



XL Fortran Advanced Edition
Version 8.1 for Mac OS X
Technology Preview



XL Fortran Advanced Edition
Version 8.1 for Mac OS X
Technology Preview

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.

Contents

Preface	v
--------------------------	----------

Compiling XL Fortran SMP Programs	1
-qsmp Option	2

Setting Run-Time Options	7
The XLSMPOPTS Environment Variable	7

OpenMP Environment Variables	13
OMP_DYNAMIC Environment Variable	13
OMP_NESTED Environment Variable	13
OMP_NUM_THREADS Environment Variable.	14
OMP_SCHEDULE Environment Variable	14

SMP Directives	17
An Introduction to SMP Directives	17
Parallel Region Construct.	17
Work-sharing Constructs	17
Combined Parallel Work-sharing Constructs	18
Synchronization Constructs	18
Other OpenMP Directives	18
Non-OpenMP SMP Directives	18
Detailed Descriptions of SMP Directives	18
ATOMIC	18
BARRIER	21
CRITICAL / END CRITICAL	22
DO / END DO	23
DO SERIAL	26
FLUSH	27
MASTER / END MASTER	29
ORDERED / END ORDERED	30
PARALLEL / END PARALLEL.	33
PARALLEL DO / END PARALLEL DO	35
PARALLEL SECTIONS / END PARALLEL SECTIONS	38
PARALLEL WORKSHARE / END PARALLEL WORKSHARE	41
SCHEDULE	42
SECTIONS / END SECTIONS	45
SINGLE / END SINGLE	48
THREADLOCAL	52
THREADPRIVATE	54

WORKSHARE	58
OpenMP Directive Clauses	61
Global Rules for Directive Clauses.	61
COPYIN	63
COPYPRIVATE	64
DEFAULT	64
IF.	66
FIRSTPRIVATE	67
LASTPRIVATE	68
NUM_THREADS	69
ORDERED.	70
PRIVATE	70
REDUCTION.	72
SCHEDULE	74
SHARED	76

OpenMP Execution Environment and Lock Routines	79
omp_destroy_lock	80
omp_destroy_nest_lock	80
omp_get_dynamic	80
omp_get_max_threads.	81
omp_get_nested	81
omp_get_num_procs	81
omp_get_num_threads	82
omp_get_thread_num	82
omp_get_wtick	83
omp_get_wtime	84
omp_in_parallel	84
omp_init_lock	85
omp_init_nest_lock.	85
omp_set_dynamic	86
omp_set_lock.	86
omp_set_nested	87
omp_set_nest_lock	87
omp_set_num_threads.	88
omp_test_lock	88
omp_test_nest_lock.	89
omp_unset_lock	89
omp_unset_nest_lock	89

Trademarks and Service Marks	91
---	-----------

Preface

This document provides information on the following:

- “Compiling XL Fortran SMP Programs” on page 1
- “Setting Run-Time Options” on page 7
- “OpenMP Environment Variables” on page 13
- “SMP Directives” on page 17
- “OpenMP Execution Environment and Lock Routines” on page 79

Note!

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

Compiling XL Fortran SMP Programs

You can use the `xlf_r`, `xlf90_r`, or `xlf95_r` commands to compile XL Fortran SMP programs. The `xlf_r` command is similar to the `xlf` command, the `xlf90_r` command is similar to the `xlf90` command, and the `xlf95_r` command is similar to the `xlf95` command. The main difference is that the thread-safe components are used to link and bind the object files if you specify the `xlf_r`, `xlf90_r`, or `xlf95_r` commands.

Note that using any of these commands alone does not imply parallelization. For the compiler to recognize the SMP directives and activate parallelization, you must also specify `-qsmp`. In turn, you can only specify the `-qsmp` option in conjunction with one of these six invocation commands. When you specify `-qsmp`, the driver links in the libraries specified on the `smplibraries` line in the active stanza of the configuration file.

A detailed description of the `-qsmp` compiler option is provided below.

-qsmp Option

Format

-qsmp[=*suboptions*]
-qnosmp

Indicates that code should be produced for an SMP system. The default is to produce code for a uniprocessor machine. When you specify this option, the compiler recognizes all directives with the trigger constants **SMP\$**, **\$OMP**, and **IBMP** (unless you specify the **omp** suboption).

Only the **xlfr_r**, **xlfr90_r**, and **xlfr95_r** invocation commands automatically link in all of the thread-safe components. You can use the **-qsmp** option with the **xlfr**, **xlfr90**, **xlfr95**, **f77**, and **fort77** invocation commands, but you are responsible for linking in the appropriate components. . If you use the **-qsmp** option to compile any source file in a program, then you must specify the **-qsmp** option at link time as well, unless you link by using the **ld** command.

Parameters

auto | **noauto**

This suboption controls automatic parallelization. By default, the compiler will attempt to parallelize explicitly coded **DO** loops as well as those that are generated by the compiler for array language. If you specify the suboption **noauto**, automatic parallelization is turned off, and only constructs that are marked with prescriptive directives are parallelized. If the compiler encounters the **omp** suboption and the **-qsmp** or **-qsmp=auto** suboptions are not explicitly specified on the command line, the **noauto** suboption is implied.

nested_par | nonested_par

If you specify the **nested_par** suboption, the compiler parallelizes prescriptive nested parallel constructs (**PARALLEL DO**, **PARALLEL SECTIONS**). This includes not only the loop constructs that are nested within a scoping unit but also parallel constructs in subprograms that are referenced (directly or indirectly) from within other parallel constructs. By default, the compiler serializes a nested parallel construct. Note that this option has no effect on loops that are automatically parallelized. In this case, at most one loop in a loop nest (in a scoping unit) will be parallelized.

Note that the implementation of the **nested_par** suboption does not comply with the OpenMP Fortran API. If you specify this suboption, the run-time library uses the same threads for the nested **PARALLEL DO** and **PARALLEL SECTIONS** constructs that it used for the enclosing **PARALLEL** constructs.

omp | noomp

If you specify this suboption, the compiler enforces compliance with the OpenMP Fortran API. Specifying this option has the following effects:

- Automatic parallelization is turned off.
- All previously recognized directive triggers are ignored.

- The **-qcclines** compiler option is turned on if you specify **-qsmp=omp**.
- The **-qcclines** compiler option is not turned on if you specify **-qnocclines** and **-qsmp=omp**.
- The only recognized directive trigger is **\$OMP**. However, you can specify additional triggers on subsequent **-qdirective** options.
- The compiler issues warning messages if your code contains any language constructs that do not conform to the OpenMP Fortran API.

Specifying this option when the C preprocessor is invoked also defines the **_OPENMP** C preprocessor macro automatically, with the value *200011*, which is useful in supporting conditional compilation. This macro is only defined when the C preprocessor is invoked.

opt | **noopt** If the **-qsmp=noopt** suboption is specified, the compiler will do the smallest amount of optimization that is required to parallelize the code. This is useful for debugging because **-qsmp** enables the **-O2** option by default, which may result in the movement of some variables into registers that are inaccessible to the debugger. However, if the **-qsmp=noopt** and **-g** options are specified, these variables will remain visible to the debugger.

rec_locks | **norec_locks** This suboption specifies whether recursive locks are used to avoid problems associated with **CRITICAL** constructs. If you specify the **rec_locks** suboption, a thread can enter a **CRITICAL** construct from within the dynamic extent of another **CRITICAL** construct that has the same name. If you specify **norec_locks**, a deadlock would occur in such a situation.

The default is **norec_locks**, or regular locks.

schedule=option

The **schedule** suboption can take any one of the following subsuboptions:

affinity[=*n*]

The iterations of a loop are initially divided into *number_of_threads* partitions, containing **CEILING**(*number_of_iterations* / *number_of_threads*) iterations. Each partition is initially assigned to a thread and is then further subdivided into chunks that each contain *n* iterations. If *n* has not been specified, then the chunks consist of **CEILING**(*number_of_iterations_left_in_partition* / 2) loop iterations.

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

The work in a partition initially assigned to a sleeping thread will be completed by threads that are active.

dynamic[=*n*]

The iterations of a loop are divided into chunks containing *n* iterations each. If *n* has not been specified, then the chunks consist of **CEILING**(number_of_iterations / number_of_threads) iterations.

Active threads are assigned these chunks on a "first-come, first-do" basis. Chunks of the remaining work are assigned to available threads until all work has been assigned.

If a thread is asleep, its assigned work will be taken over by an active thread once that thread becomes available.

guided[=*n*]

The iterations of a loop are divided into progressively smaller chunks until a minimum chunk size of *n* loop iterations is reached. If *n* has not been specified, the default value for *n* is 1 iteration.

The first chunk contains **CEILING**(number_of_iterations / number_of_threads) iterations. Subsequent chunks consist of **CEILING**(number_of_iterations_left / number_of_threads) iterations. Active threads are assigned chunks on a "first-come, first-do" basis.

runtime

Specifies that the chunking algorithm will be determined at run time.

static[=*n*]

The iterations of a loop are divided into chunks containing *n* iterations each. Each thread is assigned chunks in a "round-robin" fashion. This is known as *block cyclic scheduling*. If the value of *n* is 1, then the scheduling type is specifically referred to as *cyclic scheduling*.

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

If a thread is asleep and it has been assigned work, it will be awakened so that it may complete its work.

threshold=*n*

Controls the amount of automatic loop parallelization that occurs. The value of *n* represents the lower limit allowed for parallelization of a loop, based on the level of "work" present in a loop. Currently, the calculation of "work" is weighted heavily by the number of iterations in the loop. In general, the higher the value specified for *n*, the fewer loops are parallelized. If this suboption is not specified, the program will use the default value *n*=100.

Processing

- If you specify **-qsmp** more than once, the previous settings of all suboptions are preserved, unless overridden by the subsequent suboption setting. The compiler does not override previous suboptions that you specify. The same is true for the version of **-qsmp** without suboptions; the default options are saved.
- Specifying the **omp** suboption always implies **noauto**, unless you specify **-qsmp** or **-qsmp=auto** on the command line.

- Specifying the **noomp** suboption always implies **auto**.
- The **omp** and **noomp** suboptions only appear in the compiler listing if you explicitly set them.
- If **-qsmp** is specified without any suboptions, **-qsmp=opt** becomes the default setting. If **-qsmp** is specified after the **-qsmp=noopt** suboption has been set, the **-qsmp=noopt** setting will always be ignored.
- If the option **-qsmp** with no suboptions follows the suboption **-qsmp=noopt** on a command line, the **-qsmp=opt** and **-qsmp=auto** options are enabled.
- Specifying the **-qsmp=noopt** suboption implies that **-qsmp=noauto**. It also implies **-qnoopt**. This option overrides performance options such as **-O2**, **-O3**, **-qhot**, anywhere on the command line unless **-qsmp** appears after **-qsmp=noopt**.
- Object files generated with the **-qsmp=opt** option can be linked with object files generated with **-qsmp=noopt**. The visibility within the debugger of the variables in each object file will not be affected by linking.

Restrictions

The **-qsmp=noopt** suboption may affect the performance of the program.

Within the same **-qsmp** specification, you cannot specify the **omp** suboption before or after certain suboptions. The compiler issues warning messages if you attempt to specify them with **omp**:

auto This suboption controls automatic parallelization, but **omp** turns off automatic parallelization.

nested_par

Note that the implementation of the **nested_par** suboption does not comply with the OpenMP Fortran API. If you specify this suboption, the run-time library uses the same threads for the nested **PARALLEL DO** and **PARALLEL SECTIONS** constructs that it used for the enclosing **PARALLEL** constructs.

rec_locks

This suboption specifies a behaviour for **CRITICAL** constructs that is inconsistent with the OpenMP Fortran API.

schedule=affinity=*n*

The affinity scheduling type does not appear in the OpenMP Fortran API standard.

Examples

The **-qsmp=noopt** suboption overrides performance optimization options anywhere on the command line unless **-qsmp** appears after **-qsmp=noopt**. The following examples illustrate that all optimization options that appear after **-qsmp=noopt** are processed according to the normal rules of scope and precedence.

Example 1

```
xlf90 -qsmp=noopt -O3...
is equivalent to
xlf90 -qsmp=noopt...
```

Example 2

```
xlf90 -qsmp=noopt -O3 -qsmp...
is equivalent to
xlf90 -qsmp -O3...
```

Compiling SMP Programs

Example 3

```
xlF90 -qsmp=noopt -03 -qhot -qsmp -02...  
is equivalent to  
xlF90 -qsmp -qhot -02...
```

If you specify the following, the compiler recognizes both the **\$OMP** and **SMP\$** directive triggers and issues a warning if a directive specified with either trigger is not allowed in OpenMP.

```
-qsmp=omp -qdirective=SMP$
```

If you specify the following, the **noauto** suboption is used. The compiler issues a warning message and ignores the **auto** suboption.

```
-qsmp=omp:auto
```

In the following example, you should specify **-qsmp=rec_locks** to avoid a deadlock caused by **CRITICAL** constructs.

```
program t  
  integer i, a, b  
  
  a = 0  
  b = 0  
!smp$ parallel do  
  do i=1, 10  
!smp$ critical  
  a = a + 1  
!smp$ critical  
  b = b + 1  
!smp$ end critical  
!smp$ end critical  
  enddo  
end
```

If you use the **xlF**, **xlF_r**, **f77**, or **fort77** command with the **-qsmp** option to compile programs, specify **-qnosave** to make the default storage class automatic, and specify **-qthreaded** to tell the compiler to generate thread-safe code.

Setting Run-Time Options

The XLSMPOPTS Environment Variable

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

```
▶▶ XLSMPOPTS=" runtime_option_name=option_setting "
```

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

You can specify the following run-time options with the **XLSMPOPTS** environment variable:

schedule

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

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

dynamic or guided

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

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

affinity

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

$$\text{CEILING}(\text{number_of_iterations} / \text{number_of_threads})$$

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

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

affinity[=*n*] As described previously, the iterations of a loop are initially

Setting Run-Time Options

divided into partitions, which are then preassigned to the threads. Each of these partitions is then further subdivided into chunks that contain n iterations. If you have not specified n , a chunk consists of $\text{CEILING}(\text{number_of_iterations_left_in_local_partition} / 2)$ loop iterations.

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

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

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

The first chunk contains $\text{CEILING}(\text{number_of_iterations} / \text{number_of_threads})$ iterations. Subsequent chunks consist of $\text{CEILING}(\text{number_of_iterations_left} / \text{number_of_threads})$ iterations.

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

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

If you have not specified **schedule**, the default is set to **schedule=static**, resulting in block scheduling.

Parallel execution options

The three parallel execution options, **parthds**, **usrthds**, and **stack**, are as follows:

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

This option allows you full control over the number of execution threads. The default value for num is 1 if you did not specify **-qsmp**. Otherwise, it is the number of online processors on the machine.

usrthds=<i>num</i>	Specifies the maximum number of threads (<i>num</i>) that you expect your code will explicitly create if the code does explicit thread creation. The default value for <i>num</i> is 0.
stack=<i>num</i>	Specifies the largest amount of space in bytes (<i>num</i>) that a thread's stack will need. The default value for <i>num</i> is 4194304. Set stack=<i>num</i> so it is within the acceptable upper limit. <i>num</i> can be up to 256 MB. An application that exceeds the upper limit may cause a segmentation fault.

Performance tuning options

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

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

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

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

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

The syntax for specifying these options is as follows:

spins[=<i>num</i>]	where <i>num</i> is the number of spins before a yield. The default value for spins is 100.
yields[=<i>num</i>]	where <i>num</i> is the number of yields before a sleep. The default value for yields is 10.
delays[=<i>num</i>]	where <i>num</i> is the number of delays while busy-waiting. The default value for delays is 500.

Setting Run-Time Options

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

For instance, on a dedicated 8-way SMP, setting these options to the following:
`parthds=8 : schedule=dynamic=10 : spins=0 : yields=0`

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

Options to enable and control dynamic profiling

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

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

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

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

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

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

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

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

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

Setting Run-Time Options

OpenMP Environment Variables

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

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

OMP_DYNAMIC Environment Variable

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

►►—OMP_DYNAMIC=—

TRUE
FALSE

—►►

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

The default value for `OMP_DYNAMIC` is `TRUE`. Therefore, if your code needs to use a specific number of threads to run correctly, you should disable dynamic thread adjustment.

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

OMP_NESTED Environment Variable

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

►►—OMP_NESTED=—

TRUE
FALSE

—►►

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

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

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

OMP_NUM_THREADS Environment Variable

The **OMP_NUM_THREADS** environment variable sets the number of threads that a program will use when it runs. The syntax is as follows:

▶▶—OMP_NUM_THREADS=*num*—▶▶

num the maximum number of threads that can be used if dynamic adjustment of the number of threads is enabled. If dynamic adjustment of the number of threads is not enabled, the value of **OMP_NUM_THREADS** is the exact number of threads that can be used. It must be a positive, scalar integer.

The default number of threads that a program uses when it runs is the number of online processors on the machine.

If you specify the number of threads with both the **PARHDS** suboption of the **XLSPMPOPTS** environment variable and the **OMP_NUM_THREADS** environment variable, the **OMP_NUM_THREADS** environment variable takes precedence. The **omp_set_num_threads** subroutine takes precedence over the **OMP_NUM_THREADS** environment variable.

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

```
export OMP_NUM_THREADS=16
```

OMP_SCHEDULE Environment Variable

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

▶▶—OMP_SCHEDULE=*sched_type* [*, chunk_size*]—▶▶

sched_type is either **DYNAMIC**, **GUIDED**, or **STATIC**.

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

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

If you have not specified a schedule type either at compile time (through a directive) or at run time (through the **OMP_SCHEDULE** environment variable or the **SCHEDULE** option of the **XLSPMPOPTS** environment variable), the default schedule type is **STATIC**, and the default chunk size is set to the following for the first $N - 1$ threads:

$$\text{chunk_size} = \text{ceiling}(\text{Iters}/N)$$

It is set to the following for the N th thread, where N is the total number of threads and Iters is the total number of iterations in the **DO** loop:

$$\text{chunk_size} = \text{Iters} - ((N - 1) * \text{ceiling}(\text{Iters}/N))$$

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

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

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

OpenMP Environment Variables

SMP Directives

The Symmetric Multiprocessing (SMP) Directives section contains the following sections:

- “An Introduction to SMP Directives”
- “Detailed Descriptions of SMP Directives” on page 18
- “OpenMP Directive Clauses” on page 61

An Introduction to SMP Directives

The SMP directives described in this section allow you to exert control over parallelization. For example, the **PARALLEL DO** directive specifies that the loop immediately following the directive should be run in parallel. All SMP directives are comment form directives.

- To ensure that the compiler will recognize SMP directives, compile code using either the **xl_f_r**, **xl_f90_r**, or **xl_f95_r** invocation commands, specifying the **-qsmp** compiler option. See the directive descriptions found in this section for more details.
- To ensure that the compiler will link threadsafe libraries, compile code using either the **xl_f_r**, **xl_f90_r**, or **xl_f95_r** invocation commands.
- XL Fortran supports the OpenMP specification, as understood and interpreted by IBM®. To ensure the greatest portability of code, we recommend that you use these directives whenever possible. You should use them with the OpenMP *trigger_constant*, **\$OMP**; but you should not use this *trigger_constant* with any other directive.

XL Fortran supports the following SMP directives, divided as follows:

Parallel Region Construct

Parallel constructs form the basis for OpenMP based parallel execution in XL Fortran. The **PARALLEL/END PARALLEL** directive pair forms a basic parallel construct. Each time an executing thread enters a parallel region, it creates a team of threads and becomes master of that team. This allows parallel execution to take place within that construct by the threads in that team. The following directives are necessary for a parallel region:

PARALLEL	END PARALLEL
-----------------	---------------------

Work-sharing Constructs

Work-sharing constructs divide the execution of code enclosed by the construct between threads in a team. For work-sharing to take place, the construct must be enclosed within the dynamic extent of a parallel region. For further information on work-sharing constructs, see the following directives:

DO	END DO
SECTIONS	END SECTIONS
WORKSHARE	END WORKSHARE

Combined Parallel Work-sharing Constructs

A combined parallel work-sharing construct allows you to specify a parallel region that already contains a single work-sharing construct. These combined constructs are semantically identical to specifying a parallel construct enclosing a single work-sharing construct. For more information on implementing combined constructs, see the following directives:

PARALLEL DO	END PARALLEL DO
PARALLEL SECTIONS	END PARALLEL SECTIONS
PARALLEL WORKSHARE	END PARALLEL WORKSHARE

Synchronization Constructs

The following directives allow you to synchronize the execution of a parallel region by multiple threads in a team:

ATOMIC	
BARRIER	
CRITICAL	END CRITICAL
FLUSH	
ORDERED	END ORDERED

Other OpenMP Directives

The following OpenMP directives provide additional SMP functionality:

MASTER	END MASTER
SINGLE	END SINGLE
THREADPRIVATE	

Non-OpenMP SMP Directives

The following directives provide additional SMP functionality:

DO SERIAL	
SCHEDULE	
THREADLOCAL	

Detailed Descriptions of SMP Directives

The following section contains an alphabetical list of all SMP directives supported by XL Fortran. For information on directive clauses, see “OpenMP Directive Clauses” on page 61..

