names.

You can specify file names as either relative (such as **file**, **dir/file**, or **../file**) or absolute (such as **/file** or **/dir/file**). The maximum length of a file name (the full path name) is 4095 characters, even if you only specify a relative path name in the I/O statement. The maximum length of a file name with no path is 255 characters.

You must specify a valid file name in such places as the following:

- The **FILE=** specifier of the **OPEN** and **INQUIRE** statements
- **INCLUDE** lines

Note: To specify a file whose location depends on an environment variable, you can use the **GET_ENVIRONMENT_VARIABLE** intrinsic procedure to retrieve the value of the environment variable:

```
character(100) home, name
call get_environment_variable('HOME', value=home)
! Now home = $HOME + blank padding.
! Construct the complete path name and open the file.
name=trim(home) // '/remainder/of/path'
open (unit=10, file=name)
...
end
```

Preconnected and Implicitly Connected Files

Whether files are preconnected or implicitly connected files is dependent on their units and specific statements.

Units 0, 5, and 6 are preconnected to standard error, standard input, and standard output, respectively, before the program runs.

All other units can be implicitly connected when an **ENDFILE**, **PRINT**, **READ**, **REWIND**, or **WRITE** statement is performed on a unit that has not been opened. Unit *n* is implicitly connected to a file that is named **fort**.*n*. These files need not exist, and XL Fortran does not create them unless you use the corresponding units implicitly.

Note: Because unit 0 is preconnected for standard error, you cannot use it for the following statements: **CLOSE**, **ENDFILE**, **BACKSPACE**, **REWIND**, and direct or stream input/output. You can use it in an **OPEN** statement only to change the values of the **BLANK=**, **DELIM=**, **DECIMAL=**or **PAD=** specifiers.

You can also implicitly connect units 5 and 6 (and *) by using I/O statements that follow a **CLOSE** of these units:

```
      WRITE (6,10) "This message goes to stdout."
      CLOSE (6)
      WRITE (6,10) "This message goes in the file fort.6."
      PRINT *, "Output to * now also goes in fort.6."
10    FORMAT (A)
      END
```

The **FORM=** specifier of implicitly connected files has the value **FORMATTED** before any **READ**, **WRITE**, or **PRINT** statement is performed on the unit. The first such statement on such a file determines the **FORM=** specifier from that point on: **FORMATTED** if the formatting of the statement is format-directed, list-directed, or namelist; and **UNFORMATTED** if the statement is unformatted.

Preconnected files also have **FORM='FORMATTED'**, **STATUS='OLD'**, and **ACTION='READWRITE'** as default specifier values.

The other properties of a preconnected or implicitly connected file are the default specifier values for the **OPEN** statement. These files always use sequential access.

If you want XL Fortran to use your own file instead of the **fort.***n* file, you can either specify your file for that unit through an **OPEN** statement or create a symbolic link before running the application. In the following example, there is a symbolic link between **myfile** and **fort.10**:

```
ln -s myfile fort.10
```

When you run an application that uses the implicitly connected file **fort.10** for input/output, XL Fortran uses the file **myfile** instead. The file **fort.10** exists, but only as a symbolic link. The following command will remove the symbolic link, but will not affect the existence of **myfile**:

```
rm fort.10
```

File positioning

The position of a file pointer when a file is opened with no **POSITION=** specifier is summarized in the following table.

Table 29. Position of the file pointer when a file is opened with no POSITION= specifier

-qposition suboptions	**Implicit OPEN**		**Explicit OPEN**					
			STATUS = 'NEW'		**STATUS = 'OLD'**		**STATUS = 'UNKNOWN'**	
	File exists	**File does not exist**	**File exists**	**File does not exist**	**File exists**	**File does not exist**	**File exists**	**File does not exist**
option not specified	Start	Start	Error	Start	Start	Error	Start	Start
appendold	Start	Start	Error	Start	End	Error	Start	Start
appendunknown	Start	Start	Error	Start	Start	Error	End	Start
appendold and appendunknown	Start	Start	Error	Start	End	Error	End	Start

I/O redirection

You can use the redirection operator on the command line to redirect input to and output from your XL Fortran program.

How you specify and use this operator depends on which shell you are running. Here is a **bash** example:

```
$ cat redirect.f
        write (6,*) 'This goes to standard output'
        write (0,*) 'This goes to standard error'
        read (5,*) i
        print *,i
        end
$ bgxlf95 redirect.f
** _main    === End of Compilation 1 ===
1501-510  Compilation successful for file redirect.f.
$ # No redirection. Input comes from the terminal. Output goes to
$ # the screen.
$ a.out
 This goes to standard output
 This goes to standard error
4
 4
$ # Create an input file.
$ echo >stdin 2
$ # Redirect each standard I/O stream.
$ a.out >stdout 2>stderr <stdin
$ cat stdout
 This goes to standard output
 2
$ cat stderr
 This goes to standard error
```

Refer to your man pages for more information on redirection.

How XL Fortran I/O interacts with pipes, special files, and links

You can access regular operating system files and blocked special files by using sequential-access, direct-access, or stream-access methods.

You can only access pseudo-devices, pipes, and character special files by using sequential-access methods, or stream-access without using the **POS=** specifier.

When you use symbolic link to link files together, you can use their names interchangeably, as shown in the following example:

```
OPEN (4, FILE="file1")
OPEN (4, FILE="link_to_file1", PAD="NO") ! Modify connection
```

Do not specify the **POSITION=** specifier as **REWIND** or **APPEND** for pipes.

Do not specify **ACTION='READWRITE'** for a pipe.

Do not use the **BACKSPACE** statement on files that are pseudo-devices or character special files.

Do not use the **REWIND** statement on files that are pseudo-devices or pipes.

Default record lengths

The default record lengths for files is dependent on the file format and on the **RECL=** qualifier.

If a pseudo-device, pipe, or character special file is connected for formatted or unformatted sequential access with no **RECL=** qualifier, or for formatted stream access, the default record length is 32 768 rather than 2 147 483 647, which is the default for sequential-access files connected to random-access devices. (See the **default_recl** runtime option in the *XL Fortran Compiler Reference*.)

In certain cases, the default maximum record length for formatted files is larger, to accommodate programs that write long records to standard output. If a unit is connected to a terminal for formatted sequential access and there is no explicit **RECL=** qualifier in the **OPEN** statement, the program uses a maximum record length of 2 147 483 646 (2**31-2) bytes, rather than the usual default of 32 768 bytes. When the maximum record length is larger, formatted I/O has one restriction: **WRITE** statements that use the **T** or **TL** edit descriptors must not write more than 32 768 bytes. This is because the unit's internal buffer is flushed each 32 768 bytes, and the **T** or **TL** edit descriptors will not be able to move back past this boundary.

File permissions

A file must have the appropriate permissions (read, write, or both) for the corresponding operation being performed on it.

When a file is created, the default permissions (if the **umask** setting is 000) are both read and write for user, group, and other. You can turn off individual permission bits by changing the **umask** setting before you run the program.

Selecting error messages and recovery actions

There are various ways to control a program's behavior when errors are encountered.

By default, an XL Fortran-compiled program continues after encountering many kinds of errors, even if the statements have no **ERR=** or **IOSTAT=** specifiers. The program performs some action that might allow it to recover successfully from the bad data or other problem.

To control the behavior of a program that encounters errors, set the **XLFRTEOPTS** environment variable, which is described in *Setting runtime options* in the *XL Fortran Compiler Reference*, before running the program:

- To make the program stop when it encounters an error instead of performing a recovery action, include **err_recovery=no** in the **XLFRTEOPTS** setting.
- To make the program stop issuing messages each time it encounters an error, include **xrf_messages=no**.
- To disallow XL Fortran extensions to Fortran 90 at run time, include **langlvl=90std**. To disallow XL Fortran extensions to Fortran 95 at run time, include **langlvl=95std**. To disallow XL Fortran extensions to Fortran 2003 behavior at run time, include **langlvl=2003std**. To disallow XL Fortran extensions to Fortran 2008 behavior at run time, include **langlvl=2008std**. These settings, in conjunction with the **-qlanglvl** compiler option, can help you locate extensions when preparing to port a program to another platform.

For example:

```
# Switch defaults for some runtime settings.
XLFRTEOPTS="err_recovery=no:cnverr=no"
export XLFRTEOPTS
```

If you want a program always to work the same way, regardless of environment-variable settings, or want to change the behavior in different parts of the program, you can call the **SETRTEOPTS** procedure:

```
PROGRAM RTEOPTS
USE XLFUTILITY
CALL SETRTEOPTS("err_recovery=no") ! Change setting.
... some I/O statements ...
CALL SETRTEOPTS("err_recovery=yes") ! Change it back.
... some more I/O statements ...
END
```

Because a user can change these settings through the **XLFRTEOPTS** environment variable, be sure to use **SETRTEOPTS** to set all the runtime options that might affect the desired operation of the program.

Flushing I/O buffers

To protect data from being lost if a program ends unexpectedly, you can use the **FLUSH** statement or the **flush_** subroutine to write any buffered data to a file.

