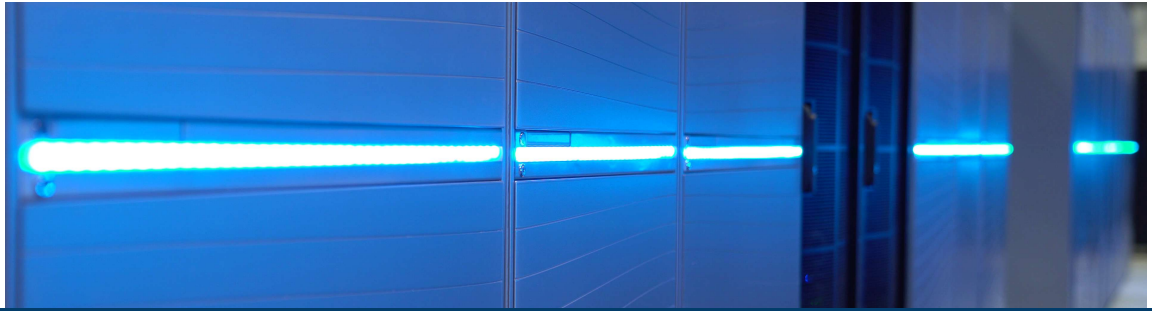
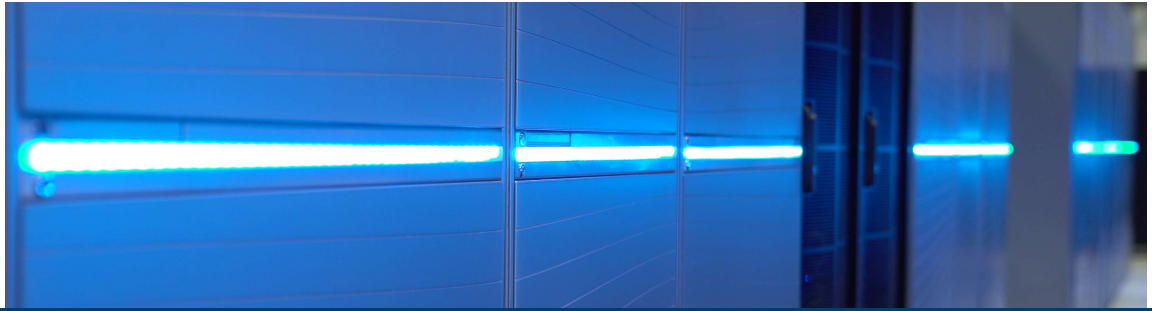


INTRODUCTION TO PARALLEL PROGRAMMING WITH MPI AND OPENMP

March 18-20 2024 | Junxian Chew, Michael Knobloch, Ilya Zhukov, Jolanta Zjupa | Jülich Supercomputing Centre



Part I: Introduction to OpenMP

WHAT IS OPENMP?

Open specifications for **Multi-Processing** (not implementations)

API (Application Program Interface) for shared memory, explicit, thread based parallelism.

Goals of OpenMP:

- Standardization
- Ease of Use
- Portability (across different platforms)

Three main API components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables

Current version of the specification: 5.2 (November 2021)
`https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf`

BRIEF HISTORY

1997 FORTRAN version 1.0

1998 C/C++ version 1.0

1999 FORTRAN version 1.1

2000 FORTRAN version 2.0

2002 C/C++ version 2.0

2005 First combined version 2.5, memory model, internal control variables, clarifications

2008 Version 3.0, tasks

2011 Version 3.1, extended task facilities

2013 Version 4.0, thread affinity, SIMD, devices, tasks (dependencies, groups, and cancellation), improved Fortran 2003 compatibility

2015 Version 4.5, extended SIMD and devices facilities, task priorities

2018 Version 5.0, memory model, base language compatibility, allocators, extended task and devices facilities

2020 Version 5.1, support for newer base languages, loop transformations, compare-and-swap, extended devices facilities

2021 Version 5.2, reorganization of the specification and improved consistency

LITERATURE

Official Resources

- OpenMP Architecture Review Board. OpenMP Application Programming Interface. Version 5.2. Nov. 2021. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf
- OpenMP Architecture Review Board. OpenMP Application Programming Interface. Examples. Version 5.1. Aug. 2021. URL: https://www.openmp.org/wp-content/uploads/openmp-examples-5.1.pdf
- https://www.openmp.org

Recommended by https://www.openmp.org/resources/openmp-books/

- Michael Klemm and Jim Cownie. High Performance Parallel Runtimes. De Gruyter Oldenbourg, 2021. ISBN: 9783110632729. DOI: doi:10.1515/9783110632729
- Timothy G. Mattson, Yun He, and Alice E. Koniges. The OpenMP Common Core. Making OpenMP Simple Again. 1st ed. The MIT Press, Nov. 19, 2019. 320 pp. ISBN: 9780262538862
- Ruud van der Pas, Eric Stotzer, and Christian Terboven. Using OpenMP—The Next Step. Affinity, Accelerators, Tasking, and SIMD. 1st ed. The MIT Press, Oct. 13, 2017. 392 pp. ISBN: 9780262534789