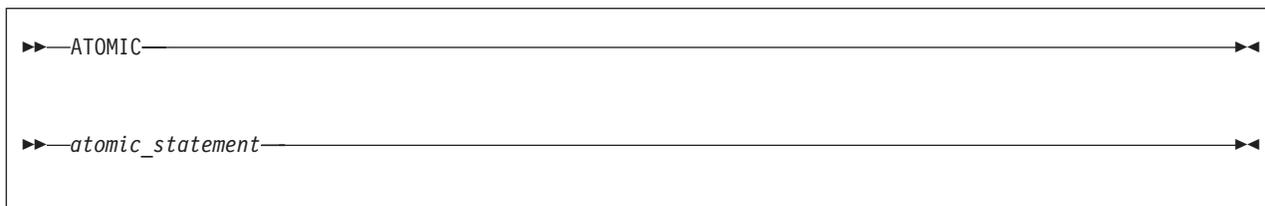
ATOMIC

You can use the **ATOMIC** directive to update a specific memory location safely within a parallel region. When you use **ATOMIC**, you ensure that only one thread is writing to the memory location at a time, avoiding errors which might occur from simultaneous writes to the same memory location.

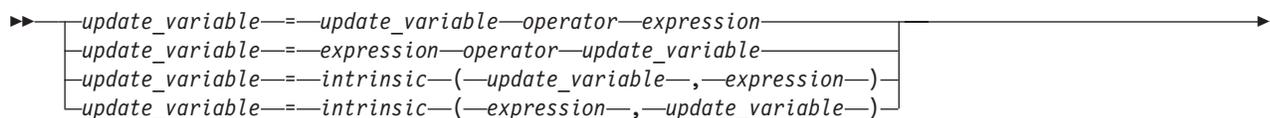
Normally, you would protect a shared variable within a **CRITICAL** construct if it is being updated by more than one thread at a time. However, certain platforms support atomic operations for updating variables. For example, some platforms might support a hardware instruction that reads from a memory location, calculates something and writes back to the location all in one atomic action. The **ATOMIC** directive instructs the compiler to use an atomic operation whenever possible. Otherwise, the compiler will use some other mechanism to perform an atomic update.

The **ATOMIC** directive only takes effect if you specify the **-qsmp** compiler option.

Syntax



where *atomic_statement* is:



update_variable
is a scalar variable of intrinsic type.

intrinsic
is one of **max**, **min**, **iand**, **ior** or **ieor**.

operator
is one of **+**, **-**, *****, **/**, **.AND.**, **.OR.**, **.EQV.**, **.NEQV.** or **.XOR.**

expression
is a scalar expression that does not reference *update_variable*.

Rules

The **ATOMIC** directive applies only to the statement which immediately follows it.

The *expression* in an *atomic_statement* is not evaluated atomically. You must ensure that no race conditions exist in the calculation.

All references made using the **ATOMIC** directive to the storage location of an *update_variable* within the entire program must have the same type and type parameters.

The function *intrinsic*, the operator *operator*, and the assignment must be the intrinsic function, operator and assignment and not a redefined intrinsic function, defined operator or defined assignment.

Examples

Example 1: In the following example, multiple threads are updating a counter. **ATOMIC** is used to ensure that no updates are lost.

ATOMIC

```
PROGRAM P
  R = 0.0
!$OMP PARALLEL DO SHARED(R)
  DO I=1, 10
!$OMP   ATOMIC
    R = R + 1.0
  END DO
  PRINT *,R
END PROGRAM P
```

Expected output:

10.0

Example 2:In the following example, an **ATOMIC** directive is required, because it is uncertain which element of array *Y* will be updated in each iteration.

```
PROGRAM P
  INTEGER, DIMENSION(10) :: Y, INDEX
  INTEGER B
  Y = 5
  READ(*,*) INDEX, B
!$OMP PARALLEL DO SHARED(Y)
  DO I = 1, 10
!$OMP   ATOMIC
    Y(INDEX(I)) = MIN(Y(INDEX(I)),B)
  END DO
  PRINT *, Y
END PROGRAM P
```

Input data:

10 10 8 8 6 6 4 4 2 2 4

Expected output:

5 4 5 4 5 4 5 4 5 4

Example 3: The following example is invalid, because you cannot use an **ATOMIC** operation to reference an array.

```
PROGRAM P
  REAL ARRAY(10)
  ARRAY = 0.0
!$OMP PARALLEL DO SHARED(ARRAY)
  DO I = 1, 10
!$OMP   ATOMIC
    ARRAY = ARRAY + 1.0
  END DO
  PRINT *, ARRAY
END PROGRAM P
```

Example 4:The following example is invalid. The *expression* must not reference the *update_variable*.

```
PROGRAM P
  R = 0.0
!$OMP PARALLEL DO SHARED(R)
  DO I = 1, 10
!$OMP   ATOMIC
    R = R + R
  END DO
  PRINT *, R
END PROGRAM P
```

Related Information

- “CRITICAL / END CRITICAL” on page 22
- “PARALLEL / END PARALLEL” on page 33
- `-qsmp` compiler option

BARRIER

The **BARRIER** directive enables you to synchronize all threads in a team. When a thread encounters a **BARRIER** directive, it will wait until all other threads in the team reach the same point.

The **BARRIER** directive only takes effect if you specify the `-qsmp` compiler option.

Syntax

►►—BARRIER—◄◄

Rules

A **BARRIER** directive binds to the closest dynamically enclosing **PARALLEL** directive, if one exists.

A **BARRIER** directive cannot appear within the dynamic extent of the **CRITICAL**, **DO** (work-sharing), **MASTER**, **PARALLEL DO**, **PARALLEL SECTIONS**, **SECTIONS**, **SINGLE** and **WORKSHARE** directives.

All threads in the team must encounter the **BARRIER** directive if any thread encounters it.

All **BARRIER** directives and work-sharing constructs must be encountered in the same order by all threads in the team.

In addition to synchronizing the threads in a team, the **BARRIER** directive implies the **FLUSH** directive.

Examples

Example 1: An example of the **BARRIER** directive binding to the **PARALLEL** directive. Note: To calculate *C*, we need to ensure that *A* and *B* have been completely assigned to, so threads need to wait.

```

SUBROUTINE SUB1
  INTEGER A(1000), B(1000), C(1000)
!$OMP PARALLEL
!$OMP DO
  DO I = 1, 1000
    A(I) = SIN(I*2.5)
  END DO
!$OMP END DO NOWAIT
!$OMP DO
  DO J = 1, 10000
    B(J) = X + COS(J*5.5)
  END DO
!$OMP END DO NOWAIT
  ...
!$OMP BARRIER
  C = A + B
!$OMP END PARALLEL
END

```


The same lock protects all **CRITICAL** constructs that do not have an explicit *lock_name*. In other words, the compiler will assign the same *lock_name*, thereby ensuring that only one thread enters any unnamed **CRITICAL** construct at a time.

The *lock_name* must not share the same name as any local entity of Class 1.

It is illegal to branch into or out of a **CRITICAL** construct.

The **CRITICAL** construct may appear anywhere in a program.

Although it is possible to nest a **CRITICAL** construct within a **CRITICAL** construct, a deadlock situation may result. The **-qsmp=rec_locks** compiler option can be used to prevent deadlocks.

The **CRITICAL** and **END CRITICAL** directives imply the **FLUSH** directive.

Examples

Example 1: Note that in this example the **CRITICAL** construct appears within a **DO** loop that has been marked with the **PARALLEL DO** directive.

```

      EXPR=0
!OMP$ PARALLEL DO PRIVATE (I)
      DO I = 1, 100
!OMP$   CRITICAL
          EXPR = EXPR + A(I) * I
!OMP$   END CRITICAL
      END DO

```

Example 2: An example specifying a *lock_name* on the **CRITICAL** construct.

```

!SMP$ PARALLEL DO PRIVATE(T)
      DO I = 1, 100
          T = B(I) * B(I-1)
!SMP$   CRITICAL (LOCK)
          SUM = SUM + T
!SMP$   END CRITICAL (LOCK)
      END DO

```

Related Information

- “FLUSH” on page 27
- “PARALLEL / END PARALLEL” on page 33
- **-qsmp** compiler option.

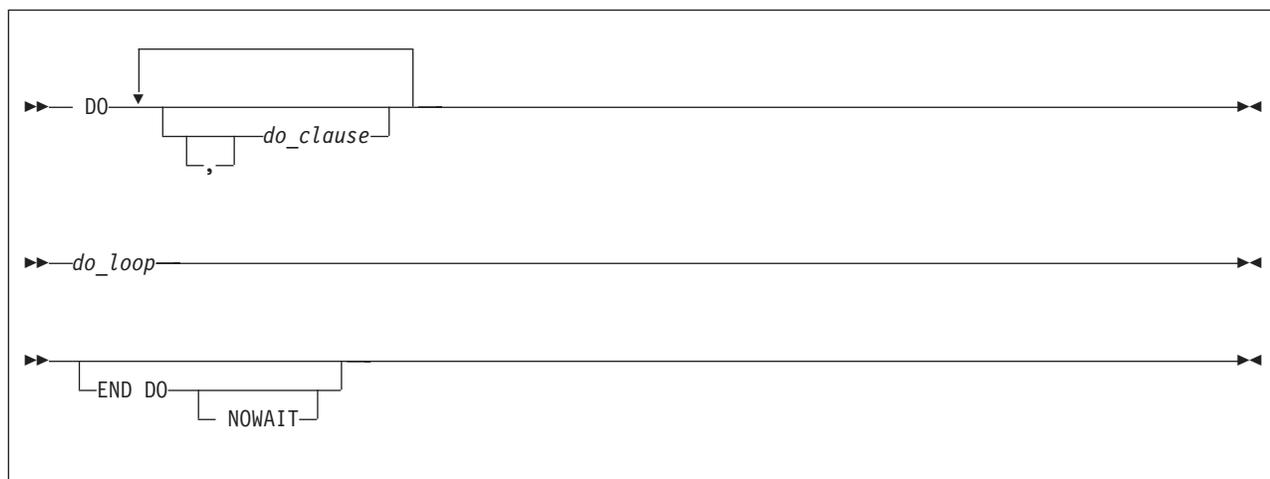
DO / END DO

The **DO** (work-sharing) construct enables you to divide the execution of the loop among the members of the team that encounter it. The **END DO** directive enables you to indicate the end of a **DO** loop that is specified by the **DO** (work-sharing) directive.

The **DO** (work-sharing) and **END DO** directives only take effect when you specify the **-qsmp** compiler option.

DO / END DO

Syntax



where *do_clause* is:



firstprivate_clause

See — “FIRSTPRIVATE” on page 67.

lastprivate_clause

See — “LASTPRIVATE” on page 68.

ordered_clause

See — “ORDERED” on page 70

private_clause

See — “PRIVATE” on page 70.

reduction_clause

See — “REDUCTION” on page 72

schedule_clause

See — “SCHEDULE” on page 74

Rules

The first noncomment line (not including other directives) that follows the **DO** (work-sharing) directive must be a **DO** loop. This line cannot be an infinite **DO** or **DO WHILE** loop. The **DO** (work-sharing) directive applies only to the **DO** loop that is immediately following the directive, and not to any nested **DO** loops.

The **END DO** directive is optional. If you use the **END DO** directive, it must immediately follow the end of the **DO** loop.

You may have a **DO** construct that contains several **DO** statements. If the **DO** statements share the same **DO** termination statement, and an **END DO** directive follows the construct, you can only specify a work-sharing **DO** directive for the outermost **DO** statement of the construct.

If you specify **NOWAIT** on the **END DO** directive, a thread that completes its iterations of the loop early will proceed to the instructions following the loop. The thread will not wait for the other threads of the team to complete the **DO** loop. If you do not specify **NOWAIT** on the **END DO** directive, each thread will wait for all other threads within the same team at the end of the **DO** loop.

If you do not specify the **NOWAIT** clause, the **END DO** directive implies the **FLUSH** directive.

All threads in the team must encounter the **DO** (work-sharing) directive if any thread encounters it. A **DO** loop must have the same loop boundary and step value for each thread in the team. All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

A **DO** (work-sharing) directive must not appear within the dynamic extent of a **CRITICAL** or **MASTER** construct. In addition, it must not appear within the dynamic extent of a **PARALLEL SECTIONS** construct, work-sharing construct, or **PARALLEL DO** loop, unless it is within the dynamic extent of a **PARALLEL** construct.

You cannot follow a **DO** (work-sharing) directive by another **DO** (work-sharing) directive. You can only specify one **DO** (work-sharing) directive for a given **DO** loop.

The **DO** (work-sharing) directive cannot appear with either an **INDEPENDENT** or **DO SERIAL** directive for a given **DO** loop.

Examples

Example 1: An example of several independent **DO** loops within a **PARALLEL** construct. No synchronization is performed after the first work-sharing **DO** loop, because **NOWAIT** is specified on the **END DO** directive.

```
!$OMP PARALLEL
!$OMP DO
    DO I = 2, N
        B(I) = (A(I) + A(I-1)) / 2.0
    END DO
!$OMP END DO NOWAIT
!$OMP DO
    DO J = 2, N
        C(J) = SQRT(REAL(J*J))
    END DO
!$OMP END DO
    C(5) = C(5) + 10
!$OMP END PARALLEL
END
```

Example 2: An example of **SHARED**, and **SCHEDULE** clauses.

```
!$OMP PARALLEL SHARED(A)
!$OMP DO SCHEDULE(STATIC,10)
    DO I = 1, 1000
        A(I) = 1 * 4
    END DO
!$OMP END DO
!$OMP END PARALLEL
```

Example 3: An example of both a **MASTER** and a **DO** (work-sharing) directive that bind to the closest enclosing **PARALLEL** directive.

DO / END DO

```
!$OMP PARALLEL DEFAULT(PRIVATE)
  Y = 100
!$OMP MASTER
  PRINT *, Y
!$OMP END MASTER
!$OMP DO
  DO I = 1, 10
    X(I) = I
    X(I) = X(I) + Y
  END DO
!$OMP END PARALLEL
END
```

Example 4: An example of both the **FIRSTPRIVATE** and the **LASTPRIVATE** clauses on **DO** (work-sharing) directives.

```
X = 100

!$OMP PARALLEL PRIVATE(I), SHARED(X,Y)
!$OMP DO FIRSTPRIVATE(X), LASTPRIVATE(X)
  DO I = 1, 80
    Y(I) = X + I
    X = I
  END DO
!$OMP END PARALLEL
END
```

Example 6: A valid example of a work-sharing **DO** directive applied to nested **DO** statements with a common **DO** termination statement.

```
!$OMP DO                ! A work-sharing DO directive can ONLY
                        ! precede the outermost DO statement.
  DO 100 I= 1,10

! !$OMP DO **Error**   ! Placing the OMP DO directive here is
                        ! invalid

  DO 100 J= 1,10

!      ...

100 CONTINUE
!$OMP END DO
```

Related Information

- “DO SERIAL”
- “FLUSH” on page 27
- “ORDERED / END ORDERED” on page 30
- “PARALLEL / END PARALLEL” on page 33
- “PARALLEL DO / END PARALLEL DO” on page 35
- “PARALLEL SECTIONS / END PARALLEL SECTIONS” on page 38
- “SCHEDULE” on page 42
- **-qsmp** compiler option.

DO SERIAL

The **DO SERIAL** directive indicates to the compiler that the **DO** loop that is immediately following the directive must not be parallelized. This directive is useful in blocking automatic parallelization for a particular **DO** loop. The **DO SERIAL** directive only takes effect if you specify the **-qsmp** compiler option.

Syntax



Rules

The first noncomment line (not including other directives) that is following the **DO SERIAL** directive must be a **DO** loop. The **DO SERIAL** directive applies only to the **DO** loop that immediately follows the directive and not to any loops that are nested within that loop.

You can only specify one **DO SERIAL** directive for a given **DO** loop. The **DO SERIAL** directive must not appear with the **DO**, or **PARALLEL DO** directive on the same **DO** loop.

White space is optional between **DO** and **SERIAL**.

You should not use the OpenMP trigger constant with this directive.

Examples

Example 1: An example with nested **DO** loops where the inner loop (the **J** loop) is not parallelized.

```
!$OMP PARALLEL DO PRIVATE(S,I), SHARED(A)
  DO I=1, 500
    S=0
    !SMP$ DOSERIAL
    DO J=1, 500
      S=S+1
    ENDDO
    A(I)=S+I
  ENDDO
```

Example 2: An example with the **DOSERIAL** directive applied in nested loops. In this case, if automatic parallelization is enabled the **I** or **K** loop may be parallelized.

```
  DO I=1, 100
!SMP$ DOSERIAL
  DO J=1, 100
    DO K=1, 100
      ARR(I,J,K)=I+J+K
    ENDDO
  ENDDO
ENDDO
```

Related Information

- “**DO / END DO**” on page 23
- “**PARALLEL DO / END PARALLEL DO**” on page 35
- **-qsmp** compiler option.

FLUSH

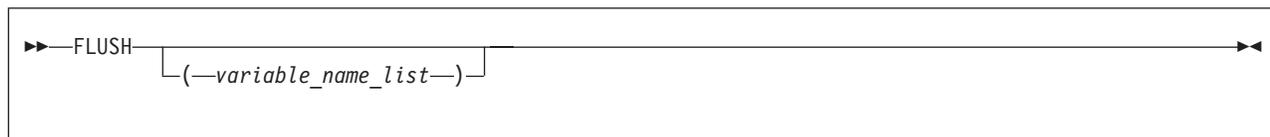
The **FLUSH** directive ensures that each thread has access to data generated by other threads. This directive is required because the compiler may keep values in processor registers if a program is optimized. The **FLUSH** directive ensures that the memory images that each thread views are consistent.

FLUSH

The **FLUSH** directive only takes effect if you specify the **-qsmp** compiler option.

You might be able to improve the performance of your program by using the **FLUSH** directive instead of the **VOLATILE** attribute. The **VOLATILE** attribute causes variables to be flushed after every update and before every use, while **FLUSH** causes variables to be written to or read from memory only when specified.

Syntax



Rules

You can specify this directive anywhere in your code; however, if you specify it outside of the dynamic extent of a parallel region, it is ignored.

If you specify a *variable_name_list*, only the variables in that list are written to or read from memory (assuming that they have not been written or read already). All variables in the *variable_name_list* must be at the current scope and must be thread visible. Thread visible variables can be any of the following:

- Globally visible variables (common blocks and module data)
- Local and host-associated variables with the **SAVE** attribute
- Local variables without the **SAVE** attribute that are specified in a **SHARED** clause in a parallel region within the subprogram
- Local variables without the **SAVE** attribute that have had their addresses taken
- All pointer dereferences
- Dummy arguments

If you do not specify a *variable_name_list*, all thread visible variables are written to or read from memory.

When a thread encounters the **FLUSH** directive, it writes into memory the modifications to the affected variables. The thread also reads the latest copies of the variables from memory if it has local copies of those variables: for example, if it has copies of the variables in registers.

It is not mandatory for all threads in a team to use the **FLUSH** directive. However, to guarantee that all thread visible variables are current, any thread that modifies a thread visible variable should use the **FLUSH** directive to update the value of that variable in memory. If you do not use **FLUSH** or one of the directives that implies **FLUSH** (see below), the value of the variable might not be the most recent one.

Note that **FLUSH** is not atomic. You must **FLUSH** shared variables that are controlled by a shared lock variable with one directive and then **FLUSH** the lock variable with another. This guarantees that the shared variables are written before the lock variable.