The **FLUSH** statement is recommended for better portability and is used in the following example:

```
INTEGER, PARAMETER :: UNIT = 10
DO I = 1, 1000000
    WRITE(UNIT, *) I
    CALL MIGHT_CRASH
! If the program ends in the middle of the loop, some data
! may be lost.
END DO
DO I = 1, 1000000
    WRITE(UNIT, *) I
    FLUSH(UNIT)
    CALL MIGHT_CRASH
! If the program ends in the middle of the loop, all data written
! up to that point will be safely in the file.
END DO
END
```

Related information:

"Mixed-language input and output" on page 254

See FLUSH in the Compiler Reference

Choosing locations and names for Input/Output files

If you need to override the default locations and names for input/output files, you can use the following methods without making any changes to the source code.

Naming files that are connected with no explicit name

To give a specific name to a file that would usually have a name of the form **fort.***unit*, you must set the runtime option **unit_vars** and then set an environment variable with a name of the form **XLFUNIT_***unit* for each scratch file. The association is between a unit number in the Fortran program and a path name in the file system.

For example, suppose that the Fortran program contains the following statements:

```
      OPEN (UNIT=1, FORM='FORMATTED', ACCESS='SEQUENTIAL', RECL=1024)
      ...
      OPEN (UNIT=12, FORM='UNFORMATTED', ACCESS='DIRECT', RECL=131072)
      ...
      OPEN (UNIT=123, FORM='UNFORMATTED', ACCESS='SEQUENTIAL', RECL=997)
```

```
XLFRTEOPTS="unit_vars=yes"     # Allow overriding default names.
XLFUNIT_1="/tmp/molecules.dat" # Use this named file.
XLFUNIT_12="../data/scratch"   # Relative to current directory.
XLFUNIT_123="/home/user/data/Users/username/data"  # Somewhere besides /tmp.
export XLFRTEOPTS XLFUNIT_1 XLFUNIT_12 XLFUNIT_123
```

Notes:

1. The **XLFUNIT_***number* variable name must be in uppercase, and *number* must not have any leading zeros.
2. **unit_vars=yes** might be only part of the value for the **XLFRTEOPTS** variable, depending on what other runtime options you have set. See *Setting runtime options* in the *XL Fortran Compiler Reference* for other options that might be part of the **XLFRTEOPTS** value.
3. If the **unit_vars** runtime option is set to **no** or is undefined or if the applicable **XLFUNIT_***number* variable is not set when the program is run, the program uses a default name (**fort.***unit*) for the file and puts it in the current directory.

Naming scratch files

To place all scratch files in a particular directory, set the **TMPDIR** environment variable to the name of the directory. The program then opens the scratch files in this directory. You might need to do this if your **/tmp** directory is too small to hold the scratch files.

To give a specific name to a scratch file, you must do the following:

1. Set the runtime option **scratch_vars**.
2. Set an environment variable with a name of the form **XLFSCRATCH_***unit* for each scratch file.

The association is between a unit number in the Fortran program and a path name in the file system. In this case, the **TMPDIR** variable does not affect the location of the scratch file.

For example, suppose that the Fortran program contains the following statements:

```
      OPEN (UNIT=1, STATUS='SCRATCH', &
            FORM='FORMATTED', ACCESS='SEQUENTIAL', RECL=1024)
      ...
      OPEN (UNIT=12, STATUS='SCRATCH', &
            FORM='UNFORMATTED', ACCESS='DIRECT', RECL=131072)
      ...
      OPEN (UNIT=123, STATUS='SCRATCH', &
            FORM='UNFORMATTED', ACCESS='SEQUENTIAL', RECL=997)
```

```
XLFRTEOPTS="scratch_vars=yes"     # Turn on scratch file naming.
XLFSCRATCH_1="/tmp/molecules.dat" # Use this named file.
XLFSCRATCH_12="../data/scratch"   # Relative to current directory.
XLFSCRATCH_123="/home/user/data/Users/username/data"  # Somewhere besides /tmp.
export XLFRTEOPTS XLFSCRATCH_1 XLFSCRATCH_12 XLFSCRATCH_123
```

Notes:

1. The **XLFSCRATCH_***number* variable name must be in uppercase, and *number* must not have any leading zeros.
2. **scratch_vars=yes** might be only part of the value for the **XLFRTEOPTS** variable, depending on what other runtime options you have set. See *Setting runtime options* in the *XL Fortran Compiler Reference* for other options that might be part of the **XLFRTEOPTS** value.
3. If the **scratch_vars** runtime option is set to **no** or is undefined or if the applicable **XLFSCRATCH_***number* variable is not set when the program is run, the program chooses a unique file name for the scratch file and puts it in the directory named by the **TMPDIR** variable or in the **/tmp** directory if the **TMPDIR** variable is not set.

XL Fortran thread-safe I/O library

The XL Fortran runtime library **libxlf90_r.so** provides support for parallel execution of Fortran I/O statements.

Synchronization of I/O operations

During parallel execution, multiple threads might perform I/O operations on the same file at the same time. If they are not synchronized, the results of these I/O operations could be shuffled or merged or both, and the application might produce incorrect results or even terminate. The XL Fortran runtime library synchronizes I/O operations for parallel applications. It performs the synchronization within the I/O library, and it is transparent to application programs. The purpose of the synchronization is to ensure the integrity and correctness of each individual I/O operation. However, the runtime does not have control over the order in which threads execute I/O statements. Therefore, the order of records read in or written out is not predictable under parallel I/O operations. Refer to “Parallel I/O issues” on page 281 for details.

External files

For external files, the synchronization is performed on a per-unit basis. The XL Fortran runtime ensures that only one thread can access a particular logical unit to prevent several threads from interfering with each other. When a thread is performing an I/O operation on a unit, other threads attempting to perform I/O operations on the same unit must wait until the first thread finishes its operation. Therefore, the execution of I/O statements by multiple threads on the same unit is serialized. However, the runtime environment does not prevent threads from operating on different logical units in parallel. In other words, parallel access to different logical units is not necessarily serialized.

Functionality of I/O under synchronization

The XL Fortran runtime sets its internal locks to synchronize access to logical units. This should not have any functional impact on the I/O operations performed by a Fortran program. Also, it will not impose any additional restrictions to the operability of Fortran I/O statements except for the use of I/O statements in a signal handler that is invoked asynchronously. Refer to "Use of I/O statements in signal handlers" on page 283 for details.

Parallel I/O issues

The order in which parallel threads perform I/O operations is not predictable. The XL Fortran runtime does not have control over the ordering. It will allow whichever thread that executes an I/O statement on a particular logical unit and obtains the lock on it first to proceed with the operation. Therefore, only use parallel I/O in cases where at least one of the following is true:

- Each thread performs I/O on a predetermined record in direct-access files.
- Each thread performs I/O on a different part of a stream-access file. Different I/O statements cannot use the same, or overlapping, areas of a file.
- The result of an application does not depend on the order in which records are written out or read in.
- Each thread performs I/O on a different file.

In these cases, results of the I/O operations are independent of the order in which threads execute. However, you might not get the performance improvements that you expect, since the I/O library serializes parallel access to the same logical unit from multiple threads. Examples of these cases are as follows:

- Each thread performs I/O on a pre-determined record in a direct-access file:

```
do i = 1, 10
  write(4, '(i4)', rec = i) a(i)
enddo
```

- Each thread performs I/O on a different part of a stream-access file. Different I/O statements cannot use the same, or overlapping, areas of a file.

```
do i = 1, 9
  write(4, '(i4)', pos = 1 + 5 * (i - 1)) a(i)
  ! We use 5 above because i4 takes 4 file storage
  ! units + 1 file storage unit for the record marker.
enddo
```

- In the case that each thread operates on a different file, since threads share the status of the logical units connected to the files, the thread still needs to obtain the lock on the logical unit for either retrieving or updating the status of the logical unit. However, the runtime allows threads to perform the data transfer between the logical unit and the I/O list item in parallel. If an application contains a large number of small I/O requests in a parallel region, you might not get the expected performance because of the lock contention. Consider the following example:

```
      program example

      use omp_lib

      integer, parameter :: num_of_threads = 4, max = 5000000
      character*10 file_name
      integer i, file_unit, thread_id
      integer, dimension(max, 2 * num_of_threads) :: aa

      call omp_set_num_threads(num_of_threads)

!$omp parallel private(file_name, thread_id, file_unit, i) shared(aa)

      thread_id = omp_get_thread_num()
      file_name = 'file_'
      file_name(6:6) = char(ichar('0') + thread_id)
      file_unit = 10 + thread_id

      open(file_unit, file = file_name, status = 'old', action = 'read')

      do i = 1, max
         read(file_unit, *) aa(i, thread_id * 2 + 1), aa(i, thread_id * 2 + 2)
      end do

      close(file_unit)

!$omp end parallel
      end
```

The XL Fortran runtime synchronizes retrieving and updating the status of the logical units while performing data transfer in parallel. In order to increase performance, it is recommended to increase the size of data transfer per I/O request. The do loop, therefore, should be rewritten as follows:

```
      read(file_unit, *) a(:, thread_id * 2 + 1 : thread_id * 2 + 2)

      do i = 1, max
         ! Do something for each element of array 'aa'.
      end do
```

- The result does not depend on the order in which records are written out or read in:

```
     real a(100)
     do i = 1, 10
       read(4) a(i)
     enddo
     call qsort_(a)
```

- Each thread performs I/O on a different logical unit of direct access, sequential access, or stream access:

```
     do i = 11, 20
       write(i, '(i4)') a(i - 10)
     enddo
```

For multiple threads to write to or read from the same sequential-access file, or to write to or read from the same stream-access file without using the **POS=** specifier, the order of records written out or read in depends on the order in which the threads execute the I/O statement on them. This order, as stated previously, is not predictable. Therefore, the result of an application could be incorrect if it assumes records are sequentially related and cannot be arbitrarily written out or read in. For example, if the following loop is parallelized, the numbers printed out will no longer be in the sequential order from 1 to 500 as the result of a serial execution:

```
do i = 1, 500
  print *, i
enddo
```

Applications that depend on numbers being strictly in the specified order will not work correctly.

