

IBM XL C/C++ Advanced Edition for Mac OS X



Technology Preview

Version 6.0

IBM XL C/C++ Advanced Edition for Mac OS X



Technology Preview

Version 6.0

First Edition (December 2003)

This document, the applications, and the functions discussed, are offered as Technology Previews.

They are provided on an "AS-IS" BASIS, WITHOUT WARRANTY OR CONDITION OF ANY KIND, INCLUDING THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

© Copyright International Business Machines Corporation 2003. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Introduction	1
-------------------------------	----------

Program Parallelization	3
--	----------

OpenMP Pragma Directives	3
------------------------------------	---

Using the smp compiler option	4
---	---

Shared and Private Variables in a Parallel Environment	5
--	---

Pragma Directives	7
-----------------------------	---

#pragma omp atomic	7
------------------------------	---

#pragma omp parallel	8
--------------------------------	---

#pragma omp for	9
---------------------------	---

#pragma omp parallel for.	13
-----------------------------------	----

#pragma omp ordered.	13
------------------------------	----

#pragma omp section, #pragma omp sections	13
---	----

#pragma omp parallel sections	14
---	----

#pragma omp single	15
------------------------------	----

#pragma omp master	15
------------------------------	----

#pragma omp critical	15
--------------------------------	----

#pragma omp barrier	16
-------------------------------	----

#pragma omp flush.	16
----------------------------	----

#pragma omp threadprivate	17
-------------------------------------	----

Parallel Processing Support	17
---------------------------------------	----

Run-time Options for Parallel Processing	18
--	----

OpenMP Run-Time Options for Parallel	
--------------------------------------	--

Processing	20
----------------------	----

Built-in Functions Used for Parallel Processing	22
---	----

Objective-C and XL C/C++ for Mac OS

X	25
--------------------	-----------

Compiler Options	25
----------------------------	----

-qsourcetype	25
------------------------	----

-framework	26
----------------------	----

-qframeworkdir	26
--------------------------	----

-lobjc	26
------------------	----

Limitations	26
-----------------------	----

Notices	27
--------------------------	-----------

Trademarks and Service Marks	27
--	----

Industry Standards	27
------------------------------	----

Introduction

This document is a technology preview of features for the IBM® XL C/C++ Advanced Edition for the Mac OS X Version 6.0. The features covered are:

- Parallel processing with the XL C/C++ compiler, implementing the OpenMP API Version 1.0
- Objective-C and XL C/C++ for Mac OS X

Note!

Features discussed in these sections, as part of the Technology Preview, are provided "as is" and not part of the XL C/C++ compiler product. The purpose of this preview is to showcase early results of development work. There is no support for these features.

Program Parallelization

Parallel regions of program code are executed by multiple threads, possibly running on multiple processors. The number of threads created is determined by the run-time options and calls to library functions. Work is distributed among available threads according to the specified scheduling algorithm.

The XL C/C++ compiler lets you explicitly identify sections of the program code to be parallelized, using OpenMP pragma directives. The compiler supports the OpenMP specification version 1.0.

For more information about the OpenMP Specification, see:

- The OpenMP Web site at <http://www.openmp.org>
- The OpenMP specifications at <http://www.openmp.org/specs>

OpenMP Pragma Directives

OpenMP pragma 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.

Parallel processing operations are controlled by pragma directives in your program source. The pragmas have effect only when parallelization is enabled with the **-qsm** compiler option. For more information, see “Using the smp compiler option” on page 4

Pragmas fall into four general categories:

- Those that allow definition of parallel regions in which work is done by threads in parallel. Most of the OpenMP directives either statically or dynamically bind to an enclosing parallel region.
- Those that allow definition of how work will be distributed or shared across the threads in a parallel region.
- Those that allow control of synchronization among threads.
- Those that allow definition of the scope of data visibility across threads.

For more information about the specific pragmas, see “Pragma Directives” on page 7.

The syntax of an omp pragma is:

```
#pragma omp directive-name [clause[ [,] clause]...] new-line
```

Pragma directives generally appear immediately before the section of code to which they apply. The **omp parallel** directive is used to define the region of program code to be parallelized. Other OpenMP directives define visibility of data variables in the defined parallel region and how work within that region is shared and synchronized.