The following directives imply a **FLUSH** directive unless you specify a **NOWAIT** clause for those directives to which it applies:

- **BARRIER**

- CRITICAL/END CRITICAL
- DO/END DO
- END SECTIONS
- END SINGLE
- PARALLEL/END PARALLEL
- PARALLEL DO/END PARALLEL DO
- PARALLEL SECTIONS/END PARALLEL SECTIONS
- PARALLEL WORKSHARE/END PARALLEL WORKSHARE
- ORDERED/END ORDERED

Examples

Example 1: In the following example, two threads perform calculations in parallel and are synchronized when the calculations are complete:

```

PROGRAM P
  INTEGER INSYNC(0:1), IAM

!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(INSYNC)
  IAM = OMP_GET_THREAD_NUM()
  INSYNC(IAM) = 0
!$OMP BARRIER
  CALL WORK
!$OMP FLUSH(INSYNC)
  INSYNC(IAM) = 1                                ! Each thread sets a flag
                                                ! once it has
!$OMP FLUSH(INSYNC)
  DO WHILE (INSYNC(1-IAM) .eq. 0)              ! completed its work.
                                                ! One thread waits for
!$OMP FLUSH(INSYNC)                            ! another to complete
  END DO                                        ! its work.
!$OMP END PARALLEL

END PROGRAM P

SUBROUTINE WORK                                ! Each thread does indep-
                                                ! endent calculations.
!
! ...
!$OMP FLUSH                                    ! flush work variables
                                                ! before INSYNC
                                                ! is flushed.

END SUBROUTINE WORK

```

Example 2: The following example is not valid, because it attempts to use **FLUSH** with a variable that is not thread visible:

```

FUNCTION F()
  INTEGER, AUTOMATIC :: i
!$OMP FLUSH(I)
END FUNCTION F

```

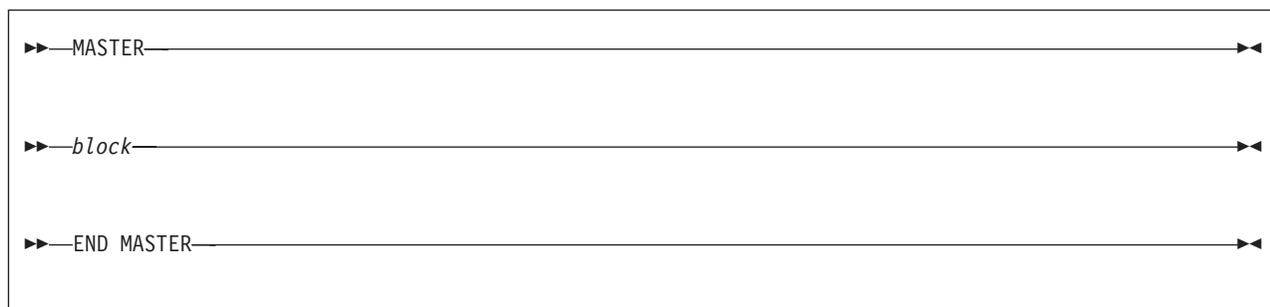
MASTER / END MASTER

The **MASTER** construct enables you to define a block of code that will be run by only the master thread of the team. It includes a **MASTER** directive that precedes a block of code and ends with an **END MASTER** directive.

The **MASTER** and **END MASTER** directives only take effect if you specify the **-qsmp** compiler option.

MASTER / END MASTER

Syntax



block represents the block of code that will be run by the master thread of the team.

Rules

It is illegal to branch into or out of a **MASTER** construct.

A **MASTER** directive binds to the closest dynamically enclosing **PARALLEL** directive, if one exists.

A **MASTER** directive cannot appear within the dynamic extent of a work-sharing construct or within the dynamic extent of the **PARALLEL DO**, **PARALLEL SECTIONS**, and **PARALLEL WORKSHARE** directives.

No implied barrier exists on entry to, or exit from, the **MASTER** construct.

Examples

Example 1: An example of the **MASTER** directive binding to the **PARALLEL** directive.

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP MASTER
    Y = 10.0
    X = 0.0
    DO I = 1, 4
        X = X + COS(Y) + I
    END DO
!$OMP END MASTER
!$OMP BARRIER
!$OMP DO PRIVATE(J)
    DO J = 1, 10000
        A(J) = X + SIN(J*2.5)
    END DO
!$OMP END DO
!$OMP END PARALLEL
END
```

Related Information

- “**PARALLEL DO / END PARALLEL DO**” on page 35
- “**PARALLEL SECTIONS / END PARALLEL SECTIONS**” on page 38
- **-qsmp** compiler option.

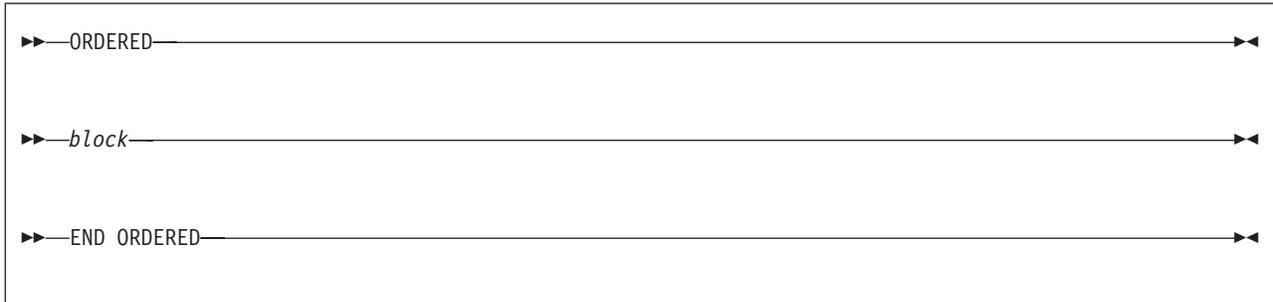
ORDERED / END ORDERED

The **ORDERED / END ORDERED** directives cause the iterations of a block of code within a parallel loop to be executed in the order that the loop would execute

in if it was run sequentially. You can force the code inside the **ORDERED** construct to run in a predictable order while code outside of the construct runs in parallel.

The **ORDERED** and **END ORDERED** directives only take effect if you specify the **-qsmp** compiler option.

Syntax



block represents the block of code that will be executed in sequence.

Rules

The **ORDERED** directive can only appear in the dynamic extent of a **DO** or **PARALLEL DO** directive. It is illegal to branch into or out of an **ORDERED** construct.

The **ORDERED** directive binds to the nearest dynamically enclosing **DO** or **PARALLEL DO** directive. You must specify the **ORDERED** clause on the **DO** or **PARALLEL DO** directive to which the **ORDERED** construct binds.

ORDERED constructs that bind to different **DO** directives are independent of each other.

Only one thread can execute an **ORDERED** construct at a time. Threads enter the **ORDERED** construct in the order of the loop iterations. A thread will enter the **ORDERED** construct if all of the previous iterations have either executed the construct or will never execute the construct.

Each iteration of a parallel loop with an **ORDERED** construct can only execute that **ORDERED** construct once. Each iteration of a parallel loop can execute at most one **ORDERED** directive. An **ORDERED** construct cannot appear within the dynamic extent of a **CRITICAL** construct.

Examples

Example 1: In this example, an **ORDERED** parallel loop counts down.

```
PROGRAM P
!$OMP PARALLEL DO ORDERED
DO I = 3, 1, -1
!$OMP ORDERED
PRINT *,I
!$OMP END ORDERED
END DO
END PROGRAM P
```

The expected output of this program is:

MASTER / END MASTER

3
2
1

Example 2: This example shows a program with two **ORDERED** constructs in a parallel loop. Each iteration can only execute a single section.

```
PROGRAM P
!$OMP PARALLEL DO ORDERED
DO I = 1, 3
  IF (MOD(I,2) == 0) THEN
!$OMP   ORDERED
        PRINT *, I*10
!$OMP   END ORDERED
  ELSE
!$OMP   ORDERED
        PRINT *, I
!$OMP   END ORDERED
  END IF
END DO
END PROGRAM P
```

The expected output of this program is:

1
20
3

Example 3: In this example, the program computes the sum of all elements of an array that are greater than a threshold. **ORDERED** is used to ensure that the results are always reproducible: roundoff will take place in the same order every time the program is executed, so the program will always produce the same results.

```
PROGRAM P
REAL :: A(1000)
REAL :: THRESHOLD = 999.9
REAL :: SUM = 0.0

!$OMP PARALLEL DO ORDERED
DO I = 1, 1000
  IF (A(I) > THRESHOLD) THEN
!$OMP   ORDERED
        SUM = SUM + A(I)
!$OMP   END ORDERED
  END IF
END DO
END PROGRAM P
```

Note: To avoid bottleneck situations when using the **ORDERED** clause, you can try using **DYNAMIC** scheduling or **STATIC** scheduling with a small chunk size. See “**SCHEDULE**” on page 42 for more information.

Related Information

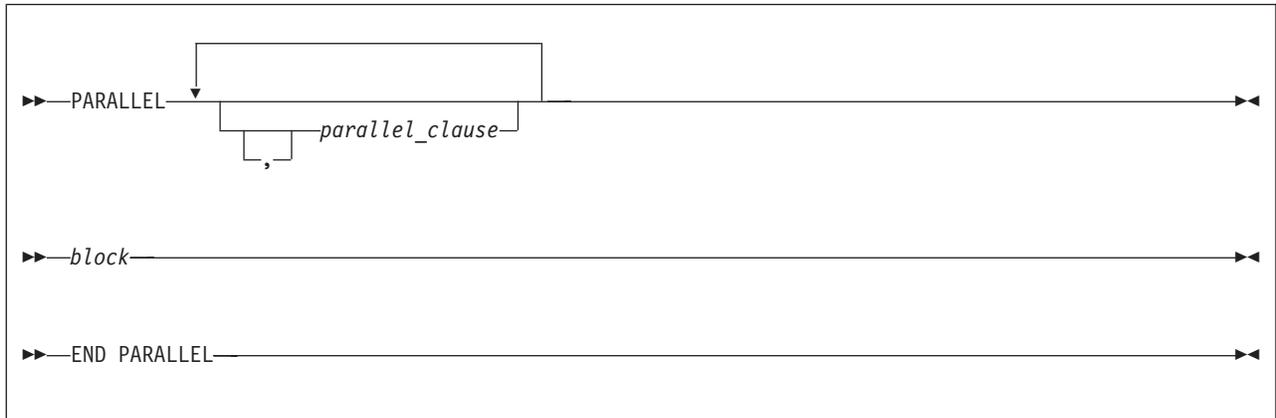
- “**PARALLEL DO / END PARALLEL DO**” on page 35
- “**DO / END DO**” on page 23
- “**CRITICAL / END CRITICAL**” on page 22
- “**SCHEDULE**” on page 42
- **-qsmp** compiler option.

PARALLEL / END PARALLEL

The **PARALLEL** construct enables you to define a block of code that can be executed by a team of threads concurrently. The **PARALLEL** construct includes a **PARALLEL** directive that is followed by one or more blocks of code, and ends with an **END PARALLEL** directive.

The **PARALLEL** and **END PARALLEL** directives only take effect if you specify the **-qsmp** compiler option.

Syntax



where *parallel_clause* is:



copyin_clause
See — “COPYIN” on page 63

default_clause
See — “DEFAULT” on page 64

if_clause
See — “IF” on page 66

firstprivate_clause
See — “FIRSTPRIVATE” on page 67.

num_threads_clause
See — “NUM_THREADS” on page 69.

private_clause
See — “PRIVATE” on page 70.

reduction_clause
See — “REDUCTION” on page 72

MASTER / END MASTER

shared_clause

See — “SHARED” on page 76

Rules

It is illegal to branch into or out of a **PARALLEL** construct.

The **IF** and **DEFAULT** clauses can appear at most once in a **PARALLEL** directive.

You should be careful when you perform input/output operations in a parallel region. If multiple threads execute a Fortran I/O statement on the same unit, you should make sure that the threads are synchronized. If you do not, the behavior is undefined. Also note that although in the XL Fortran implementation each thread has exclusive access to the I/O unit, the OpenMP specification does not require exclusive access.

Directives that bind to a parallel region will bind to that parallel region even if it is serialized.

The **END PARALLEL** directive implies the **FLUSH** directive.

Examples

Example 1: An example of an inner **PARALLEL** directive with the **PRIVATE** clause enclosing the **PARALLEL** construct. Note: The **SHARED** clause is present on the inner **PARALLEL** construct.

```
!$OMP PARALLEL PRIVATE(X)
!$OMP DO
    DO I = 1, 10
        X(I) = I
!$OMP PARALLEL SHARED (X,Y)
!$OMP DO
    DO K = 1, 10
        Y(K,I) = K * X(I)
    END DO
!$OMP END DO
!$OMP END PARALLEL
    END DO
!$OMP END DO
!$OMP END PARALLEL
```

Example 2: An example showing that a variable cannot appear in both a **PRIVATE**, and **SHARED** clause.

```
!$OMP PARALLEL PRIVATE(A), SHARED(A)
!$OMP DO
    DO I = 1, 1000
        A(I) = I * I
    END DO
!$OMP END DO
!$OMP END PARALLEL
```

Example 3: This example demonstrates the use of the **COPYIN** clause. Each thread created by the **PARALLEL** directive has its own copy of the common block **BLOCK**. The **COPYIN** clause causes the initial value of *FCTR* to be copied into the threads that execute iterations of the **DO** loop.

```
PROGRAM TT
COMMON /BLOCK/ FCTR
INTEGER :: I, FCTR
!$OMP THREADPRIVATE(/BLOCK/)
INTEGER :: A(100)

FCTR = -1
```

```

A = 0

!$OMP PARALLEL COPYIN(FCTR)
!$OMP DO
  DO I=1, 100
    FCTR = FCTR + I
    CALL SUB(A(I), I)
  ENDDO
!$OMP END PARALLEL

PRINT *, A
END PROGRAM

SUBROUTINE SUB(AA, J)
  INTEGER :: FCTR, AA, J
  COMMON /BLOCK/ FCTR
!$OMP THREADPRIVATE(/BLOCK/) ! EACH THREAD GETS ITS OWN COPY
                               ! OF BLOCK.

  AA = FCTR
  FCTR = FCTR - J
END SUBROUTINE SUB

```

The expected output is:

```
0 1 2 3 ... 96 97 98 99
```

Related Information

- “FLUSH” on page 27
- “PARALLEL DO / END PARALLEL DO”
- “THREADPRIVATE” on page 54
- “DO / END DO” on page 23
- `-qsmp` compiler option.

PARALLEL DO / END PARALLEL DO

The **PARALLEL DO** directive enables you to specify which loops the compiler should parallelize. This is semantically equivalent to:

```

!$OMP PARALLEL
!$OMP DO
  ...
!$OMP ENDDO
!$OMP END PARALLEL

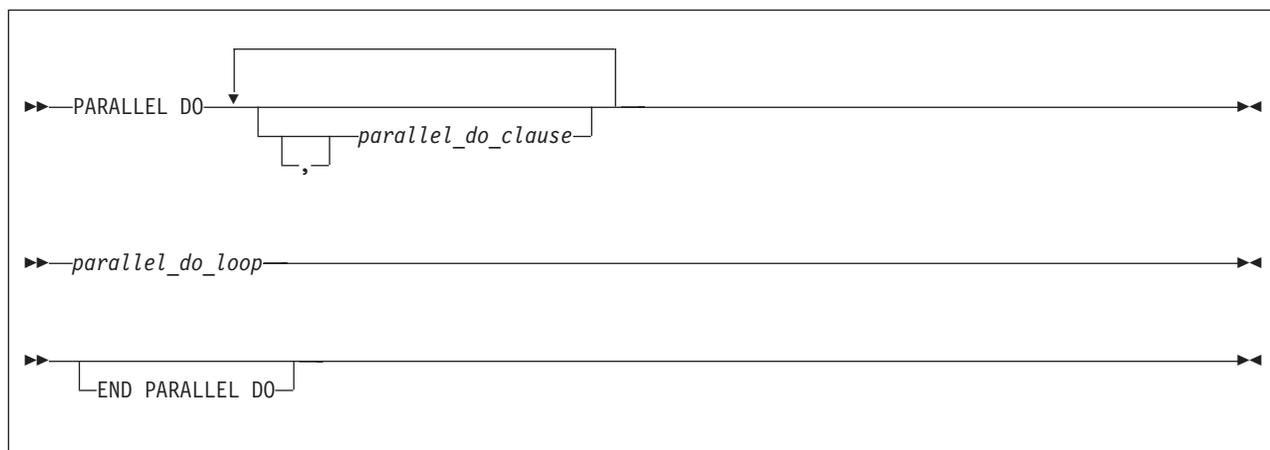
```

and is a convenient way of parallelizing loops. The **END PARALLEL DO** directive allows you to indicate the end of a **DO** loop that is specified by the **PARALLEL DO** directive.

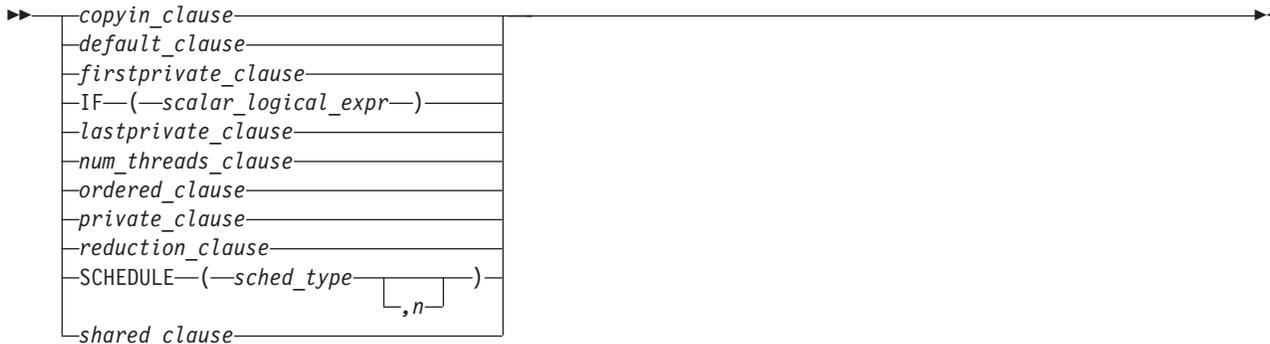
The **PARALLEL DO** and **END PARALLEL DO** directives only take effect if you specify the `-qsmp` compiler option.

PARALLEL DO / END PARALLEL DO

Syntax



where *parallel_do_clause* is:



copyin_clause

See — “COPYIN” on page 63

default_clause

See — “DEFAULT” on page 64

if_clause

See — “IF” on page 66.

firstprivate_clause

See — “FIRSTPRIVATE” on page 67.

lastprivate_clause

See — “LASTPRIVATE” on page 68.

num_threads_clause

See — “NUM_THREADS” on page 69.

ordered_clause

See — “ORDERED” on page 70

private_clause

See — “PRIVATE” on page 70

reduction_clause

See — “REDUCTION” on page 72

schedule_clause

See — “SCHEDULE” on page 74

shared_clause

See — “SHARED” on page 76

Rules

The first noncomment line (not including other directives) that is following the **PARALLEL DO** directive must be a **DO** loop. This line cannot be an infinite **DO** or **DO WHILE** loop. The **PARALLEL DO** directive applies only to the **DO** loop that is immediately following the directive, and not to any nested **DO** loops.

If you specify a **DO** loop by a **PARALLEL DO** directive, the **END PARALLEL DO** directive is optional. If you use the **END PARALLEL DO** directive, it must immediately follow the end of the **DO** loop.

You may have a **DO** construct that contains several **DO** statements. If the **DO** statements share the same **DO** termination statement, and an **END PARALLEL DO** directive follows the construct, you can only specify a **PARALLEL DO** directive for the outermost **DO** statement of the construct.

You must not follow the **PARALLEL DO** directive by a **DO** (work-sharing) or **DO SERIAL** directive. You can specify only one **PARALLEL DO** directive for a given **DO** loop.

All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

The **PARALLEL DO** directive must not appear with the **INDEPENDENT** directive for a given **DO** loop.

Note: The **INDEPENDENT** directive allows you to keep your code common with **HPF** implementations. Use the **PARALLEL DO** directive for maximum portability across multiple vendors. The **PARALLEL DO** directive is a prescriptive directive, while the **INDEPENDENT** directive is an assertion about the characteristics of the loop.

The **IF** clause may appear at most once in a **PARALLEL DO** directive.

An **IF** expression is evaluated outside of the context of the parallel construct. Any function reference in the **IF** expression must not have side effects.

By default, a nested parallel loop is serialized, regardless of the setting of the **IF** clause. You can change this default by using the **-qsmp=nested_par** compiler option.

If the **REDUCTION** variable of an inner **DO** loop appears in the **PRIVATE** or **LASTPRIVATE** clause of an enclosing **DO** loop or **PARALLEL SECTIONS** construct, the variable must be initialized before the inner **DO** loop.

A variable that appears in the **REDUCTION** clause of an **INDEPENDENT** directive of an enclosing **DO** loop must not also appear in the *data_scope_entity_list* of the **PRIVATE** or **LASTPRIVATE** clause.

You should be careful when you perform input/output operations in a parallel region. If multiple threads execute a Fortran I/O statement on the same unit, you should make sure that the threads are synchronized. If you do not, the behavior is

PARALLEL DO / END PARALLEL DO

undefined. Also note that although in the XL Fortran implementation each thread has exclusive access to the I/O unit, the OpenMP specification does not require exclusive access.

Directives that bind to a parallel region will bind to that parallel region even if it is serialized.

Examples

Example 1: A valid example with the **LASTPRIVATE** clause.

```
!$OMP PARALLEL DO PRIVATE(I), LASTPRIVATE (X)
  DO I = 1,10
    X = I * I
    A(I) = X * B(I)
  END DO
PRINT *, X                                ! X has the value 100
```

Example 2: A valid example with the **REDUCTION** clause.

```
!$OMP PARALLEL DO PRIVATE(I), REDUCTION(+:MYSUM)
  DO I = 1, 10
    MYSUM = MYSUM + IARR(I)
  END DO
```

Example 3: A valid example where more than one thread accesses a variable that is marked as **SHARED**, but the variable is used only in a **CRITICAL** construct.

```
!$OMP PARALLEL DO SHARED (X)
  DO I = 1, 10
    A(I) = A(I) * I
!$OMP CRITICAL
    X = X + A(I)
!$OMP END CRITICAL
  END DO
```

Example 4: A valid example of the **END PARALLEL DO** directive.

