IBM XL C/C++ for Blue Gene/Q, V12.1

Optimization and Programming Guide

Version 12.1

SC14-7365-00

IBM XL C/C++ for Blue Gene/Q, V12.1

Optimization and Programming Guide

Version 12.1

SC14-7365-00

Note

Before using this information and the product it supports, read the information in “Notices” on page 113.

First edition

This edition applies to IBM XL C/C++ for Blue Gene/Q, V12.1 (Program 5799-AG1) 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 guide discusses advanced topics related to the use of the IBM® XL C/C++ for Blue Gene®/Q, V12.1 compiler, with a particular focus on program portability and optimization. The guide provides both reference information and practical tips for getting the most out of the compiler's capabilities, through recommended programming practices and compilation procedures.

Who should read this information

This document is addressed to programmers building complex applications, who already have experience compiling with XL C/C++, and would like to take further advantage of the compiler's capabilities for program optimization and tuning, support for advanced programming language features, and add-on tools and utilities.

How to use this information

This document uses a "task-oriented" approach to presenting the topics, by concentrating on a specific programming or compilation problem in each section. Each topic contains extensive cross-references to the relevant sections of the reference guides in the IBM XL C/C++ for Blue Gene/Q, V12.1 documentation set, which provide detailed descriptions of compiler options and pragmas, and specific language extensions.

How this information is organized

This guide includes these topics:

- Chapter 1, “Porting from 32-bit to 64-bit mode,” on page 1 discusses common problems that arise when porting existing 32-bit applications to 64-bit mode, and provides recommendations for avoiding these problems.
- Chapter 2, “Using XL C/C++ with Fortran,” on page 5 discusses considerations for calling Fortran code from XL C/C++ programs.
- Chapter 3, “Aligning data,” on page 9 discusses the different compiler options available for controlling the alignment of data in aggregates, such as structures and classes, on all platforms.
- Chapter 4, “Handling floating-point operations,” on page 15 discusses options available for controlling the way floating-point operations are handled by the compiler.
- Chapter 5, “Using C++ constructors,” on page 19 discusses delegating constructors that can concentrate common initializations in one constructor.
- Chapter 6, “Using C++ templates,” on page 21 discusses the different options for compiling programs that include C++ templates.
- Chapter 7, “Constructing a library,” on page 29 discusses how to compile and link static and shared libraries, and how to specify the initialization order of static objects in C++ programs.
- Chapter 8, “Optimizing your applications,” on page 37 discusses the various options provided by the compiler for optimizing your programs, and provides recommendations for use of the different options.

- Chapter 9, "Debugging optimized code," on page 55 discusses the potential usability problems of the optimized programs and the options that can be used to debug the optimized code.
- Chapter 11, "Coding your application to improve performance," on page 65 discusses recommended programming practices and coding techniques for enhancing program performance and compatibility with the compiler's optimization capabilities.
- Chapter 12, "Using the high performance libraries," on page 79 discusses two performance libraries that are shipped with XL C/C++: the Mathematical Acceleration Subsystem (MASS), which contains tuned versions of standard math library functions; and the Basic Linear Algebra Subprograms (BLAS), which contains basic functions for matrix multiplication.
- Chapter 13, "Parallelizing your programs," on page 93 provides an overview of the different options offered by the XL C/C++ for creating multi-threaded programs, including OpenMP language constructs.

Conventions

Typographical conventions

The following table shows the typographical conventions used in the IBM XL C/C++ for Blue Gene/Q, V12.1 information.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options, and directives.	The compiler provides basic invocation commands, **bgxlc** and **bgxlC** (**bgxlc++**), along with several other compiler invocation commands to support various C/C++ 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.
underlining	The default setting of a parameter of a compiler option or directive.	nomaf | maf
`monospace`	Programming keywords and library functions, compiler builtins, examples of program code, command strings, or user-defined names.	To compile and optimize myprogram.c, enter: `bgxlc myprogram.c -O3`.

Qualifying elements (icons)

Most features described in this information apply to both C and C++ languages. In descriptions of language elements where a feature is exclusive to one language, or where functionality differs between languages, this information uses icons to delineate segments of text as follows:

Table 2. Qualifying elements

Qualifier/Icon	**Meaning**
C only, or C only begins ▶ C C ◀ C only ends	The text describes a feature that is supported in the C language only; or describes behavior that is specific to the C language.
C++ only, or C++ only begins ▶ C++ C++ ◀ C++ only ends	The text describes a feature that is supported in the C++ language only; or describes behavior that is specific to the C++ language.
IBM extension begins ▶ IBM IBM ◀ IBM extension ends	The text describes a feature that is an IBM extension to the standard language specifications.
C1X, or C1X begins ▶ C1X C1X ◀ C1X ends	The text describes a feature that is introduced into standard C as part of C1X.
C++0x, or C++0x begins ▶ C++0x C++0x ◀ C++0x ends	The text describes a feature that is introduced into standard C++ as part of C++0x.

Syntax diagrams

Throughout this information, diagrams illustrate XL C/C++ 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.

- Required items are shown on the horizontal line (the main path):

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

- Optional items are shown below the main path:

```
►►──keyword──┬─────────────────────┬──────────────────────────────────►◄
             └─optional_argument───┘
```

- 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.

```
►►──keyword──┬─required_argument1─┬───────────────────────────────────►◄
             └─required_argument2─┘
```

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

```
►►──keyword──┬────────────────────┬───────────────────────────────────►◄
             ├─optional_argument1─┤
             └─optional_argument2─┘
```

- 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:

```
             ┌─,──────────────────┐
             ▼                    │
►►──keyword────repeatable_argument─┴──────────────────────────────────►◄
```

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

```
             ┌─default_argument───┐
►►──keyword──┴─alternate_argument─┴───────────────────────────────────►◄
```

- 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 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 syntax diagram example shows the syntax for the **#pragma comment** directive.

```
   (1)   (2)        (3)         (4)     (5)                                                   (9)  (10)
►►─────#────────pragma────────comment──────(──┬─compiler──────────────────────────────────────────┬─)─────────────►◄
                                              ├─date──────────────────────────────────────────────┤
                                              ├─timestamp─────────────────────────────────────────┤
                                              │                   (6)                             │
                                              └─┬─copyright─┬──────────┬────────────────────────────┬─┘
                                                └─user──────┘          │  (7)                  (8)  │
                                                                       └─,──────"─token_sequence─"──┘
```

Notes:

1 This is the start of the syntax diagram.

2 The symbol # must appear first.

3 The keyword pragma must appear following the # symbol.

4 The name of the pragma `comment` must appear following the keyword `pragma`.

5 An opening parenthesis must be present.

6 The comment type must be entered only as one of the types indicated: `compiler`, `date`, `timestamp`, `copyright`, or `user`.

7 A comma must appear between the comment type `copyright` or `user`, and an optional character string.

8 A character string must follow the comma. The character string must be enclosed in double quotation marks.

9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

The following examples of the **#pragma comment** directive are syntactically correct according to the diagram shown above:

```
#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

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.

Related information

The following sections provide related information for XL C/C++:

IBM XL C/C++ information

XL C/C++ 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 C/C++ 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 C/C++ for Blue Gene/Q, V12.1 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 C/C++ for Blue Gene/Q, V12.1 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/vacpp/bg/12.1/doc/en_US/pdf/ directory. The PDF files are also available on the web at http://www.ibm.com/software/awdtools/xlcpp/features/bg/library/.

  The following files comprise the full set of XL C/C++ product information:

Table 3. XL C/C++ PDF files

Document title	PDF file name	Description
IBM XL C/C++ for Blue Gene/Q, V12.1 Installation Guide, GC14-7362-00	install.pdf	Contains information for installing XL C/C++ and configuring your environment for basic compilation and program execution.
Getting Started with IBM XL C/C++ for Blue Gene/Q, V12.1, GC14-7361-00	getstart.pdf	Contains an introduction to the XL C/C++ product, with information on setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL C/C++ for Blue Gene/Q, V12.1 Compiler Reference, GC14-7363-00	compiler.pdf	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions, including those used for parallel processing.
IBM XL C/C++ for Blue Gene/Q, V12.1 Language Reference, GC14-7364-00	langref.pdf	Contains information about the C and C++ programming languages, as supported by IBM, including language extensions for portability and conformance to nonproprietary standards.
IBM XL C/C++ for Blue Gene/Q, V12.1 Optimization and Programming Guide, SC14-7365-00	proguide.pdf	Contains information on advanced programming topics, such as application porting, interlanguage calls with Fortran code, library development, application optimization and parallelization, and the XL C/C++ 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 C/C++ including IBM Redbooks® publications, white papers, tutorials, and other articles, is available on the web at:

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

For more information about boosting performance, productivity, and portability, see the C/C++ café at http://www.ibm.com/software/rational/cafe/community/ccpp.

Standards and specifications

XL C/C++ 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.

- *Information Technology - Programming languages - C, ISO/IEC 9899:1990*, also known as *C89*.
- *Information Technology - Programming languages - C, ISO/IEC 9899:1999*, also known as *C99*.

- *Information Technology - Programming languages - C++, ISO/IEC 14882:1998*, also known as *C++98*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2003(E)*, also known as *Standard C++*.
- *Information Technology - Programming languages - Extensions for the programming language C to support new character data types, ISO/IEC DTR 19769*. This draft technical report has been accepted by the C standards committee, and is available at http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1040.pdf.
- *Draft Technical Report on C++ Library Extensions, ISO/IEC DTR 19768*. This draft technical report has been submitted to the C++ standards committee, and is available at http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1836.pdf.
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*.
- *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

Other information

- *Using the GNU Compiler Collection* available at http://gcc.gnu.org/onlinedocs

Technical support

Additional technical support is available from the XL C/C++ Support page at http://www.ibm.com/software/awdtools/xlcpp/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 C/C++, visit the product information site at http://www.ibm.com/software/awdtools/xlcpp/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 C/C++ 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 C/C++, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

Chapter 1. Porting from 32-bit to 64-bit mode

Blue Gene/Q supports 64-bit complication mode, which means you can use the XL C/C++ compiler to develop only 64-bit applications. The **-q64** compiler option is enabled by default during complication.

You might want to port existing 32-bit applications to 64-bit mode on Blue Gene/Q platforms. However, this can lead to a number of problems, mostly related to the differences in C/C++ long and pointer data type sizes and alignment between the two modes. The following table summarizes these differences.

Table 4. Size and alignment of data types in 32-bit and 64-bit modes

Data type	**32-bit mode**		**64-bit mode**	
	Size	**Alignment**	**Size**	**Alignment**
long, signed long, unsigned long	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
pointer	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
size_t (defined in the header file <cstddef>)	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
ptrdiff_t (defined in the header file <cstddef>)	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries

The following sections discuss some of the common pitfalls implied by these differences, as well as recommended programming practices to help you avoid most of these issues:
- “Assigning long values” on page 2
- “Assigning pointers” on page 3
- “Aligning aggregate data” on page 4
- “Calling Fortran code” on page 4

When compiling in 32-bit or 64-bit mode, you can use the **-qwarn64** option to help diagnose some issues related to porting applications. In either mode, the compiler immediately issues a warning if undesirable results, such as truncation or data loss, will occur when the program is executed.

For suggestions on improving performance in 64-bit mode, see "Optimize operations in 64-bit mode".

Related information in the *XL C/C++ Compiler Reference*

-q64

-qwarn64

Compile-time and link-time environment variables

Assigning long values

The limits of `long` type integers defined in the `limits.h` standard library header file are different in 32-bit and 64-bit modes, as shown in the following table.

Table 5. Constant limits of long integers in 32-bit and 64-bit modes

Symbolic constant	Mode	Value	Hexadecimal	Decimal
LONG_MIN (smallest signed long)	32-bit	$-(2^{31})$	0x80000000L	–2,147,483,648
	64-bit	$-(2^{63})$	0x8000000000000000L	–9,223,372,036,854,775,808
LONG_MAX (largest signed long)	32-bit	$2^{31}-1$	0x7FFFFFFFL	+2,147,483,647
	64-bit	$2^{63}-1$	0x7FFFFFFFFFFFFFFFL	+9,223,372,036,854,775,807
ULONG_MAX (largest unsigned long)	32-bit	$2^{32}-1$	0xFFFFFFFFUL	+4,294,967,295
	64-bit	$2^{64}-1$	0xFFFFFFFFFFFFFFFFUL	+18,446,744,073,709,551,615

Implications of these differences are:

- Assigning a `long` value to a `double` variable can cause loss of accuracy.
- Assigning constant values to `long` variables can lead to unexpected results. This issue is explored in more detail in "Assigning constant values to long variables."
- Bit-shifting long values will produce different results, as described in "Bit-shifting long values" on page 3.
- Using `int` and `long` types interchangeably in expressions will lead to implicit conversion through promotions, demotions, assignments, and argument passing, and can result in truncation of significant digits, sign shifting, or unexpected results, without warning. These operations can impact performance.

In situations where a `long` value can overflow when assigned to other variables or passed to functions, you must:

- Avoid implicit type conversion by using explicit type casting to change types.
- Ensure that all functions that return long types are properly prototyped.
- Ensure that long parameters can be accepted by the functions to which they are being passed.

Assigning constant values to long variables

Although type identification of constants follows explicit rules in C and C++, many programs use hexadecimal or unsuffixed constants as "typeless" variables and rely on a twos complement representation to exceed the limits permitted on a 32-bit system. As these large values are likely to be extended into a 64-bit `long` type in 64-bit mode, unexpected results can occur, generally at boundary areas such as:

- constant >= `UINT_MAX`
- constant < `INT_MIN`
- constant > `INT_MAX`

Some examples of unexpected boundary side effects are listed in the following table.

Table 6. Unexpected boundary results of constants assigned to long types

Constant assigned to long	Equivalent value	32 bit mode	64 bit mode
–2,147,483,649	INT_MIN–1	+2,147,483,647	–2,147,483,649
+2,147,483,648	INT_MAX+1	–2,147,483,648	+2,147,483,648
+4,294,967,726	UINT_MAX+1	0	+4,294,967,296
0xFFFFFFFF	UINT_MAX	–1	+4,294,967,295
0x100000000	UINT_MAX+1	0	+4,294,967,296
0xFFFFFFFFFFFFFFFF	ULONG_MAX	–1	–1

Unsuffixed constants can lead to type ambiguities that can affect other parts of your program, such as when the results of `sizeof` operations are assigned to variables. For example, in 32-bit mode, the compiler types a number like 4294967295 (`UINT_MAX`) as an unsigned long and `sizeof` returns 4 bytes. In 64-bit mode, this same number becomes a signed long and `sizeof` returns 8 bytes. Similar problems occur when passing constants directly to functions.

You can avoid these problems by using the suffixes `L` (for long constants), `UL` (for unsigned long constants), `LL` (for long long constants), or `ULL` (for unsigned long long constants) to explicitly type all constants that have the potential of affecting assignment or expression evaluation in other parts of your program. In the example cited in the preceding paragraph, suffixing the number as `4294967295U` forces the compiler to always recognize the constant as an `unsigned int` in 32-bit or 64-bit mode. These suffixes can also be applied to hexadecimal constants.

Bit-shifting long values

Left-bit-shifting long values produces different results in 32-bit and 64-bit modes. The examples in Table 7 show the effects of performing a bit-shift on long constants, using the following code segment:

```
long l=valueL<<1;
```

Table 7. Results of bit-shifting long values

Initial value	Symbolic constant	Value after bit shift by one bit	
		32-bit mode	64-bit mode
0x7FFFFFFFL	INT_MAX	0xFFFFFFFE	0x00000000FFFFFFFE
0x80000000L	INT_MIN	0x00000000	0x0000000100000000
0xFFFFFFFFL	UINT_MAX	0xFFFFFFFE	0x00000001FFFFFFFE

In 32-bit mode, `0xFFFFFFFE` is negative. In 64 bit mode, `0x00000000FFFFFFFE` and `0x00000001FFFFFFFE` are both positive.

Assigning pointers

In 64-bit mode, pointers and `int` types are no longer the same size. The implications of this are:

- Exchanging pointers and `int` types causes segmentation faults.
- Passing pointers to a function expecting an `int` type results in truncation.
- Functions that return a pointer, but are not explicitly prototyped as such, return an `int` instead and truncate the resulting pointer.

To avoid these types of problems:

- Prototype any functions that return a pointer, where possible by using the appropriate header file.
- Be sure that the type of parameter you are passing in a function (pointer or `int`) call matches the type expected by the function being called.
- For applications that treat pointers as an integer type, use type `unsigned long` in 64-bit mode.

Aligning aggregate data

Normally, structures are aligned according to the most strictly aligned member in both 32-bit and 64-bit modes. However, since `long` types and pointers change size and alignment in 64-bit, the alignment of a structure's strictest member can change, resulting in changes to the alignment of the structure itself.

Structures that contain pointers or `long` types cannot be shared between 32-bit and 64-bit applications. Unions that attempt to share `long` and `int` types, or overlay pointers onto `int` types can change the alignment. In general, you need to check all but the simplest structures for alignment and size dependencies.

For detailed information on aligning data structures, including structures that contain bit fields, see Chapter 3, “Aligning data,” on page 9.

Calling Fortran code

A significant number of applications use C, C++, and Fortran together, by calling each other or sharing files. It is currently easier to modify data sizes and types on the C side than on the Fortran side of such applications. The following table lists C and C++ types and the equivalent Fortran types in the different modes.

Table 8. Equivalent C/C++ and Fortran data types

C/C++ type	**Fortran type**	
	32-bit	**64-bit**
signed int	INTEGER	INTEGER
signed long	INTEGER	INTEGER*8
unsigned long	LOGICAL	LOGICAL*8
pointer	INTEGER	INTEGER*8
		integer POINTER (8 bytes)

Related information:

Chapter 2, “Using XL C/C++ with Fortran,” on page 5

Chapter 2. Using XL C/C++ with Fortran

With XL C/C++, you can call functions written in Fortran from your C and C++ programs. This section discusses some programming considerations for calling Fortran code in the following areas:

- "Identifiers"
- "Corresponding data types"
- "Character and aggregate data" on page 6
- "Function calls and parameter passing" on page 7
- "Pointers to functions" on page 7
- "Sample program: C/C++ calling Fortran" on page 7 provides an example of a C program which calls a Fortran subroutine.

Related information:

"Calling Fortran code" on page 4

Identifiers

C++ functions callable from Fortran should be declared with `extern "C"` to avoid name mangling. For details, see the appropriate section about options and conventions for mixing Fortran with C/C++ code in the Fortran optimization and programming guide.

You need to follow these recommendations when writing C and C++ code to call functions written in Fortran:

- Avoid using uppercase letters in identifiers. Although XL Fortran folds external identifiers to lowercase by default, the Fortran compiler can be set to distinguish external names by case.
- Avoid using long identifier names. The maximum number of significant characters in XL Fortran identifiers is 250[1].

Note:

1. The Fortran 90 and 95 language standards require identifiers to be no more than 31 characters; the Fortran 2003 standard requires identifiers to be no more than 63 characters.

Corresponding data types

The following table shows the correspondence between the data types available in C/C+ and Fortran. Several data types in C have no equivalent representation in Fortran. Do not use them when programming for interlanguage calls.

Table 9. Correspondence of data types among C, C++ and Fortran

C and C++ data types	Fortran data types
bool (C++)_Bool (C)	LOGICAL(1)
char	CHARACTER
signed char	INTEGER*1
unsigned char	LOGICAL*1
signed short int	INTEGER*2

Table 9. Correspondence of data types among C, C++ and Fortran (continued)

C and C++ data types	Fortran data types
unsigned short int	LOGICAL*2
signed long int	INTEGER*4
unsigned long int	LOGICAL*4
signed long long int	INTEGER*8
unsigned long long int	LOGICAL*8
float	REAL REAL*4
double	REAL*8 DOUBLE PRECISION
long double	REAL*8 DOUBLE PRECISION
float _Complex	COMPLEX*8 or COMPLEX(4)
double _Complex	COMPLEX*16 or COMPLEX(8)
long double _Complex	COMPLEX*16 or COMPLEX(8)
structure or union	derived type
enumeration	INTEGER*4
char[n]	CHARACTER*n
array pointer to type, or type []	Dimensioned variable (transposed)
pointer to function	Functional parameter
structure (with -qalign=packed)	Sequence derived type
vector4double	VECTOR(REAL(8))

Related information in the *XL C/C++ Compiler Reference*

-qldbl128, -qlongdouble

-qalign

Character and aggregate data

Most numeric data types have counterparts across C/C++ and Fortran. However, character and aggregate data types require special treatment:

- C character strings are delimited by a `'\0'` character. In Fortran, all character variables and expressions have a length that is determined at compile time. Whenever Fortran passes a string argument to another routine, it appends a hidden argument that provides the length of the string argument. This length argument must be explicitly declared in C. The C code should not assume a null terminator; the supplied or declared length should always be used.
- An n-element C/C++ array is indexed with `0...n-1`, whereas an n-element Fortran array is typically indexed with `1...n`. In addition, Fortran supports user-specified bounds while C/C++ does not.
- C stores array elements in row-major order (array elements in the same row occupy adjacent memory locations). Fortran stores array elements in ascending storage units in column-major order (array elements in the same column occupy adjacent memory locations). Table 10 on page 7 shows how a two-dimensional array declared by `A[3][2]` in C and by A(3,2) in Fortran, is stored:

Table 10. Storage of a two-dimensional array

Storage unit	C and C++ element name	Fortran element name
Lowest	A[0][0]	A(1,1)
	A[0][1]	A(2,1)
	A[1][0]	A(3,1)
	A[1][1]	A(1,2)
	A[2][0]	A(2,2)
Highest	A[2][1]	A(3,2)

- In general, for a multidimensional array, if you list the elements of the array in the order they are laid out in memory, a row-major array will be such that the rightmost index varies fastest, while a column-major array will be such that the leftmost index varies fastest.

Function calls and parameter passing

Functions must be prototyped identically in both C/C++ and Fortran.

In C, by default, all function arguments are passed by value, and the called function receives a copy of the value passed to it. In Fortran, by default, arguments are passed by reference, and the called function receives the address of the value passed to it. You can use the Fortran %VAL built-in function or the VALUE attribute to pass by value. Refer to the *XL Fortran Language Reference* for more information.

For call-by-reference (as in Fortran), the address of the parameter is passed in a register. When passing parameters by reference, if you write C or C++ functions that call a program written in Fortran, all arguments must be pointers, or scalars with the address operator.

For more information about interlanguage calls to functions or routines, see "Interlanguage calls" in the *XL Fortran Optimization and programming guide*.

Pointers to functions

A function pointer is a data type whose value is a function address. In Fortran, a dummy argument that appears in an EXTERNAL statement is a function pointer. Function pointers are supported in contexts such as the target of a call statement or an actual argument of such a statement.

Sample program: C/C++ calling Fortran

The following example illustrates how program units written in different languages can be combined to create a single program. It also demonstrates parameter passing between C/C++ and Fortran subroutines with different data types as arguments. The example includes the following source files:

- The main program source file: `example.c`
- The Fortran add function source file: `add.f`

Main program source file: `example.c`