The XL Fortran runtime option **multconn=yes** allows connection of the same file to more than one logical unit simultaneously. Since such connections can only be made for reading (**ACCESS='READ'**), access from multiple threads to logical units that are connected to the same file will produce predictable results.

Use of I/O statements in signal handlers

There are basically two kinds of signals in the POSIX signal model: *synchronously* and *asynchronously* generated signals. Signals caused by the execution of some code of a thread, such as a reference to an unmapped, protected, or bad memory (**SIGSEGV** or **SIGBUS**), floating-point exception (**SIGFPE**), execution of a trap instruction (**SIGTRAP**), or execution of illegal instructions (**SIGILL**) are said to be synchronously generated. Signals may also be generated by events outside the process: for example, **SIGINT**, **SIGHUP**, **SIGQUIT**, **SIGIO**, and so on. Such events are referred to as interrupts. Signals that are generated by interrupts are said to be asynchronously generated.

The XL Fortran runtime is asynchronous signal unsafe. This means that an XL Fortran I/O statement cannot be used in a signal handler that is entered because of an asynchronously generated signal. The behavior of the system is undefined when an XL Fortran I/O statement is called from a signal handler that interrupts an I/O statement. However, it is safe to use I/O statements in signal handlers for synchronous signals.

Sometimes an application can guarantee that a signal handler is not entered asynchronously. For example, an application might mask signals except when it runs certain known sections of code. In such situations, the signal will not interrupt any I/O statements and other asynchronous signal unsafe functions. Therefore, you can still use Fortran I/O statements in an asynchronous signal handler.

A much easier and safer way to handle asynchronous signals is to block signals in all threads and to explicitly wait (using **sigwait()**) for them in one or more separate threads. The advantage of this approach is that the **handler** thread can use Fortran I/O statements as well as other asynchronous signal unsafe routines.

Asynchronous thread cancellation

When a thread enables asynchronous thread cancellability, any cancellation request is acted upon immediately.

The XL Fortran runtime environment is not asynchronous thread cancellation safe. The behavior of the system is undefined if a thread is cancelled asynchronously while it is in the XL Fortran runtime environment.

Chapter 10. Implementation details of XL Fortran floating-point processing

This topic answers some common questions about floating-point processing.

- How can I get predictable, consistent results?
- How can I get the fastest or the most accurate results?
- How can I detect, and possibly recover from, exception conditions?
- Which compiler options can I use for floating-point calculations?

The topics describing floating-point precision make frequent reference to the compiler options that are grouped together in *Floating-point and integer control* in the *XL Fortran Compiler Reference*, especially the **-qfloat** option. The XL Fortran compiler also provides three intrinsic modules for exception handling and IEEE arithmetic support to help you write IEEE module-compliant code that can be more portable. See *IEEE Modules and Support* in the *XL Fortran Language Reference* for details.

The use of the compiler options for floating-point calculations affects the accuracy, performance, and possibly the correctness of floating-point calculations. Although the default values for the options were chosen to provide efficient and correct execution of most programs, you may need to specify nondefault options for your applications to work the way you want. We strongly advise you to read this section before using these options.

Note: The discussions of single-, double-, and extended-precision calculations in this section all refer to the default situation, with **-qrealsize=4** and no **-qautodbl** specified. If you change these settings, keep in mind that the size of a Fortran **REAL**, **DOUBLE PRECISION**, and so on may change, but single precision, double precision, and extended precision (in lowercase) still refer to 4-, 8-, and 16-byte entities respectively.

IEEE floating-point overview

The *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985 and IEEE Std 754-2008* and the details of how it applies to XL Fortran on specific hardware platforms, are summarized in the following topics.

For information on the Fortran 2003 IEEE Module and arithmetic support, see the *XL Fortran Language Reference*.

Compiling for strict IEEE conformance

By default, XL Fortran follows most, but not all of the rules in the IEEE standard. To compile for strict compliance with the standard:

- Use the compiler option **-qfloat=nomaf**.
- If the program changes the rounding mode at run time, include **rrm** among the **-qfloat** suboptions.
- If the data or program code contains signaling NaN values (NAN), include **nans** among the **-qfloat** suboptions. (A signaling NaN is different from a quiet NaN; you must explicitly code it into the program or data or create it by using the **-qinitauto** or **-qinitalloc** compiler option.)

- If you are compiling with **-O3**, or a higher base optimization level, include the **-qstrict** option. You can also use the **-qstrict** suboptions to refine the level of control for the transformations performed by the optimizers.

Related reference:

See -qstrict in the Compiler Reference

IEEE Single- and double-precision values

XL Fortran encodes single-precision and double-precision values in IEEE format. For the range and representation, see *Real* in the *XL Fortran Language Reference*.

IEEE extended-precision values

The IEEE standard suggests, but does not mandate, a format for extended-precision values. XL Fortran does not use this format. "Extended-precision values" on page 289 describes the format that XL Fortran uses.

Infinities and NaNs

For single-precision real values:

- Positive infinity is represented by the bit pattern X'7F80 0000'.
- Negative infinity is represented by the bit pattern X'FF80 0000'.
- A signaling NaN is represented by any bit pattern between X'7F80 0001' and X'7FBF FFFF' or between X'FF80 0001' and X'FFBF FFFF'.
- A quiet NaN is represented by any bit pattern between X'7FC0 0000' and X'7FFF FFFF' or between X'FFC0 0000' and X'FFFF FFFF'.

For double-precision real values:

- Positive infinity is represented by the bit pattern X'7FF00000 00000000'.
- Negative infinity is represented by the bit pattern X'FFF00000 00000000'.
- A signaling NaN is represented by any bit pattern between X'7FF00000 00000001' and X'7FF7FFFF FFFFFFFF' or between X'FFF00000 00000001' and X'FFF7FFFF FFFFFFFF'.
- A quiet NaN is represented by any bit pattern between X'7FF80000 00000000' and X'7FFFFFFF FFFFFFFF' or between X'FFF80000 00000000' and X'FFFFFFFF FFFFFFFF'.

These values do not correspond to any Fortran real constants. You can generate all of these by encoding the bit pattern directly, or by using the **ieee_value** function provided in the **ieee_arithmetic** intrinsic module. Using the **ieee_value** function is the preferred programming technique, as it is allowed by the Fortran 2003 standard and the results are portable. Encoding the bit pattern directly could cause portability problems on machines using different bit patterns for the different values. All except signaling NaN values can occur as the result of arithmetic operations:

```
$ cat fp_values.f
real plus_inf, minus_inf, plus_nanq, minus_nanq, nans
real large

data plus_inf /z'7f800000'/
data minus_inf /z'ff800000'/
data plus_nanq /z'7fc00000'/
data minus_nanq /z'ffc00000'/
data nans /z'7f800001'/

print *, 'Special values:', plus_inf, minus_inf, plus_nanq, minus_nanq, nans

! They can also occur as the result of operations.
large = 10.0 ** 200
print *, 'Number too big for a REAL:', large * large
print *, 'Number divided by zero:', (-large) / 0.0
print *, 'Nonsensical results:', plus_inf - plus_inf, sqrt(-large)

! To find if something is a NaN, compare it to itself.
print *, 'Does a quiet NaN equal itself:', plus_nanq .eq. plus_nanq
print *, 'Does a signaling NaN equal itself:', nans .eq. nans
! Only for a NaN is this comparison false.

end
$ bgxlf95 -o fp_values fp_values.f
** _main   === End of Compilation 1 ===
1501-510  Compilation successful for file fp_values.f.
$ fp_values
 Special values: INF -INF NAN -NAN NAN
 Number too big for a REAL: INF
 Number divided by zero: -INF
 Nonsensical results: NAN NAN
 Does a quiet NaN equal itself: F
 Does a signaling NaN equal itself: F
```

Exception-handling model

The IEEE standard defines several exception conditions that can occur:

OVERFLOW
The exponent of a value is too large to be represented.

UNDERFLOW
A nonzero value is so small that it cannot be represented without an extraordinary loss of accuracy. The value can be represented only as zero or a subnormal number (denorm).

ZERODIVIDE
A finite nonzero value is divided by zero.

INVALID
Operations are performed on values for which the results are not defined. These include:
- Operations on signaling NaN values
- infinity - infinity
- 0.0 * infinity
- 0.0 / 0.0
- mod(x,y) or ieee_rem(x,y) (or other remainder functions) when x is infinite or y is zero
- The square root of a negative number except -0.0
- Conversion of a floating-point number to an integer when the converted value cannot be represented faithfully

- Comparisons involving NaN values

INEXACT
A computed value cannot be represented exactly, so a rounding error is introduced. (This exception is very common.)