```
REAL A(100), B(2:100), C(100)
!$OMP PARALLEL DO
  DO I = 2, 100
    B(I) = (A(I) + A(I-1))/2.0
  END DO
!$OMP END PARALLEL DO
!$OMP PARALLEL DO
  DO J = 1, 100
    C(J) = X + COS(J*5.5)
  END DO
!$OMP END PARALLEL DO
END
```

Related Information

- “DO / END DO” on page 23
- “ORDERED / END ORDERED” on page 30
- “PARALLEL / END PARALLEL” on page 33
- “PARALLEL SECTIONS / END PARALLEL SECTIONS”
- “SCHEDULE” on page 42
- “THREADPRIVATE” on page 54
- **-qsmp** compiler option.

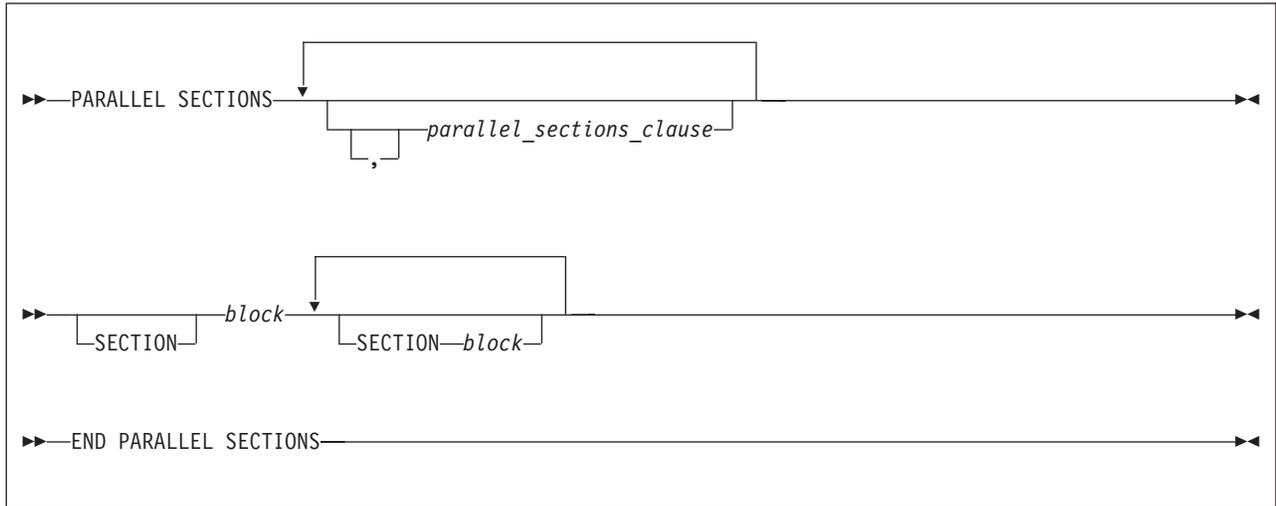
PARALLEL SECTIONS / END PARALLEL SECTIONS

The **PARALLEL SECTIONS** construct enables you to define independent blocks of code that the compiler can execute concurrently. The **PARALLEL SECTIONS**

construct includes a **PARALLEL SECTIONS** directive followed by one or more blocks of code delimited by the **SECTION** directive, and ends with an **END PARALLEL SECTIONS** directive.

The **PARALLEL SECTIONS**, **SECTION** and **END PARALLEL SECTIONS** directives only take effect if you specify the **-qsmp** compiler option.

Syntax



where *parallel_sections_clause* is:



copyin_clause

See — “COPYIN” on page 63

default_clause

See — “DEFAULT” on page 64

firstprivate_clause

See — “FIRSTPRIVATE” on page 67.

if_clause

See — “IF” on page 66

lastprivate_clause

See — “LASTPRIVATE” on page 68.

num_threads_clause

See — “NUM_THREADS” on page 69.

private_clause

See — “PRIVATE” on page 70.

PARALLEL DO / END PARALLEL DO

reduction_clause

See — “REDUCTION” on page 72

shared_clause

See — “SHARED” on page 76

Rules

The **PARALLEL SECTIONS** construct includes the delimiting directives, and the blocks of code they enclose. The rules below also refer to *sections*. You define a section as the block of code within the delimiting directives.

The **SECTION** directive marks the beginning of a block of code. At least one **SECTION** and its block of code must appear within the **PARALLEL SECTIONS** construct. Note, however, that you do not have to specify the **SECTION** directive for the first section. The end of a block is delimited by either another **SECTION** directive or by the **END PARALLEL SECTIONS** directive.

You can use the **PARALLEL SECTIONS** construct to specify parallel execution of the identified sections of code. There is no assumption as to the order in which sections are executed. Each section must not interfere with any other section in the construct unless the interference occurs within a **CRITICAL** construct.

It is illegal to branch into or out of any block of code that is defined by the **PARALLEL SECTIONS** construct.

The compiler determines how to divide the work among the threads based on a number of factors, such as the number of threads and the number of sections to be executed in parallel. Therefore, a single thread may execute more than one **SECTION**, or a thread may not execute any **SECTION**.

All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

Within a **PARALLEL SECTIONS** construct, variables that are not appearing in the **PRIVATE** clause are assumed to be **SHARED** by default.

In a **PARALLEL SECTIONS** construct, a variable that appears in the **REDUCTION** clause of an **INDEPENDENT** directive or the **PARALLEL DO** directive of an enclosing **DO** loop must not also appear in the *data_scope_entity_list* of the **PRIVATE** clause.

If the **REDUCTION** variable of the inner **PARALLEL SECTIONS** construct appears in the **PRIVATE** clause of an enclosing **DO** loop or **PARALLEL SECTIONS** construct, the variable must be initialized before the inner **PARALLEL SECTIONS** construct.

The **PARALLEL SECTIONS** construct must not appear within a **CRITICAL** construct.

You should be careful when you perform input/output operations in a parallel region. If multiple threads execute a Fortran I/O statement on the same unit, you should make sure that the threads are synchronized. If you do not, the behavior is undefined. Also note that although in the XL Fortran implementation each thread has exclusive access to the I/O unit, the OpenMP specification does not require exclusive access.

Directives that bind to a parallel region will bind to that parallel region even if it is serialized.

The **END PARALLEL SECTIONS** directive implies the **FLUSH** directive.

Examples

Example 1:

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
    DO I = 1, 10
        C(I) = MAX(A(I),A(I+1))
    END DO
!$OMP SECTION
    W = U + V
    Z = X + Y
!$OMP END PARALLEL SECTIONS
```

Example 2: In this example, the index variable I is declared as **PRIVATE**. Note also that the first optional **SECTION** directive has been omitted.

```
!$OMP PARALLEL SECTIONS PRIVATE(I)
    DO I = 1, 100
        A(I) = A(I) * I
    END DO
!$OMP SECTION
    CALL NORMALIZE (B)
    DO I = 1, 100
        B(I) = B(I) + 1.0
    END DO
!$OMP SECTION
    DO I = 1, 100
        C(I) = C(I) * C(I)
    END DO
!$OMP END PARALLEL SECTIONS
```

Example 3: This example is invalid because there is a data dependency for the variable C across sections.

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
    DO I = 1, 10
        C(I) = C(I) * I
    END DO
!$OMP SECTION
    DO K = 1, 10
        D(K) = C(K) + K
    END DO
!$OMP END PARALLEL SECTIONS
```

Related Information

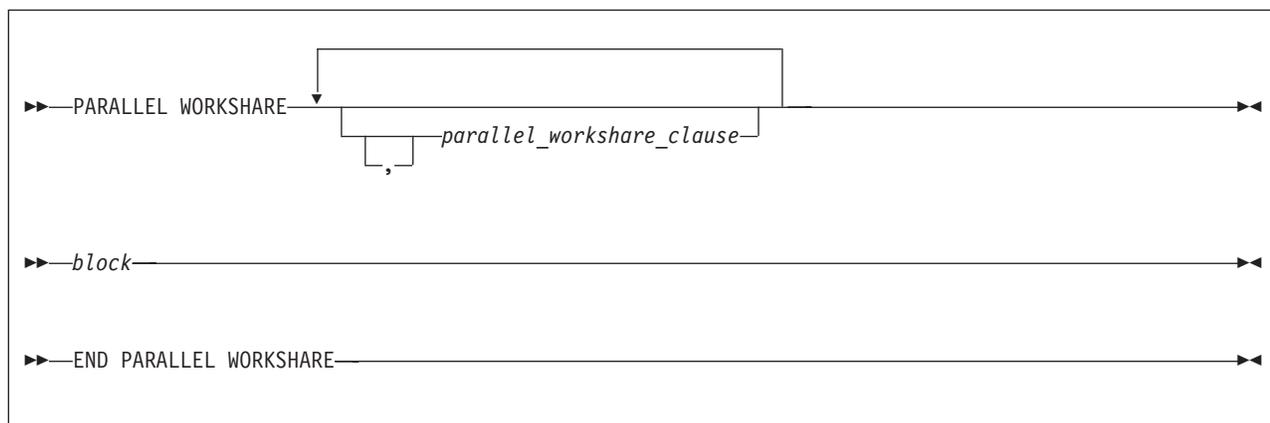
- “PARALLEL / END PARALLEL” on page 33
- “PARALLEL DO / END PARALLEL DO” on page 35
- “THREADPRIVATE” on page 54
- **-qsmp** compiler option.

PARALLEL WORKSHARE / END PARALLEL WORKSHARE

The **PARALLEL WORKSHARE** construct provides a short form method for including a **WORKSHARE** directive inside a **PARALLEL** construct.

PARALLEL DO / END PARALLEL DO

Syntax



where *parallel_workshare_clause* is any of the directives accepted by either the **PARALLEL** or **WORKSHARE** directives.

Related Information

- “PARALLEL / END PARALLEL” on page 33
- “WORKSHARE” on page 58

SCHEDULE

The **SCHEDULE** directive allows the user to specify the chunking method for parallelization. Work is assigned to threads in different manners depending on the scheduling type or chunk size used.

The **SCHEDULE** directive only takes effect if you specify the **-qsmp** Option compiler option.

Syntax



n *n* must be a positive, specification expression. You must not specify *n* for the *sched_type* **RUNTIME**.

sched_type
is **AFFINITY**, **DYNAMIC**, **GUIDED**, **RUNTIME**, or **STATIC**

For more information on *sched_type* parameters, see the **SCHEDULE** clause.

number_of_iterations
is the number of iterations in the loop to be parallelized.

number_of_threads
is the number of threads used by the program.

Rules

The **SCHEDULE** directive must appear in the specification part of a scoping unit.

Only one **SCHEDULE** directive may appear in the specification part of a scoping unit.

The **SCHEDULE** directive applies to one of the following:

- All loops in the scoping unit that do not already have explicit scheduling types specified. Individual loops can have scheduling types specified using the **SCHEDULE** clause of the **PARALLEL DO** directive.
- Loops that the compiler generates and have been chosen to be parallelized by automatic parallelization. For example, the **SCHEDULE** directive applies to loops generated for **FORALL**, **WHERE**, I/O implied-**DO**, and array constructor implied-**DO**.

Any dummy arguments appearing or referenced in the specification expression for the chunk size n must also appear in the **SUBROUTINE** or **FUNCTION** statement and in all **ENTRY** statements appearing in the given subprogram.

If the specified chunk size n is greater than the number of iterations, the loop will not be parallelized and will execute on a single thread.

If you specify more than one method of determining the chunking algorithm, the compiler will follow, in order of precedence:

1. **SCHEDULE** clause to the **PARALLEL DO** directive.
2. **SCHEDULE** directive
3. **schedule** suboption to the **-qsmp** compiler option.
4. **XLSMPOPTS** run-time option.
5. run-time default (that is, **STATIC**)

Examples

Example 1. Given the following information:

```
number of iterations = 1000
number of threads = 4
```

and using the **GUIDED** scheduling type, the chunk sizes would be as follows:

```
250 188 141 106 79 59 45 33 25 19 14 11 8 6 4 3 3 2 1 1 1 1
```

The iterations would then be divided into the following chunks:

```
chunk 1 = iterations 1 to 250
chunk 2 = iterations 251 to 438
chunk 3 = iterations 439 to 579
chunk 4 = iterations 580 to 685
chunk 5 = iterations 686 to 764
chunk 6 = iterations 765 to 823
chunk 7 = iterations 824 to 868
chunk 8 = iterations 869 to 901
chunk 9 = iterations 902 to 926
chunk 10 = iterations 927 to 945
chunk 11 = iterations 946 to 959
chunk 12 = iterations 960 to 970
chunk 13 = iterations 971 to 978
chunk 14 = iterations 979 to 984
chunk 15 = iterations 985 to 988
chunk 16 = iterations 989 to 991
chunk 17 = iterations 992 to 994
chunk 18 = iterations 995 to 996
chunk 19 = iterations 997 to 997
chunk 20 = iterations 998 to 998
chunk 21 = iterations 999 to 999
chunk 22 = iterations 1000 to 1000
```

A possible scenario for the division of work could be:

PARALLEL DO / END PARALLEL DO

```
thread 1 executes chunks 1 5 10 13 18 20
thread 2 executes chunks 2 7 9 14 16 22
thread 3 executes chunks 3 6 12 15 19
thread 4 executes chunks 4 8 11 17 21
```

Example 2. Given the following information:

```
number of iterations = 100
number of threads = 4
```

and using the **AFFINITY** scheduling type, the iterations would be divided into the following partitions:

```
partition 1 = iterations 1 to 25
partition 2 = iterations 26 to 50
partition 3 = iterations 51 to 75
partition 4 = iterations 76 to 100
```

The partitions would be divided into the following chunks:

```
chunk 1a = iterations 1 to 13
chunk 1b = iterations 14 to 19
chunk 1c = iterations 20 to 22
chunk 1d = iterations 23 to 24
chunk 1e = iterations 25 to 25
```

```
chunk 2a = iterations 26 to 38
chunk 2b = iterations 39 to 44
chunk 2c = iterations 45 to 47
chunk 2d = iterations 48 to 49
chunk 2e = iterations 50 to 50
```

```
chunk 3a = iterations 51 to 63
chunk 3b = iterations 64 to 69
chunk 3c = iterations 70 to 72
chunk 3d = iterations 73 to 74
chunk 3e = iterations 75 to 75
```

```
chunk 4a = iterations 76 to 88
chunk 4b = iterations 89 to 94
chunk 4c = iterations 95 to 97
chunk 4d = iterations 98 to 99
chunk 4e = iterations 100 to 100
```

A possible scenario for the division of work could be:

```
thread 1 executes chunks 1a 1b 1c 1d 1e 4d
thread 2 executes chunks 2a 2b 2c 2d
thread 3 executes chunks 3a 3b 3c 3d 3e 2e
thread 4 executes chunks 4a 4b 4c 4e
```

In this scenario, thread 1 finished executing all the chunks in its partition and then grabbed an available chunk from the partition of thread 4. Similarly, thread 3 finished executing all the chunks in its partition and then grabbed an available chunk from the partition of thread 2.

Example 3. Given the following information:

```
number of iterations = 1000
number of threads = 4
```

and using the **DYNAMIC** scheduling type and chunk size of 100, the chunk sizes would be as follows:

```
100 100 100 100 100 100 100 100 100 100
```

The iterations would be divided into the following chunks:

PARALLEL DO / END PARALLEL DO

```
chunk 1 = iterations 1 to 100
chunk 2 = iterations 101 to 200
chunk 3 = iterations 201 to 300
chunk 4 = iterations 301 to 400
chunk 5 = iterations 401 to 500
chunk 6 = iterations 501 to 600
chunk 7 = iterations 601 to 700
chunk 8 = iterations 701 to 800
chunk 9 = iterations 801 to 900
chunk 10 = iterations 901 to 1000
```

A possible scenario for the division of work could be:

```
thread 1 executes chunks 1 5 9
thread 2 executes chunks 2 8
thread 3 executes chunks 3 6 10
thread 4 executes chunks 4 7
```

Example 4. Given the following information:

```
number of iterations = 100
number of threads = 4
```

and using the **STATIC** scheduling type, the iterations would be divided into the following chunks:

```
chunk 1 = iterations 1 to 25
chunk 2 = iterations 26 to 50
chunk 3 = iterations 51 to 75
chunk 4 = iterations 76 to 100
```

A possible scenario for the division of work could be:

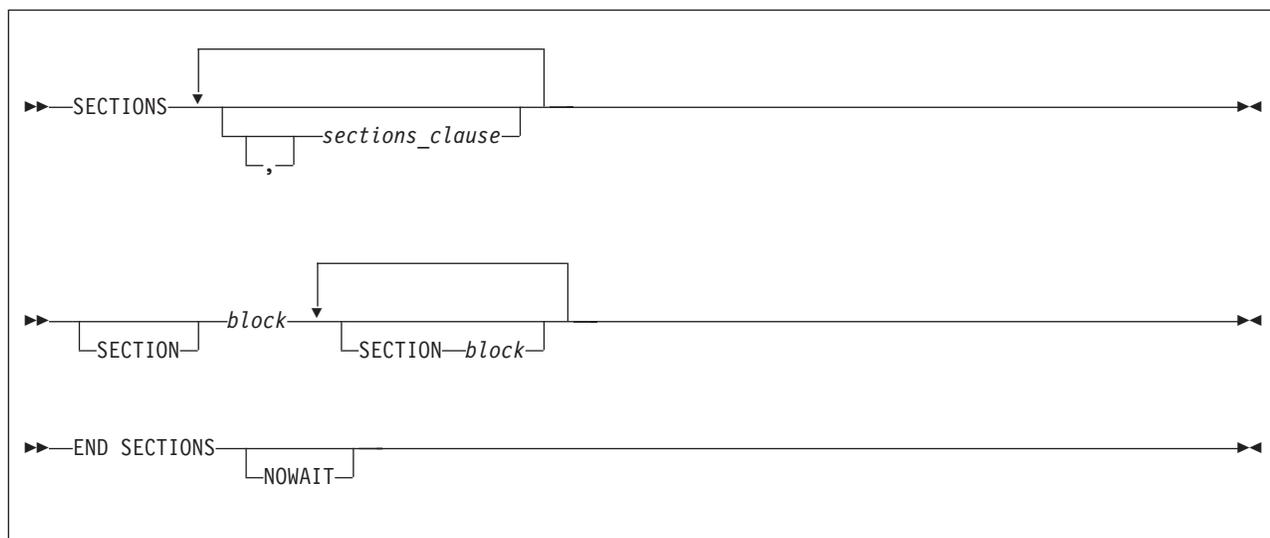
```
thread 1 executes chunks 1
thread 2 executes chunks 2
thread 3 executes chunks 3
thread 4 executes chunks 4
```

SECTIONS / END SECTIONS

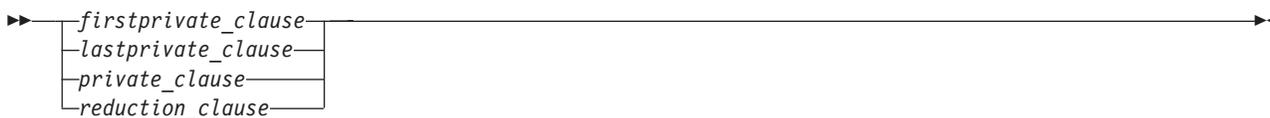
The **SECTIONS** construct defines distinct blocks of code to be executed in parallel by threads in the team.

The **SECTIONS** and **END SECTIONS** directives only take effect if you specify the **-qsmp** compiler option.

Syntax



where *sections_clause* is:



firstprivate_clause

See — “FIRSTPRIVATE” on page 67.

lastprivate_clause

See — “LASTPRIVATE” on page 68.

private_clause

See — “PRIVATE” on page 70.

reduction_clause

See — “REDUCTION” on page 72

Rules

The **SECTIONS** construct must be encountered by all threads in a team or by none of the threads in a team. All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

The **SECTIONS** construct includes the delimiting directives, and the blocks of code they enclose. At least one block of code must appear in the construct.

You must specify the **SECTION** directive at the beginning of each block of code except for the first. The end of a block is delimited by either another **SECTION** directive or by the **END SECTIONS** directive.

It is illegal to branch into or out of any block of code that is enclosed in the **SECTIONS** construct. All **SECTION** directives must appear within the lexical extent of the **SECTIONS/END SECTIONS** directive pair.

The compiler determines how to divide the work among the threads based on a number of factors, such as the number of threads in the team and the number of

sections to be executed in parallel. Therefore, a single thread might execute more than one **SECTION**. It is also possible that a thread in the team might not execute any **SECTION**.

In order for the directive to execute in parallel, you must place the **SECTIONS/END SECTIONS** pair within the dynamic extent of a parallel region. Otherwise, the blocks will be executed serially.

If you specify **NOWAIT** on the **SECTIONS** directive, a thread that completes its sections early will proceed to the instructions following the **SECTIONS** construct. If you do not specify the **NOWAIT** clause, each thread will wait for all of the other threads in the same team to reach the **END SECTIONS** directive. However, there is no implied **BARRIER** at the start of the **SECTIONS** construct.

You cannot specify a **SECTIONS** directive within the dynamic extent of a **CRITICAL** or **MASTER** directive.

You cannot nest **SECTIONS**, **DO** or **SINGLE** directives that bind to the same **PARALLEL** directive.

BARRIER and **MASTER** directives are not permitted in the dynamic extent of a **SECTIONS** directive.

The **END SECTIONS** directive implies the **FLUSH** directive.

Examples

Example 1: This example shows a valid use of the **SECTIONS** construct within a **PARALLEL** region.

