IBM XL C/C++ Alpha Edition for Multicore Acceleration
for Linux, V0.9

Using the single-source compiler

IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9

IBM

Using the single-source compiler

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 25.

First Edition

This edition applies to IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 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 document

This document contains overview and basic usage information for the IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 compiler.

Who should read this document

This document is intended for C and C++ developers who are looking for introductory overview and usage information for XL C/C++. It assumes that you have some familiarity with command-line compilers, a basic knowledge of the C and C++ programming language, and basic knowledge of operating system commands. Programmers new to XL C/C++ can use this document to find information on the capabilities and features unique to the XL C/C++ compiler.

How to use this document

Unless indicated otherwise, all of the text in this reference pertains to both C and C++ languages. Where there are differences between languages, these are indicated through qualifying text and icons, as described in "Conventions used in this document." Additionally, unless indicated otherwise, text in this document pertains to compilation targeting both the Power Processing Unit (PPU) and Synergistic Processor Units (SPUs).

While this document covers information on installing and configuring the compiler environment, and compiling and linking C and C++ applications using the XL C/C++ compiler, it does not include the following topics:

- Compiler options: see the XL C/C++ Compiler Reference for detailed information on the syntax and usage of compiler options.
- The C or C++ programming languages: see the XL C/C++ Language Reference for information on the syntax, semantics, and IBM® implementation of the C or C++ programming languages.
- Programming topics: see the XL C/C++ Programming Guide for detailed information on developing applications with XL C/C++, with a focus on program portability and optimization.

Conventions used in this document

Typographical conventions

The following table explains the typographical conventions used in this document.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options and directives.	If you specify **-O3**, the compiler assumes **-qhot=level=0**. To prevent all HOT optimizations with **-O3**, you must specify **-qnohot**.
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.

Table 1. Typographical conventions (continued)

Typeface	Indicates	Example
`monospace`	Programming keywords and library functions, compiler built-in functions, examples of program code, command strings, or user-defined names.	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.

Icons

All features described in this document apply to both C and C++ languages. Where a feature is exclusive to one language, or where functionality differs between languages, the following icons are used:

▶ C

The text describes a feature that is supported in the C language only; or describes behavior that is specific to the C language.

▶ C++

The text describes a feature that is supported in the C++ language only; or describes behavior that is specific to the C++ language.

Syntax diagrams

Throughout this document, 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——————————————————————————————————————▶◀
```

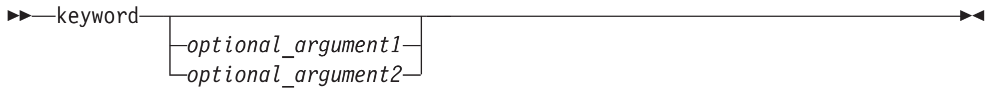
- Optional items are shown below the main path:

```
▶▶—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.

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



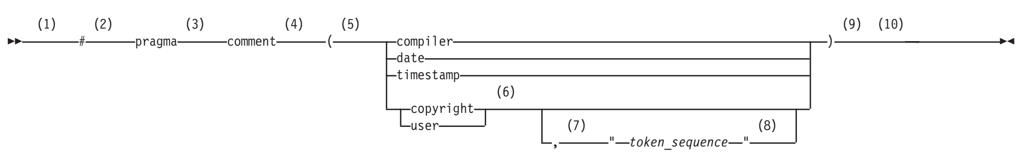
- An arrow returning to the left above the main line (a repeat arrow) indicates that you can make more than one choice from the stacked items or repeat an item. The separator character, if it is other than a blank, is also indicated:

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

- Keywords are shown in nonitalic letters and should be entered exactly as shown.
- Variables are shown in italicized lowercase letters. They represent user-supplied names or values.
- If 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.



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

The examples in this document, 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.

Related information

The following sections provide information on documentation related to XL C/C++:

- "IBM XL C/C++ publications"
- "Other IBM publications" on page ix
- "Other publications" on page ix

IBM XL C/C++ publications

This guide makes reference to other XL C/C++ publications in addition to those provided with the technical preview. The complete range of documentation for the various XL C/C++compiler products is available in the following formats and locations:

- README files

  README files contain late-breaking information, including changes and corrections to the product documentation. README files are located by default in the XL C/C++ directory and in the root directory of the installation CD.
- HTML-based information centers

  Information centers of searchable HTML files are available for many releases of XL C/C++. They can be viewed on the Web by going to the XL C/C++ product Library Web page at http://www.ibm.com/software/awdtools/xlcpp/library/.
- PDF documents

  You can access PDF versions of XL C/C++ documents on the Web at http://www.ibm.com/software/awdtools/xlcpp/library/.

  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 Web site at http://www.adobe.com.

More documentation related to XL C/C++ including IBM Redbooks, white papers, tutorials, and other articles, is available on the Web at:

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

Standards and specifications documents

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 document.

- *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.
- *AltiVec Technology Programming Interface Manual*, Motorola Inc. This specification for vector data types, to support vector processing technology, is available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.
- *OpenMP Application Program Interface Version 2.5*, available at http://www.openmp.org

Other IBM publications

- *IBM C/C++ Language Extensions for Cell Broadband Engine Architecture, Version 2.4*, available at |http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/30B3520C93F437AB87257060006FFE5E/$file/Language_Extensions_for_CBEA_2.4.pdf.
- Specifications, white papers, and other technical documents for the Cell Broadband Engine™ architecture are available at http://www.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine.
- The Cell Broadband Engine resource center, at http://www.ibm.com/developerworks/power/cell, is the central repository for technical information, including articles, tutorials, programming guides, and educational resources.