```
#include <stdio.h>
extern double add(int *, double [], int *, double []);
```

```
double ar1[4]={1.0, 2.0, 3.0, 4.0};
double ar2[4]={5.0, 6.0, 7.0, 8.0};

main()
{
int x, y;
double z;

x = 3;
y = 3;


z = add(&x, ar1, &y, ar2); /* Call Fortran add routine */
/* Note: Fortran indexes arrays 1..n */
/* C indexes arrays 0..(n-1) */

printf("The sum of %1.0f and %1.0f is %2.0f \n",
ar1[x-1], ar2[y-1], z);
}
```

Fortran add function source file: `add.h`

```
REAL*8 FUNCTION ADD (A, B, C, D)
REAL*8 B,D
INTEGER*4 A,C
DIMENSION B(4), D(4)
ADD = B(A) + D(C)
RETURN
END
```

Compile the main program and Fortran add function source files as follows:

```
xlc -c example.c
xlf -c add.f
```

Link the object files from compile step to create executable `add`:

```
xlc -o add example.o add.o
```

Execute binary:

```
./add
```

The output is as follows:

```
The sum of 3 and 7 is 10
```

Chapter 3. Aligning data

XL C/C++ provides many mechanisms for specifying data alignment at the levels of individual variables, members of aggregates, entire aggregates, and entire compilation units. If you are porting applications between different platforms, or between 32-bit and 64-bit modes, you need to take into account the differences between alignment settings available in the different environments, to prevent possible data corruption and deterioration in performance. In particular, vector types have special alignment requirements which, if not followed, can produce incorrect results.

XL C/C++ provides alignment modes and alignment modifiers for specifying data alignment. Using alignment modes, you can set alignment defaults for all data types for a compilation unit (or subsection of a compilation unit), by specifying a predefined suboption.

Using alignment modifiers, you can set the alignment for specific variables or data types within a compilation unit, by specifying the exact number of bytes that should be used for the alignment.

"Using alignment modes" discusses the default alignment modes for all data types on the different platforms and addressing models; the suboptions and pragmas you can use to change or override the defaults; and rules for the alignment modes for simple variables, aggregates, and bit fields.

"Using alignment modifiers" on page 12 discusses the different specifiers, pragmas, and attributes you can use in your source code to override the alignment mode currently in effect, for specific variable declarations. It also provides the rules governing the precedence of alignment modes and modifiers during compilation.

Using alignment modes

Each data type supported by XL C/C++ is aligned along byte boundaries according to platform-specific default alignment modes. On Blue Gene/Q, the default alignment mode is **linuxppc**.

You can change the default alignment mode, by using any of the following mechanisms:

- Set the alignment mode for all variables in a single file or multiple files during compilation

  To use this approach, you specify the **-qalign** compiler option during compilation, with the **linuxppc** (default) or **bit_packed** suboption.
- Set the alignment mode for all variables in a section of source code

  To use this approach, you specify the **#pragma align** or **#pragma options align** directives in the source files, with the **linuxppc** (default), **bit_packed**, or **reset** suboption. Each directive changes the alignment mode in effect for all variables that follow the directive until another directive is encountered, or until the end of the compilation unit.

Each of the valid alignment modes is defined in Table 11 on page 10, which provides the alignment value, in bytes, for scalar variables, for all data types.

Table 11. Alignment settings (values given in bytes)

		Valid alignment modes	
Data type	**Storage**	**linuxppc**	**bit_packed**
_Bool (C), bool (C++)	1	1	1
char, signed char, unsigned char	1	1	1
wchar_t (64-bit mode)	4	4	1
int, unsigned int	4	4	1
short int, unsigned short int	2	2	1
long int, unsigned long int (64-bit mode)	8	8	1
long long	8	8	1
float	4	4	1
double	8	8	1
long double	8	8	1
long double with **-qldbl128**	16	16	1
pointer (64-bit mode)	8	8	1
vector type (`vector4double`)	32	32	32

If you generate data with an application on one platform and read the data with an application on another platform, it is recommended that you use the **bit_packed** mode, which results in equivalent data alignment on all platforms.

Note: On Blue Gene/Q, the `vector4double` data type in a bit-packed structure cannot be aligned.

"Alignment of aggregates" discusses the rules for the alignment of entire aggregates and provide examples of aggregate layouts. "Alignment of bit-fields" on page 11 discusses additional rules and considerations for the use and alignment of bit fields, and provides an example of bit-packed alignment.

Related information in the *XL C/C++ Compiler Reference*

-qalign

-qldbl128, -qlongdouble

#pragma options

Alignment of aggregates

The data contained in Table 11 (in "Using alignment modes" on page 9) apply to scalar variables, and variables that are members of aggregates such as structures, unions, and classes. The following rules apply to aggregate variables, namely structures, unions or classes, as a whole (in the absence of any modifiers):

- For all alignment modes, the size of an aggregate is the smallest multiple of its alignment value that can encompass all of the members of the aggregate.
- ▶ C Empty aggregates are assigned a size of 0 bytes. As a result, two distinct variables might have the same address.

- C++ Empty aggregates are assigned a size of 1 byte. Note that static data members do not participate in the alignment or size of an aggregate; therefore a structure or class containing only a single static data member has a size of 1 byte.
- For all alignment modes, the alignment of an aggregate is equal to the largest alignment value of any of its members. With the exception of packed alignment modes, members whose natural alignment is smaller than that of their aggregate's alignment are padded with empty bytes.
- Aligned aggregates can be nested, and the alignment rules applicable to each nested aggregate are determined by the alignment mode that is in effect when a nested aggregate is declared.

Notes:

- C++ The C++ compiler might generate extra fields for classes that contain base classes or virtual functions. Objects of these types might not conform to the usual mappings for aggregates.
- The alignment of an aggregate must be the same in all compilation units. For example, if the declaration of an aggregate is in a header file and you include that header file into two distinct compilations units, choose the same alignment mode for both compilations units.

For rules on the alignment of aggregates containing bit fields, see "Alignment of bit-fields."

Alignment of bit-fields

You can declare a bit-field as a `_Bool` (C), `bool` (C++), `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, or `unsigned long long` data type. The alignment of a bit-field depends on its base type and the compilation mode (64-bit for Blue Gene/Q).

C The length of a bit-field cannot exceed the length of its base type. In extended mode, you can use the `sizeof` operator on a bit-field. The `sizeof` operator on a bit-field always returns the size of the base type.

C++ The length of a bit-field can exceed the length of its base type, but the remaining bits are used to pad the field, and do not actually store any value.

However, alignment rules for aggregates containing bit-fields are different depending on the alignment mode in effect. These rules are described below.

Rules for bit-packed alignment

- Bit-fields have an alignment of 1 byte, and are packed with no default padding between bit-fields.
- A zero-length bit-field causes the next member to start at the next byte boundary. If the zero-length bit-field is already at a byte boundary, the next member starts at this boundary. A non-bit-field member that follows a bit-field is aligned on the next byte boundary.

Example of bit-packed alignment

```
#pragma options align=bit_packed
struct {
   int a : 8;
   int b : 10;
   int c : 12;
   int d : 4;
   int e : 3;
   int : 0;
   int f : 1;
   char g;
   } A;

pragma options align=reset
```

The size of A is 7 bytes. The alignment of A is 1 byte. The layout of A is:

Member name	Byte offset	Bit offset
a	0	0
b	1	0
c	2	2
d	3	6
e	4	2
f	5	0
g	6	0

Using alignment modifiers

XL C/C++ also provides alignment modifiers, with which you can exercise even finer-grained control over alignment, at the level of declaration or definition of individual variables. Available modifiers are:

#pragma pack(...)

Valid application:
The entire aggregate (as a whole) immediately following the directive.

Effect: Sets the maximum alignment of the members of the aggregate to which it applies, to a specific number of bytes. Also allows a bit-field to cross a container boundary. Used to reduce the effective alignment of the selected aggregate.

Valid values:
When **-qpack_semantic=ibm** is in effect (the default for XL C/C++), **1**, **2**, **4**, **8**, **16**, **nopack**, **pop**, and empty parentheses. The use of empty parentheses has the same functionality as **nopack**. When **-qpack_semantic=gnu** is in effect, **[push,]1**, **[push,]2**, **[push,]4**, **[push,]8**, **[push,]16**, **pop**, and empty parentheses.

__attribute__((aligned(n)))

Valid application:
As a variable attribute, it applies to a single aggregate (as a whole), namely a structure, union, or class; or to an individual member of an aggregate.[1] As a type attribute, it applies to all aggregates declared of that type. If it is applied to a typedef declaration, it applies to all instances of that type.[2]

Effect:
Sets the minimum alignment of the specified variable (or variables), to a specific number of bytes. Typically used to increase the effective alignment of the selected variables.

Valid values:
n must be a positive power of 2, or NIL. NIL can be specified as either `__attribute__((aligned()))` or `__attribute__((aligned))`; this is the same as specifying the maximum system alignment based on object type and scope.

__attribute__((packed))

Valid application:
As a variable attribute, it applies to simple variables, or individual members of an aggregate, namely a structure or class[1]. As a type attribute, it applies to all members of all aggregates declared of that type.

Effect: Sets the maximum alignment of the selected variable, or variables, to which it applies, to the smallest possible alignment value, namely one byte for a variable and one bit for a bit field.

__align(n)

Effect: Sets the minimum alignment of the variable or aggregate to which it applies to a specific number of bytes; also effectively increases the amount of storage occupied by the variable. Used to increase the effective alignment of the selected variables.

Valid application:
Applies to simple static (or global) variables or to aggregates as a whole, rather than to individual members of aggregates, unless these are also aggregates.

Valid values:
n must be a positive power of 2. XL C/C++ also allows you to specify a value greater than the system maximumalignment based on object type and scope.

Notes:

- In a comma-separated list of variables in a declaration, if the modifier is placed at the beginning of the declaration, it applies to all the variables in the declaration. Otherwise, it applies only to the variable immediately preceding it.
- Depending on the placement of the modifier in the declaration of a `struct`, it can apply to the definition of the type, and hence applies to all instances of that type; or it can apply to only a single instance of the type. For details, see *Type Attributes* in the *XL C/C++ Language Reference*.

Related information in the *XL C/C++ Compiler Reference*

#pragma pack

-qpack_semantic

Related information in the *XL C/C++ Language Reference*

The aligned type attribute (IBM extension)

The packed type attribute (IBM extension)

The __align type qualifier (IBM extension)

Type attributes (IBM extension)

The aligned variable attribute (IBM extension)

The packed variable attribute (IBM extension)

Chapter 4. Handling floating-point operations

The following sections provide reference information, portability considerations, and suggested procedures for using compiler options to manage floating-point operations:

- "Floating-point formats"
- "Handling multiply-add operations"
- "Compiling for strict IEEE conformance" on page 16
- "Handling floating-point constant folding and rounding" on page 16
- "Handling floating-point exceptions" on page 18

Floating-point formats

XL C/C++ supports the following binary floating-point formats:

- 32-bit single precision, with an approximate absolute normalized range of 0 and 10^{-38} to 10^{+38} and precision of about 7 decimal digits
- 64-bit double precision, with an approximate absolute normalized range of 0 and 10^{-308} to 10^{+308} and precision of about 16 decimal digits
- 128-bit extended precision, with slightly greater range than double-precision values, and with a precision of about 32 decimal digits

Note that the `long double` type may represent either double-precision or extended-precision values, depending on the setting of the **-qldbl128** compiler option. The default is 128 bits. For compatibility with older compilations, you can use **-qnoldbl128** if you need `long double` to be 64 bits.

Related information in the *XL C/C++ Compiler Reference*

-qldbl128, -qlongdouble

Handling multiply-add operations

By default, the compiler generates a single non-IEEE 754 compatible multiply-add instruction for binary floating-point expressions such as _a+b*c_, partly because one instruction is faster than two. Because no rounding occurs between the multiply and add operations, this may also produce a more precise result. However, the increased precision might lead to different results from those obtained in other environments, and may cause _x*y-x*y_ to produce a nonzero result. To avoid these issues, you can suppress the generation of multiply-add instructions by using the **-qfloat=nomaf** option.

Related information in the *XL C/C++ Compiler Reference*

-qfloat

Compiling for strict IEEE conformance

By default, XL C/C++ follows most, but not all of the rules in the IEEE standard. If you compile with the **-qnostrict** option, which is enabled by default at optimization level **-O3** or higher, some IEEE floating-point rules are violated in ways that can improve performance but might affect program correctness. To avoid this issue, and to compile for strict compliance with the IEEE standard, use the following options:

- Use the **-qfloat=nomaf** compiler option.
- If the program changes the rounding mode at runtime, use the **-qfloat=rrm** option.
- If the data or program code contains signaling NaN values (NaNS), use the **-qfloat=nans** option. (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** compiler option.)
- If you compile with **-O3**, **-O4**, or **-O5**, include the option **-qstrict** after it.

Related information:

"Advanced optimization" on page 40

Related information in the *XL C/C++ Compiler Reference*

-qfloat

-qstrict

-qinitauto

Handling floating-point constant folding and rounding

By default, the compiler replaces most operations involving constant operands with their result at compile time. This process is known as constant folding. Additional folding opportunities might occur with optimization or with the **-qnostrict** option. The result of a floating-point operation folded at compile time normally produces the same result as that obtained at execution time, except in the following cases:

- The compile-time rounding mode is different from the execution-time rounding mode. By default, both are round-to-nearest; however, if your program changes the execution-time rounding mode, to avoid differing results, do either of the following operations:
  - Change the compile-time rounding mode to match the execution-time mode, by compiling with the appropriate **-y** option. For more information and an example, see "Matching compile-time and runtime rounding modes" on page 17.
  - Suppress folding, by compiling with the **-qfloat=nofold** option.
- Expressions like *a+b*c* are partially or fully evaluated at compile time. The results might be different from those produced at execution time, because *b*c* might be rounded before being added to *a*, while the runtime multiply-add instruction does not use any intermediate rounding. To avoid differing results, do either of the following operations:
  - Suppress the use of multiply-add instructions, by compiling with the **-qfloat=nomaf** option.
  - Suppress folding, by compiling with the **-qfloat=nofold** option.

- An operation produces an infinite or NaN result. Compile-time folding prevents execution-time detection of an exception, even if you compile with the **-qflttrap** option. To avoid missing these exceptions, suppress folding with the **-qfloat=nofold** option.

Related information:

“Handling floating-point exceptions” on page 18

Related information in the *XL C/C++ Compiler Reference*

-qfloat

-qstrict

-qflttrap

Matching compile-time and runtime rounding modes

The default rounding mode used at compile time and run time is round-to-nearest, ties to even. If your program changes the rounding mode at run time, the results of a floating-point calculation might be slightly different from those that are obtained at compile time. The following example illustrates this:

```
#include <float.h>
#include <fenv.h>
#include <stdio.h>

int main ( )
{
 volatile double one = 1.f, three = 3.f;  /* volatiles are not folded */
 double one_third;

 one_third = 1. / 3.;  /* folded */
 printf ("1/3 with compile-time rounding = %.17f\n", one_third);

 fesetround (FE_TOWARDZERO);
 one_third = one / three;  /* not folded */

 printf ("1/3 with execution-time rounding to zero = %.17f\n", one_third);

 fesetround (FE_TONEAREST);
 one_third = one / three;  /* not folded */

 printf ("1/3 with execution-time rounding to nearest = %.17f\n", one_third);

 fesetround (FE_UPWARD);
 one_third = one / three;  /* not folded */

 printf ("1/3 with execution-time rounding to +infinity = %.17f\n", one_third);

 fesetround (FE_DOWNWARD);
 one_third = one / three;  /* not folded */

 printf ("1/3 with execution-time rounding to -infinity = %.17f\n", one_third);

 return 0;
 }
```

When compiled with the default options, this code produces the following results:

```
1/3 with compile-time rounding = 0.33333333333333331
1/3 with execution-time rounding to zero = 0.33333333333333331
1/3 with execution-time rounding to nearest   = 0.33333333333333331
1/3 with execution-time rounding to +infinity = 0.33333333333333337
1/3 with execution-time rounding to -infinity = 0.33333333333333331
```

Because the fourth computation changes the rounding mode to round-to-infinity, the results are slightly different from the first computation, which is performed at compile time, using round-to-nearest. If you do not use the **-qfloat=nofold** option to suppress all compile-time folding of floating-point computations, it is recommended that you use the **-y** compiler option with the appropriate suboption to match compile-time and runtime rounding modes. In the previous example, compiling with **-yp** (round-to-infinity) produces the following result for the first computation:

```
1/3 with compile-time rounding = 0.33333333333333337
```

In general, if the rounding mode is changed to +infinity or -infinity, it is recommended that you also use the **-qfloat=rrm** option.

Related information in the *XL C/C++ Compiler Reference*

-qfloat

-y

Handling floating-point exceptions

By default, invalid operations such as division by zero, division by infinity, overflow, and underflow are ignored at run time. However, you can use the **-qflttrap** option to detect these types of exceptions. 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.

In addition, you can add suitable support code to your program to make program execution continue after an exception occurs, and to modify the results of operations causing exceptions.

Because, however, floating-point computations involving constants are usually folded at compile time, the potential exceptions that would be produced at runtime may not occur. To ensure that the **-qflttrap** option traps all runtime floating-point exceptions, consider using the **-qfloat=nofold** option to suppress all compile-time folding.

Related information in the *XL C/C++ Compiler Reference*

-qfloat

-qflttrap

Chapter 5. Using C++ constructors

▶ C++0x

Before C++0x, common initialization in multiple constructors of the same class could not be concentrated in one place in a robust, maintainable manner. A basic approach can solve this problem:

Using delegating constructors:
With the delegating constructors feature, you can concentrate common initializations in one constructor, which can make program more readable and maintainable. Delegating constructors help reduce the code size and collective size of the object files. For more information, see “Using delegating constructors (C++0x).”

Related information in the *XL C/C++ Compiler Reference*

-qlanglvl

Using delegating constructors (C++0x)

Note: C++0x is a new version of the C++ programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C++0x standard is complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C++0x standard and therefore they should not be relied on as a stable programming interface.

Syntactically, *delegating constructors* and *target constructors* present the same interface as other constructors, see "Delegating constructors (C++0x)" in the *XL C/C++ Language Reference*.

Consider the following points when you use the delegating constructors feature:

- Call the target constructor implementation in such a way that virtual bases, direct nonvirtual bases, class members and additional ABI artifacts are initialized by target constructor as appropriate.
- Respond to the exception thrown in the body of a delegating constructor by calling the destructor implementation on the object that is constructed through the target constructor. The destructor implementation must be called in such a way that it calls the destructors of subobjects as appropriate. In particular, it must call the destructors for virtual base classes if the virtual base classes are created through the target constructor.
- Perform proper construction and destruction when initializing static objects with delegating constructors and on termination of a program that does such initialization.
- When an exception is thrown, a corresponding destructor must be called. Otherwise, virtual bases may have their destructors called more than once or not at all. With a delegating constructor, the call to the target constructor does not necessarily match a specific destructor implementation.

- The feature has minimal impact on compile time and run time performance. However, use of default arguments with an existing constructor is recommended in place of a delegating constructor where possible. Without inlining and interprocedural analysis, run time performance may degrade because of function call overhead and increased opacity.

Related information in the *XL C/C++ Compiler Reference*

-qlanglvl

Related information in the *XL C/C++ Language Reference*

Delegating constructors (C++0x)

Chapter 6. Using C++ templates

In C++, you can use a template to declare a set of related:

- Classes (including structures)
- Functions
- Static data members of template classes

Reducing redundant template instantiations

Within an application, you can instantiate the same template multiple times with the same arguments or with different arguments. If you use the same arguments, the repeated instantiations are redundant. These redundant instantiations increase compilation time, increase the size of the executable, and deliver no benefit.

There are several basic approaches to the problem of redundant instantiations:

Handling redundancy during linking

The size increase of the final executable might be small enough that it does not justify changing the way you compile your program or modify the source file. Most linkers have some form of garbage collection functionality. On Blue Gene/Q, the linker performs garbage collection well, especially when you use the **-qfuncsect** option. If you use **-qtmplinst=always** or **-qtmplinst=auto** without using **-qtemplateregistry** or **-qtempinc**, no compile time management of redundant instantiations is done. In this case, you can use the **-qfuncsect** option to reduce the executable size. For details, see **-qfuncsect** in the *XL C/C++ Compiler Reference*.

Controlling implicit instantiation in the source code

Concentrating implicit instantiations of a specialization: Organize your source code so that object files contain fewer instances of each required instantiation and fewer unused instantiations. This is the least usable approach, because you must know where each template is defined, which instantiations are used, and where to declare an explicit instantiation for each instantiation.

▶ C++0x **Using explicit instantiation declarations:** With the explicit instantiation declarations feature, you can suppress the implicit instantiation of a template specialization or its members. This helps reduce the collective size of the object files. It might also reduce the size of the final executable if the suppressed symbol definitions are meant to be found in a shared library, or if the system linker is unable to always remove additional definitions of a symbol. For more information, see "Using explicit instantiation declarations (C++0x)" on page 26.

Note: If you want to control implicit instantiation in the source code, or use explicit instantiation declarations, you can use the **-qtmplinst=none** or **-qtmplinst=noinlines** option to prevent accidental implicit instantiations from occurring.

Having the compiler store instantiation information in a registry

Use the **-qtemplateregistry** compiler option. Information about each template instantiation is stored in a template registry. If the compiler is asked to instantiate the same template again with the same arguments, it points to the instantiation in the first object file instead. This approach is described in “Using the -qtemplateregistry compiler option” on page 24.

Having the compiler store instantiations in a template include directory

Use the **-qtempinc** compiler option. If the template definition and implementation files have the required structure, each template instantiation is stored in a template include directory. If the compiler is asked to instantiate the same template again with the same arguments, it uses the stored version instead. The source file created in the template include directory is compiled during the link step recursively until all instantiations are done. This approach is described in “Using the -qtempinc compiler option.”

Notes:

- The **-qtempinc** and **-qtemplateregistry** compiler options are mutually exclusive.
- **-qtemplateregistry** is a better approach than **-qtempinc** for the following reasons:
  - **-qtemplateregistry** provides better benefits than **-qtempinc**.
  - **-qtemplateregistry** does not require modifications to the header files.

The compiler generates code for an implicit instantiation unless one of the following conditions is true:

- You use either **-qtmplinst=none** or **-qtmplinst=noinlines**.
- You use **-qtmplinst=auto**, which is the default suboption of **-qtmplinst** with **-qnotemplateregistry**.
- You use **-qtmplinst=auto** with **-qtempinc** and the template source that is organized to use **-qtempinc**.
- ▶ C++0x An explicit instantiation declaration for that instantiation is in the current translation unit.

Related information in the *XL C/C++ Compiler Reference*

-qtempinc (C++ only)

-qtemplateregistry (C++ only)

-qtmplinst (C++ only)

-qlanglvl

Using the -qtempinc compiler option

To use **-qtempinc**, you must structure your application as follows:

- Declare your class templates and function templates in template declaration files, with a `.h` extension.
- For each template declaration file, create a template implementation file. This file must have the same file name as the template declaration file and an extension of `.c` or `.t`, or the name must be specified in a **#pragma implementation** directive. For a class template, the implementation file defines the member functions and static data members. For a function template, the implementation file defines the function.
- In your source program, specify an `#include` directive for each template declaration file.
- Optionally, to ensure that your code is applicable for both **-qtempinc** and **-qnotempinc** compilations, in each template declaration file, conditionally

include the corresponding template implementation file if the __TEMPINC__ macro is *not* defined. (This macro is automatically defined when you use the **-qtempinc** compilation option.) This produces the following results:

- Whenever you compile with **-qnotempinc**, the template implementation file is included.
- Whenever you compile with **-qtempinc**, the compiler does not include the template implementation file. Instead, the compiler looks for a file with the same name as the template implementation file and extension `.c` the first time it needs a particular instantiation. If the compiler subsequently needs the same instantiation, it uses the copy stored in the template include directory.

Note: You can also use **-qtemplateregistry** that provides more benefits than **-qtempinc**, and does not require modifications to your source files. For details, see "-qtemplateregistry (C++ only)" in the *XL C/C++ Compiler Reference*.

Related information in the *XL C/C++ Compiler Reference*

-qtempinc (C++ only)

-qtemplateregistry (C++ only)

-qtmplinst (C++ only)

#pragma implementation (C++ only)

Example of using -qtempinc

The following example shows how the compiler manages implicit instantiation of a template when the template declaration and definition are in separate files. This example includes the following source files:

- The template declaration file: `a.h`
- The corresponding template implementation file: `a.t`
- The main program source file: `a.cpp`

Template declaration file: `a.h`

```
struct IC {
  virtual void myfunc() = 0;
};

template <class T> struct C : public IC{
  virtual void myfunc();
};

#ifndef __TEMPINC__
  #include "a.t"
#else
  #pragma implementation("a.t")
#endif
```

Template implementation file : `a.t`

This file contains the template definition of `myfunc` function, which is called from the `main` program.

```
template <class T> void C<T>::myfunc() {}
```

Main program file: `a.cpp`

This file creates an object that requires an implicit instantiation.

```
#include "a.h"

int main() {
  IC* pIC = new C<int>();
  pIC->myfunc();
}
```

You can use the following command to compile the main program `a.cpp`:

```
xlC -qtempinc a.cpp
```

Notes:

- If **-qnotempinc** is specified, the template implementation file is included; otherwise, if **-qtempinc** is specified, the `#pragma implementation` directive instructs the compiler path to the template implementation file.
- Compiler searches for `a.c` as the template implementation file by default, if the `#pragma implementation` directive is not specified.

Regenerating the template instantiation file

The compiler builds a template instantiation file in the TEMPINC directory corresponding to each template implementation file. With each compilation, the compiler can add information to the file but it never removes information from the file.

As you develop your program, you might remove template function references or reorganize your program so that the template instantiation files become obsolete. You can periodically delete the TEMPINC destination and recompile your program.

Using -qtempinc with shared libraries

In a traditional application development environment, different applications can share both source files and compiled files. When you use templates, applications can share source files but cannot share compiled files.

If you use **-qtempinc**:

- Each application must have its own TEMPINC destination.
- You must compile all of the source files for the application, even if some of the files have already been compiled for another application.

Using the -qtemplateregistry compiler option

The template registry uses a "first-come first-served" algorithm:

- When a compiler performs an implicit instantiation for the first time, it is instantiated in the compilation unit in which it occurs.
- When another compilation unit performs the same implicit instantiation, it is not instantiated. Thus, only one copy is generated for the entire program.

The instantiation information is stored in a template registry file. You must use the same template registry file for the entire program. Two programs cannot share a template registry file.

The default file name for the template registry file is `templateregistry`, but you can specify any other valid file name to override this default. When cleaning your program build environment before starting a fresh or scratch build, you must delete the registry file along with the old object files.

You can perform multiple compilations in parallel using the same template registry file with minimal impact on compile time.

When you recompile your program, the information in the template registry file is also used to determine whether a recompilation of a source file might introduce link errors because of missing template instantiations. If the following conditions are true, the compiler will schedule the recompilation of one or more source files when you recompile a source file:

- The source file instantiated a template that other source files instantiated.
- The source file was chosen by the template registry to actually instantiate the template.
- The source file no longer instantiates the template.

When the preceding conditions are all true, the compiler chooses another source file to instantiate the template in. That file is scheduled for recompilation during the link step. If you happen to recompile a source file that is scheduled to be recompiled during the link step, the scheduled recompilation is cancelled.

You can use **-qnotemplaterecompile** to disable the scheduled recompilation during the link step. For details, see "-qtemplaterecompile (C++ only)" in the *XL C/C++ Compiler Reference*.

Related information in the *XL C/C++ Compiler Reference*

-qtempinc (C++ only)

-qtemplaterecompile (C++ only)

-qtemplateregistry (C++ only)

Recompiling related compilation units

If two compilation units, A and B, reference the same instantiation, the **-qtemplateregistry** compiler option has the following effect:

- If you compile A first, the object file for A contains the code for the instantiation.
- When you later compile B, the object file for B does not contain the code for the instantiation because object A already does.
- If you later change A so that it no longer references this instantiation, the reference in object B would produce an unresolved symbol error. When you recompile A, the compiler detects this problem and handles it as follows:
  – If the **-qtemplaterecompile** compiler option is in effect, the compiler automatically recompiles B during the link step, using the same compiler options that were specified for A. (Note, however, that if you use separate compilation and linkage steps, you need to include the compilation options in the link step to ensure the correct compilation of B.)
  – If the **-qnotemplaterecompile** compiler option is in effect, the compiler issues a warning and you must manually recompile B.

Related information in the *XL C/C++ Compiler Reference*

-qtemplateregistry (C++ only)

-qtemplaterecompile (C++ only)

Switching from -qtempinc to -qtemplateregistry

Because the **-qtemplateregistry** compiler option does not impose any restrictions on the file structure of your application, it has less administrative overhead than **-qtempinc**. You can make the switch as follows:

- If your application compiles successfully with both **-qtempinc** and **-qnotempinc**, you do not need to make any changes.
- If your application compiles successfully with **-qtempinc** but not with **-qnotempinc**, you must change it so that it will compile successfully with **-qnotempinc**. In each template definition file, conditionally include the corresponding template implementation file if the __TEMPINC__ macro is not defined. This is illustrated in "Example of using -qtempinc" on page 23.

Using explicit instantiation declarations (C++0x)

Note: C++0x is a new version of the C++ programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C++0x standard is complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C++0x standard and therefore they should not be relied on as a stable programming interface.

Syntactically, an *explicit instantiation declaration* is an *explicit instantiation definition* preceded by the `extern` keyword, see "Explicit instantiation (C++ only)" in the *XL C/C++ Language Reference*.

Consider the following points when you use the explicit instantiation declarations feature:

- ▶ IBM An explicit instantiation declaration of a class template specialization does not cause implicit instantiation of said specialization.
- In a translation unit, if a user-defined inline function is subject to an explicit instantiation declaration and not subject to an explicit instantiation definition:
  - Implicit instantiation of said function occurs no matter whether it is inlined or not.
  - ▶ IBM No out-of-line copy of the function is generated in that translation unit no matter whether the compiler option **-qkeepinlines** is specified or not.

  Note: This does not limit the behavior for functions that are implicitly generated by the compiler. Implicitly declared special members such as the default constructor, copy constructor, destructor and copy assignment operator are inline and the compiler might instantiate them. In particular, out-of-line copies might be generated.
- The degradation of the amount of inlining achieved on functions that are not inline and are subject to explicit instantiation declarations might occur.
- When a non-pure virtual member function is subject to an explicit instantiation declaration, either directly or through its class, the virtual member function must be subject to an explicit instantiation definition somewhere in the entire program. Otherwise, an unresolved symbol error might result at link time.
- When implicit instantiation of a class template specialization is allowed, the user program must be written as if the implicit instantiation of all virtual member

functions of that class specialization occurs. Otherwise, an unresolved symbol error for a virtual member function might result at link time.

- When implicit instantiation of a class template specialization is allowed and the specialization is subject to an explicit instantiation declaration, the class template specialization must be subject to an explicit instantiation definition somewhere in the user program. Otherwise, an unresolved symbol error might result at link time.

Related information in the *XL C/C++ Compiler Reference*

-qtempinc (C++ only)

#pragma implementation (C++ only)

-qlanglvl

Related information in the *XL C/C++ Language Reference*

Explicit instantiation (C++ only)

Chapter 7. Constructing a library

You can include static and shared libraries in your C and C++ applications.

“Compiling and linking a library” describes how to compile your source files into object files for inclusion in a library, how to link a library into the main program, and how to link one library into another.

“Initializing static objects in libraries (C++)” on page 30 describes how to use priorities to control the order of initialization of objects across multiple files in a C++ application.

Compiling and linking a library

Related information:

Dynamic and static linking

Compiling a static library

To compile a static library:

1. Compile each source file into an object file, with no linking. For example:
   ```
   bgxlc -c test.c example.c
   ```
2. Use the ar command to add the generated object files to an archive library file. For example:
   ```
   ar -rv libex.a test.o example.o
   ```

Compiling a shared library

To compile a shared library:

1. Compile your source files into an object file, with no linking. Note that in the case of compiling a shared library, the **-qpic** compiler option is also used. For example:
   ```
   bgxlc -qpic -c foo.c
   ```
2. Use the **-qmkshrobj** compiler option to create a shared object from the generated object files. For example:
   ```
   bgxlc -qmkshrobj -o libfoo.so foo.o
   ```
   Related information in the *XL C/C++ Compiler Reference*

   -qpic

   -qmkshrobj

Linking a library to an application

You can use the same command string to link a static or shared library to your main program. For example:

```
bgxlc -o myprogram main.c -Ldirectory [-Rdirectory] -ltest
```

where *directory* is the path to the directory containing the library `libtest.a`.

By using the **-l** option, you instruct the linker to search in the directory specified via the **-L** option (and, for a shared library, the **-R** option) for `libtest.so`; if it is not found, the linker searches for `libtest.a`. For additional linkage options, including options that modify the default behavior, see the operating system **ld** documentation.

Related information in the *XL C/C++ Compiler Reference*

-l

-L

-R

Linking a shared library to another shared library

Just as you link modules into an application, you can create dependencies between shared libraries by linking them together. For example:

```
bgxlc -qmkshrobj -o mylib.so myfile.o -Ldirectory -Rdirectory -ltest
```

Related information in the *XL C/C++ Compiler Reference*

-qmkshrobj

-R

-L

Initializing static objects in libraries (C++)

The C++ language definition specifies that, before the `main` function in a C++ program is executed, all objects with constructors, from all the files included in the program must be properly constructed. Although the language definition specifies the order of initialization for these objects within a file (which follows the order in which they are declared), it does not specify the order of initialization for these objects across files and libraries. You might want to specify the initialization order of static objects declared in various files and libraries in your program.

To specify an initialization order for objects, you assign relative priority numbers to objects. The mechanisms by which you can specify priorities for entire files or objects within files are discussed in “Assigning priorities to objects.” The mechanisms by which you can control the initialization order of objects across modules are discussed in “Order of object initialization across libraries” on page 32.

Related information:

“Assigning priorities to objects”

“Order of object initialization across libraries” on page 32

Assigning priorities to objects

You can assign a priority number to objects and files within a single library, and the objects will be initialized at run time according to the order of priority. However, because of the differences in the way modules are loaded and objects initialized on the different platforms, the levels at which you can assign priorities vary among the different platforms, as follows:

Set the priority level for an entire file

To use this approach, you specify the **-qpriority** compiler option during compilation. By default, all objects within a single file are assigned the same priority level, and are initialized in the order in which they are declared, and terminated in reverse declaration order.

Set the priority level for objects within a file

To use this approach, you include **#pragma priority** directives in the source files. Each **#pragma priority** directive sets the priority level for all objects that follow it, until another pragma directive is specified. Within a file, the first **#pragma priority** directive must have a higher priority number than the number specified in the **-qpriority** option (if it is used), and subsequent **#pragma priority** directives must have increasing numbers. While the relative priority of objects within a single file will remain the order in which they are declared, the pragma directives will affect the order in which objects are initialized across files. The objects are initialized according to their priority, and terminated in reverse priority order.

Set the priority level for individual objects

To use this approach, you use `init_priority` variable attributes in the source files. The `init_priority` attribute takes precedence over **#pragma priority** directives, and can be applied to objects in any declaration order. On Blue Gene/Q, the objects are initialized according to their priority and terminated in reverse priority across compilation units.

Using priority numbers

Priority numbers can range from 101 to 65535. The smallest priority number that you can specify, 101, is initialized first. The largest priority number, 65535, is initialized last. If you do not specify a priority level, the default priority is 65535.

The examples below show how to specify the priority of objects within a single file, and across two files. "Order of object initialization across libraries" on page 32 provides detailed information on the order of initialization of objects.

Example of object initialization within a file

The following example shows how to specify the priority for several objects within a source file.

```
...
#pragma priority(2000) //Following objects constructed with priority 2000
...

static Base a ;

House b ;
...
#pragma priority(3000) //Following objects constructed with priority 3000
...

Barn c ;
...
#pragma priority(2500) // Error - priority number must be larger
                       // than preceding number (3000)
...
#pragma priority(4000) //Following objects constructed with priority 4000
...

Garage d ;
...
```

Example of object initialization across multiple files

The following example describes the initialization order for objects in two files, `farm.C` and `zoo.C`. Both files are contained in the same shared module, and use the **-qpriority** compiler option and **#pragma priority** directives.

farm.C -qpriority=1000

```
...
Dog a ;
Dog b ;
...
#pragma priority(6000)
...
Cat c ;
Cow d ;
...
#pragma priority(7000)
Mouse e ;
...
```

zoo.C -qpriority=2000

```
...

Bear m ;
...
#pragma priority(5000)
...
Zebra n ;
Snake s ;
...
#pragma priority(8000)
Frog f ;
...
```

At runtime, the objects in these files are initialized in the following order:

Sequence	Object	Priority value	Comment
1	Dog a	1000	Takes option priority (1000).
2	Dog b	1000	Follows with the same priority.
3	Bear m	2000	Takes option priority (2000).
4	Zebra n	5000	Takes pragma priority (5000).
5	Snake s	5000	Follows with same priority.
6	Cat c	6000	Next priority number.
7	Cow d	6000	Follows with same priority.
8	Mouse e	7000	Next priority number.
9	Frog f	8000	Next priority number (initialized last).

Related information in the *XL C/C++ Compiler Reference*

-qpriority / #pragma priority (C++ only)

-qmkshrobj

Related information in the *XL C/C++ Language Reference*

The init_priority variable attribute

Order of object initialization across libraries

Each static library and shared library is loaded and initialized at runtime in reverse link order, once all of its dependencies have been loaded and initialized. Link order is the order in which each library was listed on the command line during linking into the main application. For example, if library A calls library B, library B is loaded before library A.

As each module is loaded, objects are initialized in order of priority, according to the rules outlined in “Assigning priorities to objects” on page 30. If objects do not have priorities assigned, or have the same priorities, object files are initialized in reverse link order — where link order is the order in which the files were given on the command line during linking into the library — and the objects within the files are initialized according to their declaration order. Objects are terminated in reverse order of their construction.

Example of object initialization across libraries

In this example, the following modules are used:

- main.out, the executable containing the main function
- libS1 and libS2, two shared libraries
- libS3 and libS4, two shared libraries that are dependencies of libS1
- libS5 and libS6, two shared libraries that are dependencies of libS2

The source files are compiled into object files with the following command strings:

```
bgxlC -qpriority=101 -c fileA.C -o fileA.o
bgxlC -qpriority=150 -c fileB.C -o fileB.o
bgxlC -c fileC.C -o fileC.o
bgxlC -c fileD.C -o fileD.o
bgxlC -c fileE.C -o fileE.o
bgxlC -c fileF.C -o fileF.o
bgxlC -qpriority=300 -c fileG.C -o fileG.o
bgxlC -qpriority=200 -c fileH.C -o fileH.o
bgxlC -qpriority=500 -c fileI.C -o fileI.o
bgxlC -c fileJ.C -o fileJ.o
bgxlC -c fileK.C -o fileK.o
bgxlC -qpriority=600 -c fileL.C -o fileL.o
```

The dependent libraries are created with the following command strings:

```
bgxlC -qmkshrobj -o libS3.so fileE.o fileF.o
bgxlC -qmkshrobj -o libS4.so fileG.o fileH.o
bgxlC -qmkshrobj -o libS5.so fileI.o fileJ.o
bgxlC -qmkshrobj -o libS6.so fileK.o fileL.o
```

The dependent libraries are linked with their parent libraries using the following command strings:

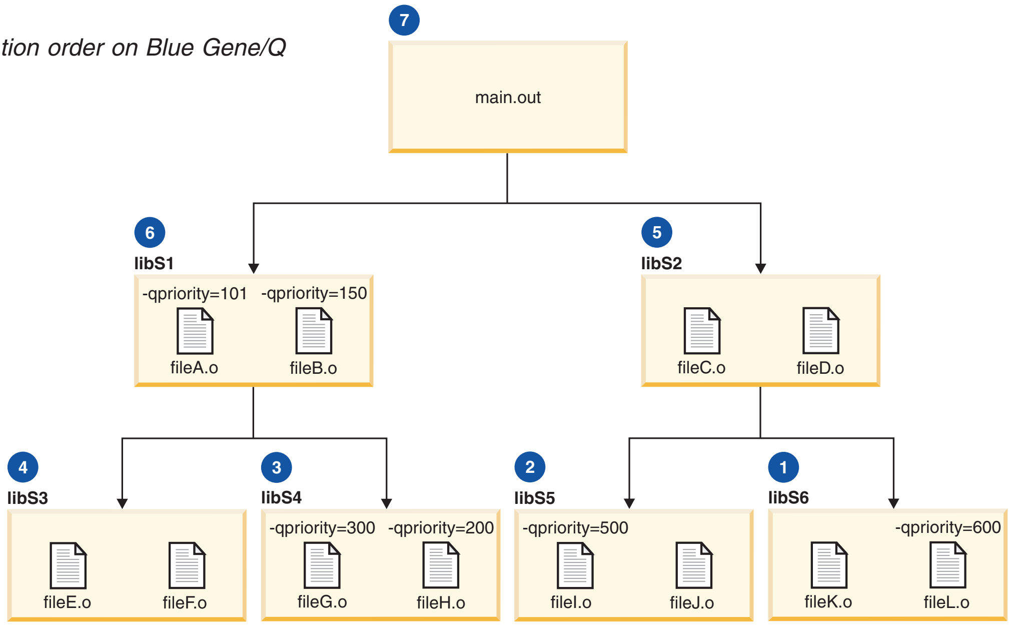
```
bgxlC -qmkshrobj -o libS1.so fileA.o fileB.o -L. -R. -lS3 -lS4
bgxlC -qmkshrobj -o libS2.so fileC.o fileD.o -L. -R. -lS5 -lS6
```

The parent libraries are linked with the main program with the following command string:

```
bgxlC main.C -o main.out -L. -R. -lS1 -lS2
```

The following diagram shows the initialization order of the shared libraries.

Figure 1. Object initialization order on Blue Gene/Q



Objects are initialized as follows:

Sequence	Object	Priority value	Comment
1	libS6	n/a	libS2 was entered last on the command line when linked with main, and so is initialized before libS1. However, libS5 and libS6 are dependencies of libS2, so they are initialized first. Since it was entered last on the command line when linked to create libS2, libS6 is initialized first. The objects in this library are initialized according to their priority.
2	fileL	600	The objects in fileL are initialized next (lowest priority number in this module).
3	fileK	65535	The objects in fileK are initialized next (next priority number in this module (default priority of 65535)).
4	libS5	n/a	libS5 was entered before libS6 on the command line when linked with libS2, so it is initialized next. The objects in this library are initialized according to their priority.
5	fileI	500	The objects in fileI are initialized next (lowest priority number in this module).
6	fileJ	65535	The objects in fileJ are initialized next (next priority number in this module (default priority of 65535)).
7	libS4	n/a	libS4 is a dependency of libS1 and was entered last on the command line when linked to create libS1, so it is initialized next. The objects in this library are initialized according to their priority.
8	fileH	200	The objects in fileH are initialized next (lowest priority number in this module).
9	fileG	300	The objects in fileG are initialized next (next priority number in this module).

Sequence	Object	Priority value	Comment
10	libS3	n/a	`libS3` is a dependency of `libS1` and was entered first on the command line during the linking with `libS1`, so it is initialized next. The objects in this library are initialized according to their priority.
11	fileF	65535	Both `fileF` and `fileE` are assigned a default priority of 65535. However, because `fileF` was listed last on the command line when the object files were linked into `libS3`, `fileF` is initialized first.
12	fileE	65535	Initialized next.
13	libS2	n/a	`libS2` is initialized next. The objects in this library are initialized according to their priority.
14	fileD	65535	Both `fileD` and `fileC` are assigned a default priority of 65535. However, because `fileD` was listed last on the command line when the object files were linked into `libS2`, `fileD` is initialized first.
15	fileC	65535	Initialized next.
16	libS1		`libS1` is initialized next. The objects in this library are initialized according to their priority.
17	fileA	101	The objects in `fileA` are initialized next (lowest priority number in this module).
18	fileB	150	The objects in `fileB` are initialized next (next priority number in this module).
19	main.out	n/a	Initialized last. The objects in `main.out` are initialized according to their priority.

Related information in the *XL C/C++ Compiler Reference*

-qmkshrobj

-W

Chapter 8. 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 38 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 Using high-order loop analysis and transformations step can help you limit loop execution time.
4. The Using interprocedural analysis 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.

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 12. 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**

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 and bad casting.

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, see “Tuning for your system architecture” on page 44.

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**.

In C, compile with **-qlibansi** unless your application defines functions with names identical to those of library functions. If you encounter problems with **-O2**, consider using **-qalias=noansi** rather than turning off optimization.

Also, ensure that pointers in your C code follow these type restrictions:

- Generic pointers can be `char*` or `void*`
- Mark all shared variables and pointers to shared variables `volatile`

Starting to tune at O2

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 “Tuning for your system architecture” on page 44 section 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.

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 38 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 13 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 13. Advanced optimizations

Optimization Level	Additional options implied	Complementary options	Options with possible benefits
-O3	**-qnostrict** **-qmaxmem=-1** **-qhot=level=0** **-qsimd=auto**	**-qarch** **-qtune**	
-O4	**-qnostrict** **-qmaxmem=-1** **-qhot** **-qipa** **-qarch=auto** **-qtune=auto** **-qcache=auto** **-qsimd=auto**	**-qarch** **-qtune** **-qcache**	**-qsmp=auto**

Table 13. Advanced optimizations (continued)

Optimization Level	Additional options implied	Complementary options	Options with possible benefits
-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 -qignerrno -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 `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 memory access 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 C/C++ Compiler Reference*).

Without **-qstrict**, the difference in computation for any one source-level operation is very small in comparison to “Basic optimization” on page 38. 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 -qoptimize" in the *XL C/C++ 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**.

For more information on **-qhot**, see “Using high-order loop analysis and transformations” on page 45.

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.

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 "Using interprocedural analysis" on page 48 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.
- **-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 IPA

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**.

Tuning for your system architecture

You can instruct the compiler to generate code for optimal execution on a given microprocessor or architecture family. By selecting appropriate target machine options, you can optimize to suit the broadest possible selection of target processors, a range of processors within a given family of processor architectures, or a specific processor.

The following table lists the optimization options that affect individual aspects of the target machine. Using a predefined optimization level sets default values for these individual options.

Table 14. Target machine options

Option	Behavior
-q64	Generates code for a 64-bit (4 byte integer / 8 byte long / 8 byte pointer) addressing model (64-bit execution mode). This is the default setting.
-qarch	Selects a family of processor architectures for which instruction code should be generated.**-qarch=qp** produces object code that runs on the Blue Gene/Q platforms.
-qtune	Biases optimization toward execution on a given microprocessor, without implying anything about the instruction set architecture to use as a target. **–qtune=qp** specifies that optimizations are tuned for the Blue Gene/Q platforms.
-qcache	Defines a specific cache or memory geometry. The defaults are determined through the setting of -qtune. See “Getting the most out of target machine options” below for more information on this option.

Related information in the *XL C/C++ Compiler Reference*

-qarch

-qipa

-qtune

-qcache

Getting the most out of target machine options

Using -qcache options

Before using the **-qcache** option, use the **-qlistopt** option to generate a listing of the current settings and verify if they are satisfactory. If you decide to specify your own **-qcache** suboptions, use **-qhot** or **-qsmp** along with it. For the full set of suboptions, option syntax, and guidelines for use, see **-qcache** in the *XL C/C++ Compiler Reference*.

Related information in the *XL C/C++ Compiler Reference*

-qhot

-qsmp

-qcache

-qlistopt

-qarch

-qtune

Using high-order loop analysis and transformations

High-order transformations are optimizations that specifically improve the performance of loops through techniques such as interchange, fusion, and unrolling.

The goals of these loop optimizations include:

- Reducing the costs of memory access through the effective use of caches and translation look-aside buffers.
- Overlapping computation and memory access through effective utilization of the data prefetching capabilities provided by the hardware.
- Improving the utilization of microprocessor resources through reordering and balancing the usage of instructions with complementary resource requirements.
- Generating QPX vector instructions.
- Generating calls to vector math library functions.

You can use the following mechanisms to enable high-order loop analysis and transformations:

- The default **-qsimd=auto** option automatically converts loop array operations into QPX vector instructions. These instructions calculate several results at one time, which is faster than calculating each result sequentially.
- Use the following suboptions available for **-qhot**, which implies an optimization level of **-O2**.

Table 15. -qhot suboptions

Suboption	Behavior
level=0	Instructs the compiler to perform a subset of high-order transformations that enhance performance by improving data locality. This suboption implies **-qhot=novector** and **-qhot=noarraypad**. This level is automatically enabled if you compile with **-O3**.
level=1	This is the default suboption if you specify **-qhot** with no suboptions. This level is also automatically enabled if you compile with **-O4** or **-O5**. This is equivalent to specifying **-qhot=vector**.
level=2	When used with **-qsmp**, 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.

Table 15. -qhot suboptions (continued)

Suboption	Behavior
vector	When specified with **-qnostrict** and **-qignerrno**, or **-O3** or a higher optimization level, instructs the compiler to transform some loops to use the optimized versions of various math functions contained in the MASS libraries, rather than use the system versions. The optimized versions make different trade-offs with respect to accuracy and exception-handling versus performance. This suboption is enabled by default if you specify **-qhot** with no suboptions. Also, specifying **-qhot=vector** with **-O3** implies **-qhot=level=1**.
arraypad	Instructs the compiler to pad any arrays where it infers there might be a benefit and to pad by whatever amount it chooses.

Related information in the *XL C/C++ Compiler Reference*

-qhot

-qstrict

-qignerrno

-qarch

-qsimd

Getting the most out of -qhot

Here are some suggestions for using **-qhot**:

- Try using **-qhot** along with **-O3** for all of your code. It is designed to have a neutral effect when no opportunities for transformation exist. However, it might increase compile time and have little benefit if the program has no loop processing vectors or arrays.
- If the runtime performance of your code can significantly benefit from automatic inlining and memory locality optimizations, try using **-O4** with **-qhot=level=0** or **-qhot=novector**.
- If you encounter unacceptably long compile time (this can happen with complex loop nests), try **-qhot=level=0** or **-qnohot**.
- If your code size is unacceptably large, try using **-qcompact** along with **-qhot**.
- You can compile some source files with the **-qhot** option and some files without the **-qhot** option, allowing the compiler to improve only the parts of your code that need optimization.
- Use **-qreport** along with **-qsimd=auto** to generate a loop transformation listing. The listing file identifies how loops are transformed in a section marked `LOOP TRANSFORMATION SECTION`. Use the listing information as feedback about how the loops in your program are being transformed. Based on this information, you may want to adjust your code so that the compiler can transform loops more effectively. For example, you can use this section of the listing to identify non-stride-one references that may prevent loop vectorization.
- Use **-qreport** along with **-qhot** or any optimization option that implies **-qhot** to generate information about nested loops in the `LOOP TRANSFORMATION SECTION` of the listing file. To generate a list of aggressive loop transformations and parallelizations performed on loop nests in the `LOOP TRANSFORMATION SECTION` of the listing file, use **-qhot=level=2** and **-qsmp** together with **-qreport**.
- If you specify **-qassert=refalign**, you assert to the compiler that all pointers inside the compilation unit only point to data that is naturally aligned with

respect to the length of the pointer types. With this assertion, the compiler might generate more efficient code. This assertion is particularly useful when you target a SIMD architecture with **-qhot=level=0** or **-qhot=level=1** with the **-qsimd=auto** option.

Related information in the *XL C/C++ Compiler Reference*

-qcompact

-qhot

-qsimd

-qstrict

Using shared-memory parallelism (SMP)

Many IBM pSeries® machines are capable of shared-memory parallel processing. You can compile with **-qsmp** to generate the threaded code needed to exploit this capability. The **-qsmp** option implies the **-qhot** option and an optimization level of **-O2** or higher.

The following table lists the most commonly used suboptions. Descriptions and syntax of all the suboptions are provided in **-qsmp** in the *XL C/C++ Compiler Reference*. An overview of automatic parallelization, as well as of OpenMP directives is provided in Chapter 13, “Parallelizing your programs,” on page 93.

Table 16. Commonly used -qsmp suboptions

suboption	Behavior
auto	Instructs the compiler to automatically generate parallel code where possible without user assistance. Any SMP programming constructs in the source code, including OpenMP directives, are also recognized. This is the default setting if you do not specify any **-qsmp** suboptions, and it also implies the **opt** suboption.
omp	Instructs the compiler to enforce strict conformance to the OpenMP API for specifying explicit parallelism. Only language constructs that conform to the OpenMP standard are recognized. Note that **-qsmp=omp** is currently incompatible with **-qsmp=auto**.
opt	Instructs the compiler to optimize as well as parallelize. The optimization is equivalent to **-O2 -qhot** in the absence of other optimization options.
noopt	All optimization is turned off. During development, it can be useful to turn off optimization to facilitate debugging.
fine_tuning	Other values for the suboption provide control over thread scheduling, nested parallelism, locking, etc.

Related information in the *XL C/C++ Compiler Reference*

-O, -qoptimize

-qsmp

-qhot

Getting the most out of -qsmp

Here are some suggestions for using the **-qsmp** option:

- Before using **-qsmp** with automatic parallelization, test your programs using optimization and **-qhot** in a single-threaded manner.
- If you are compiling an OpenMP program and do not want automatic parallelization, use **-qsmp=omp:noauto** .
- Always use the reentrant compiler invocations (the **_r** invocations) when using **-qsmp**.
- By default, the runtime environment uses all available processors. Do not set the *XLSMPOPTS=PARTHDS* or *OMP_NUM_THREADS* environment 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 settings:
  - Set the *BG_SMP_FAST_WAKEUP* environment variable to YES.
  - Set the *SPINS* and *YIELDS* environment variables (suboptions of the *XLSMPOPTS* environment variable) to 0.

  These settings prevent the operating system from intervening in the scheduling of threads across synchronization boundaries, such as barriers.
- When debugging an OpenMP program, try using **-qsmp=noopt** (without **-O**) to make the debugging information produced by the compiler more precise.

Related information in the *XL C/C++ Compiler Reference*

-qsmp

-qhot

Invoking the compiler

XLSMPOPTS

Environment variables for parallel processing

Using interprocedural analysis

Interprocedural analysis (IPA) enables the compiler to optimize across different files (whole-program analysis), and can result in significant performance improvements.

You can specify interprocedural analysis on the compilation step only or on both compilation and link steps in whole program mode. Whole program mode expands the scope of optimization to an entire program unit, which can be an executable or shared object. As IPA can significantly increase compile time, you should limit using IPA to the final performance tuning stage of development.

You enable IPA by specifying the **-qipa** option. The most commonly used suboptions and their effects are described in the following table. The full set of suboptions and syntax is described in the **-qipa** section of the *XL C/C++ Compiler Reference*.

The steps to use IPA are:

1. Do preliminary performance analysis and tuning before compiling with the **-qipa** option, because the IPA analysis uses a two-pass mechanism that increases compile and link time. You can reduce some compilation and link overhead by using the **-qipa=noobject** option.

2. Specify the **-qipa** option on both the compilation and the link steps of the entire application, or as much of it as possible. Use suboptions to indicate assumptions to be made about parts of the program *not* compiled with **-qipa**.

Table 17. Commonly used ***-qipa*** *suboptions*

Suboption	**Behavior**
level=0	Program partitioning and simple interprocedural optimization, which consists of: • Automatic recognition of standard libraries. • Localization of statically bound variables and procedures. • Partitioning and layout of procedures according to their calling relationships. (Procedures that call each other frequently are located closer together in memory.) • Expansion of scope for some optimizations, notably register allocation.
level=1	Inlining and global data mapping. Specifically: • Procedure inlining. • Partitioning and layout of static data according to reference affinity. (Data that is frequently referenced together will be located closer together in memory.) This is the default level if you do not specify any suboptions with the -qipa option.
level=2	Global alias analysis, specialization, interprocedural data flow: • Whole-program alias analysis. This level includes the disambiguation of pointer dereferences and indirect function calls, and the refinement of information about the side effects of a function call. • Intensive intraprocedural optimizations. This can take the form of value numbering, code propagation and simplification, moving code into conditions or out of loops, and elimination of redundancy. • Interprocedural constant propagation, dead code elimination, pointer analysis, code motion across functions, and interprocedural strength reduction. • Procedure specialization (cloning). • Whole program data reorganization.
inline=*suboptions*	Provides precise control over function inlining.
fine_tuning	Other values for **-qipa** provide the ability to specify the behavior of library code, tune program partitioning, read commands from a file, etc.

Related information in the *XL C/C++ Compiler Reference*

-qipa

Getting the most from -qipa

It is not necessary to compile everything with **-qipa**, but try to apply it to as much of your program as possible. Here are some suggestions:

- Specify the **-qipa** option on both the compile and link steps of the entire application. Although you can also use **-qipa** with libraries, shared objects, and executable files, be sure to use **-qipa** to compile the main and exported functions.
- When compiling and linking separately, use **-qipa=noobject** on the compile step for faster compilation.

- When specifying optimization options in a makefile, remember to use the compiler driver (**bgxlc**) to link, and to include all compiler options on the link step.
- As IPA can generate significantly larger object files than traditional compilations, ensure that there is enough space in the /tmp directory (at least 200 MB). You can use the TMPDIR environment variable to specify a directory with sufficient free space.
- Try varying the **level** suboption if link time is too long. Compiling with **-qipa=level=0** can still be very beneficial for little additional link time.
- Use **-qipa=list=long** to generate a report of functions that were previously inlined. If too few or too many functions are inlined, consider using **-qinline** or **-qnoinline**. ▶ C To control the inlining of specific functions, use **-qinline**+*function_name* or **-qinline**-*function_name*. C ◀
- To generate data reorganization information in the listing file, specify the optimization level **-qipa=level=2** or **-O5** together with **-qreport**. During the IPA link pass, the data reorganization messages for program variable data will be produced to the data reorganization section of the listing file with the label `DATA REORGANIZATION SECTION`. Reorganizations include array splitting, array transposing, memory allocation merging, array interleaving, and array coalescing.

Note: While IPA's interprocedural optimizations can significantly improve performance of a program, they can also cause incorrect but previously functioning programs to fail. Here are examples of programming practices that can work by accident without aggressive optimization but are exposed with IPA:

- Relying on the allocation order or location of automatic variables, such as taking the address of an automatic variable and then later comparing it with the address of another local variable to determine the growth direction of a stack. The C language does not guarantee where an automatic variable is allocated, or its position relative to other automatic variables. Do not compile such a function with IPA.
- Accessing a pointer that is either invalid or beyond an array's bounds. Because IPA can reorganize global data structures, a wayward pointer which might have previously modified unused memory might now conflict with user-allocated storage.

Related information in the *XL C/C++ Compiler Reference*

-qinline

-qlist

-qipa

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/vacpp/bg/12.1/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/vacpp/bg/12.1/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
- 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/vacpp/bg/12.1/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.

Other optimization options

Options are available to control particular aspects of optimization. They are often enabled as a group or given default values when you enable a more general optimization option or level.

For more information on these options, see the heading for each option in the *XL C/C++ Compiler Reference*.

Table 18. Selected compiler options for optimizing performance

Option	Description
-qignerrno	Allows the compiler to assume that `errno` is not modified by library function calls, so that such calls can be optimized. Also allows optimization of square root operations, by generating inline code rather than calling a library function. (For processors that support `sqrt`.)
-qsmallstack	Instructs the compiler to compact stack storage. Doing so might increase heap usage, which might increase execution time. However, it might be necessary for the program to run or to be optimally multithreaded.
-qinline	Controls inlining.
-qunroll	Independently controls loop unrolling. **-qunroll** is implicitly activated under -O3.
-qtbtable	Controls the generation of traceback table information.
C++ **-qnoeh**	Informs the compiler that no C++ exceptions will be thrown and that cleanup code can be omitted. If your program does not throw any C++ exceptions, use this option to compact your program by removing exception-handling code.

Table 18. Selected compiler options for optimizing performance (continued)

Option	Description
-qnounwind	Informs the compiler that the stack will not be unwound while any routine in this compilation is active. This option can improve optimization of non-volatile register saves and restores. In C++, the **-qnounwind** option implies the **-qnoeh** option. It should not be used if the program uses `setjmp/longjmp` or any other form of exception handling.
-qstrict	Disables all transformations that change program semantics. In general, compiling a program correctly with **-qstrict** and any levels of optimization produces the same results as without optimization. For details about **-qstrict** and all of its suboptions, see -qstrict in the *XL C/C++ Compiler Reference*.
-qnostrict	Allows the compiler to reorder floating-point calculations and potentially excepting instructions. A potentially excepting instruction is one that might raise an interrupt due to erroneous execution (for example, floating-point overflow, a memory access violation). **-qnostrict** is used by default for optimization levels **-O3** and higher.
-qprefetch	Inserts prefetch instructions in compiled code to improve code performance. **-qnoprefetch** is the default option. For details, see -qprefetch in the *XL C/C++ Compiler Reference*.

Related information in the *XL C/C++ Compiler Reference*

- -qignerrno
- -qsmallstack
- -qinline
- -qunroll / #pragma unroll
- -qinlglue
- -qtbtable
- -qeh (C++ only)
- -qunwind
- -qstrict
- -qprefetch

Chapter 9. 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" on page 56 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 57 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. 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* option to try to produce the incorrect results consistently. For example, using **-qinitauto=FF** gives 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.

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

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

In addition, you can also use the **snapshot** pragma to ensure that certain variables are visible to the debugger at points in your application. For details, see **#pragma ibm 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:

```
bgxlc myprogram.c -O3 -qhot -g -qoptdebug
```

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. Note the unrolled loop and the consolidation of values assigned by the `x + y` expression.

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

```
#include "stdio.h"

void foo(int x, int y, char* w)
{
 char* s = w+1;
 char* t = w+1;
 int z = x + y;
 int d = x + y;
 int a = printf("TEST\n");

 for (int i = 0; i < 4; i++)
  printf("%d %d %d %s %s\n", a, z, d, s, t);
 }

int main()
{
 char d[] = "DEBUG";
 foo(3, 4, d);
 return 0;
}
```

Example 2: gdb debugger listing

```
(gdb) list
1          3 |  void foo(long x, long y, char * w)
2          4 |  {
3          9 |    a = printf("TEST/n");
4         12 |    printf("%d %d %d %s %s/n",a,x + y,x + y,
                        ((char *)w  + 1),((char *)w  + 1));
5                 printf("%d %d %d %s %s/n",a,x + y,x + y,
                        ((char *)w  + 1),((char *)w  + 1));
6                 printf("%d %d %d %s %s/n",a,x + y,x + y,
                        ((char *)w  + 1),((char *)w  + 1));
7                 printf("%d %d %d %s %s/n",a,x + y,x + y,
                        ((char *)w  + 1),((char *)w  + 1));
8         13 |    return;
9               } /* function */
```

```
10
11
12          15 |  long main()
13          16 |  {
14          17 |    d$init$0 = "DEBUG";
15          18 |    $$PARM.x0 = 3;
16                  $$PARM.y1 = 4;
17                  $$PARM.w2 = &d;
18           9 |    a = printf("TEST/n");
19          12 |    printf("%d %d %d %s %s/n",a,$$PARM.x0 +
                           $$PARM.y1,$$PARM.x0 + $$PARM.y1,((char *)$$PARM.w2  + 1),
                           ((char *)$$PARM.w2  + 1));
20                  printf("%d %d %d %s %s/n",a,$$PARM.x0 + $$PARM.y1,
                           $$PARM.x0 + $$PARM.y1,((char *)$$PARM.w2  + 1),
                           ((char *)$$PARM.w2  + 1));
21                  printf("%d %d %d %s %s/n",a,$$PARM.x0 + $$PARM.y1,
                           $$PARM.x0 + $$PARM.y1,((char *)$$PARM.w2  + 1),
                           ((char *)$$PARM.w2  + 1));
22                  printf("%d %d %d %s %s/n",a,$$PARM.x0 + $$PARM.y1,
                           $$PARM.x0 + $$PARM.y1,((char *)$$PARM.w2  + 1),
                           ((char *)$$PARM.w2  + 1));
23          19 |    rstr = 0;
24                  return rstr;
25          20 |  } /* function */
```

Example 3: Stepping through optimized source

```
(gdb) break 17
Breakpoint 2 at 0x10000694: file myprogram.o.optdbg, line 17.
(gdb) run
Starting program: /nfs/r3lp1/home/gklou/tmp/optdbg/a.out

Breakpoint 2, main () at myprogram.o.optdbg:18
18           9 |    a = printf("TEST/n");
(gdb) continue
Continuing.
TEST
5 7 7 EBUG EBUG
5 7 7 EBUG EBUG
5 7 7 EBUG EBUG
5 7 7 EBUG EBUG

Program exited normally.
```

Chapter 10. 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 C/C++ 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 C/C++ for Blue Gene/Q, V12.1 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 C/C++ 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.

```
for(i = 0; i < 100; i++)
{
  a[i] = b[i] + c[i];
}
```

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

```
vector4double v1;
vector4double v2;
vector4double v3;

for (i = 0; i < 100; i += 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.
  vec_st(v3, 0, &a[i]);    // Stores the result to a[i], a[i+1], a[i+2], and a[i+3].
}
```

Example 2

This example shows a sample basic block.

```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
a[3] = b[3] + c[3];
```

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

```
vector4double v1 = vec_lds(0, &b[0]);  // Loads b[0], b[1], b[2], and b[3].
vector4double v2 = vec_lds(0, &c[0]);  // Loads c[0], c[1], c[2], and c[3].
vector4double v3 = vec_add(v1, v2);    // Adds the four elements.
vec_st(v3, 0, &a[0]);                  // Stores the result to a[0], a[1], a[2], and a[3].
```

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 C/C++ 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 with the **disjoint** pragma when pointers are used in your program, or use the **independent_loop** pragma to specify that the loop iterations are independent.

  ```
  #pragma ibm independent_loop
  for(i = 0; i < n; i++)
  {
    a[i + l1] = a[i + l2] + a[i + l3];
  }
  ```

- 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** pragma, 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** built-in function to assert the alignment.

  ```
  void func(double *, double *b, double *c)
  {
      __alignx(32, a);
      __alignx(32, b);
      __alignx(32, c);

      for(i = 0 ; i < 200 ; i++)
      {
          c[i] = a[i] + b[i];
      }
  }
  ```

Related information

- -qsimd
- -qhot
- -qreport
- -qassert
- #pragma disjoint
- #pragma align
- #pragma ibm independent_loop
- __alignx
- Vector built-in functions

Chapter 11. Coding your application to improve performance

Chapter 8, "Optimizing your applications," on page 37 discusses the various compiler options that the XL C/C++ compiler provides for optimizing your code with minimal coding effort. If you want to take your application a step further, to complement and take the most advantage of compiler optimizations, the following sections discuss C and C++ programming techniques that can improve performance of your code:

- "Finding faster input/output techniques"
- "Reducing function-call overhead"
- "Using delegating constructors (C++0x)" on page 67
- "Using template explicit instantiation declarations (C++0x)" on page 67
- "Managing memory efficiently" on page 68
- "Optimizing variables" on page 68
- "Manipulating strings efficiently" on page 69
- "Optimizing expressions and program logic" on page 69
- "Optimizing operations in 64-bit mode" on page 70

Finding faster input/output techniques

There are a number of ways to improve your program's performance of input and output:

- If your file I/O accesses do not exhibit locality (that is truly random access such as in a database), implement your own buffering or caching mechanism on the low-level I/O functions.
- If you do your own I/O buffering, make the buffer a multiple of 4K, which is the size of a page.
- Use buffered I/O to handle text files.
- If you know you have to process an entire file, determine the size of the data to be read in, allocate a single buffer to read it to, read the whole file into that buffer at once using `read`, and then process the data in the buffer. This reduces disk I/O, provided the file is not so big that excessive swapping will occur. Consider using the `mmap` function to access the file.

Reducing function-call overhead

When you write a function or call a library function, consider the following guidelines:

- Call a function directly, rather than using function pointers.
- Use `const` arguments in inlined functions whenever possible. Functions with constant arguments provide more opportunities for optimization.
- Use the **#pragma expected_value** preprocessor directive so that the compiler can optimize for common values used with a function.
- Use the **#pragma isolated_call** preprocessor directive to list functions that have no side effects and do not depend on side effects.
- Use the `restrict` keyword for pointers that can never point to the same memory.

- Use **#pragma disjoint** within functions for pointers or reference parameters that can never point to the same memory.
- Declare a nonmember function as static whenever possible. This can speed up calls to the function and increase the likelihood that the function will be inlined.
- C++ Usually, you should not declare all your virtual functions inline. If all virtual functions in a class are inline, the virtual function table and all the virtual function bodies will be replicated in each compilation unit that uses the class.
- C++ When declaring functions, use the `const` specifier whenever possible.
- C Fully prototype all functions. A full prototype gives the compiler and optimizer complete information about the types of the parameters. As a result, promotions from unwidened types to widened types are not required, and parameters can be passed in appropriate registers.
- C Avoid using unprototyped variable argument functions.
- Design functions so that they have few parameters and the most frequently used parameters are in the leftmost positions in the function prototype.
- Avoid passing by value large structures or unions as function parameters or returning a large structure or a union. Passing such aggregates requires the compiler to copy and store many values. This is worse in C++ programs in which class objects are passed by value because a constructor and destructor are called when the function is called. Instead, pass or return a pointer to the structure or union, or pass it by reference.
- Pass non-aggregate types such as `int` and `short` or small aggregates by value rather than passing by reference, whenever possible.
- If your function exits by returning the value of another function with the same parameters that were passed to your function, put the parameters in the same order in the function prototypes. The compiler can then branch directly to the other function.
- Use the built-in functions, which include string manipulation, floating-point, and trigonometric functions, instead of coding your own. Intrinsic functions require less overhead and are faster than a function call, and often allow the compiler to perform better optimization.

  C++ Many functions from the C++ standard libraries are mapped to optimized built-in functions by the compiler.

  C Many functions from `string.h` and `math.h` are mapped to optimized built-in functions by the compiler.
- Selectively mark your functions for inlining, using the `inline` keyword. An inlined function requires less overhead and is generally faster than a function call. The best candidates for inlining are small functions that are called frequently from a few places, or functions called with one or more compile-time constant parameters, especially those that affect `if`, `switch` or `for` statements. You might also want to put these functions into header files, which allows automatic inlining across file boundaries even at low optimization levels. Be sure to inline all functions that only load or store a value, or use simple operators such as comparison or arithmetic operators. Large functions and functions that are called rarely are generally not good candidates for inlining. Neither are medium size functions that are called from many places.
- Avoid breaking your program into too many small functions. If you must use small functions, seriously consider using the **-qipa** compiler option, which can automatically inline such functions, and uses other techniques for optimizing calls between functions.

- C++ Avoid virtual functions and virtual inheritance unless required for class extensibility. These language features are costly in object space and function invocation performance.

  Related information in the *XL C/C++ Compiler Reference*

  #pragma expected_value

  -qisolated_call / #pragma isolated_call

  #pragma disjoint

  -qipa

Using delegating constructors (C++0x)

Note: C++0x is a new version of the C++ programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C++0x standard is complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C++0x standard and therefore they should not be relied on as a stable programming interface.

Use the delegating constructors feature to concentrate common initializations in one constructor. This helps reduce the code size and make program more readable and maintainable.

This technique is described in “Using delegating constructors (C++0x)” on page 19.

Using template explicit instantiation declarations (C++0x)

Note: C++0x is a new version of the C++ programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C++0x standard is complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C++0x standard and therefore they should not be relied on as a stable programming interface.

Use the explicit instantiation declarations feature to suppress the implicit instantiation of a template specialization or its members. This helps reduce the collective size of the object files and shorten compile time.

This technique is described in “Using explicit instantiation declarations (C++0x)” on page 26.

Managing memory efficiently

Because C++ objects are often allocated from the heap and have limited scope, memory use affects performance more in C++ programs than it does in C programs. For that reason, consider the following guidelines when you develop C++ applications:

- In a structure, declare the largest aligned members first. Members of similar alignment should be grouped together where possible.
- In a structure, place variables near each other if they are frequently used together.
- ▶ C++ Ensure that objects that are no longer needed are freed or otherwise made available for reuse. One way to do this is to use an *object manager*. Each time you create an instance of an object, pass the pointer to that object to the object manager. The object manager maintains a list of these pointers. To access an object, you can call an object manager member function to return the information to you. The object manager can then manage memory usage and object reuse.
- Storage pools are a good way of keeping track of used memory (and reclaiming it) without having to resort to an object manager or reference counting.
- ▶ C++ Avoid copying large, complicated objects.
- ▶ C++ Avoid performing a *deep copy* if a *shallow copy* is all you require. For an object that contains pointers to other objects, a shallow copy copies only the pointers and not the objects to which they point. The result is two objects that point to the same contained object. A deep copy, however, copies the pointers and the objects they point to, as well as any pointers or objects contained within that object, and so on. A deep copy must be performed in multithreaded environments, because it reduces sharing and synchronization.
- ▶ C++ Use virtual methods only when absolutely necessary.
- ▶ C++ Use the "Resource Acquisition is Initialization" (RAII) pattern.
- Use `boost::shared_ptr` and `boost::weak_ptr`.

Optimizing variables

Consider the following guidelines:

- Use local variables, preferably automatic variables, as much as possible.

  The compiler must make several worst-case assumptions about global variables. For example, if a function uses external variables and also calls external functions, the compiler assumes that every call to an external function could use and change the value of every external variable. If you know that a global variable is not read or affected by any function call, and this variable is read several times with function calls interspersed, copy the global variable to a local variable and then use this local variable.
- If you must use global variables, use static variables with file scope rather than external variables whenever possible. In a file with several related functions and static variables, the optimizer can gather and use more information about how the variables are affected.
- If you must use external variables, group external data into structures or arrays whenever it makes sense to do so. All elements of an external structure use the same base address. Do not group variables whose addresses are taken with variables whose addresses are not taken.

- The **#pragma isolated_call** preprocessor directive can improve the runtime performance of optimized code by allowing the compiler to make less pessimistic assumptions about the storage of external and static variables. Isolated call functions with constant or loop-invariant parameters can be moved out of loops, and multiple calls with the same parameters can be replaced with a single call.
- Avoid taking the address of a variable. If you use a local variable as a temporary variable and must take its address, avoid reusing the temporary variable for a different purpose. Taking the address of a local variable can inhibit optimizations that would otherwise be done on calculations involving that variable.
- Use constants instead of variables where possible. The optimizer is able to do a better job reducing runtime calculations by doing them at compile time instead. For instance, if a loop body has a constant number of iterations, use constants in the loop condition to improve optimization (`for (i=0; i<4; i++)` can be better optimized than `for (i=0; i<x; i++)`).
- Use register-sized integers (`long` data type) for scalars to avoid sign extension instructions after each change in 64-bit mode. For large arrays of integers, consider using one- or two-byte integers or bit fields.
- Use the smallest floating-point precision appropriate to your computation.

Related information in the *XL C/C++ Compiler Reference*

-qisolated_call / #pragma isolated_call

Manipulating strings efficiently

The handling of string operations can affect the performance of your program.

- When you store strings into allocated storage, align the start of the string on an 8-byte boundary.
- Keep track of the length of your strings. If you know the length of a string, you can use `mem` functions instead of `str` functions. For example, `memcpy` is faster than `strcpy` because it does not have to search for the end of the string.
- If you are certain that the source and target do not overlap, use `memcpy` instead of `memmove`. This is because `memcpy` copies directly from the source to the destination, while `memmove` might copy the source to a temporary location in memory before copying to the destination (depending on the length of the string).
- When manipulating strings using `mem` functions, faster code can be generated if the *count* parameter is a constant rather than a variable. This is especially true for small count values.
- Make string literals read-only, whenever possible. This improves certain optimization techniques and reduces memory usage if there are multiple uses of the same string. You can explicitly set strings to read-only by using **#pragma strings (readonly)** in your source files or **-qro** (this is enabled by default) to avoid changing your source files.

Related information in the *XL C/C++ Compiler Reference*

-qro / #pragma strings

Optimizing expressions and program logic

Consider the following guidelines:

- If components of an expression are used in other expressions and they include function calls or there are function calls between the uses, assign the duplicated values to a local variable.
- Avoid forcing the compiler to convert numbers between integer and floating-point internal representations. For example:

  ```
  float array[10];
  float x = 1.0;
  int i;
  for (i = 0; i< 9; i++)  {     /* No conversions needed */
       array[i] = array[i]*x;
       x = x + 1.0;
      }
  for (i = 0; i< 9; i++)  {       /* Multiple conversions needed */
       array[i] = array[i]*i;
      }
  ```

  When you must use mixed-mode arithmetic, code the integer and floating-point arithmetic in separate computations whenever possible.
- Do not use global variables as loop indices or bounds.
- Avoid `goto` statements that jump into the middle of loops. Such statements inhibit certain optimizations.
- Improve the predictability of your code by making the fall-through path more probable. Code such as:

  ```
  if (error) {handle error} else {real code}
  ```

  should be written as:

  ```
  if (!error) {real code} else {error}
  ```
- If one or two cases of a `switch` statement are typically executed much more frequently than other cases, break out those cases by handling them separately before the `switch` statement. If possible, replace the `switch` statement by checking whether the value is in range to be obtained from an array.
- ▶ C++ Use `try` blocks for exception handling only when necessary because they can inhibit optimization.
- Keep array index expressions as simple as possible.

Optimizing operations in 64-bit mode

The ability to handle larger amounts of data directly in physical memory rather than relying on disk I/O is perhaps the most significant performance benefit of 64-bit machines. However, some applications compiled in 32-bit mode perform better than when they are recompiled in 64-bit mode. Some reasons for this include:

- 64-bit programs are larger. The increase in program size places greater demands on physical memory.
- 64-bit long division is more time-consuming than 32-bit integer division.
- 64-bit programs that use 32-bit signed integers as array indexes or loop counts might require additional instructions to perform sign extension each time the array is referenced or the loop count is incremented.

Some ways to compensate for the performance liabilities of 64-bit programs include:

- Avoid performing mixed 32- and 64-bit operations. For example, adding a 32-bit data type to a 64-bit data type requires that the 32-bit type be sign-extended to clear or set the upper 32 bits of the register. This slows the computation.

- Use `long` types instead of `signed`, `unsigned`, and plain `int` types for variables which will be frequently accessed, such as loop counters and array indexes. Doing so frees the compiler from having to truncate or sign-extend array references, parameters during function calls, and function results during returns.

Tracing functions in your code

You can instruct the compiler to insert calls to user-defined tracing functions to aid in debugging or timing the execution of other functions.

Using tracing functions in your program requires the following steps:

1. Writing tracing functions.
2. Specifying which functions to trace with the **-qfunctrace** option.

Using the **-qfunctrace** option causes the compiler to insert calls to these tracing functions at key points in the function body; however you are responsible for defining these tracing functions. The following list describes at which points the tracing functions are called:

- The compiler inserts calls to the tracing function at the entry point of a function. The line number passed to the routine is the line number of the first executable statement in the instrumented function.
- The compiler inserts calls to the tracing function at the exit point of a function. The line number that is passed to the function is the line number of the statement causing the exit in the instrumented function.
- The catch tracing function is called at the beginning of the C++ catch block when the exception occurs.

You can use the **-qnofunctrace** compiler option or the `#pragma nofunctrace` pragma to disable function tracing.

How to write tracing functions

To trace functions in your code, define the following tracing functions:

- `__func_trace_enter` is the entry point tracing function.
- `__func_trace_exit` is the exit point tracing function.
- `__func_trace_catch` is the catch tracing function.

The prototypes of these functions are as follows:

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

In the preceding tracing functions, the descriptions for their variables are as follows:

- `function_name` is the name of the function you want to trace.
- `file_name` is the name of the file.
- `line_number` is the line number at entry or exit point of the function. This is a 4-byte number.

- user_data is the address of a static pointer variable. The static pointer variable is generated by the compiler and initialized to NULL; in addition, because the pointer variable is static, its address is the same for all instrumentation calls inside the same function.

Notes:

- The exit function is not called if the function has an abnormal exit. The abnormal exit can be caused by C++ exception throws, raise the signal, or calls exit.
- The **-qfunctrace** option does not support setjmp and longjmp. For example, a call to longjmp() that leaves function1 and returns from setjmp() in function2 will have a missing call to __func_trace_exit in function1 and a missing a call to __func_trace_enter in function2.
- The catch function is called at the point where the C++ exception is caught by user code.
- To define tracing functions in C++ programs, use the extern "C" linkage directive before your function definition.
- The function calls are only inserted into the function definition, and if a function is inlined, no tracing is done within the inlined code.
- If you develop a multithreaded program, make sure the tracing functions have the proper synchronization. Calls to the tracing functions are not thread-safe.
- If you specify a function that does not exist with the option, the function is ignored.

Rules

The following rules apply when you trace functions in your code:

- When optimization is enabled, line numbers might not be accurate.
- The tracing function must not call instrumented function; otherwise an infinite loop might occur.
- If you instruct the compiler to trace recursive functions, make sure that your tracing functions can handle recursion.
- Inlined functions are not instrumented.
- Tracing functions are not instrumented.
- Compiler-generated functions are not instrumented, except for the outlined functions generated by optimization such as OpenMP. In those cases, the name of the outlined function contains the name of the original user function as prefix.
- Tracing functions might be called during static initialization. You must be careful that anything used in the tracing functions are initialized before the first possible call to the tracing function.

Examples

The following C example shows how you can trace functions in your code using function prototypes. Assume you want to trace the entry and exit points of function1 and function2, as well as how much time it takes the compiler to trace them in the following code:

Main program file: t1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
```

```
#ifdef __cplusplus
extern "C"
#endif
void __func_trace_enter(const char *function_name, const char *file_name,
                        int line_number, void** const user_data){
   if((*user_data)==NULL)
       (*user_data)=(time_t *)malloc(sizeof(time_t));
   (*(time_t *)*user_data)=time(NULL);
   printf("begin function: name=%s file=%s line=%d\n",function_name,file_name,
          line_number);
}
#ifdef __cplusplus
extern "C"
#endif
void __func_trace_exit(const char *function_name, const char*file_name,
                       int line_number, void** const user_data){
   printf("end function: name=%s file=%s line=%d. It took %g seconds\n",
          function_name,file_name,line_number, difftime(time(NULL),
          *(time_t *)*user_data));
}
void function2(void){
   sleep(3);
}
void function1(void){
   sleep(5);
   function2();
}
int main(){
   function1();
}
```

Compile the main program source file as follows:

```
xlc t1.c -qfunctrace+function1:function2
```

Run executable `a.out` to output function trace results:

```
begin function: name=function1 file=t.c line=27
begin function: name=function2 file=t.c line=24
end function: name=function2 file=t.c line=25. It took 3 seconds
end function: name=function1 file=t.c line=29. It took 8 seconds
```

As you see from the preceding example, the `user_data` parameter is defined to use the system time as basis for time calculation. The following steps explain how `user_data` is defined to achieve this goal:

1. The function reserves a memory area for storing the value of `user_data`.
2. The system time is used as the value for `user_data`.
3. In the `__func_trace_exit` function, the `difftime` function uses `user_data` to calculate time differences. The result is displayed in the form of `It took %g seconds` in the output.