```

        INTEGER :: I, B(500), S, SUM
! ...
        S = 0
        SUM = 0
!$OMP PARALLEL SHARED(SUM), FIRSTPRIVATE(S)
!$OMP SECTIONS REDUCTION(+: SUM), LASTPRIVATE(I)
!$OMP SECTION
        S = FCT1(B(1::2)) ! Array B is not altered in FCT1.
        SUM = SUM + S
! ...
!$OMP SECTION
        S = FCT2(B(2::2)) ! Array B is not altered in FCT2.
        SUM = SUM + S
! ...
!$OMP SECTION
        DO I = 1, 500      ! The local copy of S is initialized
            S = S + B(I)  ! to zero.
        END DO
        SUM = SUM + S
! ...
!$OMP END SECTIONS
! ...
!$OMP DO REDUCTION(-: SUM)
        DO J=I-1, 1, -1  ! The loop starts at 500 -- the last
                        ! value from the previous loop.
            SUM = SUM - B(J)
        END DO
!$OMP MASTER
        SUM = SUM - FCT1(B(1::2)) - FCT2(B(2::2))
!$OMP END MASTER
!$OMP END PARALLEL

```

PARALLEL DO / END PARALLEL DO

```
! ...
                                ! Upon termination of the PARALLEL
                                ! region, the value of SUM remains zero.
```

Example 2: This example shows a valid use of nested **SECTIONS**.

```
!$OMP PARALLEL
!$OMP MASTER
    CALL RANDOM_NUMBER(CX)
    CALL RANDOM_NUMBER(CY)
    CALL RANDOM_NUMBER(CZ)
!$OMP END MASTER

!$OMP SECTIONS
!$OMP SECTION
!$OMP PARALLEL
!$OMP SECTIONS PRIVATE(I)
!$OMP SECTION
    DO I=1, 5000
        X(I) = X(I) + CX
    END DO
!$OMP SECTION
    DO I=1, 5000
        Y(I) = Y(I) + CY
    END DO
!$OMP END SECTIONS
!$OMP END PARALLEL

!$OMP SECTION
!$OMP PARALLEL SHARED(CZ,Z)
!$OMP DO
    DO I=1, 5000
        Z(I) = Z(I) + CZ
    END DO
!$OMP END DO
!$OMP END PARALLEL
!$OMP END SECTIONS NOWAIT
```

```
! The following computations do not
! depend on the results from the
! previous section.
```

```
!$OMP DO
    DO I=1, 5000
        T(I) = T(I) * CT
    END DO
!$OMP END DO
!$OMP END PARALLEL
```

Related Information

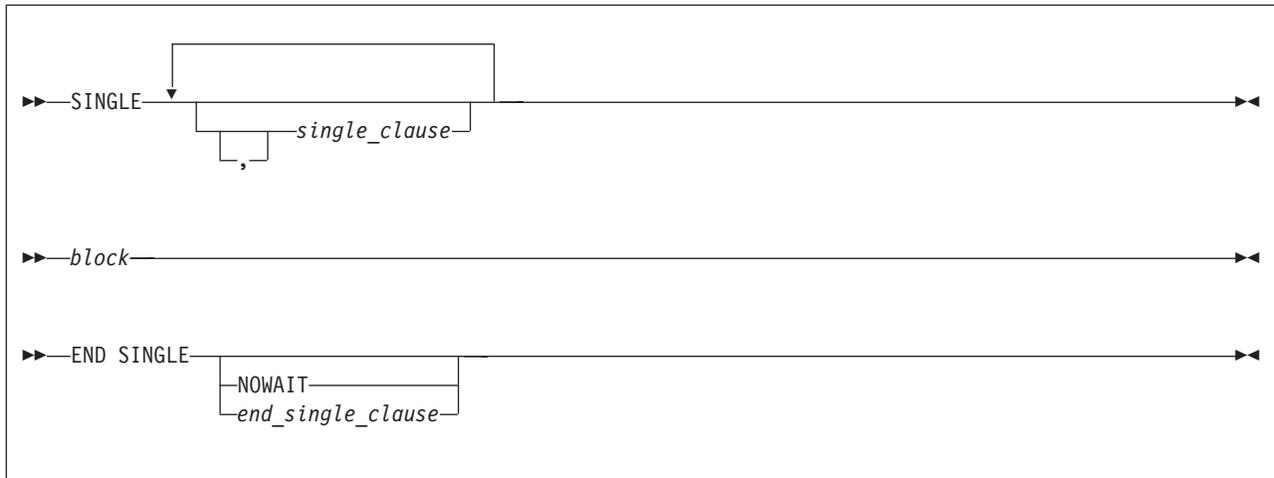
- “PARALLEL / END PARALLEL” on page 33
- “BARRIER” on page 21
- “PARALLEL DO / END PARALLEL DO” on page 35
- “THREADPRIVATE” on page 54
- **-qsmp** option.

SINGLE / END SINGLE

You can use the **SINGLE / END SINGLE** directive construct to specify that the enclosed code should only be executed by one thread in the team.

The **SINGLE** directive only takes effect if you specify the **-qsmp** compiler option.

Syntax



where *single_clause* is:



private_clause

See — “PRIVATE” on page 70.

firstprivate_clause

See — “FIRSTPRIVATE” on page 67.

where *end_single_clause* is:



NOWAIT

copyprivate_clause

Rules

It is illegal to branch into or out of a block that is enclosed within the **SINGLE** construct.

The **SINGLE** construct must be encountered by all threads in a team or by none of the threads in a team. All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

If you specify **NOWAIT** on the **END SINGLE** directive, the threads that are not executing the **SINGLE** construct will proceed to the instructions following the **SINGLE** construct. If you do not specify the **NOWAIT** clause, each thread will wait at the **END SINGLE** directive until the thread executing the construct reaches the **END SINGLE** directive. You may not specify **NOWAIT** and **COPYPRIVATE** as part of the same **END SINGLE** directive.

PARALLEL DO / END PARALLEL DO

There is no implied **BARRIER** at the start of the **SINGLE** construct. If you do not specify the **NOWAIT** clause, the **BARRIER** directive is implied at the **END SINGLE** directive.

You cannot nest **SECTIONS**, **DO** and **SINGLE** directives inside one another if they bind to the same **PARALLEL** directive.

SINGLE directives are not permitted within the dynamic extent of **CRITICAL** and **MASTER** directives. **BARRIER** and **MASTER** directives are not permitted within the dynamic extent of **SINGLE** directives.

If you have specified a variable as **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE** or **REDUCTION** in the **PARALLEL** construct which encloses your **SINGLE** construct, you cannot specify the same variable in the **PRIVATE** or **FIRSTPRIVATE** clause of the **SINGLE** construct.

The **SINGLE** directive binds to the closest dynamically enclosing **PARALLEL** directive, if one exists.

Examples

Example 1: In this example, the **BARRIER** directive is used to ensure that all threads finish their work before entering the **SINGLE** construct.

```
      REAL :: X(100), Y(50)
!
! ...
!$OMP PARALLEL DEFAULT(SHARED)
      CALL WORK(X)

!$OMP BARRIER
!$OMP SINGLE
      CALL OUTPUT(X)
      CALL INPUT(Y)
!$OMP END SINGLE

      CALL WORK(Y)
!$OMP END PARALLEL
```

Example 2: In this example, the **SINGLE** construct ensures that only one thread is executing a block of code. In this case, array *B* is initialized in the **DO** (work-sharing) construct. After the initialization, a single thread is employed to perform the summation.

```
      INTEGER :: I, J
      REAL :: B(500,500), SM
!
! ...

      J = ...
      SM = 0.0
!$OMP PARALLEL
!$OMP DO PRIVATE(I)
      DO I=1, 500
          CALL INITARR(B(I,:), I)      ! initialize the array B
      ENDDO
!$OMP END DO

!$OMP SINGLE                          ! employ only one thread
      DO I=1, 500
          SM = SM + SUM(B(J:J+1,I))
      ENDDO
!$OMP END SINGLE

!$OMP DO PRIVATE(I)
```

```

DO I=500, 1, -1
  CALL INITARR(B(I,:), 501-I)  ! re-initialize the array B
ENDDO
!$OMP END PARALLEL

```

Example 3: This example shows a valid use of the **PRIVATE** clause. Array *X* is **PRIVATE** to the **SINGLE** construct. If you were to reference array *X* immediately following the construct, it would be undefined.

```

REAL :: X(2000), A(1000), B(1000)

!$OMP PARALLEL
!   ...
!$OMP SINGLE PRIVATE(X)
  CALL READ_IN_DATA(X)
  A = X(1::2)
  B = X(2::2)
!$OMP END SINGLE
!   ...
!$OMP END PARALLEL

```

Example 4: In this example, the **LASTPRIVATE** variable *I* is used in allocating *TMP*, the **PRIVATE** variable in the **SINGLE** construct.

```

SUBROUTINE ADD(A, UPPERBOUND)
  INTEGER :: A(UPPERBOUND), I, UPPERBOUND
  INTEGER, ALLOCATABLE :: TMP(:)
!   ...
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(I)
  DO I=1, UPPERBOUND
    A(I) = I + 1
  ENDDO
!$OMP END DO

!$OMP SINGLE FIRSTPRIVATE(I), PRIVATE(TMP)
  ALLOCATE(TMP(0:I-1))
  TMP = (/ A(J),J=I,1,-1) /)
!   ...
  DEALLOCATE(TMP)
!$OMP END SINGLE
!$OMP END PARALLEL
!   ...
END SUBROUTINE ADD

```

Example 5: In this example, a value for the variable *I* is entered by the user. This value is then copied into the corresponding variable *I* for all other threads in the team using a **COPYPRIVATE** clause on an **END SINGLE** directive.

```

INTEGER I
!$OMP PARALLEL PRIVATE (I)
!   ...
!$OMP SINGLE
  READ (*, *) I
!$OMP END SINGLE COPYPRIVATE (I)  ! In all threads in the team, I
!                                   ! is equal to the value
!   ...                               ! that you entered.
!$OMP END PARALLEL

```

Example 6: In this example, variable *J* with a **POINTER** attribute is specified in a **COPYPRIVATE** clause on an **END SINGLE** directive. The value of *J*, not the value of the object that it points to, is copied into the corresponding variable *J* for all other threads in the team. The object itself is shared among all the threads in the team.

PARALLEL DO / END PARALLEL DO

```
        INTEGER, POINTER :: J
!$OMP PARALLEL PRIVATE (J)
! ...
!$OMP SINGLE
        ALLOCATE (J)
        READ (*, *) J
!$OMP END SINGLE COPYPRIVATE (J)
!$OMP ATOMIC
        J = J + OMP_GET_THREAD_NUM()
!$OMP BARRIER
!$OMP SINGLE
        WRITE (*, *) 'J = ', J    ! The result is the sum of all values added to
                                ! J. This result shows that the pointer object
                                ! is shared by all threads in the team.

        DEALLOCATE (J)
!$OMP END SINGLE
!$OMP END PARALLEL
```

Related Information

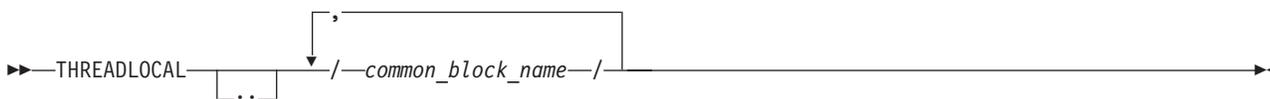
- “BARRIER” on page 21
- “CRITICAL / END CRITICAL” on page 22
- “FLUSH” on page 27
- “MASTER / END MASTER” on page 29
- “PARALLEL / END PARALLEL” on page 33

THREADLOCAL

You can use the **THREADLOCAL** directive to declare thread-specific common data. It is a possible method of ensuring that access to data that is contained within **COMMON** blocks is serialized.

In order to make use of this directive it is not necessary to specify **-qsmp** compiler option., but the invocation command must be **xlfr_r**, **xlfr90_r**, or **xlfr95_r** to link the necessary libraries.

Syntax



Rules

You can only declare named blocks as **THREADLOCAL**. All rules and constraints that normally apply to named common blocks apply to common blocks that are declared as **THREADLOCAL**.

The **THREADLOCAL** directive must appear in the *specification_part* of the scoping unit. If a common block appears in a **THREADLOCAL** directive, it must also be declared within a **COMMON** statement in the same scoping unit. The **THREADLOCAL** directive may occur before or after the **COMMON** statement.

A common block cannot be given the **THREADLOCAL** attribute if it is declared within a **PURE** subprogram.

Members of a **THREADLOCAL** common block must not appear in **NAMELIST** statements.

A common block that is use-associated must not be declared as **THREADLOCAL** in the scoping unit that contains the **USE** statement.

Any pointers declared in a **THREADLOCAL** common block are not affected by the **-qinit=f90ptr** compiler option.

Objects within **THREADLOCAL** common blocks may be used in parallel loops and parallel sections. However, these objects are implicitly shared across the iterations of the loop, and across code blocks within parallel sections. In other words, within a scoping unit, all accessible common blocks, whether declared as **THREADLOCAL** or not, have the **SHARED** attribute within parallel loops and sections in that scoping unit.

If a common block is declared as **THREADLOCAL** within a scoping unit, any subprogram that declares or references the common block, and that is directly or indirectly referenced by the scoping unit, must be executed by the same thread executing the scoping unit. If two procedures that declare common blocks are executed by different threads, then they would obtain different copies of the common block, provided that the common block had been declared **THREADLOCAL**. Threads can be created in one of the following ways:

- Explicitly, via *pthread*s library calls
- Implicitly by the compiler for parallel loop execution
- Implicitly by the compiler for parallel section execution.

If a common block is declared to be **THREADLOCAL** in one scoping unit, it must be declared to be **THREADLOCAL** in every scoping unit that declares the common block.

If a **THREADLOCAL** common block that does not have the **SAVE** attribute is declared within a subprogram, the members of the block become undefined at subprogram **RETURN** or **END**, unless there is at least one other scoping unit in which the common block is accessible that is making a direct or indirect reference to the subprogram.

You cannot specify the same *common_block_name* for both a **THREADLOCAL** directive and a **THREADPRIVATE** directive.

Example 1: The following procedure "FORT_SUB" is invoked by two threads:

```

SUBROUTINE FORT_SUB(IARG)
  INTEGER IARG

  CALL LIBRARY_ROUTINE1()
  CALL LIBRARY_ROUTINE2()
  ...
END SUBROUTINE FORT_SUB
SUBROUTINE LIBRARY_ROUTINE1()
  COMMON /BLOCK/ R
  SAVE /BLOCK/
  !IBM* THREADLOCAL /BLOCK/
  R = 1.0
  ...
END SUBROUTINE LIBRARY_ROUTINE1
SUBROUTINE LIBRARY_ROUTINE2()
  COMMON /BLOCK/ R
  SAVE /BLOCK/
  !IBM* THREADLOCAL /BLOCK/

```

! The SAVE attribute is required for the
! common block because the program requires
! that the block remain defined after
! library_routine1 is invoked.

THREADLOCAL

```
... = R
...
END SUBROUTINE LIBRARY_ROUTINE2
```

Example 2: "FORT_SUB" is invoked by multiple threads. This is an invalid example because "FORT_SUB" and "ANOTHER_SUB" both declare /BLOCK/ to be THREADLOCAL. They intend to share the common block, but they are executed by different threads.

```
SUBROUTINE FORT_SUB()
  COMMON /BLOCK/ J
  INTEGER :: J
  !IBM* THREADLOCAL /BLOCK/           ! Each thread executing FORT_SUB
                                       ! obtains its own copy of /BLOCK/

  INTEGER A(10)

  ...
  !IBM* INDEPENDENT
  DO INDEX = 1,10
    CALL ANOTHER_SUB(A(I))
  END DO
  ...

END SUBROUTINE FORT_SUB
SUBROUTINE ANOTHER_SUB(AA)           ! Multiple threads
are used to execute ANOTHER_SUB
  INTEGER AA
  COMMON /BLOCK/ J                   ! Each thread obtains a new copy of the
  INTEGER :: J                       ! common block /BLOCK/
  !IBM* THREADLOCAL /BLOCK/

  ...
  AA = J                             ! The value of 'J' is undefined.
END SUBROUTINE ANOTHER_SUB
```

THREADPRIVATE

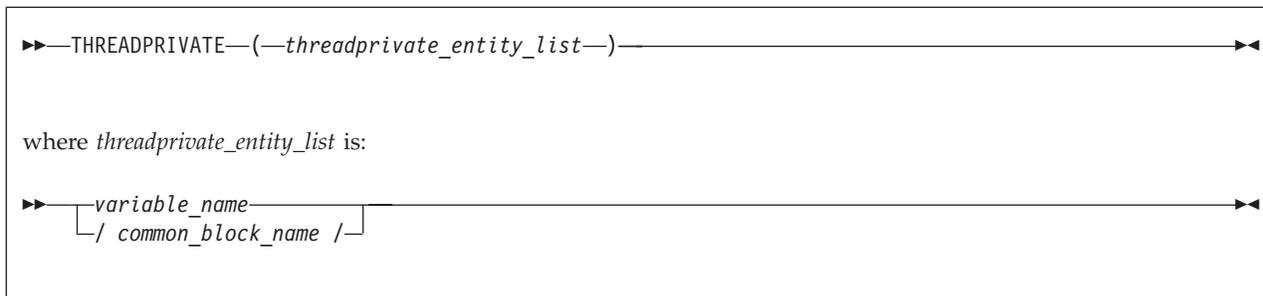
The **THREADPRIVATE** directive allows you to specify named common blocks and named variables as private to a thread but global within that thread. Once you declare a common block or variable **THREADPRIVATE**, each thread in the team maintains a separate copy of that common block or variable. Data written to a **THREADPRIVATE** common block or variable remains private to that thread and is not visible to other threads in the team.

In the serial and **MASTER** sections of a program, only the master thread's copy of the named common block and variable is accessible.

Use the **COPYIN** clause on the **PARALLEL**, **PARALLEL DO**, **PARALLEL SECTIONS** or **PARALLEL WORKSHARE** directives to specify that upon entry into a parallel region, data in the master thread's copy of a named common block or named variable is copied to each thread's private copy of that common block or variable.

The **THREADPRIVATE** directive only takes effect if you specify the **-qsmp** compiler option.

Syntax



common_block_name
is the name of a common block to be made private to a thread.

variable_name
is the name of a variable to be made private to a thread.

Rules

You cannot specify a **THREADPRIVATE** variable, common block, or the variables that comprise that common block in a **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE**, **SHARED**, or **REDUCTION** clause.

A **THREADPRIVATE** variable must have the **SAVE** attribute. For variables or common blocks declared in the scope of a module, the **SAVE** attribute is implied. If you declare the variable outside of the scope of the module, the **SAVE** attribute must be specified.

In **THREADPRIVATE** directives, you can only specify named variables and named common blocks.

A variable can only appear in a **THREADPRIVATE** directive in the scope in which it is declared, and a **THREADPRIVATE** variable or common block may only appear once in a given scope. The variable must not be an element of a common block, or be declared in an **EQUIVALENCE** statement.

You cannot specify the same *common_block_name* for both a **THREADPRIVATE** directive and a **THREADLOCAL** directive.

All rules and constraints that apply to named common blocks also apply to common blocks declared as **THREADPRIVATE**.

If you declare a common block as **THREADPRIVATE** in one scoping unit, you must declare it as **THREADPRIVATE** in all other scoping units in which it is declared.

On entry into any parallel region, a **THREADPRIVATE** variable, or a variable in a **THREADPRIVATE** common block is subject to the following criteria when declared in a **COPYIN** clause:

- If the variable has the **POINTER** attribute and the master thread's copy of the variable is associated with a target, then each copy of that variable is associated with the same target. If the master thread's pointer is disassociated, then each copy of that variable is disassociated. If the master thread's copy of the variable has an undefined association status, then each copy of that variable has an undefined association status.
- Each copy of a variable without the **POINTER** attribute is assigned the same value as the master thread's copy of that variable.

THREADLOCAL

On entry into the first parallel region of the program, **THREADPRIVATE** variables or variables within a **THREADPRIVATE** common block not specified in a **COPYIN** clause are subject to the following criteria:

- If the variable has the **ALLOCATABLE** attribute, the initial allocation status of each copy of that variable is not currently allocated.
- If the variable has the **POINTER** attribute, and that pointer is disassociated through either explicit or default initialization, the association status of each copy of that variable is disassociated. Otherwise, the association status of the pointer is undefined.
- If the variable has neither the **ALLOCATABLE** nor the **POINTER** attribute and is defined through either explicit or default initialization, then each copy of that variable is defined. If the variable is undefined, then each copy of that variable is undefined.

On entry into subsequent parallel regions of the program, **THREADPRIVATE** variables, or variables within a **THREADPRIVATE** common block not specified in a **COPYIN** clause, are subject to the following criteria:

- If you are using the **OMP_DYNAMIC** environment variable, or the **omp_set_dynamic** subroutine to enable dynamic threads, the definition and association status of a thread's copy of that variable is undefined, and the allocation status is undefined.
- If dynamic threads are disabled, the definition, association, or allocation status and definition, if the thread's copy of the variable was defined, is retained.

You cannot access the name of a common block by use association or host association. Thus, a named common block can only appear on a **THREADPRIVATE** directive if the common block is declared in the scoping unit that contains the **THREADPRIVATE** directive. However, you can access the variables in the common block by use association or host association.

The **-qinit=f90ptr** compiler option does not affect pointers that you have declared in a **THREADPRIVATE** common block.

The **DEFAULT** clause does not affect variables in **THREADPRIVATE** common blocks.

Examples

Example 1: In this example, the **PARALLEL DO** directive invokes multiple threads that call **SUB1**. The common block **BLK** in **SUB1** shares the data that is specific to the thread with subroutine **SUB2**, which is called by **SUB1**.