Other publications

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

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 document or any other XL C/C++ documentation, send your comments by e-mail to compinfo@ca.ibm.com.

Be sure to include the name of the document, the part number of the document, 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. Introducing IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9

IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 is a technical preview of an high-performance C/C++ cross-compiler that can be used for developing computationally intensive applications for use on systems based on the Cell Broadband Engine architecture.

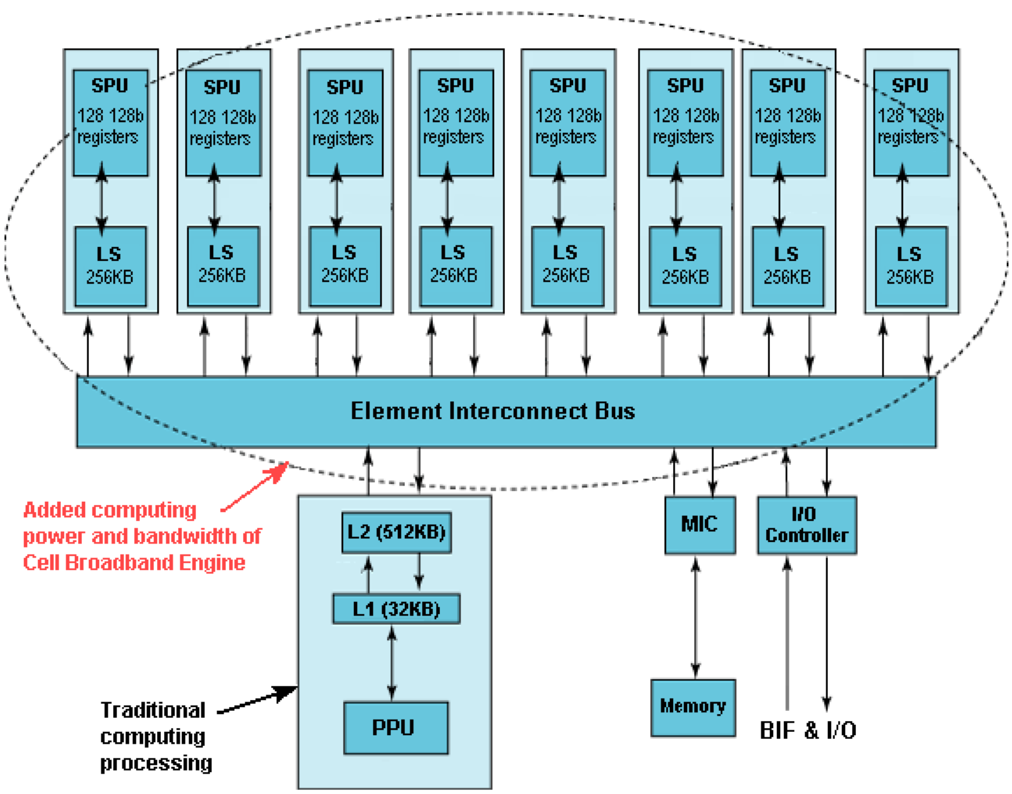
Part of a family of IBM compilers

IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 is part of a larger family of IBM C, C++, and Fortran compilers.

These compilers are derived from a common code base that shares compiler function and optimization technologies on a variety of platforms and programming languages, such as AIX®, i5/OS®, selected Linux® distributions, z/OS®, and z/VM® operating systems. The common code base, along with compliance with international programming language standards, helps support consistent compiler performance and ease of program portability across multiple operating systems and hardware platforms.

About the Cell Broadband Engine architecture

The Cell Broadband Engine architecture specification describes a new single-chip multiprocessor designed to support media-intensive applications.



At the heart of the new multiprocessor is the PowerPC® Processor Unit (PPU). The PPU is a 64-bit processor fully compliant with the Power Architecture™ standard, and capable of running both operating systems and applications. The multiprocessor also incorporates a set of eight Synergistic Processor Units (SPUs) into its design. The SPUs are optimized for running computationally intensive applications, operate independently of each other, and can access memory shared between all SPUs and the PPU.

In operation, the PPU runs the operating system and performs high-level application control, while the SPUs divide and perform an application's computational work between them.

For more information on the Cell Broadband Engine architecture, see "*Cell Broadband Engine Architecture from 20,000 feet*" at http://www.ibm.com/developerworks/power/library/pa-cbea.html.

New single-source cross-compiler technology

Earlier compilers for the Cell Broadband Engine architecture, such as the V0.8.1 and V0.8.2 compilers offered in past on the alphaWorks Web site, are considered a *dual-source* compiler. The compiler provides both PPU- and SPU-specific invocations to compile the different code segments. You write, compile, and link code segments destined to run on the PPU separately from code segments destined for the SPUs.

In contrast, a *single-source* compiler can compile and link both PPU and SPU code segments with a single compiler invocation.

The IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 offers you at an advance look at a single-source compiler with Open MP API V2.5 support that can compile applications for use on systems based on the Cell Broadband Engine architecture. With the *single-source* compiler provided in this technical preview, code destined for the PPU does not need to be written and compiled separately from code destined for the SPUs, and you can compile and link PPU and SPU code segments together with a single compiler invocation.

IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 is a cross-compiler. You develop and compile your applications on an IBM POWER or Intel x86 system running the Fedora 7 Linux operating system. When complete, you move your compiled application to a system based on the Cell Broadband Engine architecture, where that application will run.

For an overview of how the single-source compiler works to compile code optimized specifically for use on the Cell Broadband Engine architecture, see the "*Generation of Parallel Code*" section in "*Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture*", found online at http://www.research.ibm.com/journal/sj/451/eichenberger.html.