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

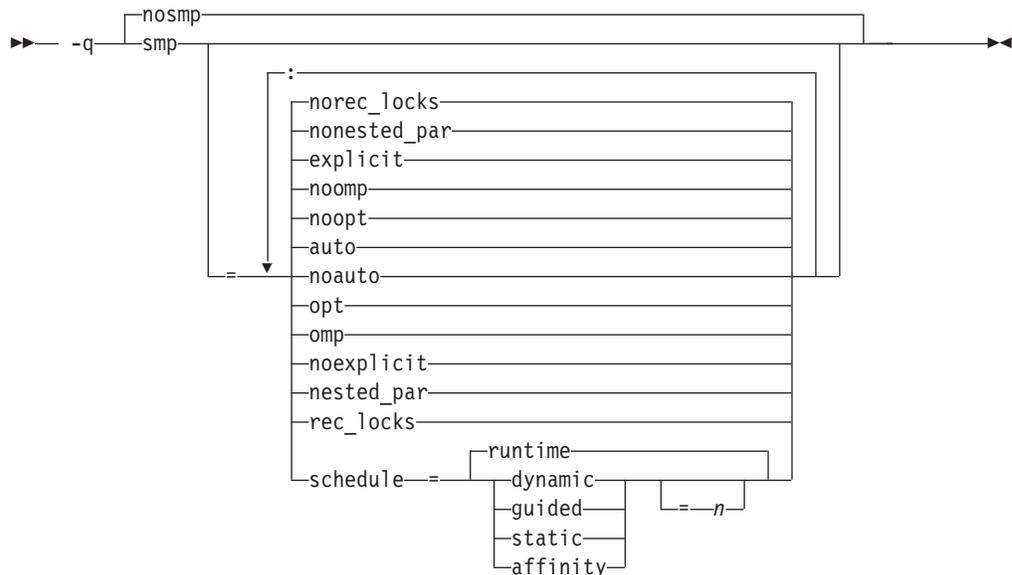
```

Using the smp compiler option

Purpose

Enables parallelization of program code.

Syntax



where:

- auto** Enables automatic parallelization and optimization of program code.
- noauto** Disables automatic parallelization of program code. Program code explicitly parallelized with SMP or OMP pragma statements is optimized.
- opt** Enables automatic parallelization and optimization of program code.

noopt	Enables automatic parallelization, but disables optimization of parallelized program code. Use this setting when debugging parallelized program code..
omp	Enables strict compliance to the OMP standard. Automatic parallelization is disabled. Parallelized program code is optimized. Only OMP parallelization pragmas are recognized.
noomp	Enables automatic parallelization and optimization of program code.
explicit	Enables pragmas controlling explicit parallelization of loops.
noexplicit	Disables pragmas controlling explicit parallelization of loops.
nested_par	If specified, nested parallel constructs are not serialized. nested_par does not provide true nested parallelism because it does not cause new team of threads to be created for nested parallel regions. Instead, threads that are currently available are re-used. This option should be used with caution. Depending on the number of threads available and the amount of work in an outer loop, inner loops could be executed sequentially even if this option is in effect. Parallelization overhead may not necessarily be offset by program performance gains.
nonested_par	Disables parallelization of nested parallel constructs.
rec_locks	If specified, recursive locks are used, and nested critical sections will not cause a deadlock.
norec_locks	If specified, recursive locks are not used.
schedule=sched_type[=n]	Specifies what kind of scheduling algorithms and chunk size (<i>n</i>) are used for loops to which no other scheduling algorithm has been explicitly assigned in the source code. If <i>sched_type</i> is not specified, runtime is assumed for the default setting.

Notes

- The **-qnosmp** default option setting specifies that no code should be generated for parallelization directives, though syntax checking will still be performed.
- Specifying **-qsmp** without suboptions is equivalent to specifying **-qsmp=auto:explicit:noomp:norec_locks:nonested_par:schedule=runtime** or **-qsmp=opt:explicit:noomp:norec_locks:nonested_par:schedule=runtime**.
- Specifying **-qsmp** implicitly sets **-O2**. The **-qsmp** option overrides **-qnooptimize**, but does not override **-O3**, **-O4**, or **-O5**. When debugging parallelized program code, you can disable optimization in parallelized program code by specifying **qsmp=noopt**.
- Specifying **-qsmp** defines the **_IBMSMP** preprocessing macro.
- **-qsmp** must be used only with thread-safe compiler mode invocations such as **xlcr**. These invocations ensure that the **pthreads**, **xlsmp**, and thread-safe versions of all default run-time libraries are linked to the resulting executable.

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 created within the thread are private.
- Variables in existence prior to entering a parallel region are shared unless specified otherwise.
- 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 */
int main (argc,...) {                  /* argc 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;                         /* b is still private here ! */
            foo (i);                     /* i is private here because it */
                                        /* is an iteration variable */
        }

        #pragma omp parallel
        {
            = b                          /* 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;                              /* c is private for the parallel */
                                        /* region it was called from */
    ... }