Additional Literature

- Michael McCool, James Reinders, and Arch Robison. Structured Parallel Programming. Patterns for Efficient Computation. 1st ed. Morgan Kaufmann, July 31, 2012. 432 pp. ISBN: 9780124159938

LITERATURE

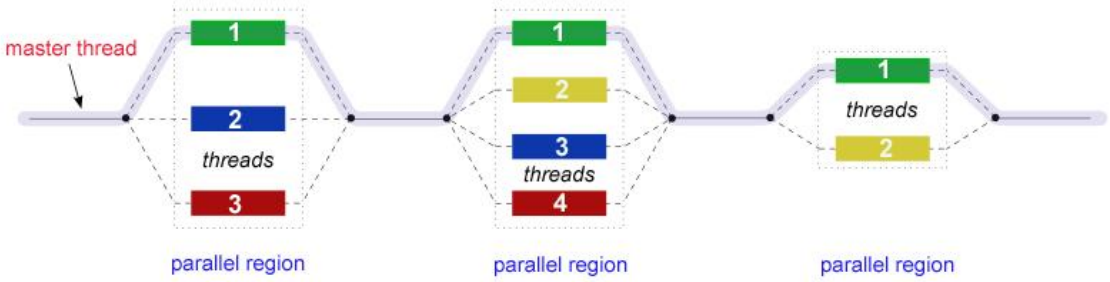
Online Tutorials

- `https://hpc-tutorials.llnl.gov/openmp/`

Older Works (`https://www.openmp.org/resources/openmp-books/`)

- Barbara Chapman, Gabriele Jost, and Ruud van der Pas. Using OpenMP. Portable Shared Memory Parallel Programming. 1st ed. Scientific and Engineering Computation. The MIT Press, Oct. 12, 2007. 384 pp. ISBN: 9780262533027
- Rohit Chandra et al. Parallel Programming in OpenMP. 1st ed. Morgan Kaufmann, Oct. 11, 2000. 231 pp. ISBN: 9781558606715
- Michael Quinn. Parallel Programming in C with MPI and OpenMP. 1st ed. McGraw-Hill, June 5, 2003. 544 pp. ISBN: 9780072822564
- Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. Patterns for Parallel Programming. 1st ed. Software Patterns. Sept. 15, 2004. 384 pp. ISBN: 9780321228116

OPENMP PROGRAMMING MODEL: FORK - JOIN



Thread Based - Explicit - Nested Parallelism - Dynamic Threads - *no* I/O

THREADS & TASKS

Thread

Smallest sequence of programmed instructions or an execution entity that can be managed independently by a scheduler (which is typically a part of the operating system).

OpenMP Thread

A thread that is managed by the OpenMP runtime system.

Team

A set of one or more threads participating in the execution of a `parallel` region.

Task

A specific instance of executable code and its data environment that the OpenMP imlementation can schedule for execution by threads.

PROGRAM

Base Language

A programming language that serves as the foundation of the OpenMP specification.

The following base languages are given in [OpenMP-5.2, 1.7]: C90, C99, C11, C18, C++98, C++11, C++14, C++17, C++20, Fortran 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008, and a subset of Fortran 2018

Base Program

A program written in the base language.

OpenMP Program

A program that consists of a base program that is annotated with OpenMP directives or that calls OpenMP API runtime library routines.

Internal Control Variables (ICVs)

ICVs are initialized by the implementation. They can be set through OpenMP environment variables, through OpenMP runtime routines, and through directive clauses. The OpenMP program can retrieve the values of ICVs only through OpenMP runtime routines.

COMPILING & LINKING

Compilers that conform to the OpenMP specification usually accept a command line argument that turns on OpenMP support, e.g.:

Intel C Compiler OpenMP Command Line Switch

```
$ icc -qopenmp ...
```

GNU C Compiler OpenMP Command Line Switch

```
$ gcc -fopenmp ...
```

GNU Fortran Compiler OpenMP Command Line Switch

```
$ gfortran -fopenmp ...
```

The name of the command line argument is not mandated by the specification and differs from one compiler to another.

RUNTIME LIBRARY DEFINITIONS

C/C++ Runtime Library Definitions

Runtime library routines and associated types are defined in the <omp.h> header file.

C
```c
#include <omp.h>
```

Fortran Runtime Library Definitions

Runtime library routines and associated types are defined in either a Fortran **include** file

F77
```
include "omp_lib.h"
```

or a Fortran 90 module

F08
```
use omp_lib
```

RUNTIME LIBRARY ROUTINES

OpenMP runtime library routines are used for a variety of purposes, a.o. to set or retrieve ICVs.

- All OpenMP runtime routine names start with a lower case *omp_*

omp_get_num_threads()
Returns the number of threads that constitute the team executing a parallel region from which this routine is called.

omp_set_num_threads()
Sets the number of threads that will be used in the following parallel region(s).

omp_get_thread_num()
Returns the thread number of the thread within a team calling this routine.