Chapter 2. Installing the XL C/C++ single-source compiler

This section describes how to install the IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 cross-compiler on its supported platforms.

Before you begin to install the compiler, be sure to:

- View the README file for any last minute updates you may need to be aware of.
- Ensure that all system prerequisites are met.
- Familiarize yourself with the installable compiler packages provided in the installation image.
- Familiarize yourself with the installation steps you will need to complete for your particular installation.
- Become either root user or a user with administration privileges.

System prerequisites

The following are the system requirements for installing IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 on your compilation host:

Supported operating system

- Fedora 7

Supported hardware

- IBM POWER technology-based systems
- Intel x86 systems

Required hard drive space

- Installed compiler packages - approximately 300 MB
- Paging space - 2 GB minimum
- Temporary files - 512 MB minimum
- Intel x86 systems

Required software prerequisites

- gcc v4.1.1
- gcc-c++ v4.1.1
- glibc v2.5
- libgcc v4.1.1
- libstdc++ v4.1.1
- IBM Software Development Kit (SDK) for Multicore Acceleration V3.0
- Perl V5.0 or higher

All software prerequisites can be obtained from your operating system's installation media and the IBM SDK for Multicore Acceleration V3.0.

Installing the compiler packages

IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 provides a set of RPM packages for each supported hardware platform. You must install the packages that correspond to your hardware platform.

By default, all packages are installed to `/opt/ibmcmp`. This technical preview does not support installation to a non-default path.

Package description	Supported hardware platforms and corresponding package names	
	IBM POWER	Intel x86
C/C++ runtime (redistributable)	cell-xlc-ssc-rte-0.9.0-0.ppc64.rpm	cell-xlc-ssc-rte-0.9.0-0.i386.rpm
C/C++ runtime links	cell-xlc-ssc-rte-lnk-0.9.0-0.ppc64.rpm	cell-xlc-ssc-rte-lnk-0.9.0-0.i386.rpm
C/C++ libraries	cell-xlc-ssc-lib-0.9.0-0.ppc64.rpm	cell-xlc-ssc-lib-0.9.0-0.i386.rpm
C/C++ OMP libraries	cell-xlc-ssc-omp-0.9.0-0.ppc64.rpm	cell-xlc-ssc-omp-0.9.0-0.i386.rpm
C/C++ compiler	cell-xlc-ssc-cmp-0.9.0-0.ppc64.rpm	cell-xlc-ssc-cmp-0.9.0-0.i386.rpm
C/C++ help and documentation	cell-xlc-ssc-help-0.9.0-0.ppc64.rpm	cell-xlc-ssc-help-0.9.0-0.i386.rpm
C/C++ manpages	cell-xlc-ssc-man-0.9.0-0.ppc64.rpm	cell-xlc-ssc-man-0.9.0-0.i386.rpm

If all prerequisites are satisfied, you can install the compiler packages to your system. To do so:

1. Log in as root or as a user with administration privileges.
2. Copy only the package files corresponding to your hardware platform to the `/rpms` directory.
3. Begin installation by issuing the following commands at the command prompt:

   ```
   # cd /rpms
   # rpm -ivh *.rpm
   ```

   Alternately, you can select and install each package manually by issuing the commands shown below in the order given.

Installation on IBM POWER	`# cd /rpms`
	`# rpm -ivh cell-xlc-ssc-rte-0.9.0-0.ppc64.rpm`
	`# rpm -ivh cell-xlc-ssc-rte-lnk-0.9.0-0.ppc64.rpm`
	`# rpm -ivh cell-xlc-ssc-lib-0.9.0-0.ppc64.rpm`
	`# rpm -ivh cell-xlc-ssc-omp-0.9.0-0.ppc64.rpm`
	`# rpm -ivh cell-xlc-ssc-cmp-0.9.0-0.ppc64.rpm`
	`# rpm -ivh cell-xlc-ssc-help-0.9.0-0.ppc64.rpm`
	`# rpm -ivh cell-xlc-ssc-man-0.9.0-0.ppc64.rpm`

Installation on Intel x86	`# cd /rpms`
	`# rpm -ivh cell-xlc-ssc-rte-0.9.0-0.i386.rpm`
	`# rpm -ivh cell-xlc-ssc-rte-lnk-0.9.0-0.i386.rpm`
	`# rpm -ivh cell-xlc-ssc-lib-0.9.0-0.i386.rpm`
	`# rpm -ivh cell-xlc-ssc-omp-0.9.0-0.i386.rpm`
	`# rpm -ivh cell-xlc-ssc-cmp-0.9.0-0.i386.rpm`
	`# rpm -ivh cell-xlc-ssc-help-0.9.0-0.i386.rpm`
	`# rpm -ivh cell-xlc-ssc-man-0.9.0-0.i386.rpm`

Coexisting with other versions of XL C/C++

In most cases, IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 can coexist on the same system with other versions of the XL C/C+ compiler without problem.

Coexisting with previous compilers for Cell Broadband Engine architecture
: There are no coexistence issues with the earlier V0.8.1 and V0.8.2 compilers offered on the alphaWorks Web site.

Having multiple instances of IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9
: Though IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 supports multiple hardware platforms, you can only install one instance of the compiler on your system, and that instance must be the version of the compiler most appropriate for your system hardware.

Coexisting with XL C/C++ Advanced Edition for Linux, any version

- There are no direct coexistence issues between this compiler and IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9.
- However, the runtime libraries for the Linux and Multicore versions of the XL C/C++ compilers both share a common name. If you have both a Linux and a Multicore version of the compiler installed on your system, and the LD_LIBRARY_PATH environment variable is set, it is possible for an application to use the wrong runtime libary.