XL Fortran always detects these exceptions when they occur, but by default does not take any special action. Calculation continues, usually with a NaN or infinity value as the result. If you want to be automatically informed when an exception occurs, you can turn on exception trapping through compiler options or calls to intrinsic subprograms. However, different results, intended to be manipulated by exception handlers, are produced:

Table 30. Results of IEEE exceptions, with and without trapping enabled

	Overflow	**Underflow**	**Zerodivide**	**Invalid**	**Inexact**
Exceptions not enabled (default)	INF	Denormalized number	INF	NaN	Rounded result
Exceptions enabled	Unnormalized number with biased exponent	Unnormalized number with biased exponent	No result	No result	Rounded result

Note: Because different results are possible, it is very important to make sure that any exceptions that are generated are handled correctly. See "Detecting and trapping floating-point exceptions" on page 294 for instructions on doing so.

Hardware-specific floating-point overview

Single- and double-precision values and extended-precision values for hardware-specific floating-point processing are described in the following topics.

Single- and double-precision values

The PowerPC floating-point hardware performs calculations in either IEEE single-precision (equivalent to **REAL(4)** in Fortran programs) or IEEE double-precision (equivalent to **REAL(8)** in Fortran programs).

Keep the following considerations in mind:

- Double precision provides greater range (approximately 10**(-308) to 10**308) and precision (about 15 decimal digits) than single precision (approximate range 10**(-38) to 10**38, with about 7 decimal digits of precision).
- Computations that mix single and double operands are performed in double precision, which requires conversion of the single-precision operands to double-precision. These conversions do not affect performance.
- Double-precision values that are converted to single-precision (such as when you specify the **SNGL** intrinsic or when a double-precision computation result is stored into a single-precision variable) require rounding operations. A rounding operation produces the correct single-precision value, which is based on the IEEE rounding mode in effect. The value may be less precise than the original double-precision value, as a result of rounding error. Conversions from double-precision values to single-precision values may reduce the performance of your code.
- Programs that manipulate large amounts of floating-point data may run faster if they use **REAL(4)** rather than **REAL(8)** variables. (You need to ensure that **REAL(4)** variables provide you with acceptable range and precision.) The

programs may run faster because the smaller data size reduces memory traffic, which can be a performance bottleneck for some applications.

The floating-point hardware also provides a special set of double-precision operations that multiply two numbers and add a third number to the product. These combined multiply-add (**MAF**) operations are performed at the same speed at which either an individual multiply or add is performed. The **MAF** functions provide an extension to the IEEE 754-1985 standard (but are in the 754-2008 standard) because they perform the multiply and add with one (rather than two) rounding errors. The **MAF** functions are faster and more accurate than the equivalent separate operations.

Extended-precision values

XL Fortran extended precision is not in the format suggested by the IEEE standard, which suggests extended formats using more bits in both the exponent (for greater range) and the fraction (for greater precision).

XL Fortran extended precision, equivalent to **REAL(16)** in Fortran programs, is implemented in software. Extended precision provides the same range as double precision (about 10**(-308) to 10**308) but more precision (a variable amount, about 31 decimal digits or more). The software support is restricted to round-to-nearest mode. Programs that use extended precision must ensure that this rounding mode is in effect when extended-precision calculations are performed. See "Selecting the rounding mode" on page 290 for the different ways you can control the rounding mode.

Programs that specify extended-precision values as hexadecimal, octal, binary, or Hollerith constants must follow these conventions:

- Extended-precision numbers are composed of two double-precision numbers with different magnitudes that do not overlap (except when the number is zero or close to zero). That is, the binary exponents differ by at least the number of fraction bits in a **REAL(8)**. The high-order double-precision value (the one that comes first in storage) must have the larger magnitude. The value of the extended-precision number is the sum of the two double-precision values.
- For a value of NaN or infinity, you must encode one of these values within the high-order double-precision value. The low-order value is not significant except that it cannot be set to NaN when the high-order value is infinity.

Because an XL Fortran extended-precision value can be the sum of two values with greatly different exponents, leaving a number of assumed zeros in the fraction, the format actually has a variable precision with a minimum of about 31 decimal digits. You get more precision in cases where the exponents of the two double values differ in magnitude by more than the number of digits in a double-precision value. This encoding allows an efficient implementation intended for applications requiring more precision but no more range than double precision.

Note:

1. In the discussions of rounding errors because of compile-time folding of expressions, keep in mind that this folding produces different results for extended-precision values more often than for other precisions.
2. Special numbers, such as NaN and infinity, are not fully supported for extended-precision values. Arithmetic operations do not necessarily propagate these numbers in extended precision.

3. XL Fortran does not always detect floating-point exception conditions (see “Detecting and trapping floating-point exceptions” on page 294) for extended-precision values. If you turn on floating-point exception trapping in programs that use extended precision, XL Fortran may also generate signals in cases where an exception condition does not really occur.

How XL Fortran rounds floating-point calculations

Understanding rounding operations in XL Fortran can help you get predictable, consistent results. It can also help you make informed decisions when you have to make tradeoffs between speed and accuracy.

In general, floating-point results from XL Fortran programs are more accurate than those from other implementations because of **MAF** operations and the higher precision used for intermediate results. If identical results are more important to you than the extra precision and performance of the XL Fortran defaults, read “Duplicating the floating-point results of other systems” on page 293.

Selecting the rounding mode

To change the rounding mode in a program, you can call the **fpsets** and **fpgets** routines, which use an array of logicals named **fpstat**, defined in the include files `/opt/ibmcmp/xlf/bg/14.1/include/fpdt.h` and `/opt/ibmcmp/xlf/bg/14.1/include/fpdc.h`. The **fpstat** array elements correspond to the bits in the floating-point status and control register.

For floating-point rounding control, the array elements **fpstat(fprn1)** and **fpstat(fprn2)** are set as specified in the following table:

Table 31. Rounding-mode bits to use with fpsets and fpgets

fpstat(fprn1)	**fpstat(fprn2)**	**Rounding Mode Enabled**
.true.	.true.	Round towards -infinity.
.true.	.false.	Round towards +infinity.
.false.	.true.	Round towards zero.
.false.	.false.	Round to nearest.

For example:

```
program fptest
  include 'fpdc.h'

  call fpgets( fpstat ) ! Get current register values.
  if ( (fpstat(fprn1) .eqv. .false.) .and. +
         (fpstat(fprn2) .eqv. .false.)) then
  print *, 'Before test: Rounding mode is towards nearest'
  print *, '            2.0 / 3.0 = ', 2.0 / 3.0
  print *, '           -2.0 / 3.0 = ', -2.0 / 3.0
  end if

  call fpgets( fpstat )   ! Get current register values.
  fpstat(fprn1) = .TRUE.  ! These 2 lines mean round towards
  fpstat(fprn2) = .FALSE. !   +infinity.
  call fpsets( fpstat )
  r = 2.0 / 3.0
  print *, 'Round towards +infinity:  2.0 / 3.0= ', r

  call fpgets( fpstat )   ! Get current register values.
  fpstat(fprn1) = .TRUE.  ! These 2 lines mean round towards
```

```
        fpstat(fprn2) = .TRUE.  !   -infinity.
        call fpsets( fpstat )
        r = -2.0 / 3.0
        print *, 'Round towards -infinity: -2.0 / 3.0= ', r
        end
! This block data program unit initializes the fpstat array, and so on.
        block data
        include 'fpdc.h'
        include 'fpdt.h'
      end
```

XL Fortran also provides several procedures that allow you to control the floating-point status and control register of the processor directly. These procedures are more efficient than the **fpsets** and **fpgets** subroutines because they are mapped into inlined machine instructions that manipulate the floating-point status and control register (fpscr) directly.

XL Fortran supplies the **get_round_mode()** and **set_round_mode()** procedures in the `xlf_fp_util` module. These procedures return and set the current floating-point rounding mode, respectively.

For example:

```
      program fptest
        use, intrinsic :: xlf_fp_util
        integer(fpscr_kind) old_fpscr
        if ( get_round_mode() == fp_rnd_rn ) then
        print *, 'Before test: Rounding mode is towards nearest'
        print *, '            2.0 / 3.0 = ', 2.0 / 3.0
        print *, '           -2.0 / 3.0 = ', -2.0 / 3.0
        end if

        old_fpscr = set_round_mode( fp_rnd_rp )
        r = 2.0 / 3.0
        print *, 'Round towards +infinity:  2.0 / 3.0 = ', r

        old_fpscr = set_round_mode( fp_rnd_rm )
        r = -2.0 / 3.0
        print *, 'Round towards -infinity: -2.0 / 3.0 = ', r
      end
```

XL Fortran supplies the **ieee_get_rounding_mode()** and **ieee_set_rounding_mode()** procedures in the `ieee_arithmetic` module. These portable procedures retrieve and set the current floating-point rounding mode, respectively.