ENVIRONMENT VARIABLES

OpenMP environment variables set ICVs of an OpenMP programs in the shell:

csh/tcsh

```
$ setenv ENV_VAR {NUM}
```

sh/bash

```
$ export ENV_VAR={NUM}
```

- Names of the environment variables must be upper case
- values given to environment variables are case insensitive and may have leading and trailing white space

OMP_NUM_THREADS
Sets number of threads to use in the OpenMP program.

Modifications to environment variables after the OpenMP program has started are ignored by the OpenMP implementation. ICVs can be however changed through directive clauses and OpenMP runtime routines.

C AND C++ DIRECTIVE FORMAT

C

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

- Directives are case-sensitive
- Only one directive-name can be specified per directive
- Each directive applies to the next statement which must be a structured block
- Clauses are optional and can be in any order
- Newline is required, and precedes the structured block which is enclosed by the directive.

Structured Block

An executable statement, possibly compound, with a single entry at the top and a single exit at the bottom, or an OpenMP construct.

FORTRAN DIRECTIVE FORMAT

F08
```
sentinel   directive-name   [clause ...]
```

- Directives are case-insensitive

Fixed Form Sentinels

F08
```
sentinel = !$OMP | C$OMP | *$OMP
```

- Must start in column 1
- The usual line length, white space, continuation and column rules apply
- Column 6 is blank for first line of directive, non-blank and non-zero for continuation

Free Form Sentinel

F08
```
sentinel = !$OMP
```

- The usual line length, white space and continuation rules apply

DIRECTIVES: PARALLEL REGION CONSTRUCT

The fundamental OpenMP: A parallel region is a block of code that will be executed by multiple threads.

C

```
#pragma omp   parallel   [clause ...]   newline
  structured_block
```

F08

```
!$OMP   PARALLEL   [clause ...]
  structured_block
!$omp end parallel
```

- Thread that reaches a `parallel` directive creates a team of threads and becomes the master of the team
  Creates a team of threads to execute the `parallel` region
- The code is duplicated and all threads in the team will execute the code contained in the structured block
- Inside the region threads are identified by consecutive numbers starting at zero
- There is an implied barrier at the end of a parallel section, only the master thread continues past this point
- Optional clauses (explained later) can be used to modify behaviour and data environment of the `parallel` region

A FIRST OPENMP PROGRAM

C

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

int main(void) {
  printf("Hello from your main thread.\n");

  #pragma omp parallel
    printf("Hello from thread %d of %d.\n", omp_get_thread_num(),
     ↪  omp_get_num_threads());

  printf("Hello again from your main thread.\n");
}
```

A FIRST OPENMP PROGRAM

Program Output

```
$ gcc -fopenmp -o hello_openmp.x hello_openmp.c
$ ./hello_openmp.x
Hello from your main thread.
Hello from thread 1 of 8.
Hello from thread 0 of 8.
Hello from thread 3 of 8.
Hello from thread 4 of 8.
Hello from thread 6 of 8.
Hello from thread 7 of 8.
Hello from thread 2 of 8.
Hello from thread 5 of 8.
Hello again from your main thread.
```

A FIRST OPENMP PROGRAM

F08

```
program hello_openmp
  use omp_lib
  implicit none

  print *, "Hello from your main thread."

  !$omp parallel
  print *, "Hello from thread ", omp_get_thread_num(), " of ",
   ↪  omp_get_num_threads(), "."
  !$omp end parallel

  print *, "Hello again from your main thread."
end program
```

PARALLEL CONTROL FLOW (IN OPENMP)

Thread 0

```
program hello_openmp
  print *, "Hello..."
  !$omp parallel
  print *, "Hello..."
  !$omp end parallel
  print *, "Hello..."
end program
```

Console

PARALLEL CONTROL FLOW (IN OPENMP)

Thread 0

```
program hello_openmp
  print *, "Hello..."
  !$omp parallel
  print *, "Hello..."
  !$omp end parallel
  print *, "Hello..."
end program
```

Console

```
Hello from your main thread.
```

PARALLEL CONTROL FLOW (IN OPENMP)

Thread 0

```
program hello_openmp
  print *, "Hello..."
  !$omp parallel
  print *, "Hello..."
  !$omp end parallel
  print *, "Hello..."
end program
```

Thread 1

```
program hello_openmp
  print *, "Hello..."
  !$omp parallel
  print *, "Hello..."
  !$omp end parallel
  print *, "Hello..."
end program
```

Console

```
Hello from your main thread.
```

PARALLEL CONTROL FLOW (IN OPENMP)

Thread 0

```
program hello_openmp
  print *, "Hello..."
  !$omp parallel
  print *, "Hello..."
  !$omp end parallel
  print *, "Hello..."
end program
```

Thread 1

```
program hello_openmp
  print *, "Hello..."
  !$omp parallel
  print *, "Hello..."
  !$omp end parallel
  print *, "Hello..."
end program
```

Console

```
Hello from your main thread.
Hello from thread 1 of 2.
```

PARALLEL CONTROL FLOW (IN OPENMP)

Thread 0

```
program hello_openmp
  print *, "Hello..."
  !$omp parallel
  print *, "Hello..."
  !$omp end parallel
  print *, "Hello..."
end program
```

Thread 1