Uninstalling the compiler

To remove IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 from your system, you must uninstall the compiler packages in reverse order of installation.

Log in as root or as a user with administration privileges, and issue the uninstallation commands below that apply to your hardware platform, in the order given:

Uninstallation on IBM POWER	`# rpm -ivh cell-xlc-ssc-man-0.9.0-0.ppc64.rpm`
	`# rpm -ivh cell-xlc-ssc-help-0.9.0-0.ppc64.rpm`
	`# rpm -ivh cell-xlc-ssc-cmp-0.9.0-0.ppc64.rpm`
	`# rpm -ivh cell-xlc-ssc-omp-0.9.0-0.ppc64.rpm`
	`# rpm -ivh cell-xlc-ssc-lib-0.9.0-0.ppc64.rpm`
	`# rpm -ivh cell-xlc-ssc-rte-lnk-0.9.0-0.ppc64.rpm`
	`# rpm -ivh cell-xlc-ssc-rte-0.9.0-0.ppc64.rpm`

Uninstallation on Intel x86	`# rpm -ivh cell-xlc-ssc-man-0.9.0-0.i386.rpm`
	`# rpm -ivh cell-xlc-ssc-help-0.9.0-0.i386.rpm`
	`# rpm -ivh cell-xlc-ssc-cmp-0.9.0-0.i386.rpm`
	`# rpm -ivh cell-xlc-ssc-omp-0.9.0-0.i386.rpm`
	`# rpm -ivh cell-xlc-ssc-lib-0.9.0-0.i386.rpm`
	`# rpm -ivh cell-xlc-ssc-rte-lnk-0.9.0-0.i386.rpm`
	`# rpm -ivh cell-xlc-ssc-rte-0.9.0-0.i386.rpm`

Chapter 3. Developing your applications

The basic steps involved in developing applications with IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 involve:

1. Writing your C/C++ program source, including using the OpenMP pragmas to mark code that you want to have run on the SPUs.
2. Compiling your C/C++ program source using the compiler invocations described later in this section.
3. Moving the compiled application to the target Cell machine for execution.

Writing your program source

The single source compiler provided in IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 helps simplify the task of writing application code destined for Cell Broadband Engine systems.

Application code intended for execution on the SPUs can reside in the same physical program source file as code intended for the PPU, and does not need to be partitioned off for separate compilation. Instead, you mark specific code segments with OpenMP pragma directives that instruct the compiler how that code segment should be parallelized for the SPUs. The OpenMP specification is described later in this section.

Otherwise, writing your program source for Cell Broadband Engine applications is little different from writing program source for any other C/C++ application. You can focus more on what you want your application to achieve, and less on the intricacies of manipulating code segments to make the best use of the PPU and SPU portions of the Cell Broadband Engine processor. The single source compiler will perform a high level of program optimization and PPU/SPU targeting for you.

For more information, see the "*Programming for the Cell BE architecture*" section in "*Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture*", found online at http://www.research.ibm.com/journal/sj/451/eichenberger.html.

Using OpenMP pragma directives in your program source

OpenMP directives are a set of API-based commands supported by IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 and many other IBM and non-IBM C, C++, and Fortran compilers.

You can use OpenMP directives to instruct the compiler how to parallelize a particular loop. The existence of the directives in the source removes the need for the compiler to perform any parallel analysis on the parallel code. OpenMP directives requires the presence of Pthread libraries to provide the necessary infrastructure for parallelization.

OpenMP directives address three important issues of parallelizing an application:

1. Clauses and directives are available for scoping variables. Frequently, variables should not be shared; that is, each processor should have its own copy of the variable.

2. Work sharing directives specify how the work contained in a parallel region of code should be distributed across the SMP processors.
3. Directives are available to control synchronization between the processors.

XL C/C++ supports the OpenMP API Version 2.5 specification.

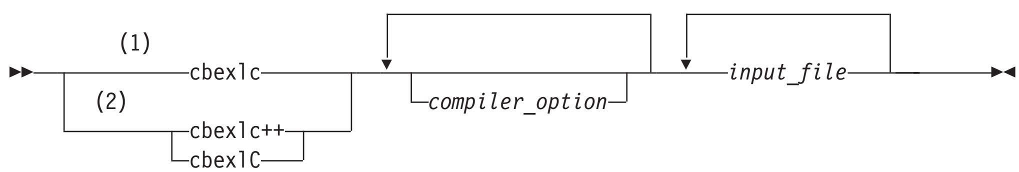
For more information, see:
- "OpenMP pragma directives provided in this technical preview," on page 13
- "Using OpenMP pragma directives in your program source" on page 7

.

Invoking the compiler

The compiler invocation commands provided with IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 perform all of the steps required to compile C or C++ source files and link the object files and libraries into an executable program.

Invoke the compiler using the basic syntax shown below:



Notes:

1 Basic invocation to compile C source code.

2 Basic invocations to compile C++ source code

Both **cbexlC** and **cbexlc++** will compile either C or C++ program source, but compiling C++ files with **cbexlc** may result in link or run time errors because libraries required for C++ code are not specified when the linker is called by the C compiler.

Specifying compiler options

Compiler options perform a variety of functions, such as setting compiler characteristics, describing the object code to be produced, controlling the diagnostic messages emitted, and performing some preprocessor functions.

You can specify compiler options:
- On the command-line with command-line compiler options
- In your source code using directive statements
- In a makefile
- In the stanzas found in a compiler configuration file
- Or by using any combination of these techniques

It is possible for option conflicts and incompatibilities to occur when multiple compiler options are specified. To resolve these conflicts in a consistent fashion, the compiler usually applies the following general priority sequence to most options:

1. Directive statements in your source file *override* command-line settings
2. Command-line compiler option settings *override* configuration file settings
3. Configuration file settings *override* default settings

Generally, if the same compiler option is specified more than once on a command-line when invoking the compiler, the last option specified prevails.