```
PROGRAM TT
  INTEGER :: I, B(50)

!$OMP PARALLEL DO SCHEDULE(STATIC, 10)
  DO I=1, 50
    CALL SUB1(I, B(I))      ! Multiple threads call SUB1.
  ENDDO
END PROGRAM TT

SUBROUTINE SUB1(J, X)
  INTEGER :: J, X, A(100)
  COMMON /BLK/ A
!$OMP THREADPRIVATE(/BLK/) ! Array a is private to each thread.
! ...
  CALL SUB2(J)
  X = A(J) + A(J + 50)
! ...
END SUBROUTINE SUB1
```

```

        SUBROUTINE SUB2(K)
            INTEGER :: C(100)
            COMMON /BLK/ C
!$OMP THREADPRIVATE(/BLK/)
! ...
        C = K
! ...
! Since each thread has its own copy of
! common block BLK, the assignment of
! array C has no effect on the copies of
! that block owned by other threads.
END SUBROUTINE SUB2

```

Example 2: In this example, each thread has its own copy of the common block **ARR** in the parallel section. If one thread initializes the common block variable **TEMP**, the initial value is not visible to other threads.

```

PROGRAM ABC
    INTEGER :: I, TEMP(100), ARR1(50), ARR2(50)
    COMMON /ARR/ TEMP
!$OMP THREADPRIVATE(/ARR/)
    INTERFACE
        SUBROUTINE SUBS(X)
            INTEGER :: X(:)
        END SUBROUTINE
    END INTERFACE
! ...
!$OMP PARALLEL SECTIONS
!$OMP SECTION
! ...
        TEMP(1:100:2) = -1
        TEMP(2:100:2) = 2
        CALL SUBS(ARR1)
! ...
!$OMP SECTION
! ...
        TEMP(1:100:2) = 1
        TEMP(2:100:2) = -2
        CALL SUBS(ARR2)
! ...
!$OMP END PARALLEL SECTIONS
! ...
    PRINT *, SUM(ARR1), SUM(ARR2)
END PROGRAM ABC

SUBROUTINE SUBS(X)
    INTEGER :: K, X(:), TEMP(100)
    COMMON /ARR/ TEMP
!$OMP THREADPRIVATE(/ARR/)
! ...
    DO K = 1, UBOUND(X, 1)
        X(K) = TEMP(K) + TEMP(K + 1)
! The thread is accessing its
! own copy of
! the common block.
    ENDDO
! ...
END SUBROUTINE SUBS

```

The expected output for this program is:

```
50 -50
```

Example 3: In the following example, local variables outside of a common block are declared **THREADPRIVATE**.

THREADLOCAL

```
MODULE MDL
  INTEGER          :: A(2)
  INTEGER, POINTER :: P
  INTEGER, TARGET  :: T
!$OMP THREADPRIVATE(A, P)
END MODULE MDL

PROGRAM MVAR
USE MDL

INTEGER :: I
INTEGER OMP_GET_THREAD_NUM

CALL OMP_SET_NUM_THREADS(2)
A = (/1, 2/)
T = 4
P => T

!$OMP PARALLEL PRIVATE(I) COPYIN(A, P)
  I = OMP_GET_THREAD_NUM()
  IF (I .EQ. 0) THEN
    A(1) = 100
    T = 5
  ELSE IF (I .EQ. 1) THEN
    A(2) = 200
  END IF
!$OMP END PARALLEL

!$OMP PARALLEL PRIVATE(I)
  I = OMP_GET_THREAD_NUM()
  IF (I .EQ. 0) THEN
    PRINT *, 'A(2) = ', A(2)
  ELSE IF (I .EQ. 1) THEN
    PRINT *, 'A(1) = ', A(1)
    PRINT *, 'P => ', P
  END IF
!$OMP END PARALLEL

END PROGRAM MVAR
```

If dynamic threads mechanism is disabled, the expected output is:

```
A(2) = 2
A(1) = 1
P => 5
or
A(1) = 1
P => 5
A(2) = 2
```

Related Information

- **OMP_DYNAMIC** environment variable.
- “omp_set_dynamic” on page 86
- “PARALLEL / END PARALLEL” on page 33
- “PARALLEL DO / END PARALLEL DO” on page 35
- “PARALLEL SECTIONS / END PARALLEL SECTIONS” on page 38

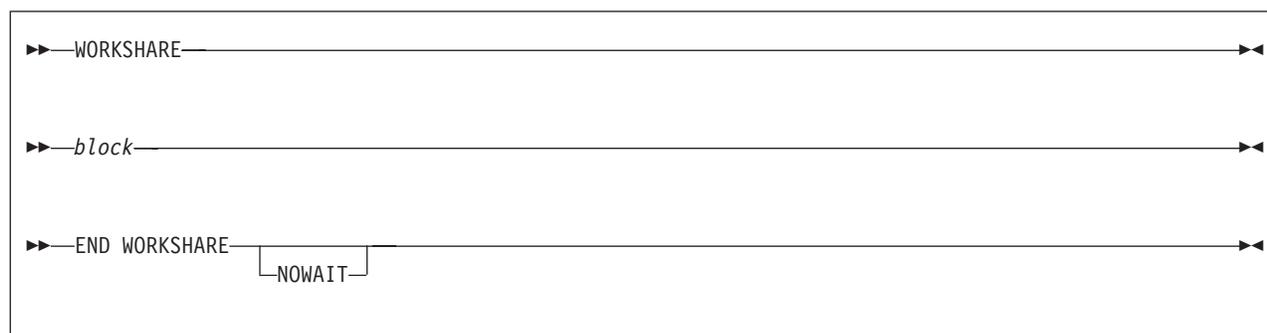
WORKSHARE

The **WORKSHARE** directive allows you to parallelize the execution of array operations. A **WORKSHARE** directive divides the tasks associated with an

enclosed block of code into *units of work*. When a team of threads encounters a **WORKSHARE** directive, the threads in the team share the tasks, so that each *unit of work* executes exactly once.

The **WORKSHARE** directive only takes effect if you specify the **-qsmp** compiler option.

Syntax



block is a structured block of statements that allows work sharing within the lexical extent of the **WORKSHARE** construct. The execution of statements are synchronized so that statements whose result is a dependent on another statement are evaluated before that result is required. The *block* can contain any of the following:

- Array assignment statements
- **ATOMIC** directives
- **CRITICAL** constructs
- **FORALL** constructs
- **FORALL** statements
- **PARALLEL** construct
- **PARALLEL DO** construct
- **PARALLEL SECTION** construct
- **PARALLEL WORKSHARE** construct
- Scalar assignment statements
- **WHERE** constructs
- **WHERE** statements

The transformational intrinsic functions you can use as part of an array operation are:

- | | | |
|----------------------|-----------------|--------------------|
| • ALL | • MATMUL | • PRODUCT |
| • ANY | • MAXLOC | • RESHAPE |
| • COUNT | • MAXVAL | • SPREAD |
| • CSHIFT | • MINLOC | • SUM |
| • DOT_PRODUCT | • MINVAL | • TRANSPOSE |
| • EOSHIFT | • PACK | • UNPACK |

The *block* can also contain statements bound to lexically enclosed **PARALLEL** constructs. These statements are not restricted.

Any user-defined function calls within the *block* must be elemental.

THREADLOCAL

Statements enclosed in a **WORKSHARE** directive are divided into *units of work*. The definition of a *unit of work* varies according to the statement evaluated. A *unit of work* is defined as follows:

- **Array expressions:** Evaluation of each element of an array expression is a *unit of work*. Any of the transformational intrinsic functions listed above may be divided into any number of *units of work*.
- **Assignment statements:** In an array assignment statement, the assignment of each element in the array is a *unit of work*. For scalar assignment statements, the assignment operation is a *unit of work*.
- **Constructs:** Evaluation of each **CRITICAL** construct is a *unit of work*. Each **PARALLEL** construct contained within a **WORKSHARE** construct is a single *unit of work*. New teams of threads execute the statements contained within the lexical extent of the enclosed **PARALLEL** constructs. In **FORALL** constructs or statements, the evaluation of the mask expression, expressions occurring in the specification of the iteration space, and the masked assignments are *units of work*. In **WHERE** constructs or statements, the evaluation of the mask expression and the masked assignments are *units of work*.
- **Directives:** The update of each scalar variable for an **ATOMIC** directive and its assignments is a *unit of work*.
- **ELEMENTAL functions:** If the argument to an **ELEMENTAL** function is an array, then the application of the function to each element of an array is a *unit of work*.

If none of the above definitions apply to a statement within the *block*, then that statement is a *unit of work*.

Rules

In order to ensure that the statements within a **WORKSHARE** construct execute in parallel, the construct must be enclosed within the dynamic extent of a parallel region. Threads encountering a **WORKSHARE** construct outside the dynamic extent of a parallel region will evaluate the statements within the construct serially.

A **WORKSHARE** directive binds to the closest dynamically enclosing **PARALLEL** directive if one exists.

You must not nest **DO**, **SECTIONS**, **SINGLE** and **WORKSHARE** directives that bind to the same **PARALLEL** directive

You must not specify a **WORKSHARE** directive within the dynamic extent of **CRITICAL**, **MASTER**, or **ORDERED** directives.

You must not specify **BARRIER**, **MASTER**, or **ORDERED** directives within the dynamic extent of a **WORKSHARE** construct.

If an array assignment, scalar assignment, a masked array assignment or a **FORALL** assignment assigns to a private variable in the *block*, the result is undefined.

If an array expression in the *block* references the value, association status or allocation status of private variables, the value of the expression is undefined unless each thread computes the same value.

If you do not specify a **NO WAIT** clause at the end of a **WORKSHARE** construct, a **BARRIER** directive is implied.

A **WORKSHARE** construct must be encountered by all threads in the team or by none at all.

Examples

Example 1: In the following example, the **WORKSHARE** directive evaluates the masked expressions in parallel.

```
!$OMP WORKSHARE
  FORALL (I = 1 : N, AA(1, I) == 0) AA(1, I) = I
  BB = TRANSPOSE(AA)
  CC = MATMUL(AA, BB)
!$OMP ATOMIC
  S = S + SUM(CC)
!$OMP END WORKSHARE
```

Example 2: The following example includes a user defined **ELEMENTAL** as part of a **WORKSHARE** construct.

```
!$OMP WORKSHARE
  WHERE (AA(1, :) /= 0.0) AA(1, :) = 1 / AA(1, :)
  DD = TRANS(AA(1, :))
!$OMP END WORKSHARE

ELEMENTAL REAL FUNCTION TRANS(ELM) RESULT(RES)
REAL, INTENT(IN) :: ELM
RES = ELM * ELM + 4
END FUNCTION
```

Related Information

- “ATOMIC” on page 18
- “BARRIER” on page 21
- “CRITICAL / END CRITICAL” on page 22
- “PARALLEL WORKSHARE / END PARALLEL WORKSHARE” on page 41
- **-qsmp** compiler option.

OpenMP Directive Clauses

The following OpenMP directive clauses allow you to specify the scope attributes of variables within a parallel construct. The **IF**, **NUM_THREADS**, **ORDERED**, and **SCHEDULE** clauses, also in this section, allow you to control the parallel environment of a parallel region. See the detailed directive descriptions for more information.

COPYIN	FIRSTPRIVATE	PRIVATE
COPYPRIVATE	LASTPRIVATE	REDUCTION
DEFAULT	NUM_THREADS	SCHEDULE
IF	ORDERED	SHARED

Global Rules for Directive Clauses

You must not specify a variable or common block name more than once in a clause.

A variable, common block name, or variable name that is a member of a common block must not appear in more than one clause on the same directive, with the following exceptions:

- You can define a named common block or named variable as **FIRSTPRIVATE** and **LASTPRIVATE** for the same directive.

OpenMP Directive Clauses

- A variable appearing in a **NUM_THREADS** clause can appear in another clause for the same directive.
- A variable appearing in a **IF** clause can appear in another clause for the same directive.

If you do not specify a clause that changes the scope of a variable, the default scope for variables affected by a directive is **SHARED**.

A local variable with the **SAVE** or **STATIC** attribute declared in a procedure referenced within the dynamic extent of a parallel region has an implicit **SHARED** attribute. A local variable without the **SAVE** or **STATIC** attribute declared in a procedure referenced within the dynamic extent of a parallel region has an implicit **PRIVATE** attribute.

Members of common blocks and variables of modules declared in procedure referenced within the dynamic extent of a parallel region have an implicit **SHARED** attribute, unless they are **THREADLOCAL** or **THREADPRIVATE** common blocks and module variables.

While a parallel or work-sharing construct is running, a variable or variable subobject used in a **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE** or **REDUCTION** clause of the directive must not be referenced, become defined, become undefined, have its association status or allocation status changed, or appear as an actual argument:

- In a scoping unit other than the one in which the directive construct appears
- In a variable format expression

You can declare a variable as **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE**, or **REDUCTION**, even if that variable is already storage associated with other variables. Storage association may exist for variables declared in **EQUIVALENCE** statements or in **COMMON** blocks. If a variable is storage associated with a **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE**, or **REDUCTION** variable, then:

- The contents, allocation status and association status of the variable that is storage associated with the **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE** or **REDUCTION** variable are undefined on entry to the parallel construct.
- The allocation status, association status and the contents of the associated variable become undefined if you define the **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE** or **REDUCTION** variable or if you define that variable's allocation or association status.
- The allocation status, association status and the contents of the **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE** or **REDUCTION** variable become undefined if you define the associated variable or if you define the associated variable's allocation or association status.

Pointers and OpenMP Fortran API Version 2.0

OpenMP Fortran API Version 2.0 allows a variable or variable subobject of a **PRIVATE** clause can have the **POINTER** attribute. The association status of the pointer is undefined at thread creation and when the thread is destroyed.

XL Fortran provides an extension which allows a variable or variable subobject of a **FIRSTPRIVATE** or **LASTPRIVATE** clause to have the **POINTER** attribute. For **FIRSTPRIVATE** pointers at thread creation, each copy of the pointer receives the same association status as the original. If the pointer is used in a **LASTPRIVATE** clause, the pointer retains its association status at the end of the last iteration or **SECTION**.

To maintain full compliance with the OpenMP Fortran API Version 2.0 standard, ensure that a **POINTER** variable applies only to a **PRIVATE** clause.

COPYIN

If you specify the **COPYIN** clause, the master thread’s copy of each variable, or common block declared in the *copyin_entity_list* is duplicated at the beginning of a parallel region. Each thread in the team that will execute within that parallel region receives a private copy of all entities in the *copyin_entity_list*. All variables declared in the *copyin_entity_list* must be **THREADPRIVATE** or members of a common block that appears in a **THREADPRIVATE** directive.

Syntax

▶▶—COPYIN—(—*copyin_entity_list*—)————▶▶

copyin_entity



variable

is a **THREADPRIVATE** variable, or **THREADPRIVATE** variable in a common block

common_block_name

is a **THREADPRIVATE** common block name

Rules

If you specify a **COPYIN** clause, you cannot:

- specify the same entity name more than once in a *copyin_entity_list*.
- specify the same entity name in separate **COPYIN** clauses on the same directive.
- specify both a common block name and any variable within that same named common block in a *copyin_entity_list*.
- specify both a common block name and any variable within that same named common block in different **COPYIN** clauses on the same directive.
- specify a variable with the **ALLOCATABLE** attribute.

When the master thread of a team of threads reaches a directive containing the **COPYIN** clause, thread’s private copy of a variable or common block specified in the **COPYIN** clause will have the same value as the master thread’s copy.

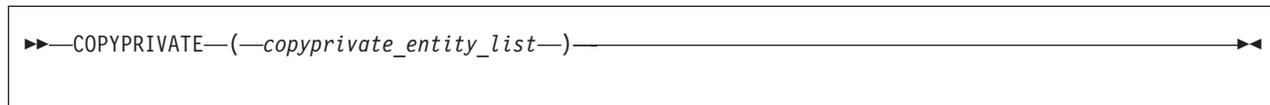
The **COPYIN** clause applies to:

- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**

COPYPRIVATE

If you specify the **COPYPRIVATE** clause, the value of a private variable or pointer to a shared object from one thread in a team is copied into the corresponding variables of all other threads in that team. If the variable in *copyprivate_entity_list* is not a pointer, then the corresponding variables of all threads within that team are defined with the value of that variable. If the variable is a pointer, then the corresponding variables of all threads within that team are defined with the association status of the pointer. Integer pointers and assumed-size arrays must not appear in *copyprivate_entity_list*.

Syntax



copyprivate_entity



variable

is a private variable within the enclosing parallel region

common_block_name

is a **THREADPRIVATE** common block name

Rules

If a common block is part of the *copyprivate_entity_list*, then it must appear in a **THREADPRIVATE** directive. Furthermore, the **COPYPRIVATE** clause treats a common block as if all variables within its *object_list* were specified in the *copyprivate_entity_list*.

A **COPYPRIVATE** clause must occur on an **END SINGLE** directive at the end of a **SINGLE** construct. The compiler evaluates a **COPYPRIVATE** clause before any threads have passed the implied **BARRIER** directive at the end of that construct. The variables you specify in *copyprivate_entity_list* must not appear in a **PRIVATE** or **FIRSTPRIVATE** clause for the **SINGLE** construct. If the **END SINGLE** directive occurs within the dynamic extent of a parallel region, the variables you specify in *copyprivate_entity_list* must be private within that parallel region.

A **COPYPRIVATE** clause must not appear on the same **END SINGLE** directive as a **NOWAIT** clause.

A **THREADLOCAL** common block, or members of that common block, are not permitted as part of a **COPYPRIVATE** clause.

A **COPYPRIVATE** clause applies to the following directives:

- **END SINGLE**

DEFAULT

If you specify the **DEFAULT** clause, all variables in the lexical extent of the parallel construct will have a scope attribute of *default_scope_attr*.

If you specify **DEFAULT(NONE)**, there is no default scope attribute. Therefore, you must explicitly list each variable you use in the lexical extent of the parallel construct in a data scope attribute clause on the parallel construct, unless the variable is:

- **THREADPRIVATE**
- A member of a **THREADPRIVATE** common block.
- A pointee
- A loop iteration variable used only as a loop iteration variable for:
 - Sequential loops in the lexical extent of the parallel region, or,
 - Parallel do loops that bind to the parallel region
- A variable that is only used in work-sharing constructs that bind to the parallel region, and is specified in a data scope attribute clause for each of the work-sharing constructs.

The **DEFAULT** clause specifies that all variables in the parallel construct share the same default scope attribute of either **PRIVATE**, **SHARED**, or no default scope attribute.

Syntax

```
▶▶—DEFAULT—(—default_scope_attr—)————▶▶
```

default_scope_attr

is one of **PRIVATE**, **SHARED**, or **NONE**

Rules

If you specify **DEFAULT(NONE)** on a directive you must specify all named variables and all the leftmost names of referenced array sections, array elements, structure components, or substrings in the lexical extent of the directive construct in a **FIRSTPRIVATE**, **LASTPRIVATE**, **PRIVATE**, **REDUCTION**, or **SHARED** clause.

If you specify **DEFAULT(PRIVATE)** on a directive, all named variables and all leftmost names of referenced array sections, array elements, structure components, or substrings in the lexical extent of the directive construct, including common block and use associated variables, but excluding **POINTEES** and **THREADLOCAL** common blocks, have a **PRIVATE** attribute to a thread as if they were listed explicitly in a **PRIVATE** clause.

If you specify **DEFAULT(SHARED)** on a directive, all named variables and all leftmost names of referenced array sections, array elements, structure components, or substrings in the lexical extent of the directive construct, excluding **POINTEES** have a **SHARED** attribute to a thread as if they were listed explicitly in a **SHARED** clause.

The default behavior will be **DEFAULT(SHARED)** if you do not explicitly indicate a **DEFAULT** clause on a directive.

The **DEFAULT** clause applies to:

- **PARALLEL**

OpenMP Directive Clauses

- PARALLEL DO
- PARALLEL SECTIONS
- PARALLEL WORKSHARE

Examples

The following example demonstrates the use of **DEFAULT(NONE)**, and some of the rules for specifying the data scope attributes of variables in the parallel region.