```
program hello_openmp
  print *, "Hello..."
  !$omp parallel
  print *, "Hello..."
  !$omp end parallel
  print *, "Hello..."
end program
```

Console

```
Hello from your main thread.
Hello from thread 1 of 2.
Hello from thread 0 of 2.
```

PARALLEL CONTROL FLOW (IN OPENMP)

Thread 0

```
program hello_openmp
  print *, "Hello..."
  !$omp parallel
  print *, "Hello..."
  !$omp end parallel
  print *, "Hello..."
end program
```

Thread 1

```
program hello_openmp
  print *, "Hello..."
  !$omp parallel
  print *, "Hello..."
  !$omp end parallel
  print *, "Hello..."
end program
```

Console

```
Hello from your main thread.
Hello from thread 1 of 2.
Hello from thread 0 of 2.
```

PARALLEL CONTROL FLOW (IN OPENMP)

Thread 0

```
program hello_openmp
  print *, "Hello..."
  !$omp parallel
  print *, "Hello..."
  !$omp end parallel
  print *, "Hello..."
end program
```

Console

```
Hello from your main thread.
Hello from thread 1 of 2.
Hello from thread 0 of 2.
Hello again from your main thread.
```

SETTING NUMBER OF THREADS

// ENVIRONMENTAL VARIABLE

```
$ export OMP_NUM_THREADS={NUM}
```

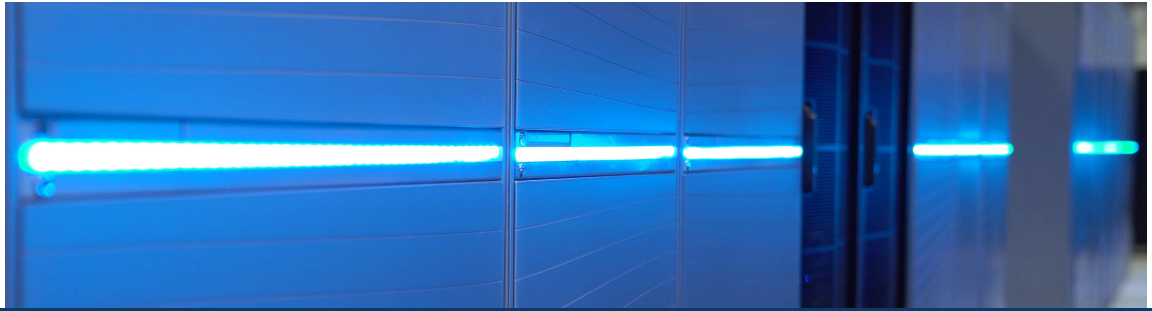
C

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

int main(){
  printf("master thread: hello world.\n");

  omp_set_num_threads({NUM}); // RUNTIME LIBRARY ROUTINE
  #pragma omp parallel num_threads({NUM}) // DIRECTIVE CLAUSE
    printf("thread %d out of %d threads: hello world. \n",
      ↪  omp_get_thread_num(),omp_get_num_threads());

  return(0);
}
```



Part II: Low-Level OpenMP Concepts

DATA-SHARING ATTRIBUTES [OpenMP-5.2, 5.1]

Variable

A named data storage block, for which the value can be defined and redefined during the execution of a program.

Private Variable

With respect to a given set of task regions that bind to the same `parallel` region, a variable for which the name provides access to a **different** block of storage for each task region.

Shared Variable

With respect to a given set of task regions that bind to the same `parallel` region, a variable for which the name provides access to the **same** block of storage for each task region.

DATASHARING ATTRIBUTE CLAUSES

PRIVATE Clause declares variables in its list to be private to each thread. A new object of the same type is declared and referenced once for each thread in the team. Private variables should be assumed to be uninitialized for each thread. *C/C++: private(list), F: PRIVATE(list).*

FIRSTPRIVATE Clause equals the private clause with automatic initialization of the listed variables accoding to the value of their original objects prior to entry into the parallel or work-sharing construct. *C/C++: firstprivate(list), F: FIRSTPRIVATE(list).*

LASTPRIVATE Clause equals the private clause with a copy from the last loop iteration or section to the original variable object. *C/C++: lastprivate(list), F: LASTPRIVATE(list).*

SHARED Clause declares variables in its list to be shared among all threads in the team. A shared variable exists in only one memory location and all threads can read or write to that address. *C/C++: shared(list), F: SHARED(list).*

DEFAULT Clause specifies a default scope for all variables in the lexical extent of any parallel region.
C/C++: default (shared | none), F: DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE).

REDUCTION CLAUSE [OpenMP-5.2, 5.5.8]

```
* reduction(reduction-identifier : list)
```

- Listed variables are declared private.
- At the end of the construct, the original variable is updated by combining the private copies using the operation given by `reduction-identifier`.
- `reduction-identifier` may be `+`, `-`, `*`, `&`, `|`, `^`, `&&`, `||`, `min` or `max` (C and C++) or an identifier (C) or an id-expression (C++)
- `reduction-identifier` may be a base language identifier, a user-defined operator, or one of `+`, `-`, `*`, `.and.`, `.or.`, `.eqv.`, `.neqv.`, `max`, `min`, `iand`, `ior` or `ieor` (Fortran)
- Private versions of the variable are initialized with appropriate values

THREAD SYNCHRONIZATION

- In MPI, exchange of data between processes implies synchronization through the message metaphor.
- In OpenMP, threads exchange data through shared parts of memory.
- Explicit synchronization is needed to coordinate access to shared memory.