```

The compiler can privatize some shared variables if it is possible to do so without changing the semantics of the program. For example, if each loop iteration uses a unique value of a shared variable, that variable can be privatized. Privatized shared variables are reported by the **-qinfo=private** option. Use critical sections to synchronize access to all shared variables not listed in this report.

Some OpenMP preprocessor directives 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. That is, it causes each thread to initialize its private copy of a variable with the original object on entry to the parallel construct. In this case, the initializer is the value of the variable's original object.
lastprivate	The lastprivate clause provides a superset of the functionality provided by the private clause. That is, it causes the last iteration of the parallel construct to assign its value to the external variable. In this way, the value of the private variable is known upon exiting the parallel construct. In this case, the last value in the last loop or section is assigned to the variable's original object.
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 informs the compiler that the variables in the clause will behave as an arithmetic reduction. This allows the compiler to optimize the parallelization, increasing the performance.
default	The default clause allows the user to affect the data scope attributes of variables.

Pragma Directives

The directives in this section control how the compiler handles parallel processing in your program. These directives apply only to the statement or statement block immediately following the directive. For more information about how to use the pragmas, see "OpenMP Pragma Directives" on page 3.

#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

The syntax of this pragma is:

```
#pragma omp atomic
    <statement_block>
```

statement_block is either a unary or binary statement of scalar type. The possible binary statement is:

```
x bin_op = expr
```

where:

- *bin_op* is +, *, -, /, &, ^, |, <<, or >>
- *expr* is an scalar expression that does not reference *x*.

The possible unary statement is x++, ++x, x--, or --x.

Notes:

1. Load and store operations are atomic only for object *x*. Evaluation of *expr* is not atomic.
2. All atomic references to a given object in your program must have a compatible type.
3. 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 parallel

Description

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

Syntax

The syntax of this pragma is:

```
#pragma omp parallel [clause [clause] ...]
<statement_block>
```

The following table lists and describes the *clause* options:

Clause	Description
if (<i>exp</i>)	When the if argument is specified, the program code executes in parallel only if the scalar expression represented by <i>exp</i> evaluates to a non-zero value at run-time. Only one if clause can be specified.
private (<i>list</i>)	Declares the scope of the data variables in <i>list</i> to be private to each thread. Data variables in <i>list</i> are separated by commas.
firstprivate (<i>list</i>)	Declares the scope of the data variables in <i>list</i> 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 <i>list</i> are separated by commas.
shared (<i>list</i>)	Declares the scope of the data variables in <i>list</i> 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 explicitly listed in a data scope clause, with the exception of those variables that are: <ul style="list-style-type: none">• 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.