The following C++ example shows how tracing functions are called. The following example traces class `myStack`, function `foo`, and disables tracing for `int main()` using `#pragma nofunctrace`:

Main program file: t2.cpp

```
#include <iostream>
#include <vector>
#include <stdexcept>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
```

```
extern "C"
void __func_trace_enter(const char *function_name, const char *file_name,
                        int line_number, void** const user_data){
  if((*user_data)==NULL)
    (*user_data)=(time_t *)malloc(sizeof(time_t));
  (*(time_t *)*user_data)=time(NULL);
  printf("enter function: name=%s file=%s line=%d\n",function_name,file_name,
         line_number);
}
extern "C"
void __func_trace_exit(const char *function_name, const char*file_name,
                       int line_number, void** const user_data){
  printf("exit function: name=%s file=%s line=%d. It took %g seconds\n",
         function_name, file_name, line_number, difftime(time(NULL),
         *(time_t *)*user_data));
}
extern "C"
void __func_trace_catch(const char *function_name, const char*file_name,
                        int line_number, void** const user_data){
  printf("catch function: name=%s file=%s line=%d. It took %g seconds\n",
         function_name, file_name,line_number, difftime(time(NULL),
         *(time_t *)*user_data));
}

template <typename T> class myStack{
  private:
  std::vector<T> elements;
  public:
  void push(T const&);
  void pop();
};

template <typename T>
void myStack<T>::push(T const& value){
  sleep(3);
  std::cout<< "\tpush(" << value << ")" <<std::endl;
  elements.push_back(value);
}
template <typename T>
void myStack<T>::pop(){
  sleep(5);
  std::cout<< "\tpop()" <<std::endl;
  if(elements.empty()){
    throw std::out_of_range("myStack is empty");
  }
  elements.pop_back();
}
void foo(){
  myStack<int> intValues;
  myStack<float> floatValues;
  myStack<double> doubleValues;
  intValues.push(4);
  floatValues.push(5.5f);
  try{
    intValues.pop();
    floatValues.pop();
    doubleValues.pop(); // cause exception
  } catch(std::exception const& e){
    std::cout<<"\tException: "<<e.what()<<std::endl;
  }
  std::cout<<"\tdone"<<std::endl;
}
#pragma nofunctrace(main)
int main(){
  foo();
}
```

Compile the main program source file as follows:

```
xlC t2.cpp -qfunctrace+myStack:foo
```

Run executable `a.out` to output function trace results:

```
  enter function: name=_Z3foov file=t2.cpp line=56
  enter function: name=_ZN7myStackIiE4pushERKi file=t2.cpp line=42
          push(4)
  exit function: name=_ZN7myStackIiE4pushERKi file=t2.cpp line=45. It took 3 seconds
  enter function: name=_ZN7myStackIfE4pushERKf file=t2.cpp line=42
          push(5.5)
  exit function: name=_ZN7myStackIfE4pushERKf file=t2.cpp line=45. It took 3 seconds
  enter function: name=_ZN7myStackIiE3popEv file=t2.cpp line=48
          pop()
  exit function: name=_ZN7myStackIiE3popEv file=t2.cpp line=54. It took 5 seconds
  enter function: name=_ZN7myStackIfE3popEv file=t2.cpp line=48
          pop()
  exit function: name=_ZN7myStackIfE3popEv file=t2.cpp line=54. It took 5 seconds
  enter function: name=_ZN7myStackIdE3popEv file=t2.cpp line=48
          pop()
  catch function: name=_Z3foov file=t2.cpp line=65. It took 21 seconds
          Exception: myStack is empty
          done
  exit function: name=_Z3foov file=t2.cpp line=69. It took 21 seconds
```

Related information

- For details about the **-qfunctrace** compiler option, see -qfunctrace in the *XL C/C++ Compiler Reference*.
- See #pragma nofunctrace in the *XL C/C++ Compiler Reference* for details about the `#pragma nofunctrace`.

Using rvalue references (C++0x)

Note: C++0x is a new version of the C++ programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C++0x standard is complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C++0x standard and therefore they should not be relied on as a stable programming interface.

In C++0x, you can overload functions based on the value categories of arguments and similarly have lvalueness detected by template argument deduction. You can also have an rvalue bound to an rvalue reference and modify the rvalue through the reference. This enables a programming technique with which you can reuse the resources of expiring objects and therefore improve the performance of your libraries, especially if you use generic code with class types, for example, template data structures. Additionally, the value category can be considered when writing a forwarding function.

Move semantics

When you want to optimize the use of temporary values, you can use a move operation in what is known as destructive copying. Consider the following string concatenation and assignment:

```
std::string a, b, c;
c = a + b;
```

In this program, the compiler first stores the result of a + b in an internal temporary variable, that is, an rvalue.

The signature of a normal copy assignment operator is as follows:

```
string& operator = (const string&)
```

With this copy assignment operator, the assignment consists of the following steps:

1. Copy the temporary variable into c using a deep-copy operation.
2. Discard the temporary variable.

Deep copying the temporary variable into c is not efficient because the temporary variable is discarded at the next step.

To avoid the needless duplication of the temporary variable, you can implement an assignment operator that moves the variable instead of copying the variable. That is, the argument of the operator is modified by the operation. A move operation is faster because it is done through pointer manipulation, but it requires a reference through which the source variable can be manipulated. However, a + b is a temporary value, which is not easily differentiated from a const-qualified value in C++ before C++0x for the purposes of overload resolution.

With rvalue references, you can create a move assignment operator as follows:

```
string& operator= (string&&)
```

With this move assignment operator, the memory allocated for the underlying C-style string in the result of a + b is assigned to c. Therefore, it is not necessary to allocate new memory to hold the underlying string in c and to copy the contents to the new memory.

The following code can be an implementation of the string move assignment operator:

```
string& string::operator=(string&& str)
{
  // The named rvalue reference str acts like an lvalue
  std::swap(_capacity, str._capacity);
  std::swap(_length, str._length);

  // char* _str points to a character array and is a
  // member variable of the string class
  std::swap(_str, str._str);
  return *this;
}
```

However, in this implementation, the memory originally held by the string being assigned to is not freed until str is destroyed. The following implementation that uses a local variable is more memory efficient:

```
string& string::operator=(string&& parm_str)
{
  // The named rvalue reference parm_str acts like an lvalue
  string sink_str;
  std::swap(sink_str, parm_str);
  std::swap(*this, sink_str);
  return *this;
}
```

In a similar manner, the following program is a possible implementation of a string concatenation operator:

```
string operator+(string&& a, const string& b)
{
  return std::move(a+=b);
}
```

Note: The `std::move` function only casts the result of `a+=b` to an rvalue reference, without moving anything. The return value is constructed using a move constructor because the expression `std::move(a+=b)` is an rvalue. The relationship between a move constructor and a copy constructor is analogous to the relationship between a move assignment operator and a copy assignment operator.

Perfect forwarding

The `std::forward` function is a helper template, much like `std::move`. It returns a reference to its function argument, with the resulting value category determined by the template type argument. In an instantiation of a forwarding function template, the value category of an argument is encoded as part of the deduced type for the related template type parameter. The deduced type is passed to the `std::forward` function.

The `wrapper` function in the following example is a forwarding function template that forwards to the `do_work` function. Use `std::forward` in forwarding functions on the calls to the target functions. The following example also uses the decltype and trailing return type features to produce a forwarding function that forwards to one of the `do_work` functions. Calling the `wrapper` function with any argument results in a call to a `do_work` function if a suitable overload function exists. Extra temporaries are not created and overload resolution on the forwarding call resolves to the same overload as it would if the `do_work` function were called directly.

```
struct s1 *do_work(const int&);              // #1
struct s2 *do_work(const double&);           // #2
struct s3 *do_work(int&&);                   // #3
struct s4 *do_work(double&&);                // #4
template <typename T> auto wrapper(T && a)->
   decltype(do_work(std::forward<T>(*static_cast<typename std::remove_reference<T>
   ::type*>(0))))
{
   return do_work(std::forward<T>(a));
}
template <typename T> void tPtr(T *t);
int main()
{
   int x;
   double y;
   tPtr<s1>(wrapper(x));     // calls #1
   tPtr<s2>(wrapper(y));     // calls #2
   tPtr<s3>(wrapper(0));     // calls #3
   tPtr<s4>(wrapper(1.0));   // calls #4
}
```

Related information in the *XL C/C++ Compiler Reference*

-qlanglvl

Related information in the *XL C/C++ Language Reference*

Reference collapsing(C++0x)

The decltype(expression) type specifier (C++0x)

Trailing return type (C++0x)

Chapter 12. Using the high performance libraries

IBM XL C/C++ for Blue Gene/Q, V12.1 is shipped with a set of libraries for high-performance mathematical computing:

- The Mathematical Acceleration Subsystem (MASS) is a set of libraries of tuned mathematical intrinsic functions that provide improved performance over the corresponding standard system math library functions. MASS is described in "Using the Mathematical Acceleration Subsystem libraries (MASS)."
- The Basic Linear Algebra Subprograms (BLAS) are a set of routines which provide matrix/vector multiplication functions tuned for Blue Gene architectures. The BLAS functions are described in "Using the Basic Linear Algebra Subprograms – BLAS" on page 89.

Using the Mathematical Acceleration Subsystem libraries (MASS)

XL C/C++ 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 C/C++ functions described in "Using the scalar library" on page 80, a set of vector libraries tuned for specific architectures described in "Using the vector libraries" on page 82, and a SIMD library described in "Using the SIMD library" on page 87. 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 -qignerrno -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 `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 89 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. Provide the prototypes for the functions (except `anint`, `cosisin`, `dnint`, `sincos`, and `rsqrt`), by including `math.h` in your source files.
2. Provide the prototypes for `anint`, `cosisin`, `dnint`, `sincos`, and `rsqrt`, by including `mass.h` in your source files.
3. Link the MASS scalar library libmass.a with your application. For instructions, see "Compiling and linking a program with MASS" on page 89.

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 19.

Table 19. MASS scalar functions

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
acos	acosf	Returns the arccosine of `x`	double acos (double x);	float acosf (float x);
acosh	acoshf	Returns the hyperbolic arccosine of `x`	double acosh (double x);	float acoshf (float x);
	anint	Returns the rounded integer value of `x`		float anint (float x);
asin	asinf	Returns the arcsine of `x`	double asin (double x);	float asinf (float x);
asinh	asinhf	Returns the hyperbolic arcsine of `x`	double asinh (double x);	float asinhf (float x);
atan2	atan2f	Returns the arctangent of `x/y`	double atan2 (double x, double y);	float atan2f (float x, float y);
atan	atanf	Returns the arctangent of `x`	double atan (double x);	float atanf (float x);
atanh	atanhf	Returns the hyperbolic arctangent of `x`	double atanh (double x);	float atanhf (float x);
cbrt	cbrtf	Returns the cube root of `x`	double cbrt (double x);	float cbrtf (float x);
copysign	copysignf	Returns `x` with the sign of `y`	double copysign (double x,double y);	float copysignf (float x);
cos	cosf	Returns the cosine of `x`	double cos (double x);	float cosf (float x);
cosh	coshf	Returns the hyperbolic cosine of `x`	double cosh (double x);	float coshf (float x);

Table 19. MASS scalar functions (continued)

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

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.h` 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 89.)

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 20 on page 83. The integer functions contained in the vector libraries are summarized in Table 21 on page 86. Note that in C and C++ applications, only call by reference is supported, even for scalar arguments.

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

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

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 parameters *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.h>

double x[500], y[500];
int n;
n = 500;
...
vexp (y, x, &n);
```

outputs a vector *y* of length 500 whose elements are exp(x[i]), where i=0,...,499.

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=0,..,*n–1. The function `vpow` outputs a vector *z* whose elements are $x[i]^{y[i]}$, where i=0,..,*n–1. The function `vatan2` outputs a vector *z* whose elements are atan(x[i]/y[i]), where i=0,..,*n–1. 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 elements of the form (cos(x[i]),sin(x[i])).

Table 20. MASS floating-point vector functions

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vacos	vsacos	Sets y[i] to the arc cosine of x[i], for i=0,..,*n-1	void vacos (double y[], double x[], int *n);	void vsacos (float y[], float x[], int *n);
vacosh	vsacosh	Sets y[i] to the hyperbolic arc cosine of x[i], for i=0,..,*n-1	void vacosh (double y[], double x[], int *n);	void vsacosh (float y[], float x[], int *n);
vasin	vsasin	Sets y[i] to the arc sine of x[i], for i=0,..,*n-1	void vasin (double y[], double x[], int *n);	void vsasin (float y[], float x[], int *n);
vasinh	vsasinh	Sets y[i] to the hyperbolic arc sine of x[i], for i=0,..,*n-1	void vasinh (double y[], double x[], int *n);	void vsasinh (float y[], float x[], int *n);
vatan2	vsatan2	Sets z[i] to the arc tangent of x[i]/y[i], for i=0,..,*n-1	void vatan2 (double z[], double x[], double y[], int *n);	void vsatan2 (float z[], float x[], float y[], int *n);
vatanh	vsatanh	Sets y[i] to the hyperbolic arc tangent of x[i], for i=0,..,*n-1	void vatanh (double y[], double x[], int *n);	void vsatanh (float y[], float x[], int *n);
vcbrt	vscbrt	Sets y[i] to the cube root of x[i], for i=0,..,*n-1	void vcbrt (double y[], double x[], int *n);	void vscbrt (float y[], float x[], int *n);
vcos	vscos	Sets y[i] to the cosine of x[i], for i=0,..,*n-1	void vcos (double y[], double x[], int *n);	void vscos (float y[], float x[], int *n);
vcosh	vscosh	Sets y[i] to the hyperbolic cosine of x[i], for i=0,..,*n-1	void vcosh (double y[], double x[], int *n);	void vscosh (float y[], float x[], int *n);
vcosisin	vscosisin	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=0,..,*n-1	void vcosisin (double _Complex y[], double x[], int *n);	void vscosisin (float _Complex y[], float x[], int *n);
vdint		Sets y[i] to the integer truncation of x[i], for i=0,..,*n-1	void vdint (double y[], double x[], int *n);	
vdiv	vsdiv	Sets z[i] to x[i]/y[i], for i=0,..,*n–1	void vdiv (double z[], double x[], double y[], int *n);	void vsdiv (float z[], float x[], float y[], int *n);
vdiv_fast [(1)]	vsdiv_fast [(2)]	Sets z[i] to x[i]/y[i], for i=0,..,*n–1.	void vdiv_fast (double z[], double x[], double y[], int *n);	void vsdiv_fast (float z[], float x[],float y[], int *n);
vdnint		Sets y[i] to the nearest integer to x[i], for i=0,..,*n-1	void vdnint (double y[], double x[], int *n);	

Table 20. MASS floating-point vector functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
verf	vserf	Sets y[i] to the error function of x[i], for i=0,..,*n-1	void verf (double y[], double x[], int *n)	void vserf (float y[], float x[], int *n)
verfc	vserfc	Sets y[i] to the complimentary error function of x[i], for i=0,..,*n-1	void verfc (double y[], double x[], int *n)	void vserfc (float y[], float x[], int *n)
vexp	vsexp	Sets y[i] to the exponential function of x[i], for i=0,..,*n-1	void vexp (double y[], double x[], int *n);	void vsexp (float y[], float x[], int *n);
vexp2	vsexp2	Sets y[i] to 2 raised to the power of x[i], for i=1,..,*n-1	void vexp2 (double y[], double x[], int *n);	void vsexp2 (float y[], float x[], int *n);
vexpm1	vsexpm1	Sets y[i] to (the exponential function of x[i])-1, for i=0,..,*n-1	void vexpm1 (double y[], double x[], int *n);	void vsexpm1 (float y[], float x[], int *n);
vexp2m1	vsexp2m1	Sets y[i] to (2 raised to the power of x[i]) - 1, for i=1,..,*n-1	void vexp2m1 (double y[], double x[], int *n);	void vsexp2m1 (float y[], float x[], int *n);
vhypot	vshypot	Sets z[i] to the square root of the sum of the squares of x[i] and y[i], for i=0,..,*n-1	void vhypot (double z[], double x[], double y[], int *n)	void vshypot (float z[], float x[], float y[], int *n)
vlog	vslog	Sets y[i] to the natural logarithm of x[i], for i=0,..,*n-1	void vlog (double y[], double x[], int *n);	void vslog (float y[], float x[], int *n);
vlog2	vslog2	Sets y[i] to the base-2 logarithm of x[i], for i=1,..,*n-1	void vlog2 (double y[], double x[], int *n);	void vslog2 (float y[], float x[], int *n);
vlog10	vslog10	Sets y[i] to the base-10 logarithm of x[i], for i=0,..,*n-1	void vlog10 (double y[], double x[], int *n);	void vslog10 (float y[], float x[], int *n);
vlog1p	vslog1p	Sets y[i] to the natural logarithm of (x[i]+1), for i=0,..,*n-1	void vlog1p (double y[], double x[], int *n);	void vslog1p (float y[], float x[], int *n);
vlog21p	vslog21p	Sets y[i] to the base-2 logarithm of (x[i]+1), for i=1,..,*n-1	void vlog21p (double y[], double x[], int *n);	void vslog21p (float y[], float x[], int *n);
vpow	vspow	Sets z[i] to x[i] raised to the power y[i], for i=0,..,*n-1	void vpow (double z[], double x[], double y[], int *n);	void vspow (float z[], float x[], float y[], int *n);
vqdrt	vsqdrt	Sets y[i] to the fourth root of x[i], for i=0,..,*n-1	void vqdrt (double y[], double x[], int *n);	void vsqdrt (float y[], float x[], int *n);
vrcbrt	vsrcbrt	Sets y[i] to the reciprocal of the cube root of x[i], for i=0,..,*n-1	void vrcbrt (double y[], double x[], int *n);	void vsrcbrt (float y[], float x[], int *n);
vrec	vsrec	Sets y[i] to the reciprocal of x[i], for i=0,..,*n-1	void vrec (double y[], double x[], int *n);	void vsrec (float y[], float x[], int *n);

Table 20. MASS floating-point vector functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vrec_fast [3]	vsrec_fast [4]	Sets y[i] to the reciprocal of x[i], for i=0,..,*n–1.	void vrec_fast (double y[], double x[], int *n);	void vsrec_fast (float y[], float x[], int *n);
vrqdrt	vsrqdrt	Sets y[i] to the reciprocal of the fourth root of x[i], for i=0,..,*n-1	void vrqdrt (double y[], double x[], int *n);	void vsrqdrt (float y[], float x[], int *n);
vrsqrt	vsrsqrt	Sets y[i] to the reciprocal of the square root of x[i], for i=0,..,*n-1	void vrsqrt (double y[], double x[], int *n);	void vsrsqrt (float y[], float x[], int *n);
vsin	vssin	Sets y[i] to the sine of x[i], for i=0,..,*n-1	void vsin (double y[], double x[], int *n);	void vssin (float y[], float x[], int *n);
vsincos	vssincos	Sets y[i] to the sine of x[i] and z[i] to the cosine of x[i], for i=0,..,*n-1	void vsincos (double y[], double z[], double x[], int *n);	void vssincos (float y[], float z[], float x[], int *n);
vsinh	vssinh	Sets y[i] to the hyperbolic sine of x[i], for i=0,..,*n-1	void vsinh (double y[], double x[], int *n);	void vssinh (float y[], float x[], int *n);
vsqrt	vssqrt	Sets y[i] to the square root of x[i], for i=0,..,*n-1	void vsqrt (double y[], double x[], int *n);	void vssqrt (float y[], float x[], int *n);
vtan	vstan	Sets y[i] to the tangent of x[i], for i=0,..,*n-1	void vtan (double y[], double x[], int *n);	void vstan (float y[], float x[], int *n);
vtanh	vstanh	Sets y[i] to the hyperbolic tangent of x[i], for i=0,..,*n-1	void vtanh (double y[], double x[], int *n);	void vstanh (float y[], float x[], int *n);

Notes:

1. vdiv_fast arguments must satisfy all the following conditions for i=0,...,*n-1:
   - $2^{-1021} \le |y[i]| \le 2^{1020}$
   - 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=0,...,*n-1:
   - $2^{-125} \le |y[i]| \le 2^{124}$
   - 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=0,...,*n-1:
   - $2^{-1021} \le |x[i]| \le 2^{1020}$
4. vsrec_fast arguments must satisfy the following condition for i=0,...,*n-1:
   - $2^{-125} \le |x[i]| \le 2^{124}$

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 (integral or floating-point), and *n is the vector length.

Table 21. MASS integer vector library functions

Function	Description	Prototype
vpopcnt4	Returns the total number of 1 bits in the concatenation of the binary representation of `x[i]`, for i=0,..,*n–1 , where `x` is a vector of 32-bit objects.	unsigned int vpopcnt4 (void *x, int *n)
vpopcnt8	Returns the total number of 1 bits in the concatenation of the binary representation of `x[i]`, for i=0,..,*n–1 , where `x` is a vector of 64-bit objects.	unsigned int vpopcnt8 (void *x, int *n)

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[0:n-1]` and `y[0:n-1]` 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[0:n-1]` and `x1[0:n-1]` must be either disjoint or identical; and `y[0:n-1]` and `x2[0:n-1]` 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[0:n-1]` and `x[0:n-1]` must be either disjoint or identical; and `y2[0:n-1]` and `x[0:n-1]` must be either disjoint or identical. Also, the vectors `y1[0:n-1]` and `y2[0:n-1]` 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 information in the *XL C/C++ Compiler Reference*

-D

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 prototypes for the functions by including mass_simd.h 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 89.

The single/double-precision MASS SIMD functions accept single/double-precision arguments and return single/double-precision results. They are summarized in Table 22.

Table 22. MASS SIMD functions

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
acosd4	acosf4	Computes the arc cosine of each element of vx.	vector4double acosd4 (vector4double vx);	vector4double acosf4 (vector4double vx);
acoshd4	acoshf4	Computes the arc hyperbolic cosine of each element of vx.	vector4double acoshd4 (vector4double vx);	vector4double acoshf4 (vector4double vx);
asind4	asinf4	Computes the arc sine of each element of vx.	vector4double asind4 (vector4double vx);	vector4double asinf4 (vector4double vx);
asinhd4	asinhf4	Computes the arc hyperbolic sine of each element of vx.	vector4double asinhd4 (vector4double vx);	vector4double asinhf4 (vector4double vx);
atand4	atanf4	Computes the arc tangent of each element of vx.	vector4double atand4 (vector4double vx);	vector4double atanf4 (vector4double vx);
atan2d4	atan2f4	Computes the arc tangent of each element of vy/vx.	vector4double atan2d4 (vector4double vx, vector4double vy);	vector4double atan2f4 (vector4double vx, vector4double vy);
atanhd4	atanhf4	Computes the arc hyperbolic tangent of each element of vx.	vector4double atanhd4 (vector4double vx);	vector4double atanhf4 (vector4double vx);
cbrtd4	cbrtf4	Computes the cube root of each element of vx.	vector4double cbrtd4 (vector4double vx);	vector4double cbrtf4 (vector4double vx);
cosd4	cosf4	Computes the cosine of each element of vx.	vector4double cosd4 (vector4double vx);	vector4double cosf4 (vector4double vx);
coshd4	coshf4	Computes the hyperbolic cosine of each element of vx.	vector4double coshd4 (vector4double vx);	vector4double coshf4 (vector4double vx);
cosisind4	cosisinf4	Computes the cosine and sine of each element of x, and stores the results in y and z as follows: Sets y to {cos(x1), sin(x1), cos(x2), sin(x2)} and z to {cos(x3), sin(x3), cos(x4), sin(x4)} where x={x1,x2,x3,x4}.	void cosisind4 (vector4double x, vector4double *y, vector4double *z)	void cosisinf4 (vector4double x, vector4double *y, vector4double *z)
divd4	divf4	Computes the quotient vx/vy.	vector4double divd4 (vector4double vx, vector4double vy);	vector4double divf4 (vector4double vx, vector4double vy);
div_fastd4 (1)	div_fastf4 (2)	Computes the quotient vx/vy.	vector4double div_fastd4 (vector4double vx, vector4double vy);	vector4double div_fastf4 (vector4double vx, vector4double vy);
erfcd4	erfcf4	Computes the complementary error function of each element of vx.	vector4double erfcd4 (vector4double vx);	vector4double erfcf4 (vector4double vx);
erfd4	erff4	Computes the error function of each element of vx.	vector4double erfd4 (vector4double vx);	vector4double erff4 (vector4double vx);
expd4	expf4	Computes the exponential function of each element of vx.	vector4double expd4 (vector4double vx);	vector4double expf4 (vector4double vx);
exp2d4	exp2f4	Computes 2 raised to the power of each element of vx.	vector4double exp2d4 (vector4double vx);	vector4double exp2f4 (vector4double vx);
expm1d4	expm1f4	Computes (the exponential function of each element of vx) - 1.	vector4double expm1d4 (vector4double vx);	vector4double expm1f4 (vector4double vx);
exp2m1d4	exp2m1f4	Computes (2 raised to the power of each element of vx) -1.	vector4double exp2m1d4 (vector4double vx);	vector4double exp2m1f4 (vector4double vx);
hypotd4	hypotf4	For each element of vx and the corresponding element of vy, computes sqrt(x*x+y*y).	vector4double hypotd4 (vector4double vx, vector4double vy);	vector4double hypotf4 (vector4double vx, vector4double vy);

Table 22. MASS SIMD functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
lgammad4	lgammaf4	Computes the natural logarithm of the absolute value of the Gamma function of each element of vx .	vector4double lgammad4 (vector4double vx);	vector4double lgammaf4 (vector4double vx);
logd4	logf4	Computes the natural logarithm of each element of vx.	vector4double logd4 (vector4double vx);	vector4double logf4 (vector4double vx);
log2d4	log2f4	Computes the base-2 logarithm of each element of vx.	vector4double log2d4 (vector4double vx);	vector4double log2f4 (vector4double vx);
log10d4	log10f4	Computes the base-10 logarithm of each element of vx.	vector4double log10d4 (vector4double vx);	vector4double log10f4 (vector4double vx);
log1pd4	log1pf4	Computes the natural logarithm of each element of (vx +1).	vector4double log1pd4 (vector4double vx);	vector4double log1pf4 (vector4double vx);
log21pd4	log21pf4	Computes the base-2 logarithm of each element of (vx +1).	vector4double log21pd4 (vector4double vx);	vector4double log21pf4 (vector4double vx);
powd4	powf4	Computes each element of vx raised to the power of the corresponding element of vy.	vector4double powd4 (vector4double vx, vector4double vy);	vector4double powf4 (vector4double vx, vector4double vy);
qdrtd4	qdrtf4	Computes the quad root of each element of vx.	vector4double qdrtd4 (vector4double vx);	vector4double qdrtf4 (vector4double vx);
rcbrtd4	rcbrtf4	Computes the reciprocal of the cube root of each element of vx.	vector4double rcbrtd4 (vector4double vx);	vector4double rcbrtf4 (vector4double vx);
recipd4	recipf4	Computes the reciprocal of each element of vx.	vector4double recipd4 (vector4double vx);	vector4double recipf4 (vector4double vx);
recip_fastd4 (3)	recip_fastf4 (4)	Computes the reciprocal of each element of vx.	vector4double recip_fastd4 (vector4double vx);	vector4double recip_fastf4 (vector4double vx);
rqdrtd4	rqdrtf4	Computes the reciprocal of the quad root of each element of vx.	vector4double rqdrtd4 (vector4double vx);	vector4double rqdrtf4 (vector4double vx);
rsqrtd4	rsqrtf4	Computes the reciprocal of the square root of each element of vx.	vector4double rsqrtd4 (vector4double vx);	vector4double rsqrtf4 (vector4double vx);
sincosd4	sincosf4	Computes the sine and cosine of each element of vx.	void sincosd4 (vector4double vx, vector4double *vs, vector4double *vc);	void sincosf4 (vector4double vx, vector4double *vs, vector4double *vc);
sind4	sinf4	Computes the sine of each element of vx.	vector4double sind4 (vector4double vx);	vector4double sinf4 (vector4double vx);
sinhd4	sinhf4	Computes the hyperbolic sine of each element of vx.	vector4double sinhd4 (vector4double vx);	vector4double sinhf4 (vector4double vx);
sqrtd4	sqrtf4	Computes the square root of each element of vx.	vector4double sqrtd4 (vector4double vx);	vector4double sqrtf4 (vector4double vx);
tand4	tanf4	Computes the tangent of each element of vx.	vector4double tand4 (vector4double vx);	vector4double tanf4 (vector4double vx);
tanhd4	tanhf4	Computes the hyperbolic tangent of each element of vx.	vector4double tanhd4 (vector4double vx);	vector4double tanhf4 (vector4double 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, $2^{-102} \le |xi| < \infty$ and $2^{-124} \le |xi/yi| \le 2^{126}$
3. recip_fastd4 arguments must satisfy the following condition for each element xi of x:
   - $2^{-1021} \le |xi| \le 2^{1020}$
4. recip_fastf4 arguments must satisfy the following condition for each element xi of x:
   - $2^{-125} \le |xi| \le 2^{124}$

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