Data Race

A data race occurs when

- multiple threads write to the same memory unit without synchronization or
- at least one thread writes to and at least one thread reads from the same memory unit without synchronization.

- Data races result in unspecified program behavior.
- OpenMP offers several synchronization mechanism which range from high-level/general to low-level/specialized.

THE BARRIER CONSTRUCT [OpenMP-5.2, 15.3.1]

C
```
#pragma omp barrier
```

F08
```
!$omp barrier
```

- Threads are only allowed to continue execution of code after the `barrier` once all threads in the current team have reached the `barrier`.
- A barrier region must be executed by all threads in the current team or none.

BARRIER CONTROL FLOW

Thread 0

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

Thread 1

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

BARRIER CONTROL FLOW

Thread 0

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

Thread 1

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

BARRIER CONTROL FLOW

Thread 0

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

Thread 1

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

BARRIER CONTROL FLOW

Thread 0

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

Thread 1

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

BARRIER CONTROL FLOW

Thread 0

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

Thread 1

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

BARRIER CONTROL FLOW

Thread 0

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

Thread 1

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

BARRIER CONTROL FLOW

Thread 0

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

Thread 1

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

BARRIER CONTROL FLOW

Thread 0

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

Thread 1

```
program hello_barrier
  ...
  statement1
  !$omp barrier
  statement2
  ...
end program
```

THE CRITICAL CONSTRUCT [OpenMP-5.2, 15.2]

C

```
#pragma omp critical [(name)]
  structured-block
```

F08

```
!$omp critical [(name)]
  structured-block
!$omp end critical [(name)]
```

- Execution of `critical` regions with the same name are restricted to one thread at a time.
- name is a compile time constant.
- In C, names live in their own name space.
- In Fortran, names of critical regions can collide with other identifiers.

CRITICAL CONTROL FLOW

Thread 0

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Thread 1

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Console

CRITICAL CONTROL FLOW

Thread 0

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Thread 1

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Console

CRITICAL CONTROL FLOW

Thread 0

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Thread 1

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Console

CRITICAL CONTROL FLOW

Thread 0

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Thread 1

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Console

```
Hello from thread 1 of 2.
```

CRITICAL CONTROL FLOW

Thread 0

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Thread 1

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Console

```
Hello from thread 1 of 2.
Again, hello from thread 1 of 2.
```

CRITICAL CONTROL FLOW

Thread 0

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Thread 1

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Console

```
Hello from thread 1 of 2.
Again, hello from thread 1 of 2.
```

CRITICAL CONTROL FLOW

Thread 0

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Thread 1

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Console

```
Hello from thread 1 of 2.
Again, hello from thread 1 of 2.
Hello from thread 0 of 2.
```

CRITICAL CONTROL FLOW

Thread 0

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Thread 1

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Console

```
Hello from thread 1 of 2.
Again, hello from thread 1 of 2.
Hello from thread 0 of 2.
Again, hello from thread 0 of 2.
```

CRITICAL CONTROL FLOW

Thread 0

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Thread 1

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Console

```
Hello from thread 1 of 2.
Again, hello from thread 1 of 2.
Hello from thread 0 of 2.
Again, hello from thread 0 of 2.
```

CRITICAL CONTROL FLOW

Thread 0

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Thread 1

```
program hello_critical
  ...
  statement1
  !$omp critical
  print *, "Hello..."
  print *, "Again..."
  !$omp end critical
  statement2
end program
```

Console

```
Hello from thread 1 of 2.
Again, hello from thread 1 of 2.
Hello from thread 0 of 2.
Again, hello from thread 0 of 2.
```

ORDERED DIRECTIVE

The `ordered` directive specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor.

C

```
#pragma omp   for   ordered [clauses...]
   (loop region)
   #pragma omp   ordered   newline
      structured_block
   (endo of loop region)
```

F08

```
!$OMP   DO   ORDERED [clauses...]
   (loop region)
   !$OMP   ORDERED
      structured_block
   !$OMP END   ORDERED
   (end of loop region)
!$OMP END   DO
```

ORDERED DIRECTIVE

The `ordered` directive specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor.