Clause	Description
copyin (<i>list</i>)	<p>For each data variable specified in <i>list</i>, 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 <i>list</i> are separated by commas.</p> <p>Each data variable specified in the copyin clause must be a threadprivate variable.</p>
reduction (<i>operator: list</i>)	<p>Performs a reduction on all scalar variables in <i>list</i> using the specified <i>operator</i>. Reduction variables in <i>list</i> are separated by commas.</p> <p>A private copy of each variable in <i>list</i> 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.</p> <p>Variables specified in the reduction clause:</p> <ul style="list-style-type: none"> • 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 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[ clause] ...]
<for_loop>
```

where *clause* is any of the following:

private (<i>list</i>)	Declares the scope of the data variables in <i>list</i> to be private to each thread. Data variables in <i>list</i> are separated by commas.
firstprivate (<i>list</i>)	Declares the scope of the data variables in <i>list</i> 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 <i>list</i> are separated by commas.

lastprivate (<i>list</i>)	Declares the scope of the data variables in <i>list</i> to be private to each thread. The final value of each variable in <i>list</i> , 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 <i>list</i> are separated by commas.
reduction (<i>operator:list</i>)	Performs a reduction on all scalar variables in <i>list</i> using the specified <i>operator</i> . Reduction variables in <i>list</i> are separated by commas.
	<p>A private copy of each variable in <i>list</i> 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.</p> <p>Variables specified in the reduction clause:</p> <ul style="list-style-type: none"> • 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 $\text{ceiling}(\text{number_of_iterations}/\text{number_of_threads})$.

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 $\text{ceiling}(\text{number_of_iterations} / \text{number_of_threads})$. Remaining chunks are of size $\text{ceiling}(\text{number_of_iterations_remaining} / \text{number_of_threads})$.

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 $\text{ceiling}(\text{number_of_iterations}/\text{number_of_threads})$. 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*.

static,1

Iterations of a loop are divided into chunks of size 1. Each chunk is assigned to a thread in *round-robin* fashion.

This scheduling policy is also known as *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:

<i>init_expr</i>	takes form:	<i>iv</i> = <i>b</i> <i>integer-type iv</i> = <i>b</i>
<i>exit_cond</i>	takes form:	<i>iv</i> <= <i>ub</i> <i>iv</i> < <i>ub</i> <i>iv</i> >= <i>ub</i> <i>iv</i> > <i>ub</i>
<i>incr_expr</i>	takes form:	++ <i>iv</i> <i>iv</i> ++ -- <i>iv</i> <i>iv</i> -- <i>iv</i> += <i>incr</i> <i>iv</i> -= <i>incr</i> <i>iv</i> = <i>iv</i> + <i>incr</i> <i>iv</i> = <i>incr</i> + <i>iv</i> <i>iv</i> = <i>iv</i> - <i>incr</i>

and where:

<i>iv</i>	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..
<i>b, ub, incr</i>	Loop invariant signed integer expressions. No synchronization is performed when evaluating these expressions and evaluated side effects may result in indeterminate values..

Notes

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 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 parallel for [clause[ clause] ...]
<for_loop>
```

Notes

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

#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
statement_block
```

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 section, #pragma omp sections

Description

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

Syntax

```
#pragma omp sections [clause[ clause] ...]
{
    [#pragma omp section]
    statement-block
    [#pragma omp section]
    statement-block
    .
    .
    .
}
```

where *clause* is any of the following:

<code>private (list)</code>	Declares the scope of the data variables in <i>list</i> to be private to each thread. Data variables in <i>list</i> are separated by commas.
<code>firstprivate (list)</code>	Declares the scope of the data variables in <i>list</i> 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 <i>list</i> are separated by commas.
<code>lastprivate (list)</code>	Declares the scope of the data variables in <i>list</i> to be private to each thread. The final value of each variable in <i>list</i> , 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 <i>list</i> are separated by commas.
<code>reduction (operator: list)</code>	Performs a reduction on all scalar variables in <i>list</i> using the specified <i>operator</i> . Reduction variables in <i>list</i> 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 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 [clause] ...]
{
    [#pragma omp section]
    statement-block
    [#pragma omp section]
```

```

        statement-block
        .
        .
        .
    ]
}

```

Notes

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

#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[ clause] ...]
    statement_block

```

where *clause* is any of the following:

private (<i>list</i>)	Declares the scope of the data variables in <i>list</i> to be private to each thread. Data variables in <i>list</i> are separated by commas.
firstprivate (<i>list</i>)	Declares the scope of the data variables in <i>list</i> 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 <i>list</i> are separated by commas.
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.

Notes

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

#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
    statement_block

```

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 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)]
    statement_block
```

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 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 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 threadprivate

Description

The **omp threadprivate** directive defines the scope of selected file-scope data variables as being private to a thread, but file-scope visible within that thread.

Syntax