```
PROGRAM MAIN
  COMMON /COMBLK/ ABC(10), DEF

      ! THE LOOP ITERATION VARIABLE, I, IS NOT
      ! REQUIRED TO BE IN DATA SCOPE ATTRIBUTE CLAUSE

!$OMP  PARALLEL DEFAULT(NONE) SHARED(ABC)

      ! DEF IS SPECIFIED ON THE WORK-SHARING DO AND IS NOT
      ! REQUIRED TO BE SPECIFIED IN A DATA SCOPE ATTRIBUTE
      ! CLAUSE ON THE PARALLEL REGION.

!$OMP  DO FIRSTPRIVATE(DEF)
      DO I=1,10
        ABC(I) = DEF
      END DO
!$OMP  END PARALLEL
END
```

IF

If you specify the **IF** clause, the run-time environment performs a test to determine whether to run the block in serial or parallel. If *scalar_logical_expression* is true, then the block is run in parallel; if not, then the block is run in serial.

Syntax

▶—IF—(*scalar_logical_expression*)—▶

Rules

Within a **PARALLEL SECTIONS** construct, variables that are not appearing in the **PRIVATE** clause are assumed to be **SHARED** by default.

The **IF** clause may appear at most once in the a any directive.

By default, a nested parallel loop is serialized, regardless of the setting of the **IF** clause. You can change this default by using the **-qsmp=nested_par** compiler option.

An **IF** expression is evaluated outside of the context of the parallel construct. Any function reference in the **IF** expression must not have side effects.

The **IF** clause applies to the following directives:

- “PARALLEL / END PARALLEL” on page 33
- “PARALLEL DO / END PARALLEL DO” on page 35
- “PARALLEL SECTIONS / END PARALLEL SECTIONS” on page 38
- “PARALLEL WORKSHARE / END PARALLEL WORKSHARE” on page 41

FIRSTPRIVATE

If you use the **FIRSTPRIVATE** clause, each thread has its own initialized local copy of the variables and common blocks in *data_scope_entity_list*.

The **FIRSTPRIVATE** clause can be specified for the same variables as the **PRIVATE** clause, and functions in a manner similar to the **PRIVATE** clause. The exception is the status of the variable upon entry into the directive construct; the **FIRSTPRIVATE** variable exists and is initialized for each thread entering the directive construct.

Syntax

▶▶—FIRSTPRIVATE—(—*data_scope_entity_list*—)————▶▶

Rules

A variable in a **FIRSTPRIVATE** clause must not be any of the following elements:

- A pointer
- An assumed-size array
- A **THREADLOCAL** common block
- A **THREADPRIVATE** common block or its members
- A **THREADPRIVATE** variable
- An allocatable object

You cannot specify a variable in a **FIRSTPRIVATE** clause of a parallel construct if:

- the variable appears in a namelist statement, variable format expression or in an expression for a statement function definition, and,
- you reference the statement function, the variable format expression through formatted I/O, or the namelist through namelist I/O, within the parallel construct.

When individual members of a common block are privatized, the storage of the specified variable is no longer associated with the storage of the common block.

Any variable that is storage associated with a **FIRSTPRIVATE** variable is undefined on entrance into the parallel construct.

If a directive construct contains a **FIRSTPRIVATE** argument to a Message Passing Interface (MPI) routine performing non-blocking communication, the MPI communication must complete before the end of the construct.

The **FIRSTPRIVATE** clause applies to the following directives:

- **DO**
- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**
- **SECTIONS**
- **SINGLE**

LASTPRIVATE

If you use the **LASTPRIVATE** clause, each variable and common block in *data_scope_entity_list* is **PRIVATE**, and the last value of each variable in *data_scope_entity_list* can be referred to outside of the construct of the directive. If you use the **LASTPRIVATE** clause with **DO** or **PARALLEL DO**, the last value is the value of the variable after the last sequential iteration of the loop. If you use the **LASTPRIVATE** clause with **SECTIONS** or **PARALLEL SECTIONS**, the last value is the value of the variable after the last **SECTION** of the construct. If the last iteration of the loop or last section of the construct does not define a **LASTPRIVATE** variable, the variable is undefined after the loop or construct.

The **LASTPRIVATE** clause functions in a manner similar to the **PRIVATE** clause and you should specify it for variables that match the same criteria. The exception is in the status of the variable on exit from the directive construct. The compiler determines the last value of the variable, and takes a copy of that value which it saves in the named variable for use after the construct. A **LASTPRIVATE** variable is undefined on entry to the construct if it is not a **FIRSTPRIVATE** variable.

Syntax

▶▶—LASTPRIVATE—(—*data_scope_entity_list*—)————▶▶

Rules

A variable in a **LASTPRIVATE** clause must not be any of the following elements:

- A pointee
- An allocatable object
- An assumed-size array
- A **THREADLOCAL** common block
- A **THREADPRIVATE** common block or its members
- A **THREADPRIVATE** variable

You cannot specify a variable in a **LASTPRIVATE** clause of a parallel construct if:

- the variable appears in a namelist statement, variable format expression or in an expression for a statement function definition, and,
- you reference the statement function, the variable format expression through formatted I/O, or the namelist through namelist I/O, within the parallel construct.

When individual members of a common block are privatized, the storage of the specified variable is no longer associated with the storage of the common block.

Any variable that is storage associated with a **LASTPRIVATE** variable is undefined on entrance into the parallel construct.

If a directive construct contains a **LASTPRIVATE** argument to a Message Passing Interface (MPI) routine performing non-blocking communication, the MPI communication must complete before the end of that construct.

If you specify a variable as **LASTPRIVATE** on a work-sharing directive, and you have specified a **NOWAIT** clause on that directive, you cannot use that variable between the end of the work-sharing construct and a **BARRIER**.

Variables that you specify as **LASTPRIVATE** to a parallel construct become defined at the end of the construct. If you have concurrent definitions or uses of **LASTPRIVATE** variables on multiple threads, you must ensure that the threads are synchronized at the end of the construct when the variables become defined. For example, if multiple threads encounter a **PARALLEL** construct with a **LASTPRIVATE** variable, you must synchronize the threads when they reach the **END PARALLEL** directive, because the **LASTPRIVATE** variable becomes defined at **END PARALLEL**. Therefore the whole **PARALLEL** construct must be enclosed within a synchronization construct.

The **LASTPRIVATE** clause applies to the following directives:

- **DO**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **SECTIONS**

Examples

The following example shows the proper use of a **LASTPRIVATE** variable after a **NOWAIT** clause.

```
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(K)

        DO I=1,10
           K=I+1
        END DO

!$OMP END DO NOWAIT

! PRINT *, K **ERROR** ! The reference to K must occur after a
                        ! barrier.

!$OMP BARRIER
PRINT *, K           ! This reference to K is legal.
!$OMP END PARALLEL
END
```

NUM_THREADS

The **NUM_THREADS** clause allows you to specify the number of threads used in a parallel region. Subsequent parallel regions are not affected. The **NUM_THREADS** clause takes precedence over the number of threads specified using the `omp_set_num_threads` library routine or the environment variable **OMP_NUM_THREADS**.

Syntax

►►—**NUM_THREADS**—(—*scalar_integer_expression*—)——►►

Rules

The value of *scalar_integer_expression* must be a positive. Evaluation of the expression occurs outside the context of the parallel region. Any function calls that appear in the expression and change the value of a variable referenced in the expression will have unspecified results.

LASTPRIVATE

If you are using the environment variable `OMP_DYNAMIC` to enable dynamic threads, *scalar_integer_expression* defines the maximum number of threads available in the parallel region.

You must specify the `omp_set_nested` library routine or set the `OMP_NESTED` environment variable when including the `NUM_THREADS` clause as part of a nested parallel regions otherwise, the execution of that parallel region is serialized.

The `NUM_THREADS` clause applies to the following work-sharing constructs:

- `PARALLEL`
- `PARALLEL DO`
- `PARALLEL SECTIONS`
- `PARALLEL WORKSHARE`

ORDERED

Specifying the `ORDERED` clause on a work-sharing construct allows you to specify the `ORDERED` directive within the dynamic extent of a parallel loop.

Syntax

```
▶▶—ORDERED—▶▶
```

Rules

The `ORDERED` clause applies to the following directives:

- “`DO / END DO`” on page 23
- “`PARALLEL DO / END PARALLEL DO`” on page 35

PRIVATE

If you specify the `PRIVATE` clause on one of the directives listed below, each thread in a team has its own uninitialized local copy of the variables and common blocks in *data_scope_entity_list*.

You should specify a variable with the `PRIVATE` attribute if its value is calculated by a single thread and that value is not dependent on any other thread, if it is defined before it is used in the construct, and if its value is not used after the construct ends. Copies of the `PRIVATE` variable exist, locally, on each thread. Each thread receives its own uninitialized copy of the `PRIVATE` variable. A `PRIVATE` variable has an undefined value or association status on entry to, and exit from, the directive construct. All thread variables within the lexical extent of the directive construct have the `PRIVATE` attribute by default.

Syntax

```
▶▶—PRIVATE—(—data_scope_entity_list—)——▶▶
```

Rules

A variable in the `PRIVATE` clause must not be any of the following elements:

- A pointee

- An assumed-size array
- A **THREADLOCAL** common block
- A **THREADPRIVATE** common block or its members
- A **THREADPRIVATE** variable

You cannot specify a variable in a **PRIVATE** clause of a parallel construct if:

- the variable appears in a namelist statement, variable format expression or in an expression for a statement function definition, and,
- you reference the statement function, the variable format expression through formatted I/O, or the namelist through namelist I/O, within the parallel construct.

When individual members of a common block are privatized, the storage of the specified variable is no longer associated with the storage of the common block.

Any variable that is storage associated with a **PRIVATE** variable is undefined on entrance into the parallel construct.

If a directive construct contains a **PRIVATE** argument to a Message Passing Interface (MPI) routine performing non-blocking communication, the MPI communication must complete before the end of that construct.

A variable name in the *data_scope_entity_list* of the **PRIVATE** clause can be an allocatable object. It must not be allocated on initial entry to the directive construct, and you must allocate and deallocate the object for every thread that executes the construct.

Local variables without the **SAVE** or **STATIC** attributes in referenced subprograms in the dynamic extent of a directive construct have an implicit **PRIVATE** attribute.

The **PRIVATE** clause applies to the following directives:

- **DO**
- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **SECTIONS**
- **SINGLE**
- **PARALLEL WORKSHARE**

Examples

The following example demonstrates the proper use of a **PRIVATE** variable that is used to define a statement function. A commented line shows the invalid use. Since *J* appears in a statement function, the statement function cannot be referenced within the parallel construct for which *J* is **PRIVATE**.

```

      INTEGER :: ARR(10), J = 17
      ISTFNC() = J

!$OMP PARALLEL DO PRIVATE(J)
      DO I = 1, 10
         J=I
         ARR(I) = J
         ! ARR(I) = ISTFNC() **ERROR**   A reference to ISTFNC would
                                         ! make the PRIVATE(J) clause

```

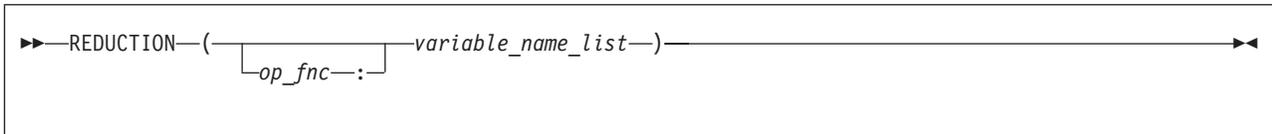
```

                                ! invalid.
                                END DO
                                PRINT *, ARR
                                END
    
```

REDUCTION

The **REDUCTION** clause updates named variables declared on the clause within the directive construct. Intermediate values of **REDUCTION** variables are not used within the parallel construct, other than in the updates themselves.

Syntax



op_fnc is a *reduction_op* or a *reduction_function* that appears in all **REDUCTION** statements involving this variable. You must not specify more than one **REDUCTION** operator or function for a variable in the directive construct. To maintain OpenMP Fortran API Version 2.0 compliance, you must specify *op_fnc* for the **REDUCTION** clause.

A **REDUCTION** statement can have one of the following forms:

```

▶▶ reduction_var_ref = expr reduction_op reduction_var_ref
    
```

```

▶▶ reduction_var_ref = reduction_var_ref reduction_op expr
    
```

```

▶▶ reduction_var_ref = reduction_function (expr, reduction_var_ref)
    
```

```

▶▶ reduction_var_ref = reduction_function (reduction_var_ref, expr)
    
```

where:

reduction_var_ref
is a variable or subobject of a variable that appears in a **REDUCTION** clause

reduction_op
is one of the intrinsic operators: +, -, *, .AND., .OR., .EQV., .NEQV., or .XOR.

reduction_function
is one of the intrinsic procedures: MAX, MIN, IAND, IOR, or IEOB.

The canonical initialization value of each of the operators and intrinsics are shown in the following table. The actual initialization value will be consistent with the data type of your corresponding **REDUCTION** variable.

Intrinsic Operator	Initialization
+	0

*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
.XOR.	.FALSE.
Intrinsic Procedure	Initialization
MAX	Smallest representable number
MIN	Largest representable number
IAND	All bits on
IOR	0
IEOR	0

Rules

The following rules apply to **REDUCTION** statements:

- A variable in the **REDUCTION** clause must only occur in a **REDUCTION** statement within the directive construct on which the **REDUCTION** clause appears.
- The two *reduction_var_refs* that appear in a **REDUCTION** statement must be lexically identical.
- You cannot use the following form of the **REDUCTION** statement:
 $reduction_var_ref = expr\ operator\ reduction_var_ref$

When you specify individual members of a common block in a **REDUCTION** clause, the storage of the specified variable is no longer associated with the storage of the common block.

Any variable you specify in a **REDUCTION** clause of a work-sharing construct must be shared in the enclosing **PARALLEL** construct.

If you use a **REDUCTION** clause on a construct that has a **NOWAIT** clause, the **REDUCTION** variable remains undefined until a barrier synchronization has been performed to ensure that all threads have completed the **REDUCTION** clause.

A **REDUCTION** variable must not appear in a **FIRSTPRIVATE**, **PRIVATE** or **LASTPRIVATE** clause of another construct within the dynamic extent of the construct in which it appeared as a **REDUCTION** variable.

If you specify *op_fnc* for the **REDUCTION** clause, each variable in the *variable_name_list* must be of intrinsic type. The variable can only appear in a **REDUCTION** statement within the lexical extent of the directive construct. You must specify *op_fnc* if the directive uses the *trigger_constant* **\$OMP**.

The **REDUCTION** clause specifies named variables that appear in reduction operations. The compiler will maintain local copies of such variables, but will combine them upon exit from the construct. The intermediate values of the **REDUCTION** variables are combined in random order, dependent on which threads finish their calculations first. Therefore, there is no guarantee that

LASTPRIVATE

bit-identical results will be obtained from one parallel run to another. This is true even if the parallel runs use the same number of threads, scheduling type, and chunk size.

Variables that you specify as **REDUCTION** or **LASTPRIVATE** to a parallel construct become defined at the end of the construct. If you have concurrent definitions or uses of **REDUCTION** or **LASTPRIVATE** variables on multiple threads, you must ensure that the threads are synchronized at the end of the construct when the variables become defined. For example, if multiple threads encounter a **PARALLEL** construct with a **REDUCTION** variable, you must synchronize the threads when they reach the **END PARALLEL** directive, because the **REDUCTION** variable becomes defined at **END PARALLEL**. Therefore the whole **PARALLEL** construct must be enclosed within a synchronization construct.

A variable in the **REDUCTION** clause must be of intrinsic type. A variable in the **REDUCTION** clause, or any element thereof, must not be any of the following:

- A pointer
- An assumed-size array
- A **THREADLOCAL** common block
- A **THREADPRIVATE** common block or its members
- A **THREADPRIVATE** variable
- An Allocatable object
- A Fortran 90 pointer

These rules describe the use of **REDUCTION** on OpenMP directives. If you are using the **REDUCTION** clause on the **INDEPENDENT** directive, see the **INDEPENDENT** directive.

The OpenMP implementation of the **REDUCTION** clause applies to:

- **DO**
- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**
- **SECTIONS**

SCHEDULE

The **SCHEDULE** clause allows you to specify the chunking method for parallelization. Work is assigned to threads in different manners depending on the scheduling type or chunk size used.

Syntax

```
►► SCHEDULE ( ( sched_type [ , n ] ) ) ►►
```

sched_type

is one of **AFFINITY**, **DYNAMIC**, **GUIDED**, **RUNTIME**, or **STATIC**

n

must be a positive scalar integer expression; it must not be specified for the

RUNTIME *sched_type*. If you are using the *trigger_constant* **\$OMP**, do not specify the scheduling type **AFFINITY**.

AFFINITY

The iterations of a loop are initially divided into *number_of_threads* partitions, containing

$\text{CEILING}(\text{number_of_iterations} / \text{number_of_threads})$

iterations. Each partition is initially assigned to a thread, and is then further subdivided into chunks containing *n* iterations, if *n* has been specified. If *n* has not been specified, then the chunks consist of

$\text{CEILING}(\text{number_of_iterations_remaining_in_partition} / 2)$

loop iterations.

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

Threads that are active will complete the work in a partition that is initially assigned to a sleeping thread.

DYNAMIC

If *n* has been specified, the iterations of a loop are divided into chunks containing *n* iterations each. If *n* has not been specified, then the default chunk size is 1 iteration.

Threads are assigned these chunks on a "first-come, first-do" basis. Chunks of the remaining work are assigned to available threads, until all work has been assigned.

If a thread is asleep, its assigned work will be taken over by an active thread, once that other thread becomes available.

GUIDED

If you specify a value for *n*, the iterations of a loop are divided into chunks such that the size of each successive chunk is exponentially decreasing. *n* specifies the size of the smallest chunk, except possibly the last. If you do not specify a value for *n*, the default value is 1.

The size of the initial chunk is

$\text{CEILING}(\text{number_of_iterations} / \text{number_of_threads})$

iterations. Subsequent chunks consist of

$\text{CEILING}(\text{number_of_iterations_remaining} / \text{number_of_threads})$

iterations. As each thread finishes a chunk, it dynamically obtains the next available chunk.

You can use guided scheduling in a situation in which multiple threads in a team might arrive at a **DO** work-sharing construct at varying times, and each iteration requires roughly the same amount of work. For example, if you have a **DO** loop preceded by one or more work-sharing **SECTIONS** or **DO** constructs with **NOWAIT** clauses, you can guarantee that no thread waits at the barrier longer than it takes another thread to execute its final

LASTPRIVATE

iteration, or final k iterations if a chunk size of k is specified. The **GUIDED** schedule requires the fewest synchronizations of all the scheduling methods.

An n expression is evaluated outside of the context of the **DO** construct. Any function reference in the n expression must not have side effects.

The value of the n parameter on the **SCHEDULE** clause must be the same for all of the threads in the team.

RUNTIME

Determine the scheduling type at run time.

At run time, the scheduling type can be specified using the environment variable **XLSMPOPTS**. If no scheduling type is specified using that variable, then the default scheduling type used is **STATIC**.

STATIC

If n has been specified, the iterations of a loop are divided into chunks that contain n iterations. Each thread is assigned chunks in a "round robin" fashion. This is known as block cyclic scheduling. If the value of n is 1, then the scheduling type is specifically referred to as cyclic scheduling.

If n has not been specified, the chunks will contain

$\text{CEILING}(\text{number_of_iterations} / \text{number_of_threads})$

iterations. Each thread is assigned one of these chunks. This is known as block cyclic scheduling.

If a thread is asleep and it has been assigned work, it will be awakened so that it may complete its work.

STATIC is the default scheduling type if the user has not specified any scheduling type at compile-time or run time.

Rules

You must not specify the **SCHEDULE** clause more than once for a particular **DO** directive.

The **SCHEDULE** clause applies to the following directives:

- "DO / END DO" on page 23
- "PARALLEL DO / END PARALLEL DO" on page 35

SHARED

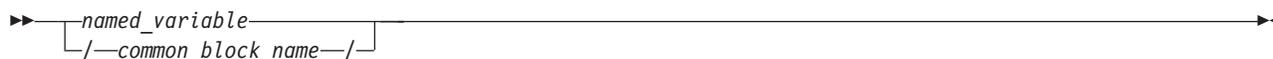
All sections use the same copy of the variables and common blocks you specify in *data_scope_entity_list*.

The **SHARED** clause specifies variables that must be available to all threads. If you specify a variable as **SHARED**, you are stating that all threads can safely share a single copy of the variable.

Syntax

►►—SHARED—(*—data_scope_entity_list—*)—

data_scope_entity



named_variable

is a named variable that is accessible in the directive construct

common_block_name

is a common block name that is accessible in the directive construct

Rules

A variable in the **SHARED** clause must not be either:

- A pointee
- A **THREADLOCAL** common block.
- A **THREADPRIVATE** common block or its members.
- A **THREADPRIVATE** variable.

If a **SHARED** variable, a subobject of a **SHARED** variable, or an object associated with a **SHARED** variable or subobject of a **SHARED** variable appears as an actual argument in a reference to a non-intrinsic procedure and:

- The actual argument is an array section with a vector subscript; or
- The actual argument is
 - An array section,
 - An assumed-shape array, or,
 - A pointer array

and the associated dummy argument is an explicit-shape or assumed-size array;

then any references to or definitions of the shared storage that is associated with the dummy argument by any other thread must be synchronized with the procedure reference. You can do this, for example, by placing the procedure reference after a **BARRIER**.

The **SHARED** clause applies to:

- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**

Examples

In the following example, the procedure reference with an array section actual argument is required to be synchronized with references to the dummy argument by placing the procedure reference in a critical section, because the associated dummy argument is an explicit-shape array.