C

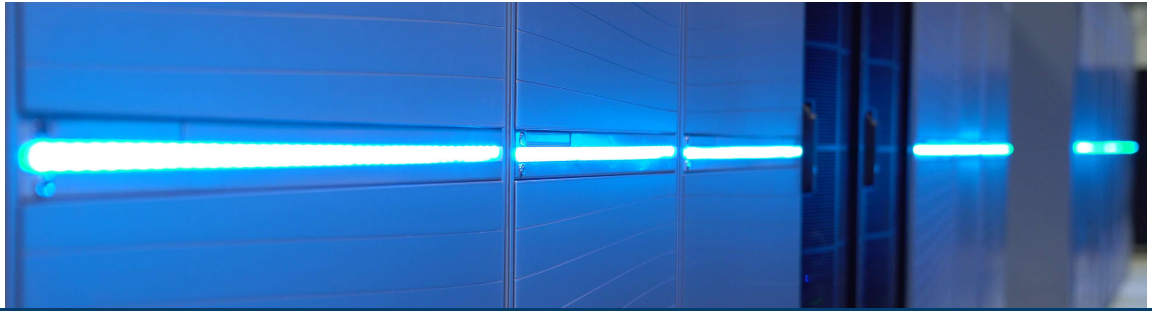
```
#pragma omp   for   ordered [clauses...]
   (loop region)
   #pragma omp   ordered   newline
      structured_block
   (endo of loop region)
```

An `ordered` directive can only appear in the dynamic extent of the for or parallel for (C/C++) directives and equivalently DO or PARALLEL DO (Fortran) directives.

A loop which contains an `ordered` ***directive***, must be a loop with an `ordered` ***clause***.

THE ATOMIC AND FLUSH CONSTRUCTS [OpenMP-5.2, 15.8.4, 15.8.5]

- `barrier`, `critical`, and locks implement synchronization between general blocks of code
- If blocks become very small, synchronization overhead could become an issue
- The `atomic` and `flush` constructs implement low-level, fine grained synchronization for certain limited operations on scalar variables:
  - `read`
  - `write`
  - `update`, writing a new value based on the old value
  - `capture`, like update and the old or new value is available in the subsequent code
- Correct use requires knowledge of the OpenMP Memory Model [OpenMP-5.2, 1.4]
- See also: C11 and C++11 Memory Models



Part III: Worksharing

WORKSHARING CONSTRUCTS

- Decompose work for concurrent execution by multiple threads
- Used inside `parallel` regions
- Available worksharing constructs:
  - `single` and `sections` construct
  - loop construct
  - `workshare` construct
  - `task` worksharing

THE SINGLE CONSTRUCT [OpenMP-5.2, 11.1]

C

```
#pragma omp single [clause[[,] clause]...]
  structured-block
```

F08

```
!$omp single [clause[[,] clause]...]
  structured-block
!$omp end single [end_clause[[,] end_clause]...]
```

- The structured block is executed by a single thread in the encountering team.
- Permissible clauses are `firstprivate`, `private`, `copyprivate` and `nowait`.
- `nowait` and `copyprivate` are `end_clauses` in Fortran.

SINGLE CONTROL FLOW

Thread 0

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single
  print *, "Again..."
  !$omp end parallel
end program
```

Thread 1

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single
  print *, "Again..."
  !$omp end parallel
end program
```

Console

SINGLE CONTROL FLOW

Thread 0

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single
  print *, "Again..."
  !$omp end parallel
end program
```

Thread 1

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single
  print *, "Again..."
  !$omp end parallel
end program
```

Console

SINGLE CONTROL FLOW

Thread 0

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single
  print *, "Again..."
  !$omp end parallel
end program
```

Thread 1

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single
  print *, "Again..."
  !$omp end parallel
end program
```

Console

```
Hello from thread 1 of 2.
```

SINGLE CONTROL FLOW

Thread 0

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single
  print *, "Again..."
  !$omp end parallel
end program
```

Thread 1

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single
  print *, "Again..."
  !$omp end parallel
end program
```

Console

```
Hello from thread 1 of 2.
```

SINGLE CONTROL FLOW

Thread 0

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single
  print *, "Again..."
  !$omp end parallel
end program
```

Thread 1

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single
  print *, "Again..."
  !$omp end parallel
end program
```

Console

```
Hello from thread 1 of 2.
Again, hello from thread 0 of 2.
```

SINGLE CONTROL FLOW

Thread 0

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single
  print *, "Again..."
  !$omp end parallel
end program
```

Thread 1

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single
  print *, "Again..."
  !$omp end parallel
end program
```

Console

```
Hello from thread 1 of 2.
Again, hello from thread 0 of 2.
Again, hello from thread 1 of 2.
```

SINGLE CONTROL FLOW

Thread 0

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single
  print *, "Again..."
  !$omp end parallel
end program
```

Thread 1

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single
  print *, "Again..."
  !$omp end parallel
end program
```

Console

```
Hello from thread 1 of 2.
Again, hello from thread 0 of 2.
Again, hello from thread 1 of 2.
```

WORK SHARING CONSTRUCT: SECTIONS DIRECTIVE

The `sections` directive specifies that the enclosed section(s) of code are divided among the threads of a team.

C

```
#pragma omp   sections   [clause ...]   newline
  {
    #pragma omp   section   newline
      structured_block
    #pragma omp   section   newline
      structured_block
  }
```

F08

```
!$OMP   SECTIONS   [clause ...]
!$OMP  SECTION
  structured_block
!$OMP  SECTION
  structured_block
!$OMP END   SECTIONS
```

WORK SHARING CONSTRUCT: SECTIONS DIRECTIVE

The `sections` directive specifies that the enclosed section(s) of code are divided among the threads of a team.