```
#pragma omp threadprivate (list)
```

where *list* is a comma-separated list of variables.

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.
- 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**, **schedule**, and **if** 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.

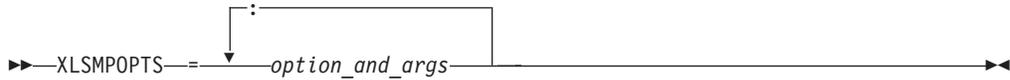
Parallel Processing Support

This section contains information on environment variables and built-in functions used to control parallel processing. Topics in this section are:

- “Run-time Options for Parallel Processing” on page 18
- “OpenMP Run-Time Options for Parallel Processing” on page 20
- “Built-in Functions Used for Parallel Processing” on page 22

Run-time Options for Parallel Processing

Run-time options affecting parallel processing can be specified with the XLSMPOPTS environment variable. This environment variable must be set before you run an application, and uses basic syntax of the form:



Parallelization run-time options can also be specified using OMP environment variables. When run-time options specified by OMP- and XLSMPOPTS-specific environment variables conflict, OMP options will prevail.

Note: You must use thread-safe compiler mode invocations when compiling parallelized program code.

Run-time option settings for the XLSMPOPTS environment variable are shown below, grouped by category:

Scheduling Algorithm Options

XLSMPOPTS Environment Variable Option	Description
<code>schedule=algorithm=[n]</code>	<p>This option specifies the scheduling algorithm used for loops not explicitly assigned a scheduling algorithm.</p> <p>Valid options for <i>algorithm</i> are:</p> <ul style="list-style-type: none"> • guided • affinity • dynamic • static <p>If specified, the chunk size <i>n</i> must be an integer value of 1 or greater.</p> <p>The default is scheduling algorithm is static.</p>

Parallel Environment Options

XLSMPOPTS Environment Variable Option	Description
<code>parthds=num</code>	<p><i>num</i> represents the number of parallel threads requested, which is usually equivalent to the number of processors available on the system.</p> <p>Some applications cannot use more threads than the maximum number of processors available. Other applications can experience significant performance improvements if they use more threads than there are processors. This option gives you full control over the number of user threads used to run your program.</p> <p>The default value for <i>num</i> is the number of processors available on the system.</p>

XLSMPOPTS Environment Variable Option	Description
usrthds= <i>num</i>	<p><i>num</i> represents the number of user threads expected.</p> <p>This option should be used if the program code explicitly creates threads, in which case <i>num</i> should be set to the number of threads created.</p> <p>The default value for <i>num</i> is 0.</p>
stack= <i>num</i>	<p><i>num</i> specifies the largest amount of space required for a thread's stack.</p> <p>The default value for <i>num</i> is 41943042097152.</p> <p>The glibc library is compiled by default to allow a stack size of 2 Mb. Setting <i>num</i> to a value greater than this will cause the default stack size to be used. If larger stack sizes are required, you should link the program to a glibc library compiled with the <code>FLOATING_STACKS</code> parameter turned on.</p>

Performance Tuning Options

XLSMPOPTS Environment Variable Option	Description
spins= <i>num</i>	<p><i>num</i> represents the number of loop spins before a yield occurs.</p> <p>When a thread completes its work, the thread continues executing in a tight loop looking for new work. One complete scan of the work queue is done during each busy-wait state. An extended busy-wait state can make a particular application highly responsive, but can also harm the overall responsiveness of the system unless the thread is given instructions to periodically scan for and yield to requests from other applications.</p> <p>A complete busy-wait state for benchmarking purposes can be forced by setting both spins and yields to 0.</p> <p>The default value for <i>num</i> is 100.</p>
yields= <i>num</i>	<p><i>num</i> represents the number of yields before a sleep occurs.</p> <p>When a thread sleeps, it completely suspends execution until another thread signals that there is work to do. This provides better system utilization, but also adds extra system overhead for the application.</p> <p>The default value for <i>num</i> is 100.</p>
delays= <i>num</i>	<p><i>num</i> represents a period of do-nothing delay time between each scan of the work queue. Each unit of delay is achieved by running a single no-memory-access delay loop.</p> <p>The default value for <i>num</i> is 500.</p>

Dynamic Profiling Options

XLSMPOPTS Environment Variable Option	Description
<code>profilefreq=num</code>	<p><i>num</i> represents the sampling rate at which each loop is revisited to determine appropriateness for parallel processing.</p> <p>The run-time library uses dynamic profiling to dynamically tune the performance of automatically-parallelized loops. Dynamic profiling gathers information about loop running times to determine if the loop should be run sequentially or in parallel the next time through. Threshold running times are set by the parthreshold and seqthreshold dynamic profiling options, described below.</p> <p>If <i>num</i> is 0, all profiling is turned off, and overheads that occur because of profiling will not occur. If <i>num</i> is greater than 0, running time of the loop is monitored once every <i>num</i> times through the loop.</p> <p>The default for <i>num</i> is 16. The maximum sampling rate is 32. Higher values of <i>num</i> are changed to 32.</p>
<code>parthreshold=mSec</code>	<p><i>mSec</i> specifies the expected running time in milliseconds below which a loop must be run sequentially. <i>mSec</i> can be specified using decimal places.</p> <p>If parthreshold is set to 0, a parallelized loop will never be serialized by the dynamic profiler.</p> <p>The default value for <i>mSec</i> is 0.2 milliseconds.</p>
<code>seqthreshold=mSec</code>	<p><i>mSec</i> specifies the expected running time in milliseconds beyond which a loop that has been serialized by the dynamic profiler must revert to being run in parallel mode again. <i>mSec</i> can be specified using decimal places.</p> <p>The default value for <i>mSec</i> is 5 milliseconds.</p>

OpenMP Run-Time Options for Parallel Processing

OpenMP run-time options affecting parallel processing are set by specifying OMP environment variables. These environment variables, which must be set before you run an application, use syntax of the form:

►►—*env_variable*—=—*option_and_args*—◄◄

Note: You must use thread-safe compiler mode invocations when compiling parallelized program code.

OpenMP run-time options fall into different categories as described below:

Scheduling Algorithm Environment Variable

`OMP_SCHEDULE=algorithm` This option specifies the scheduling algorithm used for loops not explicitly assigned a scheduling algorithm with the `omp schedule` directive. For example:

```
OMP_SCHEDULE="guided, 4"
```

Valid options for *algorithm* are:

- `dynamic[, n]`
- `guided[, n]`
- `runtime`
- `static[, n]`

If specified, the value of *n* must be an integer value of 1 or greater.

The default is scheduling algorithm is **static**.

Parallel Environment Environment Variables

`OMP_NUM_THREADS=num` *num* represents the number of parallel threads requested, which is usually equivalent to the number of processors available on the system.

This number can be overridden during program execution by calling the `omp_set_num_threads()` runtime library function.

Some applications cannot use more threads than the maximum number of processors available. Other applications can experience significant performance improvements if they use more threads than there are processors. This option gives you full control over the number of user threads used to run your program.

The default value for *num* is the number of processors available on the system.

`OMP_NESTED=TRUE | FALSE`

This environment variable enables or disables nested parallelism. The setting of this environment variable can be overridden by calling the `omp_set_nested()` runtime library function.

If nested parallelism is disabled, nested parallel regions are serialized and run in the current thread.

In the current implementation, nested parallel regions are always serialized. As a result, `OMP_SET_NESTED` does not have any effect, and `omp_get_nested()` always returns 0. If `-qsmp=nested_par` option is on (only in non-strict OMP mode), nested parallel regions may employ additional threads as available. However, no new team will be created to run nested parallel regions.

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

Dynamic Profiling Environment Variable

OMP_DYNAMIC=TRUE | FALSE This environment variable enables or disables dynamic adjustment of the number of threads available for running parallel regions.

If set to TRUE, the number of threads available for executing parallel regions may be adjusted at runtime to make the best use of system resources.

If set to FALSE, dynamic adjustment is disabled.

The default setting is TRUE.

Built-in Functions Used for Parallel Processing

Use these built-in functions to obtain information about the parallel environment. Function definitions for the **omp_** functions can be found in the **omp.h** header file.

Function Prototype	Description
int omp_get_num_threads(void);	This function returns the number of threads currently in the team executing the parallel region from which it is called.
int omp_get_max_threads(void);	This function returns the maximum value that can be returned by calls to omp_get_num_threads.
int omp_get_thread_num(void);	This function returns the thread number, within its team, of the thread executing the function. The thread number lies between 0 and omp_get_num_threads()-1, inclusive. The master thread of the team is thread 0.
int omp_get_num_procs(void);	This function returns the maximum number of processors that could be assigned to the program.
int omp_in_parallel(void);	This function returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns 0.
void omp_set_dynamic(int dynamic_threads);	This function enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.
int omp_get_dynamic(void);	This function returns non-zero if dynamic thread adjustments enabled and returns 0 otherwise.
void omp_set_nested(int nested);	This function enables or disables nested parallelism.
int omp_get_nested(void);	This function returns non-zero if nested parallelism is enabled and 0 if it is disabled.