```
bgxlc progc.c -o progc -lmass -lmassv
```

Link with SIMD library libmass_simd.a

```
bgxlc progc.c -o progc -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 **bgxlc**, specifying **-lfasttan** instead of **-lmass**:

   ```
   bgxlc sample.c -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 C/C++ pow function 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 the XL C/C++ compiler 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

Because the BLAS routines are written in Fortran, all parameters are passed to them by reference, and all arrays are stored in column-major order.

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 prototypes and parameters for the XL C/C++ BLAS functions. The interfaces for these functions are similar to those of the equivalent BLAS functions shipped in IBM's Engineering and Scientific Subroutine Library (ESSL); for more information and examples of usage of these functions, see *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 92 describes how to link to the XL C/C++ `libxlopt` library if you are also using a third-party BLAS library.

BLAS function syntax

The prototypes for the `sgemv` and `dgemv` functions are as follows:

```
void sgemv(const char *trans, int *m, int *n, float *alpha,
     void *a, int *lda, void *x, int *incx,
     float *beta, void *y, int *incy);
```

```
void dgemv(const char *trans, int *m, int *n, double *alpha,
     void *a, int *lda, void *x, int *incx,
      double *beta, void *y, int *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 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 for matrix *a*

a is the input matrix of `float` (for `sgemv`) or `double` (for `dgemv`) values

lda
: is the leading dimension of the array specified by *a*. The leading dimension must be greater than zero. The leading dimension must be greater than or equal to 1 and greater than or equal to the value specified in *m*.

x is the input vector of `float` (for `sgemv`) or `double` (for `dgemv`) values.

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

beta
is the scaling constant for vector *y*

y is the output vector of `float` (for `sgemv`) or `double` (for `dgemv`) 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:

```
void sgemm(const char *transa, const char *transb,
   int *l, int *n, int *m, float *alpha,
   const void *a, int *lda, void *b, int *ldb,
   float *beta, void *c, int *ldc);
```

```
void dgemm(const char *transa, const char *transb,
   int *l, int *n, int *m, double *alpha,
   const void *a, int *lda, void *b, int *ldb,
   double *beta, void *c, int *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 greater than or equal to zero, and less than 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 for matrix *a*

a is the input matrix *a* of `float` (for `sgemm`) or `double` (for `dgemm`) 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 `float` (for `sgemm`) or `double` (for `dgemm`) 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 for matrix *c*

c is the output matrix *c* of `float` (for `sgemm`) or `double` (for `dgemm`) 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 0 and 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 the XL C/C++ compiler. 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.a`, you would compile your code with the following command:

```
bgxlc app.c -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.a` library.

Chapter 13. Parallelizing your programs

The compiler offers you the following methods of implementing shared memory program parallelization:

- Automatic parallelization of countable program loops, which are defined in "Countable loops" on page 94. An overview of the compiler's automatic parallelization capabilities is provided in "Enabling automatic parallelization" on page 95.
- Explicit parallelization of C and C++ program code using pragma directives compliant to the OpenMP Application Program Interface specification. An overview of the OpenMP directives is provided in "Using OpenMP directives" on page 95.

All methods of program parallelization are enabled when the **-qsmp** compiler option is in effect without the **omp** suboption. You can enable strict OpenMP compliance with the **-qsmp=omp** compiler option, but doing so will disable automatic parallelization.

Note: The **-qsmp** option must only be used together with thread-safe compiler invocation modes (those that contain the **_r** suffix).

Parallel regions of program code are executed by multiple threads, possibly running on multiple processors. The number of threads created is determined by environment variables and calls to library functions. Work is distributed among available threads according to scheduling algorithms specified by the environment variables. For any of the methods of parallelization, you can use the XLSMPOPTS environment variable and its suboptions to control thread scheduling; for more information on this environment variable, see *XLSMPOPTS* in the *XL C/C++ Compiler Reference*. If you are using OpenMP constructs, you can use the OpenMP environment variables to control thread scheduling; for information on OpenMP environment variables, see *OpenMP environment variables for parallel processing* in the *XL C/C++ Compiler Reference*. For more information on OpenMP built-in functions, see *Built-in functions for parallel processing* in the *XL C/C++ Compiler Reference*.

For a complete discussion on how threads are created and utilized, refer to the *OpenMP Application Program Interface Specification*, available at http://www.openmp.org.

Related information:

"Using shared-memory parallelism (SMP)" on page 47

Related information in the *XL C/C++ Compiler Reference*

XLSMPOPTS

OpenMP environment variables for parallel processing

Built-in functions for parallel processing

Related external information

OpenMP Application Program Interface Language Specification, available at http://www.openmp.org

Countable loops

Loops are considered to be countable if they take any of the following forms:

Countable for loop syntax with single statement

```
►►──for──(──┬────────────────────┬──;──exit_condition──;──increment_expression──)──────►
            └─iteration_variable─┘

►─statement──────────────────────────────────────────────────────────────────────────►◄
```

Countable for loop syntax with statement block

```
►►──for──(──┬────────────────────┬──;──┬────────────┬──)──────────────────────────────►
            └─iteration_variable─┘     └─expression─┘

►─{──┬──────────────────┬──┬────────────────┬──increment_expression──┬────────────────┬──}──────►◄
     └─declaration_list─┘  └─statement_list─┘                        └─statement_list─┘
```

Countable while loop syntax

```
►►──while──(──exit_condition──)──────────────────────────────────────►

►─{──┬──────────────────┬──┬────────────────┬──increment_expression──}──────►◄
     └─declaration_list─┘  └─statement_list─┘
```

Countable do while loop syntax

```
►►──do──{──┬──────────────────┬──┬────────────────┬──increment_expression──}──while──(──exit_condition──)──────►◄
           └─declaration_list─┘  └─statement_list─┘
```

The following definitions apply to these syntax diagrams:

iteration_variable
: is a signed integer that has either automatic or register storage class, does not have its address taken, and is not modified anywhere in the loop except in the *increment_expression*.

exit_condition
: takes the following form:

```
├──increment_variable──┬─<=─┬──expression──────────────────────────────┤
                       ├─<──┤
                       ├─>=─┤
                       └─>──┘
```

where *expression* is a loop-invariant signed integer expression. *expression* cannot reference external or static variables, pointers or pointer expressions, function calls, or variables that have their address taken.

increment_expression
: takes any of the following forms:

- *++iteration_variable*
- *--iteration_variable*
- *iteration_variable++*
- *iteration_variable--*
- *iteration_variable* += *increment*

- *iteration_variable* -= *increment*
- *iteration_variable* = *iteration_variable* + *increment*
- *iteration_variable* = *increment* + *iteration_variable*
- *iteration_variable* = *iteration_variable* - *increment*

where *increment* is a loop-invariant signed integer expression. The value of the expression is known at run time and is not 0. *increment* cannot reference external or static variables, pointers or pointer expressions, function calls, or variables that have their address taken.

Enabling automatic parallelization

The compiler can automatically locate and parallelize all countable loops where possible in your program code. A loop is considered to be countable if it has any of the forms shown in "Countable loops" on page 94, and:

- There is no branching into or out of the loop.
- The *increment_expression* is not within a critical section.

In general, a countable loop is automatically parallelized only if all of the following conditions are met:

- The order in which loop iterations start or end does not affect the results of the program.
- The loop does not contain I/O operations.
- Floating point reductions inside the loop are not affected by round-off error, unless the **-qnostrict** option is in effect.
- The **-qnostrict_induction** compiler option is in effect.
- The **-qsmp=auto** compiler option is in effect.
- The compiler is invoked with a thread-safe compiler mode.

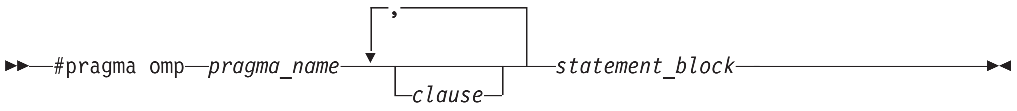
Using OpenMP directives

OpenMP directives exploit shared memory parallelism by defining various types of parallel regions. Parallel regions can include both iterative and non-iterative segments of program code.

Pragmas fall into these general categories:

1. Pragmas that let you define parallel regions in which work is done by threads in parallel (**#pragma omp parallel**). Most of the OpenMP directives either statically or dynamically bind to an enclosing parallel region.
2. Pragmas that let you define how work is distributed or shared across the threads in a parallel region (**#pragma omp section**, **#pragma omp for**, **#pragma omp single**, **#pragma omp task**).
3. Pragmas that let you control synchronization among threads (**#pragma omp atomic**, **#pragma omp master**, **#pragma omp barrier**, **#pragma omp critical**, **#pragma omp flush**, **#pragma omp ordered**) .
4. Pragmas that let you define the scope of data visibility across threads (**#pragma omp threadprivate**).
5. Pragmas for task synchronization (**#pragma omp taskwait**, **#pragma omp barrier**)

OpenMP directive syntax



Pragmas can be controlled by clauses. For example, a `num_threads` clause can be used to control a parallel region pragma.

Pragma directives generally appear immediately before the section of code to which they apply. For example, the following example defines a parallel region in which iterations of a for loop can run in parallel:

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

This example defines a parallel region in which two or more non-iterative sections of program code can run in parallel:

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
       structured_block_1
           ...
    #pragma omp section
       structured_block_2
           ...
         ....
  }
}
```

For a pragma-by-pragma description of the OpenMP directives, refer to *Pragma directives for parallel processing* in the *XL C/C++ Compiler Reference*.

Related information in the *XL C/C++ Compiler Reference*

Pragma directives for parallel processing

OpenMP built-in functions

OpenMP environment variables for parallel processing

Shared and private variables in a parallel environment

Variables can have either shared or private context in a parallel environment. Variables in shared context are visible to all threads running in associated parallel loops. Variables in private context are hidden from other threads. Each thread has its own private copy of the variable, and modifications made by a thread to its copy are not visible to other threads.

The default context of a variable is determined by the following rules:

- Variables with `static` storage duration are shared.
- Dynamically allocated objects are shared.

- Variables with automatic storage duration are private.
- Variables in heap allocated memory are shared. There can be only one shared heap.
- All variables defined outside a parallel construct become shared when the parallel loop is encountered.
- Loop iteration variables are private within their loops. The value of the iteration variable after the loop is the same as if the loop were run sequentially.
- Memory allocated within a parallel loop by the `alloca` function persists only for the duration of one iteration of that loop, and is private for each thread.

The following code segments show examples of these default rules:

```
int E1;                        /* shared static     */

void main (argvc,...) {        /* argvc is shared   */
   int i;                       /* shared automatic  */

void *p = malloc(...);       /* memory allocated by malloc   */
                                /* is accessible by all threads */
                                /* and cannot be privatized     */

#pragma omp parallel firstprivate (p)
   {
     int b;                     /* private automatic  */
     static int s;              /* shared static      */

     #pragma omp for
     for (i =0;...) {
       b = 1;                   /* b is still private here !    */
       foo (i);                 /* i is private here because it */
                                /* is an iteration variable     */
      }


#pragma omp parallel
     {
       b = 1;                   /* b is shared here because it  */
                                /* is another parallel region   */
     }
   }
 }


int E2;                        /*shared static */

void foo (int x) {             /* x is private for the parallel */
                                /* region it was called from     */

int c;                       /* the same */
 ... }
```

Some OpenMP clauses let you specify visibility context for selected data variables. A brief summary of data scope attribute clauses are listed below:

Data scope attribute clause	Description
private	The **private** clause declares the variables in the list to be private to each thread in a team.
firstprivate	The **firstprivate** clause provides a superset of the functionality provided by the private clause.

Data scope attribute clause	Description
lastprivate	The **lastprivate** clause provides a superset of the functionality provided by the private clause.
shared	The **shared** clause shares variables that appear in the list among all the threads in a team. All threads within a team access the same storage area for shared variables.
reduction	The **reduction** clause performs a reduction on the scalar variables that appear in the list, with a specified operator.
default	The **default** clause allows the user to affect the data scope attributes of variables.

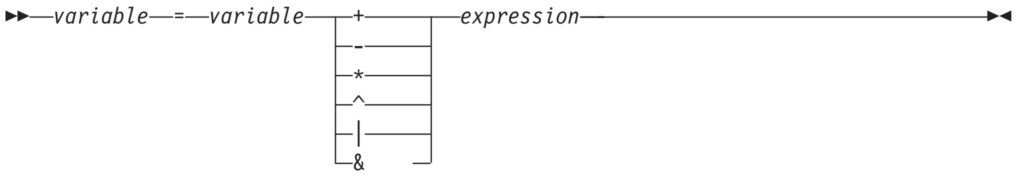
For more information, see the OpenMP directive descriptions in "Pragma directives for parallel processing" in the *XL C/C++ Compiler Reference*. You can also refer to the *OpenMP Application Program Interface Language Specification*, which is available at http://www.openmp.org.

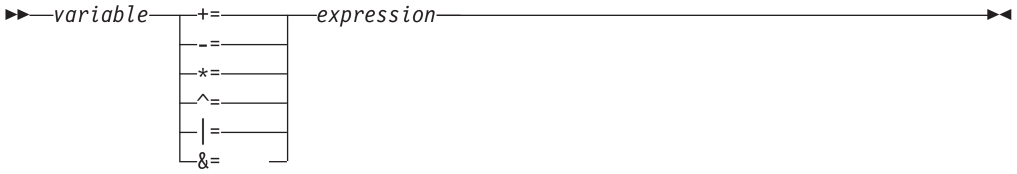
Related information in the *XL C/C++ Compiler Reference*

Pragma directives for parallel processing

Reduction operations in parallelized loops

The compiler can recognize and properly handle most reduction operations in a loop during both automatic and explicit parallelization. In particular, it can handle reduction statements that have either of the following forms:





where:

variable
is an identifier designating an automatic or register variable that does not have its address taken and is not referenced anywhere else in the loop, including all loops that are nested. For example, in the following code, only S in the nested loop is recognized as a reduction:

```
int i,j, S=0;
for (i= 0 ;i < N; i++) {
    S = S+ i;
     for (j=0;j< M; j++) {
        S = S + j;
    }
}
```

expression
is any valid expression.

Recognized reductions are listed by the **-qinfo=reduction** option. OpenMP directives provide you with mechanisms to specify reduction variables explictly.

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 C/C++ 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 **omp parallel for** and **omp parallel sections** directives apply to the **speculative for** and **speculative sections** directives.

Speculative threads are not able to detect access conflicts in OpenMP **threadprivate** data or variables that are marked with the `__thread` specifier. 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
- Built-in functions for thread-level speculative execution
- Environment variables for thread-level speculative execution
- #pragma speculative for
- #pragma speculative sections
- #pragma omp threadprivate
- The __thread storage class specifier (IBM extension)

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
- Built-in functions for transactional memory
- Environment variables for transactional memory
- #pragma 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.

On Blue Gene/Q platforms, the POMP implementation is available as part of the SMP runtime of XL C/C++. The POMP implementation does not require any changes in the source code. Instead, you must provide a timer probe library and link your program with a POMP-enabled SMP runtime.

Timer probe library

A timer probe library is a separately compiled library that provides implementation of all POMP callback functions. You can write the timer probe library from scratch, so that the callback events can be handled in the way that your program determines.

For a full list of POMP callback functions that are supported by the SMP runtime, see POMP callback functions. For a complete sample timer probe library, see Creating a sample timer probe library.

Performance consideration

Due to some possible performance overhead, POMP is not enabled in the default runtime `libxlsmp.a`. This might cause a few extra branches and function calls. Be aware that the performance overhead might affect your results.

When you write a timer probe library, do not use any lock on, for example, an output file; otherwise, your application might be slow down. For example, if the `POMP_Parallel_begin` function uses a lock, threads are forced to enter a parallel region one by one. The sample timer probe takes extra effort to support asynchronous printing, and also tries to print output with atomic write calls so that the lines may be atomically interleaved, but the output will not contain a partial line.

Related information

- POMP callback functions
- Sample timer probe library

Creating a sample timer probe library

This section provides four sample source files that you can use to create a complete timer probe library:

- `probe.c` and `probe.h` provide POMP callback functions. For a full list of POMP callback functions supported by the SMP runtime, see POMP callback functions.
- `printing.c` and `printing.h` provide output functions.

Sample working directory: `/opt/ibmcmp/vacpp/bg/12.1/bin`

You can compile and create the timer probe library `libprobe.a` by using the following commands:

```
/opt/ibmcmp/vacpp/bg/12.1/bin/bgxlc -qsmp=omp -c -o probe.o probe.c

/opt/ibmcmp/vacpp/bg/12.1/bin/bgxlc -qsmp=omp -c -o printing.o printing.c

ar cr libprobe.a probe.o printing.o
```

probe.c

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#include "probe.h"
#include "printing.h"

int32_t POMP_Init(void) {
    print("initializing POMP timer probe\n");

    init_timing();
}

void POMP_Finalize(void) {
    print("exiting\n");
}

int32_t POMP_Get_handle(POMP_Handle_t *handle, char ctc[]) {
    enum {
        MODE_SEPARATOR,  /* looking for and skipping a '*' */
        MODE_KEY,        /* looking for a key, ending with '=' */
        MODE_VALUE,      /* looking for a value, ending with '*' */
        MODE_INTERESTING_VALUE,  /* the value we're looking for */
        MODE_DONE        /* finished parsing */
    } mode = MODE_SEPARATOR;

    int length;
    char *start = ctc, *current = ctc;

    *handle = "unknown";  /* assume nothing found at the beginning */

    length = strtol(current, &current, 0);
    if(current && *current == '*') current ++;  /* skip first '*' */

    while(current && *current && mode != MODE_DONE) {
        switch(mode) {
        case MODE_SEPARATOR:
            if(*current == '*') {  /* two '*' in a row */
```

```
                mode = MODE_DONE;
            }
            else {
                start = current;
                mode = MODE_KEY;
            }
            break;
        case MODE_KEY:
            if(*current == '=') {
                if(current - start == 4 && strncmp(start, "sscl", 4) == 0) {
                    mode = MODE_INTERESTING_VALUE;
                }
                else mode = MODE_VALUE;

                current ++;
                start = current;
            }
            else current ++;
            break;
        case MODE_VALUE:
        case MODE_INTERESTING_VALUE:
            if(*current == '*') {
                if(mode == MODE_INTERESTING_VALUE) {
                    char *data = malloc(current - start + 1);
                    memcpy(data, start, current - start);
                    data[current - start] = 0;

                    *handle = data;
                }

                current ++;
                mode = MODE_SEPARATOR;
            }
            else current ++;
            break;
        default:
            break;
        }
    }
}


int32_t POMP_Parallel_enter(
    POMP_Handle_t *handle,
    int32_t thread_id,
    int32_t num_threads,
    int32_t if_expr_result,
    char ctc[]) {

    print_callback(*handle, thread_id,
        "enter parallel region with %d threads, if_expr_result=%d\n",
        num_threads, if_expr_result);
}

int32_t POMP_Parallel_begin(POMP_Handle_t *handle, int32_t thread_id) {
    print_callback(*handle, thread_id,
        "begin parallel region\n");
    add_indent(thread_id);
}

int32_t POMP_Parallel_end(POMP_Handle_t *handle, int32_t thread_id) {
    subtract_indent(thread_id);
    print_callback(*handle, thread_id,
        "end parallel region\n");
}

int32_t POMP_Parallel_exit(POMP_Handle_t *handle, int32_t thread_id) {
    print_callback(*handle, thread_id,
        "exit parallel region\n");
}


int32_t POMP_Loop_enter(
    POMP_Handle_t *handle,
    int32_t thread_id,
    int64_t chunk_size,
    int64_t init_iter,
    int64_t final_iter,
    int64_t incr,
    char ctc[]) {

    print_callback(*handle, thread_id,
        "enter loop [%ld,%ld) chunk size %ld\n",
        init_iter, final_iter, chunk_size);
    add_indent(thread_id);
}

int32_t POMP_Loop_chunk_begin(POMP_Handle_t *handle,
    int32_t thread_id,
    int64_t init_iter,
    int64_t final_iter) {

    print_callback(*handle, thread_id,
        "begin chunk [%d,%d)\n",
        init_iter, final_iter);
}

int32_t POMP_Loop_chunk_end(POMP_Handle_t *handle, int32_t thread_id) {
    print_callback(*handle, thread_id,
        "end chunk\n");
}

int32_t POMP_Loop_exit(POMP_Handle_t *handle, int32_t thread_id) {
    subtract_indent(thread_id);
    print_callback(*handle, thread_id,
        "exit loop\n");
}
```

probe.h

```
#ifndef PROBE_H
#define PROBE_H

#include <stdint.h>  /* for int32_t */

typedef void *POMP_Handle_t;

/*
    POMP functions for initialization/etc.

    POMP_Init() is called by the runtime.
    POMP_Finalize() must be called manually (with atexit()).
    POMP_Get_handle() is called by the runtime.
*/

int32_t POMP_Init(void);
void POMP_Finalize(void);
int32_t POMP_Get_handle(POMP_Handle_t *handle, char ctc[]);

/*
    POMP_Parallel_*() callback functions
*/
int32_t POMP_Parallel_enter(
    POMP_Handle_t *handle,
    int32_t thread_id,
    int32_t num_threads,
    int32_t if_expr_result,
    char ctc[]);
int32_t POMP_Parallel_begin(POMP_Handle_t *handle, int32_t thread_id);
int32_t POMP_Parallel_end(POMP_Handle_t *handle, int32_t thread_id);
int32_t POMP_Parallel_exit(POMP_Handle_t *handle, int32_t thread_id);

/*
    POMP_Loop_*() callback functions
*/
int32_t POMP_Loop_enter(
    POMP_Handle_t *handle,
    int32_t thread_id,
    int64_t chunk_size,
    int64_t init_iter,
    int64_t final_iter,
    int64_t incr,
    char ctc[]);
int32_t POMP_Loop_chunk_begin(POMP_Handle_t *handle,
    int32_t thread_id,
    int64_t init_iter,
    int64_t final_iter);
int32_t POMP_Loop_chunk_end(POMP_Handle_t *handle, int32_t thread_id);
int32_t POMP_Loop_exit(POMP_Handle_t *handle, int32_t thread_id);

#endif
```

printing.c

```
#include <stdio.h>
#include <stdarg.h>

#include "printing.h"

#ifdef HAVE_WRITE
    #include <unistd.h>
#endif

#ifdef HAVE_GETTIMEOFDAY
    #include <sys/time.h>
#else
    #include <time.h>
#endif

static int thread_indent[MAX_THREADS];

void print(const char *format, ...) {
    char buffer[MAX_MESSAGE_LENGTH];
    va_list arg;
    va_start(arg, format);

    strcpy(buffer, "probe: ");  /* 7 chars */

#ifdef HAVE_SNPRINTF
    vsnprintf(buffer+7, MAX_MESSAGE_LENGTH, format, arg);
#else
    vsprintf(buffer+7, format, arg);
#endif

    /* We format the string into a buffer and then call write directly,
       because this has the least chance of interleaving output due to
       threading. The output might still be interleaved but this is about
       the best we can do without using locks and possibly affecting the
       behaviour of the program drastically, or implementing a complex
       non-locking data structure.
    */
    write(STDERR_FILENO, buffer, strlen(buffer));

    va_end(arg);
}

void print_callback(const char *source, int32_t thread,
    const char *format, ...) {

    char buffer[MAX_MESSAGE_LENGTH];
    int length = 0;
    int indent = thread_indent[thread];
    va_list arg;
    va_start(arg, format);

    strcpy(buffer, "probe: ");  /* 7 chars */
    length += 7;

#ifdef HAVE_SNPRINTF
    length += snprintf(buffer + length, MAX_MESSAGE_LENGTH - length,
#else
    length += sprintf(buffer + length,
#endif
        "[%8.6f %s thd %2d] ", current_time(), source, thread);

    while(indent) {
        buffer[length ++] = ' ';
        buffer[length ++] = ' ';
        indent --;
    }

#ifdef HAVE_VSNPRINTF
    vsnprintf(buffer + length, MAX_MESSAGE_LENGTH - length, format, arg);
#else
    vsprintf(buffer + length, format, arg);
#endif

    /* We format the string into a buffer and then call write directly,
       because this has the least chance of interleaving output due to
       threading. The output might still be interleaved but this is about
       the best we can do without using locks and possibly affecting the
       behaviour of the program drastically, or implementing a complex
       non-locking data structure.
    */
    write(STDERR_FILENO, buffer, strlen(buffer));

    va_end(arg);
}

void add_indent(int32_t thread) {
    thread_indent[thread] ++;
}

void subtract_indent(int32_t thread) {
    if(thread_indent[thread]) {
        thread_indent[thread] --;
    }
}

#ifdef HAVE_GETTIMEOFDAY
static double initial_time;

void init_timing(void) {
    initial_time = current_time();
}

double current_time(void) {
    struct timeval tv;
    double seconds;

    gettimeofday(&tv, NULL);

    seconds = tv.tv_sec;
    seconds += tv.tv_usec / 1000000.0;
    return seconds - initial_time;
}
#else
static clock_t initial_time;

void init_timing(void) {
    initial_time = clock();
}

double current_time(void) {
    clock_t now = clock();

    return (double)(now - initial_time) / CLOCKS_PER_SEC;
}
#endif
```

printing.h

```
#ifndef PRINTING_H
#define PRINTING_H

#include <stdint.h>  /* for int32_t, int64_t */

#define HAVE_SNPRINTF
#define HAVE_VSNPRINTF
#define HAVE_WRITE
#define HAVE_GETTIMEOFDAY

#define MAX_THREADS 128
#define MAX_MESSAGE_LENGTH 256

void print(const char *format, ...);
void print_callback(const char *source, int32_t thread,
    const char *format, ...);

void add_indent(int32_t thread);
void subtract_indent(int32_t thread);

void init_timing(void);
double current_time(void);

#endif
```

Related information

- Profiler for OpenMP
- POMP callback functions

Linking with POMP-enabled SMP runtime

To use POMP, you must link your program with a POMP-enabled SMP runtime (libxlsmp_pomp.a) and specify a timer probe library. To link with the SMP runtime, simply pass -lxlsmp_pomp to XL C/C++. For example:

```
bgxlc_r -qsmp=omp ptest.c -o ptest -lxlsmp_pomp -L . -lprobe
```

If you are linking with libxlsmp_pomp.a, but do not specify a timer probe library, the complier issues linker errors similar as follows:

```
/path/to/libxlsmp_pomp.a(pardo.pomp64.o):(.text+0x4658):
undefined reference to `POMP_Loop_chunk_end'
/path/to/libxlsmp_pomp.a(pardo.pomp64.o):(.text+0x4684):
undefined reference to `POMP_Loop_chunk_begin'
/path/to/libxlsmp_pomp.a(pardo.pomp64.o):(.text+0x6350):
undefined reference to `POMP_Init'
```

The compiler also issues linker errors for each callback function defined in the POMP implementation. You can also see a full list of symbols by looking for undefined symbols in `libxlsmp_pomp.a`:

```
nm /path/to/libxlsmp_pomp.a | grep POMP | sort | uniq
                 POMP_Finalize
                 POMP_Get_handle
                 POMP_InitU POMP_Loop_chunk_begin
                 POMP_Loop_chunk_end
                 POMP_Loop_enter
                 POMP_Loop_exit
                 POMP_Parallel_begin
                 POMP_Parallel_end
                 POMP_Parallel_enter
                 POMP_Parallel_exit
```

Related information

- Profiler for OpenMP
- Creating a sample timer probe library
- POMP callback functions

Running a complete OpenMP program

You must build a timer probe library before you can compile and run an OpenMP program with POMP. For details, see Creating a sample timer probe library.

Use the following command to compile and link the sample OpenMP program `ptest`:

```
/opt/ibmcmp/vacpp/bg/12.1/bin/bgxlc_r -qsmp=omp ptest.c -o ptest -lxlsmp_pomp -L . -lprobe
```

It is assumed that `libprobe.a` is in the current directory; otherwise, adjust the `-L` path accordingly.

Use the following command to run `ptest`:

```
runjob --block R00-M0-N06 --corner R00-M0-N06-J12 --shape 1x1x1x1x1
--envs OMP_NUM_THREADS=4 : ptest | tee output
```

ptest.c

```
#include <omp.h>
#include <stdio.h>

#define SCREEN_WIDTH 80
#define TABLE_SIZE SCREEN_WIDTH*20

int main() {
 int i, j;
 int prime[TABLE_SIZE + 1];

 #pragma omp parallel
 {
  printf("Greetings from thread %d.\n", omp_get_thread_num());
 }
 /* Also try schedule(dynamic, SCREEN_WIDTH) for more random output */
 #pragma omp parallel for shared(prime) private(i, j)
 for(i = 1; i <= TABLE_SIZE; i ++) {
  prime[i] = 1; /* assume i is prime until proven otherwise */
  for(j = 2; j < i; j ++) {
   if(i % j == 0) {
    prime[i] = 0; /* j divides i, so i is not prime */
    break;
   }
  }
 }

 printf("Table of primes, marked with '*':\n");
 for(i = 1; i <= TABLE_SIZE; i ++) {
  putchar(prime[i] ? '*' : '-');
  if(i % SCREEN_WIDTH == 0) printf("\n");
 }

 return 0;
}
```

Output

When you run the program ptest, the compiler issues the output similar as follows. The exact output varies depending on the number of threads and the way threads interleave.

```
probe: initializing POMP timer probe
probe: [0.000100 ptest.c:11 thd 0] enter parallel region with 4 threads,
if_expr_result=1
probe: [0.000453 ptest.c:11 thd 0] begin parallel region
probe: [0.000471 ptest.c:11 thd 1] begin parallel region
probe: [0.000465 ptest.c:11 thd 2] begin parallel region
probe: [0.000459 ptest.c:11 thd 3] begin parallel region
Greetings from thread 0.
Greetings from thread 1.
probe: [0.001448 ptest.c:11 thd 0] end parallel region
probe: [0.001726 ptest.c:11 thd 1] end parallel region
Greetings from thread 2.
Greetings from thread 3.
probe: [0.002187 ptest.c:11 thd 2] end parallel region
probe: [0.002429 ptest.c:11 thd 3] end parallel region
probe: [0.002812 ptest.c:11 thd 0] exit parallel region
probe: [0.003074 ptest.c:18 thd 0] enter parallel region with 4 threads,
if_expr_result=0
probe: [0.003278 ptest.c:18 thd 0] begin parallel region
probe: [0.003428 ptest.c:18 thd 0] enter loop [0,1600) chunk size 0
probe: [0.003678 ptest.c:18 thd 3] begin parallel region
probe: [0.003677 ptest.c:18 thd 0] begin chunk [0,400)
probe: [0.003684 ptest.c:18 thd 2] begin parallel region
probe: [0.003907 ptest.c:18 thd 3] enter loop [0,1600) chunk size 0
probe: [0.003690 ptest.c:18 thd 1] begin parallel region
probe: [0.004348 ptest.c:18 thd 3] begin chunk [1200,1600)
probe: [0.004563 ptest.c:18 thd 0] end chunk
probe: [0.004204 ptest.c:18 thd 2] enter loop [0,1600) chunk size 0
probe: [0.004494 ptest.c:18 thd 1] enter loop [0,1600) chunk size 0
probe: [0.004932 ptest.c:18 thd 2] begin chunk [800,1200)
probe: [0.005075 ptest.c:18 thd 1] begin chunk [400,800)
probe: [0.006649 ptest.c:18 thd 1] end chunk
probe: [0.006841 ptest.c:18 thd 1] exit loop
probe: [0.007046 ptest.c:18 thd 1] end parallel region
probe: [0.007212 ptest.c:18 thd 2] end chunk
probe: [0.007333 ptest.c:18 thd 3] end chunk
probe: [0.007403 ptest.c:18 thd 2] exit loop
probe: [0.007545 ptest.c:18 thd 3] exit loop
probe: [0.007691 ptest.c:18 thd 2] end parallel region
probe: [0.007837 ptest.c:18 thd 3] end parallel region
```

```
probe: [0.008167 ptest.c:18 thd 0] exit loop
probe: [0.008360 ptest.c:18 thd 0] end parallel region
probe: [0.008566 ptest.c:18 thd 0] exit parallel region
Table of primes, marked with '*':
***-*-*---*-*---*-*---*-----*-*-----*---*-*---*-----*-----*-*-----*---*-*-----*-
--*-----*-------*---*-*---*-*---*-------------*---*-----*-*---------*-*-----*---
--*---*-----*-----*-*---------*-*---*-*-----------*-----------*---*-*---*-----*-
*---------*-----*-----*-----*-*-----*---*-*---------*-------------*---*-*---*---
----------*-----*---------*-*---*-----*-------*-----*-----*---*-----*-------*---
*-------*---------*-*---------*-*-----*---*-----*-------*---*-*---*-----------*-
------*---*-------*---*-----*-----------*-*-----------------*-----*---------*---
--*-----*-*-----*---------*-----*-----*-*-----*-----*---*-*-----------*---------
*-*---*-----*-----*-*-----------*---*-----*-------*---------*-------*---------*-
------*-----*-----*---*-------*-----*---*-------*---*-------------*---------*---
--------*-*---------*-*---*-*---------*-------------*---*-*---*-------------*---
*-*---*-------------------*---*-------*---------*-------*---*-----*-----*-------
------*---*-----*-----*-------*-----*-----------*---*-----*-*---------*-*-----*-
--------*-*---------*-*-----*-----------------*---*-*---*-----*-----*-------*---
--*-----*---------------------*-*---------*-------*---------*-----*-----*-------
*-----------*---*-----*-----*-*-----*-----------*---------*-----------------*-*-
--*-----*-*-----*---*-*---*-----------*-*-----*---------------------------------
*-----*-----*-------*-----------------*---------*-------------*---*-*---*-----*-
------*---*-*-----*-----------*---------*-*---*-*---*-----*-----------*---------
--*-------*-----------*-----*---*-----*-------*---*-------*---*-------------*---
probe: exiting
```

You can use the following command to verify the output:

```
tail -n 21 output > ptest.verify
```

Related information

- Profiler for OpenMP
- POMP callback functions

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

Product Number: 5799-AG1

Printed in USA

SC14-7365-00