Note: Some compiler options do not follow the priority sequence described above.

For example, the **-I** compiler option is a special case. The compiler searches any directories specified with **-I** in the vac.cfg file before it searches the directories specified with **-I** on the command-line. The option is cumulative rather than preemptive.

See the XL C/C++ Compiler Reference for more information about compiler options and their usage.

You can also pass compiler options to the linker, assembler, and preprocessor. See ″Compiler options reference″ in the XL C/C++ Compiler Reference for more information about compiler options and how to specify them.

Compiler options and pragmas specific to this technical preview

The IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 technical preview includes support for compiler options and pragma directives not documented in the XL C/C++ Compiler Reference. These include:

-qarch=cell, -qtune-cell

The **cell** suboption to the **-qarch** and **-qtune** options instructs the compiler to generate code targeted for processors based on the Cell Broadband Engine architecture.

Specifying **-qarch=cell** sets the following macros:

```
_ARCH_COM
_ARCH_PPC
_ARCH_PPCGR
_ARCH_PPC64
_ARCH_PPC64GR
_ARCH_PPC64GRSQ
_ARCH_CBEPPE
_ARCH_CELLPPU
_ARCH_CELL
```

-qarch=celledp, -qtune=celledp

The **celledp** suboption to the **-qarch** and **-qtune** options instructs the compiler to generate code targeted for processors based on the Cell Broadband Engine architecture that also incorporate SPUs with enhanced double precision capability.

Specifying **-qarch=celledp** sets the following macros:

```
_ARCH_COM
_ARCH_PPC
_ARCH_PPCGR
_ARCH_PPC64
_ARCH_PPC64GR
_ARCH_PPC64GRSQ
```

```
_ARCH_CBEPPE
_ARCH_CELLPPU
_ARCH_CELL
_ARCH_CELLEDP
```

-qipa=overlay
Ordinarily, the compiler default setting is **-qipa=nooverlay**. For IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9, the default setting changes to **-qipa=overlay**. See **-qipa** in the XL C/C++ Compiler Reference for more information.

OpenMP pragma directives

The OpenMP directives are a set of API-based commands supported by IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9 and many other IBM and non-IBM C, C++, and Fortran compilers. These pragmas instruct the compiler how specific sections of your application code should be parallized for use by the SPUs.

For more information, see:

- “OpenMP pragma directives provided in this technical preview,” on page 13
- “Using OpenMP pragma directives in your program source” on page 7

.

XL C/C++ input and output files

The file types listed below are recognized by XL C/C++. For detailed information about these and additional file types used by the compiler, see *"Types of input files""Types of input files"* and *"Types of output files""Types of output files"* in the XL C/C++ Compiler Reference.

Table 2. Input file types

Filename extension	Description
.c	C source files
.C, .cc, .cp, .cpp, .cxx, .c++	C++ source files
.i	Preprocessed source files
.o	Object files
.s	Assembler files
.S	Unpreprocessed assembler files

Table 3. Output file types

Filename extension	Description
a.out	Default name for executable file created by the compiler
.d	Make dependency file
.i	Preprocessed source files
.lst	Listing files
.o	Object files

Linking your compiled applications

By default, you do not need to do anything special to link an XL C/C++ program. The compiler invocation commands automatically call the linker to produce an executable output file. For example, running the following command:

```
cbexlc++ file1.C file2.o file3.C
```

compiles and produces the object files `file1.o` and `file3.o`, then all object files (including `file2.o`) are submitted to the linker to produce one executable.

Compiling and linking in separate steps

To produce object files that can be linked later, use the **-c** option.

```
xlc++ -c file1.C              # Produce one object file (file1.o)
xlc++ -c file2.C file3.C      # Or multiple object files (file1.o, file3.o)
xlc++ file1.o file2.o file3.o  # Link object files with default libraries
```

For more information about compiling and linking your programs, see the documentation provided with the IBM SDK for Multicore Acceleration 3.0.

Appendix. OpenMP pragma directives provided in this technical preview

This section describes the OpenMP directives supported by IBM XL C/C++ Alpha Edition for Multicore Acceleration for Linux, V0.9.

The OpenMP pragmas fall into different categories of effect. These are:

Defines code segments in which work is done by threads in parallel
- "#pragma omp parallel" on page 19
- "#pragma omp parallel for" on page 21
- "#pragma omp parallel sections" on page 21

Defines how work will be distributed across threads
- "#pragma omp for" on page 16
- "#pragma omp ordered" on page 19
- "#pragma omp section, #pragma omp sections" on page 22
- "#pragma omp single" on page 23

Controls synchronization among threads
- "#pragma omp atomic"
- "#pragma omp barrier" on page 14
- "#pragma omp critical" on page 15
- "#pragma omp flush" on page 15
- "#pragma omp master" on page 19

Defines scope of data visibility across threads
- "#pragma omp threadprivate" on page 24

You can use these pragmas to mark specific sections of application code for use by the SPUs. For more information about these pragmas and the OpenMP specification, see www.openmp.org.

#pragma omp atomic

Description

The **omp atomic** directive identifies a specific memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.

Syntax

```
>>--#--pragma--omp atomic--+-----------+------------------------------><
                           '-statement-'
```

where *statement* is an expression statement of scalar type that takes one of the forms that follow:

statement	**Conditions**
x bin_op = *expr*	where: *bin_op* is one of: `+ * - / & ^ | << >>` *expr* is an expression of scalar type that does not reference *x*.
x++	
++x	
x--	
--x	

Notes

Load and store operations are atomic only for object *x*. Evaluation of *expr* is not atomic.