For example:

```
      program fptest
        use, intrinsic :: ieee_arithmetic
        type(ieee_round_type) current_mode
        call ieee_get_rounding_mode( current_mode )
        if ( current_mode == ieee_nearest ) then
        print *, 'Before test: Rounding mode is towards nearest'
        print *, '            2.0 / 3.0 = ', 2.0 / 3.0
        print *, '           -2.0 / 3.0 = ', -2.0 / 3.0
        end if

        call ieee_set_rounding_mode( ieee_up )
        r = 2.0 / 3.0
        print *, 'Round towards +infinity:  2.0 / 3.0 = ', r

        call ieee_set_rounding_mode( ieee_down )
        r = -2.0 / 3.0
        print *, 'Round towards -infinity: -2.0 / 3.0 = ', r
      end
```

Notes:

1. Extended-precision floating-point values must only be used in round-to-nearest mode.
2. For thread-safety and reentrancy, the include file `/opt/ibmcmp/xlf/bg/14.1/include/fpdc.h` contains a **THREADLOCAL** directive that is protected by the trigger constant **IBMT**. The invocation commands **bgxlf_r**, **bgxlf90_r**, **bgxlf95_r**, **bgxlf2003_r**, and **bgxlf2008_r** turn on the **-qthreaded** compiler option by default, which in turn implies the trigger constant **IBMT**. If you are including the file `/opt/ibmcmp/xlf/bg/14.1/include/fpdc.h` in code that is not intended to be threadsafe, do not specify **IBMT** as a trigger constant.
3. Compile a program that changes the rounding mode with **-qfloat=rrm**.

Minimizing rounding errors

There are several strategies for handling rounding errors and other unexpected, slight differences in calculated results. You may want to consider one or more of the following strategies:

- Minimizing the amount of overall rounding
- Delaying as much rounding as possible to run time
- Ensuring that if some rounding is performed in a mode other than round-to-nearest, *all* rounding is performed in the same mode

Minimizing overall rounding

Rounding operations, especially in loops, reduce code performance and may have a negative effect on the precision of computations. Consider using double-precision variables instead of single-precision variables when you store the temporary results of double-precision calculations, and delay rounding operations until the final result is computed.

Delaying rounding until run time

The compiler evaluates floating-point expressions during compilation when it can, so that the resulting program does not run more slowly due to unnecessary runtime calculations. However, the results of the compiler's evaluation might not match exactly the results of the runtime calculation. To delay these calculations until run time, specify the **nofold** suboption of the **-qfloat** option.

The results may still not be identical; for example, calculations in **DATA** and **PARAMETER** statements are still performed at compile time.

The differences in results due to **fold** or **nofold** are greatest for programs that perform extended-precision calculations or are compiled with the **-O** option or both.

Ensuring that the rounding mode is consistent

You can change the rounding mode from its default setting of round-to-nearest. (See for examples.) If you do so, you must be careful that *all* rounding operations for the program use the same mode:

- Specify the equivalent setting on the **-qieee** option, so that any compile-time calculations use the same rounding mode.
- Specify the **rrm** suboption of the **-qfloat** option, so that the compiler does not perform any optimizations that require round-to-nearest rounding mode to work correctly.

For example, you might compile a program like the one in "Selecting the rounding mode" on page 290 with this command if the program consistently uses round-to-plus-infinity mode:

```
bgxlf95 -qieee=plus -qfloat=rrm changes_rounding_mode.f
```

Duplicating the floating-point results of other systems

To duplicate the double-precision results of programs on systems with different floating-point architectures (without multiply-add instructions), specify the **nomaf** suboption of the **-qfloat** option. This suboption prevents the compiler from generating any multiply-add instructions. This results in decreased accuracy and performance but provides strict conformance to the IEEE standard for double-precision arithmetic.

To duplicate the results of programs where the default size of **REAL** items is different from that on systems running XL Fortran, use the **-qrealsize** option to change the default **REAL** size when compiling with XL Fortran.

If the system whose results you want to duplicate preserves full double precision for default real constants that are assigned to **DOUBLE PRECISION** variables, use the **-qdpc** or **-qrealsize** option.

If results consistent with other systems are important to you, include **norsqrt** and **nofold** in the settings for the **-qfloat** option. If you specify the option **-O3**, **-O4**, or **-O5**, include **-qstrict** and any necessary suboptions too.

Related information:

See -qarch in the Compiler Reference

See -qfloat in the Compiler Reference

See -qrealsize in the Compiler Reference

See -qstrict in the Compiler Reference

Maximizing floating-point performance

If performance is your primary concern and you want your program to be relatively safe but do not mind if results are slightly different (generally more precise) from what they would be otherwise, optimize the program with the **-O** option, and specify **-qfloat=rsqrt:hssngl:fltint**.

The following topics describe the functions of these suboptions:

- The **rsqrt** suboption replaces division by a square root with multiplication by the reciprocal of the root, a faster operation that may not produce precisely the same result.
- The **hssngl** suboption improves the performance of single-precision (**REAL(4)**) floating-point calculations by suppressing rounding operations that are required by the Fortran language but are not necessary for correct program execution. The results of floating-point expressions are kept in double precision where the original program would round them to single-precision. These results are then used in some later expressions instead of the rounded results.

  To detect single-precision floating-point overflows and underflows, rounding operations are still inserted when double-precision results are stored into single-precision memory locations. However, if optimization removes such a store operation, **hssngl** also removes the corresponding rounding operation, possibly preventing the exception. (Depending on the characteristics of your program, you may or may not care whether the exception happens.)

The **hssngl** suboption is safe for all types of programs because it always only *increases* the precision of floating-point calculations. Program results may differ because of the increased precision and because of avoidance of some exceptions.

- The **fltint** suboption speeds up float-to-integer conversions by reducing error checking for overflows when the program is compiled to run on older processors. You should make sure that any floats that are converted to integers are not outside the range of the corresponding integer types.

Detecting and trapping floating-point exceptions

The IEEE standard for floating-point arithmetic defines a number of exception (or error) conditions that might require special care to avoid or recover from. The following topics are intended to help you make your programs work safely in the presence of such exception conditions while sacrificing the minimum amount of performance.

The floating-point hardware always detects a number of floating-point exception conditions (which the IEEE standard rigorously defines): overflow, underflow, zerodivide, invalid, and inexact.

By default, the only action that occurs is that a status flag is set. The program continues without a problem (although the results from that point on may not be what you expect). If you want to know when an exception occurs, you can arrange for one or more of these exception conditions to generate a signal.

The signal causes a branch to a handler routine. The handler receives information about the type of signal and the state of the program when the signal occurred. It can produce a core dump, display a listing showing where the exception occurred, modify the results of the calculation, or carry out some other processing that you specify.

The XL Fortran compiler uses the operating system facilities for working with floating-point exception conditions. These facilities indicate the presence of floating-point exceptions by generating **SIGFPE** signals.

Compiler features for trapping floating-point exceptions

To turn on XL Fortran exception trapping, compile the program with the **-qflttrap** option and some combination of suboptions that includes **enable**. This option uses trap operations to detect floating-point exceptions and generates **SIGFPE** signals when exceptions occur, provided that a signal handler for **SIGFPE** is installed.

-qflttrap also has suboptions that correspond to the names of the exception conditions. In particular, use the **-qflttrap=qpxstore** suboption to detect NaN or infinity values in QPX vectors. The compiler generates stores with indicating instructions for QPX vectors in registers. For example, if you are concerned with handling overflow, underflow, and QPX floating point exceptions, you can specify the following command:

```
bgxlf -qflttrap=overflow:underflow:qpxstore:enable compute_pi.f
```

You only need **enable** when you are compiling the main program. However, it is very important and does not cause any problems if you specify it for other files, so always include it when you use **-qflttrap**.

An advantage of this approach is that performance impact is relatively low. However, this approach only traps exceptions that occur in code that you compiled with **-qflttrap**, which does not include system library routines.

Notes:

1. If your program depends on floating-point exceptions occurring for particular operations, also specify **-qfloat** suboptions that include **nofold**. Otherwise, the compiler might replace an exception-producing calculation with a constant NaN or infinity value, or it might eliminate an overflow in a single-precision operation.
2. The suboptions of the **-qflttrap** option replace an earlier technique that required you to modify your code with calls to the **fpsets** and **fpgets** procedures. You no longer require these calls for exception handling if you use the appropriate **-qflttrap** settings.

   Attention: If your code contains **fpsets** calls that enable checking for floating-point exceptions and you do not use the **-qflttrap** option when compiling the whole program, the program will produce unexpected results if exceptions occur, as explained in Table 30 on page 288.

Installing an exception handler

When a program that uses the XL Fortran or Blue Gene/Q exception-detection facilities encounters an exception condition, it receives a signal from the operating system. This causes a branch to whatever handler is specified by the program.

By default, programs on Blue Gene/Q do not trap on floating-point exceptions unless a signal handler is installed. To produce a core file, you can use the **xl__trcedump** signal handler described below. If you want to install a **SIGTRAP** or **SIGFPE** signal handler, use the **-qsigtrap** option. It allows you to specify an XL Fortran handler that produces a traceback or to specify a handler you have written:

```
bgxlf95 -qflttrap=ov:und:en pi.f                          # Dump core on an exception
bgxlf95 -qflttrap=ov:und:en -qsigtrap pi.f                # Uses the xl__trce handler
bgxlf95 -qflttrap=ov:und:en -qsigtrap=return_22_over_7 pi.f  # Uses any other handler
```

You can also install an alternative exception handler, either one supplied by XL Fortran or one you have written yourself, by calling the **SIGNAL** subroutine (defined in `/opt/ibmcmp/xlf/bg/14.1/include/fexcp.h`):

```
  INCLUDE 'fexcp.h'
  CALL SIGNAL(SIGTRAP,handler_name)
  CALL SIGNAL(SIGFPE,handler_name)
```

The XL Fortran exception handlers and related routines are:

xl__ieee
: Produces a traceback and an explanation of the signal and continues execution by supplying the default IEEE result for the failed computation. This handler allows the program to produce the same results as if exception detection was not turned on.

xl__trce
: Produces a traceback and stops the program.

xl__trcedump
: Produces a traceback and a core file and stops the program.

xl__sigdump
: Provides a traceback that starts from the point at which it is called and provides information about the signal. You can only call it from inside a

user-written signal handler. It does not stop the program. To successfully continue, the signal handler must perform some cleanup after calling this subprogram.

xl__trbk
Provides a traceback that starts from the point at which it is called. You call it as a subroutine from your code, rather than specifying it with the **-qsigtrap** option. It requires no parameters. It does not stop the program.