```

      INTEGER:: ABC(10)
      I=2; J=5
!$OMP PARALLEL DEFAULT(NONE), SHARED(ABC,I,J)
!$OMP  CRITICAL
      CALL SUB1(ABC(I:J))    ! ACTUAL ARGUMENT IS AN ARRAY
                           ! SECTION; THE PROCEDURE
                           ! REFERENCE MUST BE IN A CRITICAL SECTION.
!$OMP  END CRITICAL
!$OMP END PARALLEL
CONTAINS
```

LASTPRIVATE

```
SUBROUTINE SUB1(ARR)
  INTEGER:: ARR(1:4)
  DO I=1, 4
    ARR(I) = I
  END DO
END SUBROUTINE
END
```

OpenMP Execution Environment and Lock Routines

The OpenMP specification provides a number of routines which allow you to control and query the parallel execution environment.

Parallel threads created by the run-time environment through the OpenMP interface are considered independent of the threads you create and control using calls to the **Fortran Pthreads library module**. References within the following descriptions to "serial portions of the program" refer to portions of the program that are executed by only one of the threads that have been created by the run-time environment. For example, you can create multiple threads by using **f_pthread_create**. However, if you then call **omp_get_num_threads** from outside of an OpenMP parallel block, or from within a serialized nested parallel region, the function will return **1**, regardless of the number of threads that are currently executing.

The OpenMP execution environment routines are:

- **omp_get_dynamic**: see "omp_get_dynamic" on page 80
- **omp_get_max_threads**: see "omp_get_max_threads" on page 81
- **omp_get_nested**: see "omp_get_nested" on page 81
- **omp_get_num_procs**: see "omp_get_num_procs" on page 81
- **omp_get_num_threads**: see "omp_get_num_threads" on page 82
- **omp_get_thread_num**: see "omp_get_thread_num" on page 82
- **omp_in_parallel**: see "omp_in_parallel" on page 84
- **omp_set_dynamic**: see "omp_set_dynamic" on page 86
- **omp_set_nested**: see "omp_set_nested" on page 87
- **omp_set_num_threads**: see "omp_set_num_threads" on page 88

Included in the OpenMP run-time library are two routines that support a portable wall-clock timer. The OpenMP timing routines are:

- **omp_get_wtick**: see "omp_get_wtick" on page 83
- **omp_get_wtime**: see "omp_get_wtime" on page 84

The OpenMP run-time library also supports a set of simple and nestable lock routines. You must only lock variables through these routines. Simple locks may not be locked if they are already in a locked state. Simple lock variables are associated with simple locks and may only be passed to simple lock routines. Nestable locks may be locked multiple times by the same thread. Nestable lock variables are associated with nestable locks and may only be passed to nestable lock routines.

For all the routines listed below, the lock variable is an integer whose **KIND** type parameter is denoted either by the symbolic constant **omp_lock_kind**, or by **omp_nest_lock_kind**. Please note that the predefined lock variables are defined inside the `omp_lib` module which is not an intrinsic data type.

This variable is sized according to the compilation mode. It is set either to '4'.

OpenMP provides the following simple lock routines:

- **omp_destroy_lock**: see "omp_destroy_lock" on page 80

OpenMP Routines

- **omp_init_lock**: see “omp_init_lock” on page 85
- **omp_set_lock**: see “omp_set_lock” on page 86
- **omp_test_lock**: see “omp_test_lock” on page 88
- **omp_unset_lock**: see “omp_unset_lock” on page 89

OpenMP provides the following nestable lock routines:

- **omp_destroy_nest_lock**: see “omp_destroy_nest_lock”
- **omp_init_nest_lock**: see “omp_init_nest_lock” on page 85
- **omp_set_nest_lock**: see “omp_set_nest_lock” on page 87
- **omp_test_nest_lock**: see “omp_test_nest_lock” on page 89
- **omp_unset_nest_lock**: see “omp_unset_nest_lock” on page 89

Note: You can define and implement your own versions of the OpenMP routines. However, by default, the compiler will substitute the XL Fortran versions of the OpenMP routines regardless of the existence of other implementations, unless you specify the **-qnoswapomp** compiler option.

omp_destroy_lock

This subroutine disassociates a given lock variable from all locks. You have to use **omp_init_lock** to reinitialize a lock variable that has been destroyed with a call to **omp_destroy_lock** before using it again as a lock variable.

Note: If you call **omp_destroy_lock** with a lock variable that has not been initialized, the result of the call is undefined.

Argument Type and Attributes

Type integer with kind **omp_lock_kind**.

Examples

For an example of how to use **omp_destroy_lock**, see “omp_init_lock” on page 85

omp_destroy_nest_lock

This subroutine initializes a nestable lock variable causing the lock variable to become undefined. The variable *nvar* must be an unlocked and initialized nestable lock variable.

Note: : If you call **omp_destroy_nest_lock** using a variable that is not initialized, the result is undefined.

Argument Type and Attributes

Type integer with kind **omp_nest_lock_kind**.

Examples

```
USE omp_lib
INTEGER(kind=omp_nest_lock_kind) LOCK
CALL omp_destroy_nest_lock(LOCK)
```

omp_get_dynamic

The **omp_get_dynamic** function returns **.TRUE.** if dynamic thread adjustment by the run-time environment is enabled, and **.FALSE.** otherwise.

Argument Type and Attributes

There are no arguments to this function.

Result Type and Attributes

Default logical.

omp_get_max_threads

This function returns the maximum number of threads that can execute concurrently in a single parallel region. The return value is equal to the maximum value that can be returned by the **omp_get_num_threads** function. If you use **omp_set_num_threads** to change the number of threads, subsequent calls to **omp_get_max_threads** will return the new value.

The function has global scope, which means that the maximum value it returns applies to all functions, subroutines, and compilation units in the program. It returns the same value whether executing from a serial or parallel region.

You can use **omp_get_max_threads** to allocate maximum-sized data structures for each thread when you have enabled dynamic thread adjustment by passing **omp_set_dynamic** an argument which evaluates to **.TRUE**.

Argument Type and Attributes

There are no arguments to this function.

Result Type and Attributes

Default integer.

Result Value

The maximum number of threads that can execute concurrently in a single parallel region.

omp_get_nested

The **omp_get_nested** function returns **.TRUE**, if nested parallelism is enabled and **.FALSE**, if nested parallelism is disabled.

Argument Type and Attributes

There are no arguments to this function.

Result Type and Attributes

Default logical.

omp_get_num_procs

The **omp_get_num_procs** function returns the number of online processors on the machine.

Argument Type and Attributes

There are no arguments to this function.

Result Type and Attributes

Default integer.

Result Value

The number of online processors on the machine.

omp_get_num_threads

The **omp_get_num_threads** function returns the number of threads in the team currently executing the parallel region from which it is called. The function binds to the closest enclosing **PARALLEL** directive.

The **omp_set_num_threads** subroutine and the **OMP_NUM_THREADS** environment variable control the number of threads in a team. If you do not explicitly set the number of threads, the run-time environment will use the number of online processors on the machine by default.

If you call **omp_get_num_threads** from a serial portion of your program or from a nested parallel region that is serialized, the function returns 1.

Argument Type and Attributes

There are no arguments to this function.

Result Type and Attributes

Default integer.

Result Value

The number of threads in the team currently executing the parallel region from which the function is called.

Examples

```
USE omp_lib
INTEGER N1, N2

N1 = omp_get_num_threads()
PRINT *, N1
!$OMP PARALLEL PRIVATE(N2)
N2 = omp_get_num_threads()
PRINT *, N2
!$OMP END PARALLEL
```

The **omp_get_num_threads** call returns 1 in the serial section of the code, so N1 is assigned the value 1. N2 is assigned the number of threads in the team executing the parallel region, so the output of the second print statement will be an arbitrary number less than or equal to the value returned by **omp_get_max_threads**.

omp_get_thread_num

This function returns the number of the currently executing thread within the team. The number returned will always be between 0 and *NUM_PARTHDS* - 1. *NUM_PARTHDS* is the number of currently executing threads within the team. The master thread of the team returns a value of 0.

If you call `omp_get_thread_num` from within a serial region, from within a serialized nested parallel region, or from outside the dynamic extent of any parallel region, this function will return a value of 0.

This function binds to the closest parallel region.

Result Type and Attributes

Default integer.

Result Value

The value of the currently executing thread within the team between 0 and `NUM_PARTHDS - 1`. `NUM_PARTHDS` is the number of currently executing threads within the team. A call to `omp_get_thread_num` from a serialized nested parallel region, or from outside the dynamic extent of any parallel region returns 0.

Examples

```

USE omp_lib
INTEGER NP

!$OMP PARALLEL PRIVATE(NP)
  NP = omp_get_thread_num()
  CALL WORK(NP)
!$OMP MASTER
  NP = omp_get_thread_num()
  CALL WORK(NP)
!$OMP END MASTER
!$OMP END PARALLEL
END

SUBROUTINE WORK(THD_NUM)
  INTEGER THD_NUM
  PRINT *, THD_NUM
END

```

omp_get_wtick

The `omp_get_wtick` function returns a double precision value equal to the number of seconds between consecutive clock ticks.

Argument Type and Attributes

There are no arguments to this function.

Result Type and Attributes

Double precision real.

Result Value

The number of seconds between consecutive ticks of the operating system real-time clock.

Examples

```

USE omp_lib
DOUBLE PRECISION WTICKS
WTICKS = omp_get_wtick()
PRINT *, 'The clock ticks ', 10 / WTICKS, &
' times in 10 seconds.'

```

omp_get_wtime

The **omp_get_wtime** function returns a double precision value equal to the number of seconds since the initial value of the operating system real-time clock. This value is guaranteed not to change during execution of the program.

The value returned by the **omp_get_wtime** function is not consistent across all threads in the team.

Argument Type and Attributes

There are no arguments to this function.

Result Type and Attributes

Double precision real.

Result Value

The number of seconds since the initial value of the operating system real-time clock.

Examples

```
USE omp_lib
DOUBLE PRECISION START, END
START = omp_get_wtime()
! Work to be timed
END = omp_get_wtime()
PRINT *, 'Stuff took ', END - START, ' seconds.'
```

omp_in_parallel

The **omp_in_parallel** function returns **.TRUE.** if you call it from the dynamic extent of a region executing in parallel and returns **.FALSE.** otherwise. If you call **omp_in_parallel** from a region that is serialized but nested within the dynamic extent of a region executing in parallel, the function will still return **.TRUE.**. (Nested parallel regions are serialized by default. See “omp_set_nested” on page 87 and the environment variable **OMP_NESTED** for more information.)

Result Type and Attributes

Default logical.

Result Value

.TRUE. if called from the dynamic extent of a region executing in parallel. **.FALSE.** otherwise.

Examples

```
USE omp_lib
INTEGER N, M
N = 4
M = 3
PRINT*, omp_in_parallel()
!$OMP PARALLEL DO
DO I = 1,N
!$OMP PARALLEL DO
DO J=1, M
PRINT *, omp_in_parallel()
```

```

        END DO
!$OMP  END PARALLEL DO
        END DO
!$OMP END PARALLEL DO

```

The first call to `omp_in_parallel` returns `.FALSE.` because the call is outside the dynamic extent of any parallel region. The second call returns `.TRUE.`, even if the nested `PARALLEL DO` loop is serialized, because the call is still inside the dynamic extent of the outer `PARALLEL DO` loop.

omp_init_lock

The `omp_init_lock` subroutine initializes a lock and associates it with the lock variable passed in as a parameter. After the call to `omp_init_lock`, the initial state of the lock variable is unlocked.

Note: If you call this routine with a lock variable that you have already initialized, the result of the call is undefined.

Argument Type and Attributes

Integer of kind `omp_lock_kind`.

Examples

```

USE omp_lib
INTEGER(kind=omp_lock_kind) LCK
INTEGER ID
CALL omp_init_lock(LCK)
!$OMP PARALLEL SHARED(LCK), PRIVATE(ID)
  ID = omp_get_thread_num()
  CALL omp_set_lock(LCK)
  PRINT *, 'MY THREAD ID IS', ID
  CALL omp_unset_lock(LCK)
!$OMP END PARALLEL
CALL omp_destroy_lock(LCK)

```

In the above example, one at a time, the threads gain ownership of the lock associated with the lock variable `LCK`, print the thread ID, and release ownership of the lock.

omp_init_nest_lock

The `omp_init_nest_lock` subroutine allows you to initialize a nestable lock and associate it with the lock variable you specify. The initial state of the lock variable is unlocked, and the initial nesting count is zero. The value of `nvar` must be an uninitialized nestable lock variable.

Note: If you call `omp_init_nest_lock` using a variable that is already initialized, the result is undefined.

Argument Type and Attributes

Integer of kind `omp_nest_lock_kind`.

Examples

```

USE omp_lib
INTEGER(kind=omp_nest_lock_kind) LCK
CALL omp_init_nest_lock(LCK)

```

For an example of how to use `omp_init_nest_lock`, see “`omp_set_nest_lock`” on page 87.

`omp_set_dynamic`

The `omp_set_dynamic` subroutine enables or disables dynamic adjustment, by the run-time environment, of the number of threads available to execute parallel regions.

If you call `omp_set_dynamic` with a *scalar_logical_expression* that evaluates to `.TRUE.`, the run-time environment can automatically adjust the number of threads that are used to execute subsequent parallel regions to obtain the best use of system resources. The number of threads you specify using `omp_set_num_threads` becomes the maximum, not exact, thread count.

If you call the subroutine with a *scalar_logical_expression* which evaluates to `.FALSE.`, dynamic adjustment of the number of threads is disabled. The run-time environment cannot automatically adjust the number of threads used to execute subsequent parallel regions. The value you pass to `omp_set_num_threads` becomes the exact thread count.

By default, dynamic thread adjustment is enabled. If your code depends on a specific number of threads for correct execution, you should explicitly disable dynamic threads.

Note: The number of threads remains fixed for each parallel region. The `omp_get_num_threads` function returns that number.

This subroutine has precedence over the `OMP_DYNAMIC` environment variable.

Result Type and Attributes

Default logical.

`omp_set_lock`

The `omp_set_lock` subroutine forces the calling thread to wait until the specified lock is available before executing subsequent instructions. The calling thread is given ownership of the lock when it becomes available.

Note: If you call this routine with a lock variable that has not been initialized, the result of the call is undefined. Also, if a thread that owns a lock tries to lock it again by issuing a call to `omp_set_lock`, it will produce a deadlock.

Argument Type and Attributes

`omp_lock_kind`.

Examples

```
USE omp_lib
INTEGER A(100)
INTEGER(kind=omp_lock_kind) LCK_X
CALL omp_init_lock (LCK_X)
!$OMP PARALLEL PRIVATE (I), SHARED (A, X)
!$OMP DO
DO I = 3, 100
  A(I) = I * 10
  CALL omp_set_lock (LCK_X)
```

```

        X = X + A(I)
        CALL omp_unset_lock (LCK_X)
    END DO
!$OMP END DO
!$OMP END PARALLEL
    CALL omp_destroy_lock (LCK_X)

```

In this example, the lock variable LCK_X is used to avoid race conditions when updating the shared variable X. By setting the lock before each update to X and unsetting it after the update, you ensure that only one thread is updating X at a given time.

omp_set_nested

The **omp_set_nested** subroutine enables or disables nested parallelism.

If you call the subroutine with a *scalar_logical_expression* that evaluates to **.FALSE.**, nested parallelism is disabled. Nested parallel regions are serialized, and they are executed by the current thread. This is the default setting.

If you call the subroutine with a *scalar_logical_expression* that evaluates to **.TRUE.**, nested parallelism is enabled. Parallel regions that are nested can deploy additional threads to the team. It is up to the run-time environment to determine whether additional threads should be deployed. Therefore, the number of threads used to execute parallel regions may vary from one nested region to the next.

This subroutine takes precedence over the **OMP_NESTED** environment variable.

Result Type and Attributes

Default logical.

omp_set_nest_lock

The **omp_set_nest_lock** subroutine allows you to set a nestable lock. The thread executing the subroutine will wait until the lock becomes available and then set that lock, incrementing the nesting count. A nestable lock is available if it is owned by the thread executing the subroutine, or is unlocked.

Argument Type and Attributes

Integer of kind **omp_nest_lock_kind**.

Examples

```

USE omp_lib
INTEGER P
INTEGER A
INTEGER B
INTEGER ( kind=omp_nest_lock_kind ) LCK

CALL omp_init_nest_lock ( LCK )

!$OMP PARALLEL SECTIONS
!$OMP SECTION
CALL omp_set_nest_lock ( LCK )
P = P + A
CALL omp_set_nest_lock ( LCK )
P = P + B
CALL omp_unset_nest_lock ( LCK )
CALL omp_unset_nest_lock ( LCK )

```

OpenMP Routines

```
!$OMP SECTION
CALL omp_set_nest_lock ( LCK )
P = P + B
CALL omp_unset_nest_lock ( LCK )
!$OMP END PARALLEL SECTIONS

CALL omp_destroy_nest_lock ( LCK )
END
```

omp_set_num_threads

The **omp_set_num_threads** subroutine tells the run-time environment how many threads to use in the next parallel region. The *scalar_integer_expression* that you pass to the subroutine is evaluated, and its value is used as the number of threads. If you have enabled dynamic adjustment of the number of threads (see “omp_set_dynamic” on page 86), **omp_set_num_threads** sets the maximum number of threads to use for the next parallel region. The run-time environment then determines the exact number of threads to use. However, when dynamic adjustment of the number of threads is disabled, **omp_set_num_threads** sets the exact number of threads to use in the next parallel region.

This subroutine takes precedence over the **OMP_NUM_THREADS** environment variable.

Note: If you call this subroutine from the dynamic extent of a region executing in parallel, the behavior of the subroutine is undefined.

Argument Type and Attributes

Integer.

omp_test_lock

The **omp_test_lock** function attempts to set the lock associated with the specified lock variable. It returns **.TRUE.** if it was able to set the lock and **.FALSE.** otherwise. In either case, the calling thread will continue to execute subsequent instructions in the program.

Note: If you call **omp_test_lock** with a lock variable that has not yet been initialized, the result of the call is undefined.

Result Type and Attributes

Default logical.

Argument Type and Attributes

Integer of kind **omp_lock_kind**.

Result Value

.TRUE. if the function was able to set the lock. **.FALSE.** otherwise.

Examples

```
USE omp_lib
INTEGER LCK
INTEGER ID
CALL omp_init_lock (LCK)
!$OMP PARALLEL SHARED(LCK), PRIVATE(ID)
  ID = omp_get_thread_num()
```

```

DO WHILE (.NOT. omp_test_lock(LCK))
  CALL WORK_A (ID)
END DO
CALL WORK_B (ID)
CALL omp_unset_lock (LCK)
!$OMP END PARALLEL
CALL omp_destroy_lock (LCK)

```

In this example, a thread repeatedly executes `WORK_A` until it can set `LCK`, the lock variable. When it succeeds in setting the lock variable, it executes `WORK_B`.

omp_test_nest_lock

The `omp_test_nest_lock` subroutine allows you to attempt to set a lock using the same method as `omp_set_nest_lock` but the execution thread does not wait for confirmation that the lock is available. If the lock is successfully set, the function will increment the nesting count. If the lock is unavailable the function returns a value of zero. The result value is always a default integer.

Argument Type and Attributes

Integer of kind `omp_nest_lock_kind`.

Result Value

`.TRUE.` if the function was able to set the lock. `.FALSE.` otherwise.

omp_unset_lock

This subroutine causes the executing thread to release ownership of the specified lock. The lock can then be set by another thread as required.

Note: The behavior of the `omp_unset_lock` subroutine is undefined if either:

- The calling thread does not own the lock specified, or
- The routine is called with a lock variable that has not been initialized.

Result Type and Attributes

Integer of kind `omp_lock_kind`.

Examples

For an example of how to use `omp_unset_lock`, see “`omp_set_lock`” on page 86.

omp_unset_nest_lock

The `omp_unset_nest_lock` subroutine allows you to release ownership of a nestable lock. The subroutine decrements the nesting count and releases the associated thread from ownership of the nestable lock.

Result Type and Attributes

Integer of kind `omp_nest_lock_kind`.

Examples

For an example of how to use `omp_unset_nest_lock`, see “`omp_set_nest_lock`” on page 87.

Trademarks and Service Marks

The following terms, used in this publication, are trademarks or service marks 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.

IBM