C

```
#pragma omp   sections   [clause ...]   newline
  {
    #pragma omp   section   newline
      structured_block
    #pragma omp   section   newline
      structured_block
  }
```

- Multiple `section` directives are nested within a `sections` directive.
- Calculations done in the individual `section`(s) must be independent(!)
- Each `section` is executed by one thread in the team, a thread can execute more than one section.

Represents a type of functional parallelism.

IMPLICIT BARRIERS & THE NOWAIT CLAUSE [OpenMP-5.2, 15.3.2, 15.6]

- Worksharing constructs (and the `parallel` construct) contain an implied barrier at their exit.
- The `nowait` clause can be used on worksharing constructs to disable this implicit barrier.

SINGLE CONTROL FLOW

Thread 0

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single nowait
  print *, "Again..."
  !$omp end parallel
end program
```

Thread 1

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single nowait
  print *, "Again..."
  !$omp end parallel
end program
```

Console

SINGLE CONTROL FLOW

Thread 0

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single nowait
  print *, "Again..."
  !$omp end parallel
end program
```

Thread 1

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single nowait
  print *, "Again..."
  !$omp end parallel
end program
```

Console

SINGLE CONTROL FLOW

Thread 0

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single nowait
  print *, "Again..."
  !$omp end parallel
end program
```

Thread 1

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single nowait
  print *, "Again..."
  !$omp end parallel
end program
```

Console

```
Again, hello from thread 0 of 2.
Hello from thread 1 of 2.
```

SINGLE CONTROL FLOW

Thread 0

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single nowait
  print *, "Again..."
  !$omp end parallel
end program
```

Thread 1

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single nowait
  print *, "Again..."
  !$omp end parallel
end program
```

Console

```
Again, hello from thread 0 of 2.
Hello from thread 1 of 2.
Again, hello from thread 1 of 2.
```

SINGLE CONTROL FLOW

Thread 0

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single nowait
  print *, "Again..."
  !$omp end parallel
end program
```

Thread 1

```
program hello_single
  !$omp parallel
  !$omp single
  print *, "Hello..."
  !$omp end single nowait
  print *, "Again..."
  !$omp end parallel
end program
```

Console

```
Again, hello from thread 0 of 2.
Hello from thread 1 of 2.
Again, hello from thread 1 of 2.
```

THE COPYPRIVATE CLAUSE [OpenMP-5.2, 5.7.2]

```
*  copyprivate(list)
```

- list contains variables that are `private` in the enclosing parallel region.
- At the end of the `single` construct, the values of all list items on the single thread are copied to all other threads.
- E.g. serial initialization
- `copyprivate` cannot be combined with `nowait`.

WORKSHARING-LOOP CONSTRUCT [OpenMP-5.2, 11.5]

C

```
#pragma omp for [clause[[,] clause]...]
  for-loops
```

F08

```
!$omp do [clause[[,] clause]...]
  do-loops
[!$omp end do [nowait]]
```

Declares the iterations of a loop to be suitable for concurrent execution on multiple threads.

Data-environment clauses

- `private`
- `firstprivate`
- `lastprivate`
- `reduction`

Worksharing-Loop-specific clauses

- `schedule`
- `collapse`

WORKSHARING-LOOP CONTROL FLOW

Thread 0

```
...
!$omp parallel
!$omp do
do i = 1, 4
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Console

WORKSHARING-LOOP CONTROL FLOW

Thread 0

```
...
!$omp parallel
!$omp do
do i = 1, 4
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Console

WORKSHARING-LOOP CONTROL FLOW

Thread 0

```
...
!$omp parallel
!$omp do
do i = 1, 4
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Thread 1

```
...
!$omp parallel
!$omp do
do i = 1, 4
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Console

WORKSHARING-LOOP CONTROL FLOW

Thread 0

```
...
!$omp parallel
!$omp do
do i = 1, 2
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Thread 1

```
...
!$omp parallel
!$omp do
do i = 3, 4
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Console

WORKSHARING-LOOP CONTROL FLOW

Thread 0

```
...
!$omp parallel
!$omp do
do i = 1, 2
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Thread 1

```
...
!$omp parallel
!$omp do
do i = 3, 4
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Console

```
iteration 3 on thread 1
```

WORKSHARING-LOOP CONTROL FLOW

Thread 0

```
...
!$omp parallel
!$omp do
do i = 1, 2
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Thread 1

```
...
!$omp parallel
!$omp do
do i = 3, 4
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Console

```
iteration 3 on thread 1
iteration 1 on thread 0
```

WORKSHARING-LOOP CONTROL FLOW

Thread 0

```
...
!$omp parallel
!$omp do
do i = 1, 2
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Thread 1

```
...
!$omp parallel
!$omp do
do i = 3, 4
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Console

```
iteration 3 on thread 1
iteration 1 on thread 0
iteration 2 on thread 0
```

WORKSHARING-LOOP CONTROL FLOW

Thread 0

```
...
!$omp parallel
!$omp do
do i = 1, 2
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Thread 1

```
...
!$omp parallel
!$omp do
do i = 3, 4
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Console

```
iteration 3 on thread 1
iteration 1 on thread 0
iteration 2 on thread 0
iteration 4 on thread 1
```

WORKSHARING-LOOP CONTROL FLOW

Thread 0