All atomic references to a given object in your program must have a compatible type.

Objects that can be updated in parallel and may be subject to race conditions should be protected with the **omp atomic** directive.

Examples

```
extern float x[], *p = x, y;

/* Protect against race conditions among multiple updates.  */
#pragma omp atomic
x[index[i]] += y;

/* Protect against races with updates through x.            */
#pragma omp atomic
p[i] -= 1.0f;
```

#pragma omp barrier

Description

The **omp barrier** directive identifies a synchronization point at which threads in a parallel region will wait until all other threads in that section reach the same point. Statement execution past the **omp barrier** point then continues in parallel.

Syntax

```
►►──#──pragma──omp barrier──────────────────────────────────────────►◄
```

Notes

The **omp barrier** directive must appear within a block or compound statement. For example:

```
if (x!=0) {
   #pragma omp barrier    /* valid usage    */
}
```

```
if (x!=0)
   #pragma omp barrier    /* invalid usage  */
```

#pragma omp critical

Description

The **omp critical** directive identifies a section of code that must be executed by a single thread at a time.

Syntax

```
                                 ┌─,────┐
                                 ▼      │
►►──#──pragma──omp critical───(name)────┴──────────────────────────►◄
```

where *name* can optionally be used to identify the critical region. Identifiers naming a critical region have external linkage and occupy a namespace distinct from that used by ordinary identifiers.

Notes

A thread waits at the start of a critical region identified by a given name until no other thread in the program is executing a critical region with that same name. Critical sections not specifically named by **omp critical** directive invocation are mapped to the same unspecified name.

#pragma omp flush

Description

The **omp flush** directive identifies a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.

Syntax

```
                          ┌──────────┐
                          │ ┌─,──┐   │
                          ▼ │    │   │
►►──#──pragma──omp flush──┴──────────┴──────────────────────────────►◄
                            └─list─┘
```

where *list* is a comma-separated list of variables that will be synchronized.

Notes

If *list* includes a pointer, the pointer is flushed, not the object being referred to by the pointer. If *list* is not specified, all shared objects are synchronized except those inaccessible with automatic storage duration.

An implied **flush** directive appears in conjunction with the following directives:

- **omp barrier**
- Entry to and exit from **omp critical**.
- Exit from **omp parallel**.

- Exit from **omp for**.
- Exit from **omp sections**.
- Exit from **omp single**.

The **omp flush** directive must appear within a block or compound statement. For example:

```
if (x!=0) {
   #pragma omp flush    /* valid usage    */
}
if (x!=0)
   #pragma omp flush    /* invalid usage  */
```

#pragma omp for

Description

The **omp for** directive instructs the compiler to distribute loop iterations within the team of threads that encounters this work-sharing construct.

Syntax

```
                          ┌─,──────────────┐
                          ▼                │
►►─#─pragma─omp for────────┬──────────────┬┴────────────────────►◄
                          └─clause─for-loop─┘
```

where *clause* is any of the following:

private (*list*)
: Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list*)
: Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

lastprivate (*list*)
: Declares the scope of the data variables in *list* to be private to each thread. The final value of each variable in *list*, if assigned, will be the value assigned to that variable in the last iteration. Variables not assigned a value will have an indeterminate value. Data variables in *list* are separated by commas.

reduction (*operator*:*list*)
: Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

  A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable.

  Variables specified in the **reduction** clause:
  - must be of a type appropriate to the operator.
  - must be shared in the enclosing context.
  - must not be const-qualified.
  - must not have pointer type.

ordered
: Specify this clause if an ordered construct is present within the dynamic extent of the **omp for** directive.

schedule (*type*)

Specifies how iterations of the **for** loop are divided among available threads. Acceptable values for *type* are:

dynamic
: Iterations of a loop are divided into chunks of size **ceiling**(*number_of_iterations*/*number_of_threads*).

  Chunks are dynamically assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.

dynamic,*n*
: As above, except chunks are set to size *n*. *n* must be an integral assignment expression of value 1 or greater.

guided
: Chunks are made progressively smaller until the default minimum chunk size is reached. The first chunk is of size **ceiling**(*number_of_iterations*/*number_of_threads*). Remaining chunks are of size **ceiling**(*number_of_iterations_left*/*number_of_threads*).

  The minimum chunk size is 1.

  Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.

guided,*n*
: As above, except the minimum chunk size is set to *n*. *n* must be an integral assignment expression of value 1 or greater.

runtime
: Scheduling policy is determined at run time. Use the OMP_SCHEDULE environment variable to set the scheduling type and chunk size.

static
: Iterations of a loop are divided into chunks of size **ceiling**(*number_of_iterations*/*number_of_threads*). Each thread is assigned a separate chunk.

  This scheduling policy is also known as *block scheduling*.

static,*n*
: Iterations of a loop are divided into chunks of size *n*. Each chunk is assigned to a thread in *round-robin* fashion.

  n must be an integral assignment expression of value 1 or greater.