All of these handler names contain double underscores to avoid duplicating names that you declared in your program. All of these routines work for both **SIGTRAP** and **SIGFPE** signals.

You can use the **-g** compiler option to get line numbers in the traceback listings. The file `/opt/ibmcmp/xlf/bg/14.1/include/fsignal.h` defines a Fortran derived type similar to the `ucontext_t` structure in `/usr/include/sys/ucontext.h` system header. You can write a Fortran signal handler that accesses this derived type.

"Sample programs for exception handling" on page 299 lists some sample programs that illustrate how to use these signal handlers or write your own. Also see the **SIGNAL** procedure in the *XL Fortran Language Reference* for more information.

Producing a core file

To produce a core file, specify the **xl__trcedump** handler.

Controlling the floating-point status and control register

Before the introduction of **-qflttrap** suboptions or the **-qsigtrap** options, most of the processing for floating-point exceptions required you to change your source files to turn on exception trapping or install a signal handler. Although you can still do so, for any new applications, we recommend that you use the options instead.

To control exception handling at run time, compile without the **enable** suboption of the **-qflttrap** option:

```
bgxlf95 -qflttrap compute_pi.f     # Check all exceptions, but do not trap.
bgxlf95 -qflttrap=ov compute_pi.f  # Check one type, but do not trap.
```

Then, inside your program, manipulate the **fpstats** array (defined in the include file `/opt/ibmcmp/xlf/bg/14.1/include/fpdc.h`) and call the **fpsets** subroutine to specify which exceptions should generate traps.

See the sample program that uses **fpsets** and **fpgets** in "Selecting the rounding mode" on page 290.

Another method is to use the **set_fpscr_flags()** subroutine in the **xlf_fp_util** module. This subroutine allows you to set the floating-point status and control register flags you specify in the **MASK** argument. Flags that you do not specify in **MASK** remain unaffected. MASK must be of type **INTEGER(FPSCR_KIND)**. For example:

```
USE, INTRINSIC :: xlf_fp_util
INTEGER(FPSCR_KIND) SAVED_FPSCR
INTEGER(FP_MODE_KIND) FP_MODE

SAVED_FPSCR = get_fpscr()          ! Saves the current value of
                                   ! the fpscr register.
```

```
 CALL set_fpscr_flags(TRP_DIV_BY_ZERO) ! Enables trapping of
 ! ...                                 ! divide-by-zero.
 SAVED_FPSCR=set_fpscr(SAVED_FPSCR)    ! Restores fpscr register.
```

Another method is to use the **ieee_set_halting_mode** subroutine in the `ieee_exceptions` module. This portable subroutine allows you to set the halting (trapping) status for any **FPSCR** exception flags. For example:

```
USE, INTRINSIC :: ieee_exceptions
 TYPE(IEEE_STATUS_TYPE) SAVED_FPSCR
 CALL ieee_get_status(SAVED_FPSCR)   ! Saves the current value of the
                                     ! fpscr register

 CALL ieee_set_halting_mode(IEEE_DIVIDE_BY_ZERO, .TRUE.)  ! Enabled trapping
 ! ...                                                    ! of divide-by-zero.

 CALL IEEE_SET_STATUS(SAVED_FPSCR)  ! Restore fpscr register
```

xlf_fp_util procedures

The **xlf_fp_util** procedures allow you to query and control the floating-point status and control register (fpscr) of the processor directly. These procedures are more efficient than the **fpsets** and **fpgets** subroutines because they are mapped into inlined machine instructions that manipulate the floating-point status and control register directly.

The intrinsic module, **xlf_fp_util**, contains the interfaces and data type definitions for these procedures and the definitions for the named constants that are needed by the procedures. This module enables type checking of these procedures at compile time rather than link time. The following files are supplied for the **xlf_fp_util** module:

File names	**File type**	**Locations**
xlf_fp_util.mod	module symbol file (64-bit)	`/opt/ibmcmp/xlf/bg/14.1/include64`

To use the procedures, you must add a **USE XLF_FP_UTIL** statement to your source file. For more information, see the **USE** statement in the *XL Fortran Language Reference*.

When compiling with the **-U** option, you must code the names of these procedures in all lowercase.

For a list of the **xlf_fp_util** procedures, see the *Service and utility procedures* section in the *XL Fortran Language Reference*.

fpgets and fpsets subroutines

The **fpsets** and **fpgets** subroutines provide a way to manipulate or query the floating-point status and control register. Instead of calling the operating system routines directly, you pass information back and forth in **fpstat**, an array of logicals. The following table shows the most commonly used array elements that deal with exceptions:

Table 32. Exception bits to use with fpsets and fpgets

Array Element to Set to Enable	Array Element to Check if Exception Occurred	Exception Indicated When .TRUE.
n/a	fpstat(fpfx)	Floating-point exception summary
n/a	fpstat(fpfex)	Floating-point enabled exception summary
fpstat(fpve)	fpstat(fpvx)	Floating-point invalid operation exception summary
fpstat(fpoe)	fpstat(fpox)	Floating-point overflow exception
fpstat(fpue)	fpstat(fpux)	Floating-point underflow exception
fpstat(fpze)	fpstat(fpzx)	Zero-divide exception
fpstat(fpxe)	fpstat(fpxx)	Inexact exception
fpstat(fpve)	fpstat(fpvxsnan)	Floating-point invalid operation exception (signaling NaN)
fpstat(fpve)	fpstat(fpvxisi)	Floating-point invalid operation exception (INF-INF)
fpstat(fpve)	fpstat(fpvxidi)	Floating-point invalid operation exception (INF/INF)
fpstat(fpve)	fpstat(fpvxzdz)	Floating-point invalid operation exception (0/0)
fpstat(fpve)	fpstat(fpvximz)	Floating-point invalid operation exception (INF*0)
fpstat(fpve)	fpstat(fpvxvc)	Floating-point invalid operation exception (invalid compare)
n/a	fpstat(fpvxsoft)	Floating-point invalid operation exception (software request), PowerPC only
n/a	fpstat(fpvxsqrt)	Floating-point invalid operation exception (invalid square root), PowerPC only
n/a	fpstat(fpvxcvi)	Floating-point invalid operation exception (invalid integer convert), PowerPC only

To explicitly check for specific exceptions at particular points in a program, use **fpgets** and then test whether the elements in **fpstat** have changed. Once an exception has occurred, the corresponding exception bit (second column in the preceding table) is set until it is explicitly reset, except for **fpstat(fpfx)**, **fpstat(fpvx)**, and **fpstat(fpfex)**, which are reset only when the specific exception bits are reset.

An advantage of using the **fpgets** and **fpsets** subroutines (as opposed to controlling everything with suboptions of the **-qflttrap** option) includes control over granularity of exception checking. For example, you might only want to test if an exception occurred anywhere in the program when the program ends.

The disadvantages of this approach include the following:
- You have to change your source code.
- These routines differ from what you may be accustomed to on other platforms.

For example, to trap floating-point overflow exceptions but only in a certain section of the program, you would set `fpstat(fpoe)` to `.TRUE.` and call **fpsets**. After the exception occurs, the corresponding exception bit, `fpstat(fpox)`, is `.TRUE.` until the program runs:

```
call fpgets(fpstat)
fpstat(fpox) = .FALSE.
call fpsets(fpstat)   ! resetting fpstat(fpox) to .FALSE.
```

Sample programs for exception handling

Sample programs contained in `/opt/ibmcmp/xlf/bg/14.1/samples/floating_point` illustrate different aspects of exception handling:

flttrap_handler.c and flttrap_test.f
: A sample exception handler that is written in C and a Fortran program that uses it.

xl__ieee.F and xl__ieee.c
: Exception handlers that are written in Fortran and C that show how to substitute particular values for operations that produce exceptions. Even when you use support code such as this, the implementation of XL Fortran exception handling does not fully support the exception-handling environment that is suggested by the IEEE floating-point standard.

check_fpscr.f and postmortem.f
: Show how to work with the **fpsets** and **fpgets** procedures and the **fpstats** array.

fhandler.F
: Shows a sample Fortran signal handler and demonstrates the **xl__sigdump** procedure.

xl__trbk_test.f
: Shows how to use the **xl__trbk** procedure to generate a traceback listing without stopping the program.

The sample programs are strictly for illustrative purposes only.

Causing exceptions for particular variables

To mark a variable as "do not use", you can encode a special value called a signaling NaN in it. This causes an invalid exception condition any time that variable is used in a calculation.