```
...
!$omp parallel
!$omp do
do i = 1, 2
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Thread 1

```
...
!$omp parallel
!$omp do
do i = 3, 4
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Console

```
iteration 3 on thread 1
iteration 1 on thread 0
iteration 2 on thread 0
iteration 4 on thread 1
```

WORKSHARING-LOOP CONTROL FLOW

Thread 0

```
...
!$omp parallel
!$omp do
do i = 1, 2
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Thread 1

```
...
!$omp parallel
!$omp do
do i = 3, 4
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Console

```
iteration 3 on thread 1
iteration 1 on thread 0
iteration 2 on thread 0
iteration 4 on thread 1
```

WORKSHARING-LOOP CONTROL FLOW

Thread 0

```
...
!$omp parallel
!$omp do
do i = 1, 2
  print *, "iteration: ", i, ...
end do
!$omp end do
!$omp end parallel
...
```

Console

```
iteration 3 on thread 1
iteration 1 on thread 0
iteration 2 on thread 0
iteration 4 on thread 1
```

THE COLLAPSE CLAUSE [OpenMP-5.2, 4.4.3]

```
*  collapse(n)
```

- The `loop` directive applies to the outermost loop of a set of nested loops, by default
- `collapse(n)` extends the scope of the `loop` directive to the n outer loops
- All associated loops must be perfectly nested, i.e.:

```c
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < M; ++j) {
    // ...
  }
}
```

THE SCHEDULE CLAUSE [OpenMP-5.2, 11.5.3]

```
schedule(kind[, chunk_size])
```

Determines how the iteration space is divided into chunks and how these chunks are distributed among threads.

`static`	Divide iteration space into chunks of `chunk_size` iterations and distribute them in a round-robin fashion among threads. If `chunk_size` is not specified, chunk size is chosen such that each thread gets at most one chunk.
`dynamic`	Divide into chunks of size `chunk_size` (defaults to 1). When a thread is done processing a chunk it acquires a new one.
`guided`	Like dynamic but chunk size is adjusted, starting with large sizes for the first chunks and decreasing to `chunk_size` (default 1).
`auto`	Let the compiler and runtime decide.
`runtime`	Schedule is chosen based on ICV run-sched-var.

If no `schedule` clause is present, the default schedule is implementation defined.

COMBINED CONSTRUCTS [OpenMP-5.2, 17]

Some constructs that often appear as nested pairs can be combined into one construct, e.g.

C
```
#pragma omp parallel
#pragma omp for
for (...; ...; ...) {
  ...
}
```

can be turned into

C
```
#pragma omp parallel for
for (...; ...; ...) {
  ...
}
```

Similarly, `parallel` and `workshare` can be combined.
Combined constructs usually accept the clauses of either of the base constructs.

EXERCISES

Exercise 1 – Hello World

1.1 Hello World & Setting number of threads

Compile and execute the file `openmp_hello_world.{c|f90}`. It contains a ‘hello world’ from all available OpenMP threads.

How many OpenMP threads are used? Implement different ways to set the number of OpenMP threads. Which way overwrites which other? Make a hierarchical list.

EXERCISES

Exercise 2 – Manual work sharing

2.1 Manual work sharing

Compile and execute the file `openmp_ws_manual.{c|f90}`. It containes a small loop inside parallel region.

Implement manual worksharing using `omp_get_thread_num()`, `omp_get_num_threads()` and division operations. If the equal distribution is not possible assign reminder iterations to the last thread.

EXERCISES

Exercise 3 – Race Conditions

3.1 A Simple Sum

Compile and execute the file `openmp_simple_sum.{c|f90}`. It containes a serial version of a simple sum from 0 to `large_number`.

Implement an OpenMP parallelised version of the loop.

What result do you get? How long does it take? How does the runtime scale with the number of OpenMP threads and the value of `large_number` (degrees of freedom)?

EXERCISES

Exercise 4 – Race Conditions

4.1 A Simple Sum

Let's return to the example in `openmp_simple_sum.{c|f90}`. What changes can you do to the code to avoid the race condition?

What result do you get? How long does it take? How does the runtime scale with the number of OpenMP threads and the matrix `size` (degrees of freedom)?

4.2 Double Loop Tensor Contraction

Compile and execute the file `openmp_double_loop.{c|f90}`. It containes a serial version of a double contraction for a second-rank tensor initialised with random values between 0 and 1.

Implement an OpenMP parallelised version of the double loop.

What result do you get? How long does it take? How does the runtime scale with the number of OpenMP threads and the matrix `size` (degrees of freedom)?

EXERCISES

Exercise 5 – Data attribute clauses

5.1 Data attribute clauses

Compile and execute the file `openmp_data.{c|f90}`.

Try to understand what is wrong, which data should be shared/private.

What first/lastprivate suppose to do?

Set `default(none)` clause. What has changed?

Experiment!

EXERCISES

Exercise 6 – Scheduling

6.1 Scheduling

Compile and execute the file `openmp_scheduling.{c|f90}`.

Play with various variants, e.g. array size, number of threads, chunk size.

What was the fastest implementation? Can you explain results?