  This scheduling policy is also known as *block cyclic scheduling*.
  Note: if *n*=1, iterations of a loop are divided into chunks of size 1 and each chunk is assigned to a thread in *round-robin* fashion. This scheduling policy is also known as *block cyclic scheduling*

nowait

Use this clause to avoid the implied **barrier** at the end of the **for** directive. This is useful if you have multiple independent work-sharing sections or iterative loops within a given parallel region. Only one **nowait** clause can appear on a given **for** directive.

and where *for_loop* is a `for` loop construct with the following canonical shape:

```
for (init_expr; exit_cond; incr_expr)
 statement
```

where:

init_expr	takes form:	`iv = b` `integer-type iv = b`

exit_cond	takes form:	`iv <= ub`
		`iv <  ub`
		`iv >= ub`
		`iv >  ub`

incr_expr	takes form:	`++iv`
		`iv++`
		`--iv`
		`iv--`
		`iv += incr`
		`iv -= incr`
		`iv = iv + incr`
		`iv = incr + iv`
		`iv = iv - incr`

and where:

iv
: Iteration variable. The iteration variable must be a `signed integer` not modified anywhere within the `for` loop. It is implicitly made private for the duration of the `for` operation. If not specified as **lastprivate**, the iteration variable will have an indeterminate value after the operation completes.

b, *ub*, *incr*
: Loop invariant signed integer expressions. No synchronization is performed when evaluating these expressions and evaluated side effects may result in indeterminate values.

Notes

This pragma must appear immediately before the loop or loop block directive to be affected.

Program sections using the **omp for** pragma must be able to produce a correct result regardless of which thread executes a particular iteration. Similarly, program correctness must not rely on using a particular scheduling algorithm.

The `for` loop iteration variable is implicitly made private in scope for the duration of loop execution. This variable must not be modified within the body of the `for` loop. The value of the increment variable is indeterminate unless the variable is specified as having a data scope of **lastprivate**.

An implicit barrier exists at the end of the `for` loop unless the **nowait** clause is specified.

Restrictions are:
- The `for` loop must be a structured block, and must not be terminated by a `break` statement.
- Values of the loop control expressions must be the same for all iterations of the loop.
- An **omp for** directive can accept only one **schedule** clauses.
- The value of *n* (chunk size) must be the same for all threads of a parallel region.

#pragma omp master

Description

The **omp master** directive identifies a section of code that must be run only by the master thread.

Syntax

```
►►—#—pragma—omp master————————————————————————————————————————►◄
```

Notes

Threads other than the master thread will not execute the statement block associated with this construct.

No implied barrier exists on either entry to or exit from the master section.

#pragma omp ordered

Description

The **omp ordered** directive identifies a structured block of code that must be executed in sequential order.

Syntax

```
►►—#—pragma—omp ordered———————————————————————————————————————►◄
```

Notes

The **omp ordered** directive must be used as follows:

- It must appear within the extent of a **omp for** or **omp parallel for** construct containing an **ordered** clause.
- It applies to the statement block immediately following it. Statements in that block are executed in the same order in which iterations are executed in a sequential loop.
- An iteration of a loop must not execute the same **omp ordered** directive more than once.
- An iteration of a loop must not execute more than one distinct **omp ordered** directive.

#pragma omp parallel

Description

The **omp parallel** directive explicitly instructs the compiler to parallelize the chosen block of code.

Syntax

```
                                  ┌─,─────┐
                                  ↓       │
►►─#─pragma─omp parallel─────────clause───┴──────────────────────────────────►◄
```

where *clause* is any of the following:

if (*exp*)
: When the `if` argument is specified, the program code executes in parallel only if the scalar expression represented by *exp* evaluates to a non-zero value at run time. Only one `if` clause can be specified.

private (*list*)
: Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list*)
: Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized with the value of the original variable as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

num_threads (*int_exp*)
: The value of *int_exp* is an integer expression that specifies the number of threads to use for the parallel region. If dynamic adjustment of the number of threads is also enabled, then *int_exp* specifies the maximum number of threads to be used.

shared (*list*)
: Declares the scope of the comma-separated data variables in *list* to be shared across all threads.

default (shared | none)
: Defines the default data scope of variables in each thread. Only one **default** clause can be specified on an **omp parallel** directive.

  Specifying **default(shared)** is equivalent to stating each variable in a **shared(***list***)** clause.

  Specifying **default(none)** requires that each data variable visible to the parallelized statement block must be explcitly listed in a data scope clause, with the exception of those variables that are:

  - const-qualified,
  - specified in an enclosed data scope attribute clause, or,
  - used as a loop control variable referenced only by a corresponding **omp for** or **omp parallel for** directive.

copyin (*list*)
: For each data variable specified in *list*, the value of the data variable in the master thread is copied to the thread-private copies at the beginning of the parallel region. Data variables in *list* are separated by commas.

  Each data variable specified in the **copyin** clause must be a **threadprivate** variable.

reduction (*operator*: *list*)
: Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

  A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable.

  Variables specified in the **reduction** clause:

  - must be of a type appropriate to the operator.
  - must be shared in the enclosing context.
  - must not be const-qualified.
  - must not have pointer type.

Notes

When a parallel region is encountered, a logical team of threads is formed. Each thread in the team executes all statements within a parallel region except for work-sharing constructs. Work within work-sharing constructs is distributed among the threads in a team.

Loop iterations must be independent before the loop can be parallelized. An implied barrier exists at the end of a parallelized statement block.

Nested parallel regions are always serialized.

#pragma omp parallel for

Description

The **omp parallel for** directive effectively combines the **omp parallel** and **omp for** directives. This directive lets you define a parallel region containing a single **for** directive in one step.

Syntax

```
                                ┌─,─┐
>>──#──pragma──omp for──────────┴───┴─────────────────────────────────><
                                └─clause──for-loop─┘
```

Notes

With the exception of the **nowait** clause, clauses and restrictions described in the **omp parallel** and **omp for** directives also apply to the **omp parallel for** directive.

#pragma omp parallel sections

Description

The **omp parallel sections** directive effectively combines the **omp parallel** and **omp sections** directives. This directive lets you define a parallel region containing a single **sections** directive in one step.