If you use this technique, use the **nans** suboption of the **-qfloat** option, so that the program properly detects all cases where a signaling NaN is used, and one of the methods already described to generate corresponding **SIGFPE** signals.

Minimizing the performance impact of floating-point exception trapping

If you need to deal with floating-point exception conditions but are concerned that doing so will make your program too slow, here are some techniques that can help minimize the performance impact:

- Consider using only a subset of the **overflow**, **underflow**, **zerodivide**, **invalid**, and **inexact** suboptions with the **-qflttrap** option if you can identify some conditions that will never happen or you do not care about. In particular, because an **inexact** exception occurs for each rounding error, you probably should not check for it if performance is important.

Chapter 11. Porting programs to XL Fortran

XL Fortran provides many features intended to make it easier to take programs that were originally written for other computer systems or compilers and recompile them with XL Fortran.

Outline of the porting process

The process of porting a typical program is described in this topic.

The process for porting a typical program looks like this:

1. Identify any nonportable language extensions or subroutines that you used in the original program. Check to see if any of them are supported by XL Fortran:
   - Language extensions are identified in the *XL Fortran Language Reference*.
   - Some extensions require you to specify an XL Fortran compiler option; you can find these options listed in the *Portability and migration options* table in the *XL Fortran Compiler Reference*.
2. For any nonportable features that XL Fortran does not support, modify the source files to remove or work around them.
3. Do the same for any implementation-dependent features. For example, if your program relies on exact bit-pattern representation of floating-point values or uses system-specific file names, you may need to change it.
4. Compile the program with XL Fortran. If any compilation problems occur, fix them and recompile and fix any additional errors until the program compiles successfully.
5. Run the XL Fortran-compiled program and compare the output with the output from the other system. If the results are substantially different, there are probably still some implementation-specific features that need to be changed. If the results are only marginally different (for example, if XL Fortran produces a different number of digits of precision or a number differs in the last decimal place), decide whether the difference is significant enough to investigate further. You may be able to fix these differences.

Before porting programs to XL Fortran, read the tips in the following sections so that you know in advance what compatibility features XL Fortran offers.

Portability of directives

XL Fortran supports many directives available with other Fortran products. This ensures easy portability between products.

If your code contains *trigger_constants* other than the defaults in XL Fortran, you can use the **-qdirective** compiler option to specify them. For instance, if you are porting CRAY code contained in a file `xx.f`, you would use the following command to add the CRAY *trigger_constant*:

```
bgxlf95 xx.f -qdirective=mic\$
```

For fixed source form code, in addition to the **!** value for the *trigger_head* portion of the directive, XL Fortran also supports the *trigger_head* values **C**, **c**, and *****.

For more information, see the **-qdirective** option in the *XL Fortran Compiler Reference*.

XL Fortran supports a number of programming terms as synonyms to ease the effort of porting code from other Fortran products. Those terms that are supported are dependent on context, as indicated in the following tables:

Table 33. PARALLEL DO Clauses and their XL Fortran synonyms

PARALLEL DO Clause	**XL Fortran Synonym**
LASTLOCAL	LASTPRIVATE
LOCAL	PRIVATE
MP_SCHEDTYPE *and* CHUNK	SCHEDULE
SAVELAST	LASTPRIVATE
SHARE	SHARED
NEW	PRIVATE

Table 34. PARALLEL DO scheduling types and their XL Fortran synonyms

Scheduling Type	**XL Fortran Synonym**
GSS	GUIDED
INTERLEAVE	STATIC(1)
INTERLEAVED	STATIC(1)
INTERLEAVE(n)	STATIC(n)
INTERLEAVED(n)	STATIC(n)
SIMPLE	STATIC

Table 35. PARALLEL SECTIONS clauses and their XL Fortran synonyms

PARALLEL SECTIONS Clause	**XL Fortran Synonym**
LOCAL	PRIVATE
SHARE	SHARED
NEW	PRIVATE

Common industry extensions that XL Fortran supports

XL Fortran allows many of the same FORTRAN 77 extensions as other popular compilers.

These extensions include:

Extension	**Refer to** *XL Fortran Language Reference* **Section(s)**
Typeless constants	Typeless literal constants
**len* length specifiers for types	Data types
BYTE data type	Byte

Extension	**Refer to** *XL Fortran Language Reference* **Section(s)**
Long variable names	Names
Lower case	Names
Mixing integers and logicals (with **-qintlog** option)	Evaluation of expressions
Character-count **Q** edit descriptor (with **-qqcount** option)	Q (Character Count) Editing
Intrinsics for counting set bits in registers and determining data-object parity	POPCNT, POPPAR
64-bit data types (**INTEGER(8)**, **REAL(8)**, **COMPLEX(8)**, and **LOGICAL(8)**), including support for default 64-bit types (with **-qintsize** and **-qrealsize** options)	Integer Real Complex Logical
Integer **POINTER**s, similar to those supported by CRAY and Sun compilers. (XL Fortran integer pointer arithmetic uses increments of one byte, while the increment on CRAY computers is eight bytes. You may need to multiply pointer increments and decrements by eight to make programs ported from CRAY computers work properly.)	POINTER(integer)
Conditional vector merge (CVMGx) intrinsic functions	CVMGx (TSOURCE, FSOURCE, MASK)
Date and time service and utility functions (rtc, irtc, jdate, clock_, timef, and date)	Service and utility procedures
STRUCTURE, **UNION**, and **MAP** constructs	Structure components, Union and map

Finding nonstandard extensions

XL Fortran supports a number of extensions to various language standards. Many of these extensions are so common that you need to keep in mind, when you port programs to other systems, that not all compilers have them. To find such extensions in your XL Fortran programs before beginning a porting effort, use the **-qlanglvl** option:

```
$ # -qnoobject stops the compiler after parsing all the source,
$ # giving a fast way to check for errors.
$ # Look for anything above the base F77 standard.
$ bgxlf -qnoobject -qlanglvl=77std f77prog.f
  ...
$ # Look for anything above the F90 standard.
$ bgxlf90 -qnoobject -qlanglvl=90std use_in_2000.f
  ...
$ # Look for anything above the F95 standard.
$ bgxlf95 -qnoobject -qlanglvl=95std use_in_2000.f
  ...
```

Related reference:

See -langlvl in the Compiler Reference

See -qport in the Compiler Reference

Mixing data types in statements

The **-qctyplss** option lets you use character constant expressions in the same places that you use typeless constants. The **-qintlog** option lets you use integer expressions where you can use logicals, and vice versa. A kind type parameter must not be replaced with a logical constant even if **-qintlog** is on, nor by a character constant even if **-qctyplss** is on, nor can it be a typeless constant.

Date and time routines

Date and time routines, such as **dtime**, **etime**, and **jdate**, are accessible as Fortran subroutines.

Other libc routines

A number of other popular routines from the **libc** library, such as **flush**, **getenv**, and **system**, are also accessible as Fortran subroutines.

Changing the default sizes of data types

For porting from machines with larger or smaller word sizes, the **-qintsize** option lets you specify the default size for integers and logicals.The **-qrealsize** option lets you specify the default size for reals and complex components.

Name conflicts between your procedures and XL Fortran intrinsic procedures

If you have procedures with the same names as any XL Fortran intrinsic procedures, the program calls the intrinsic procedure. This situation is more likely with the addition of the many new Fortran 90, Fortran 95, Fortran 2003, and Fortran 2008 intrinsic procedures.

If you still want to call your procedure, add explicit interfaces, **EXTERNAL** statements, or **PROCEDURE** statements for any procedures with conflicting names, or use the **-qextern** option when compiling.

Reproducing results from other systems

XL Fortran provides settings through the **-qfloat** option that help make floating-point results consistent with those from other IEEE systems; this subject is discussed in “Duplicating the floating-point results of other systems” on page 293.

Chapter 12. Sample Fortran programs

The programs in the topics referenced here are provided as coding examples for XL Fortran.

A number of these samples illustrate various aspects of SMP programming that may be new to many users. If you are new to SMP programming, you should examine these samples to gain a better understanding of the SMP coding style.

You can compile and execute the first program to verify that the compiler is installed correctly and your user ID is set up to execute Fortran programs.