void omp_init_lock(omp_lock_t *lock); void omp_init_nest_lock(omp_nest_lock_t *lock);	These functions provide the only means of initializing a lock. Each function initializes the lock associated with the parameter lock for use in subsequent calls.
void omp_destroy_lock (omp_lock_t *lock); void omp_destroy_nest_lock (omp_nest_lock_t *lock);	These functions ensure that the targeted lock variable is uninitialized.
void omp_set_lock(omp_lock_t *lock); void omp_set_nest_lock(omp_nest_lock_t *lock);	Each of these functions blocks the thread executing the function until the specified lock is available and then sets the lock. A simple lock is available if it is unlocked. A nestable lock is available if it is unlocked or if it is already owned by the thread executing the function.

Function Prototype	Description
<pre>void omp_unset_lock(omp_lock_t *lock); void omp_unset_nest_lock(omp_nest_lock_t *lock);</pre>	These functions provide the means of releasing ownership of a lock.
<pre>int omp_test_lock(omp_lock_t *lock); int omp_test_nest_lock(omp_nest_lock_t *lock);</pre>	These functions attempt to set a lock but do not block execution of the thread.

Note: In the current implementation, nested parallel regions are always serialized. As a result, **omp_set_nested** does not have any effect, and **omp_get_nested** always returns 0.

Objective-C and XL C/C++ for Mac OS X

Objective-C is an object oriented extension to the C language. IBM XL C/C++ provides Objective-C as a technology preview to allow application developers to use XL C/C++ with the Cocoa framework. The extended capability of XL C/C++ allows greater compatibility with the Apple integrated development environments and the ability to use XL C/C++ to compile the graphical components of an application.

The compiler recognizes an input file with the extension *.m* as an Objective-C file, and compiles it without requiring a specific compiler option. Please note that Objective-C++ files are not supported. If the input file has the file name extension *c*, *C*, *cpp*, *cxx*, *cc*, *cp* or *c++* and contains Objective-C code, you can inform the compiler to compile the file as an Objective-C file by specifying the `-qsource=objc` compiler option. This option overrides the source type implied by the suffix of the source file.

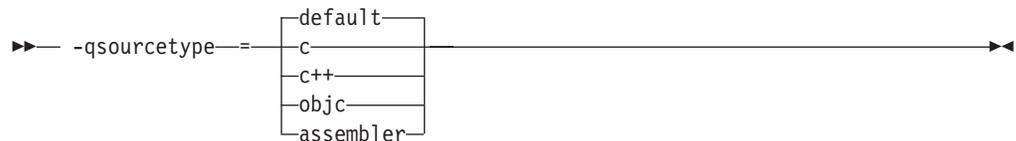
The Objective-C Technology Preview is provided to demonstrate new language functionality and has not been tuned for performance.

Compiler Options

The following are commonly used options when compiling Objective-C programs:

`-qsource`

The `-qsource` option supports the `objc` sub-option as a technology preview. The `-qsource` option instructs the compiler to not rely on the filename suffix, and to instead assume a source type as specified by the sub-option.



where:

<code>default</code>	The compiler assumes that the programming language of a source file will be implied by its filename suffix.
<code>c</code>	The compiler compiles all source files following this option as if they are C language source files.
<code>c++</code>	The compiler compiles all source files following this option as if they are C++ language source files.
<code>objc</code>	The compiler compiles all source files following this option as if they are Objective-C language source files.
<code>assembler</code>	The compiler compiles all source files following this option as if they are Assembler language source files.

Notices

Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States, or other countries, or both:

IBM

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 9899-1999 (E)).
- The C language is consistent with the OpenMP C Application Programming Interface, Version 1.0.
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).



Program Number: 5724-G12