Syntax

```
                                         ┌─,─┐
>>──#──pragma──omp parallel sections─────┴───┴────────────────────────><
                                         └─clause─┘
```

Notes

All clauses and restrictions described in the **omp parallel** and **omp sections** directives apply to the **omp parallel sections** directive.

#pragma omp section, #pragma omp sections

Description

The **omp sections** directive distributes work among threads bound to a defined parallel region.

Syntax

```
         ┌─,──────┐
         ▼        │
►►─#─pragma─omp sections───clause─┴──────────────────────────────►◄
```

where *clause* is any of the following:

private (*list*)	Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.
firstprivate (*list*)	Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.
lastprivate (*list*)	Declares the scope of the data variables in *list* to be private to each thread. The final value of each variable in *list*, if assigned, will be the value assigned to that variable in the last **section**. Variables not assigned a value will have an indeterminate value. Data variables in *list* are separated by commas.
reduction (*operator*: *list*)	Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas. A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable. Variables specified in the **reduction** clause: • must be of a type appropriate to the operator. • must be shared in the enclosing context. • must not be const-qualified. • must not have pointer type.
nowait	Use this clause to avoid the implied **barrier** at the end of the **sections** directive. This is useful if you have multiple independent work-sharing sections within a given parallel region. Only one **nowait** clause can appear on a given **sections** directive.

Notes

The **omp section** directive is optional for the first program code segment inside the **omp sections** directive. Following segments must be preceded by an **omp section** directive. All **omp section** directives must appear within the lexical construct of the program source code segment associated with the **omp sections** directive.

When program execution reaches a **omp sections** directive, program segments defined by the following **omp section** directive are distributed for parallel execution among available threads. A barrier is implicitly defined at the end of the larger program region associated with the **omp sections** directive unless the **nowait** clause is specified.

#pragma omp single

Description

The **omp single** directive identifies a section of code that must be run by a single available thread.

Syntax

```
                            ┌─────────┐
                            │  └─,─┘  │
                            ▼         │
►►──#──pragma──omp single───┴─clause──┴──────────────────────────────►◄
```

where *clause* is any of the following:

private (*list*)
: Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

  A variable in the **private** clause must not also appear in a **copyprivate** clause for the same **omp single** directive.

copyprivate (*list*)
: Broadcasts the values of variables specified in *list* from one member of the team to other members. This occurs after the execution of the structured block associated with the **omp single** directive, and before any of the threads leave the barrier at the end of the construct. For all other threads in the team, each variable in the *list* becomes defined with the value of the corresponding variable in the thread that executed the structured block. Data variables in *list* are separated by commas. Usage restrictions for this clause are:

  - A variable in the **copyprivate** clause must not also appear in a **private** or **firstprivate** clause for the same **omp single** directive.
  - If an **omp single** directive with a **copyprivate** clause is encountered in the dynamic extent of a parallel region, all variables specified in the **copyprivate** clause must be private in the enclosing context.
  - Variables specified in **copyprivate** clause within dynamic extent of a parallel region must be private in the enclosing context.
  - A variable that is specified in the **copyprivate** clause must have an accessible and unambiguous copy assignment operator.
  - The **copyprivate** clause must not be used together with the **nowait** clause.

firstprivate (*list*)
: Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

  A variable in the **firstprivate** clause must not also appear in a **copyprivate** clause for the same **omp single** directive.

nowait
: Use this clause to avoid the implied **barrier** at the end of the **single** directive. Only one **nowait** clause can appear on a given **single** directive. The **nowait** clause must not be used together with the **copyprivate** clause.

Notes

An implied barrier exists at the end of a parallelized statement block unless the **nowait** clause is specified.

#pragma omp threadprivate

Description

The **omp threadprivate** directive makes the named file-scope, namespace-scope, or static block-scope variables private to a thread.

Syntax

```
                                     ┌─,──────────┐
                                     ▼            │
►►──#──pragma──omp threadprivate────(identifier)──┴──────────────────────►◄
```

where *identifier* is a file-scope, name space-scope or static block-scope variable.

Notes

Each copy of an **omp threadprivate** data variable is initialized once prior to first use of that copy. If an object is changed before being used to initialize a **threadprivate** data variable, behavior is unspecified.

A thread must not reference another thread's copy of an **omp threadprivate** data variable. References will always be to the master thread's copy of the data variable when executing serial and master regions of the program.

Use of the **omp threadprivate** directive is governed by the following points:

- An **omp threadprivate** directive must appear at file scope outside of any definition or declaration.
- The **omp threadprivate** directive is applicable to static-block scope variables and may appear in lexical blocks to reference those block-scope variables. The directive must appear in the scope of the variable and not in a nested scope, and must precede all references to variables in its list.
- A data variable must be declared with file scope prior to inclusion in an **omp threadprivate** directive *list*.
- An **omp threadprivate** directive and its *list* must lexically precede any reference to a data variable found in that *list*.
- A data variable specified in an **omp threadprivate** directive in one translation unit must also be specified as such in all other translation units in which it is declared.
- Data variables specified in an **omp threadprivate** *list* must not appear in any clause other than the **copyin**, **copyprivate**, **if**, **num_threads**, and **schedule** clauses.
- The address of a data variable in an **omp threadprivate** *list* is not an address constant.
- A data variable specified in an **omp threadprivate** *list* must not have an incomplete or reference type.

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 Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites 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

Company, product, or service names identified in the text may be trademarks or service marks of IBM or other companies. Information on the trademarks of International Business Machines Corporation in the United States, other countries, or both is located at http://www.ibm.com/legal/copytrade.shtml.

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

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

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

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

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

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

Industry standards

The following standards are supported:

- The C language is consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1990).
- The C language is also consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1999 (E)).
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The C++ language is also consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:2003 (E)).

Printed in USA