Example 1 - XL Fortran source file

This is an example of an XL Fortran source file

```
      PROGRAM CALCULATE
!
! Program to calculate the sum of up to n values of x**3
! where negative values are ignored.
!
      IMPLICIT NONE
      INTEGER I,N
      REAL SUM,X,Y
      READ(*,*) N
      WRITE(*,*) N
      SUM=0
      DO I=1,N
         READ(*,*) X
         WRITE(*,*) X
         IF (X.GE.0.0) THEN
            Y=X**3
            SUM=SUM+Y
         END IF
      END DO
      WRITE(*,*) 'This is the sum of the positive cubes:',SUM
      END
```

Execution results

Running the program yields the following results:

```
$ a.out
5
37
22
-4
19
6
 This is the sum of the positive cubes:  68376.00000
```

Example 2 - valid C routine source file

This is an example of a valid C routine source file used to execute Fortran test subroutines.

```
/*
 * *******************************************************************
 * This is a main function that creates threads to execute the Fortran
 * test subroutines.
 * ********************************************************************
 */
#include <pthread.h>
#include <stdio.h>
#include <errno.h>


extern char *optarg;
extern int optind;

static char *prog_name;

#define MAX_NUM_THREADS 100

void *f_mt_exec(void *);
void f_pre_mt_exec(void);
void f_post_mt_exec(int *);

void
usage(void)
{
    fprintf(stderr, "Usage: %s -t number_of_threads.\n", prog_name);
    exit(-1);
}

main(int argc, char *argv[])
{
    int i, c, rc;
    int num_of_threads, n[MAX_NUM_THREADS];
    char *num_of_threads_p;
    pthread_attr_t attr;
    pthread_t tid[MAX_NUM_THREADS];

    prog_name = argv[0];
    while ((c = getopt(argc, argv, "t")) != EOF)
    {
        switch (c)
        {
        case 't':
            break;

        default:
            usage();
            break;
        }
    }
    argc -= optind;
    argv += optind;
    if (argc < 1)
    {
        usage();
    }

    num_of_threads_p = argv[0];
    if ((num_of_threads = atoi(num_of_threads_p)) == 0)
    {
        fprintf(stderr,
         "%s: Invalid number of threads to be created <\n", prog_name,
                num_of_threads_p);
        exit(1);
    }
    else if (num_of_threads > MAX_NUM_THREADS)
    {
        fprintf(stderr,
                "%s: Cannot create more than 100 threads.\n", prog_name);
        exit(1);
    }
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    /* ****************************************************************
     * Execute the Fortran subroutine that prepares for multi-threaded
     * execution.
     * ****************************************************************
     */
    f_pre_mt_exec();

    for (i = 0; i < num_of_threads; i++)
    {
        n[i] = i;
        rc = pthread_create(&tid[i], &attr, f_mt_exec, (void *)&n[i]);
        if (rc != 0)
        {
            fprintf(stderr, "Failed to create thread %d.\n", i);

            exit(1);
        }
    }
    /* The attribute is no longer needed after threads are created. */
    pthread_attr_destroy(&attr);
    for (i = 0; i < num_of_threads; i++)
    {
        rc = pthread_join(tid[i], NULL);
        if (rc != 0)
        {
            fprintf(stderr, "Failed to join thread %d. \n", i);

        }
    }
    /*
     * Execute the Fortran subroutine that does the check after
     * multi-threaded execution.
     */
    f_post_mt_exec(&num_of_threads);

    exit(0);
}

! ***********************************************************************
! This test case tests the writing list-directed to a single external
! file by many threads.
! ***********************************************************************

        subroutine f_pre_mt_exec()
        integer array(1000)
        common /x/ array

        do i = 1, 1000
          array(i) = i
        end do

        open(10, file="fun10.out", form="formatted", status="replace")
        end
        subroutine f_post_mt_exec(number_of_threads)
        integer array(1000), array1(1000)
        common /x/ array

        close(10)
        open(10, file="fun10.out", form="formatted")
        do j = 1, number_of_threads
          read(10, *) array1

          do i = 1, 1000
            if (array1(i) /= array(i)) then
              print *, "Result is wrong."
              stop
            endif
          end do
        end do
        close(10, status="delete")
        print *, "Normal ending."
        end

        subroutine f_mt_exec(thread_number)
        integer thread_number
        integer array(1000)
        common /x/ array

        write(10, *) array
        end
```

Example 3 - valid Fortran SMP source file

This is an example of a valid Fortran SMP source file used to calculate the value of pi.

```
!****************************************************************
!* This example uses a PARALLEL construct and a DO construct     *
!* to calculate the value of pi.                                 *
!****************************************************************
      program compute_pi
      integer n, i
      real*8 w, x, pi, f, a
      f(a) = 4.d0 /(1.d0 + a*a)  !! function to integrate

      pi = 0.0d0
!$OMP PARALLEL private(x, w, n), shared(pi)
      n = 10000                    !! number of intervals
      w = 1.0d0/n                  !! calculate the interval size
!$OMP DO reduction(+: pi)
      do i = 1, n
         x = w * (i - 0.5d0)
         pi = pi + f(x)
      enddo
!$OMP END DO
!$OMP END PARALLEL
      print *, "Computed pi = ", pi
     end
```

Example 4 - invalid Fortran SMP source file

This is an example of an invalid Fortran SMP source file.

```
!****************************************************************
!* In this example, fort_sub is invoked by multiple threads.     *
!*                                                               *
!* This example is not valid because                             *
!*  fort_sub and another_sub both declare /block/ to be          *
!*  THREADPRIVATE. They intend to share the common block, but    *
!*  they are executed via different threads.                     *
!*                                                               *
!* To "fix" this problem, one of the following approaches can    *
!* be taken:                                                     *
!*  (1) The code for another_sub should be brought into the loop.*
!*  (2) "j" should be passed as an argument to another_sub, and  *
!*       the declaration for /block/ should be removed from      *
!*       another_sub.                                            *
!*  (3) The loop should be marked as "do not parallelize" by     *
!*      using the directive "!$OMP PARALLEL DO  IF(.FALSE.)".    *
```

```
!*****************************************************************

subroutine fort_sub()

  common /block/ j
  integer :: j
  !$OMP THREADPRIVATE(/block/)     ! Each thread executing fort_sub
                                    ! obtains its own copy of /block/.
  integer a(10)

  ...
  !$OMP PARALLEL DO
  do index = 1,10
    call another_sub(a(i))
  enddo
  ...

end subroutine fort_sub

subroutine another_sub(aa)          ! Multiple threads are used to
  integer aa                        ! execute another_sub.
  common /block/ j                  ! Each thread obtains a new copy
  integer :: j                      ! of the common block /block/.
  !$OMP THREADPRIVATE(/block/)

  aa = j                            ! The value of "j" is undefined.
end subroutine another_sub
```

Programming examples using the Pthreads library module

These examples demonstrate the use of the Pthreads library module.

```
!******************************************************************
!* Example 5 : Create a thread with Round_Robin scheduling policy.*
!* For simplicity, we do not show any codes for error checking,   *
!* which would be necessary in a real program.                    *
!******************************************************************
        use, intrinsic::f_pthread
        integer(4) ret_val
        type(f_pthread_attr_t) attr
        type(f_pthread_t)      thr

        ret_val = f_pthread_attr_init(attr)
        ret_val = f_pthread_attr_setschedpolicy(attr, SCHED_RR)
        ret_val = f_pthread_attr_setinheritsched(attr, PTHREAD_EXPLICIT_SCHED)
        ret_val = f_pthread_create(thr, attr, FLAG_DEFAULT, ent, integer_arg)
        ret_val = f_pthread_attr_destroy(attr)
        ......
```

Before you can manipulate a pthread attribute object, you need to create and initialize it. The appropriate interfaces must be called to manipulate the attribute objects. A call to **f_pthread_attr_setschedpolicy** sets the scheduling policy attribute to Round_Robin. Note that this does not affect newly created threads that inherit the scheduling property from the creating thread. For these threads, we explicitly call **f_pthread_attr_setinheritsched** to override the default inheritance attribute. The rest of the code is self-explanatory.

```
!*****************************************************************
!* Example 6 : Thread safety                                     *
!* In this example, we show that thread safety can be achieved   *
!* by using the push-pop cleanup stack for each thread. We       *
!* assume that the thread is in deferred cancellability-enabled  *
!* state.  This means that any thread-cancel requests will be    *
!* put on hold until a cancellation point is encountered.       *
!* Note that f_pthread_cond_wait provides a                      *
!* cancellation point.                                           *
```

```
!******************************************************************
        use, intrinsic::f_pthread
        integer(4) ret_val
        type(f_pthread_mutex_t) mutex
        type(f_pthread_cond_t) cond
        pointer(p, byte)
        ! Initialize mutex and condition variables before using them.
        ! For global variables this should be done in a module, so that they
        ! can be used by all threads. If they are local, other threads
        ! will not see them. Furthermore, they must be managed carefully
        ! (for example, destroy them before returning, to avoid dangling and
        ! undefined objects).
        mutex = PTHREAD_MUTEX_INITIALIZER
        cond  = PTHREAD_COND_INITIALIZER

        ......
        ! Doing something

        ......

        ! This thread needs to allocate some memory area used to
        ! synchronize with other threads. However, when it waits on a
        ! condition variable, this thread may be canceled by another
        ! thread. The allocated memory may be lost if no measures are
        ! taken in advance. This will cause memory leakage.

        ret_val = f_pthread_mutex_lock(mutex)
        p = malloc(%val(4096))

        ! Check condition. If it is not true, wait for it.
        ! This should be a loop.

        ! Since memory has been allocated, cleanup must be registered
        ! for safety during condition waiting.

        ret_val = f_pthread_cleanup_push(mycleanup, FLAG_DEFAULT, p)
        ret_val = f_pthread_cond_wait(cond, mutex)

        ! If this thread returns from condition waiting, the cleanup
        ! should be de-registered.

        call f_pthread_cleanup_pop(0)      ! not execute
        ret_val = f_pthread_mutex_unlock(mutex)

        ! This thread will take care of p for the rest of its life.
        ......


        ! mycleanup looks like:

        subroutine mycleanup(passed_in)
            pointer(passed_in, byte)
            external free

            call free(%val(passed_in))
        end subroutine mycleanup
```

Notices

Trademarks and service marks

Index

Product Number: 5799-AH1

Printed in USA

SC14-7370-00