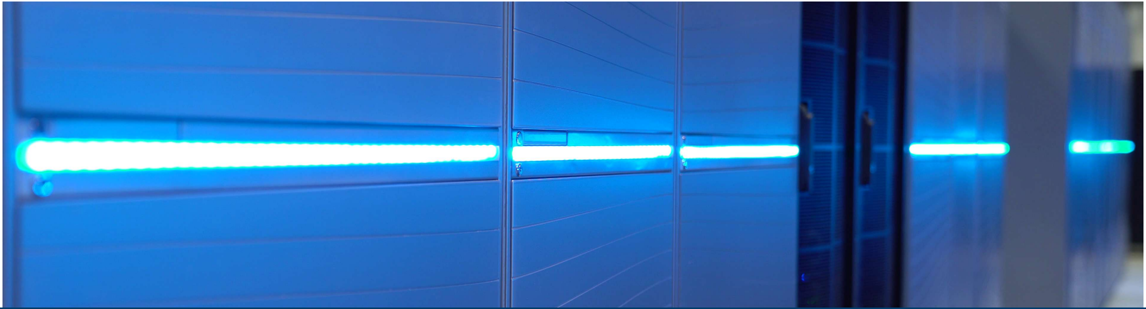
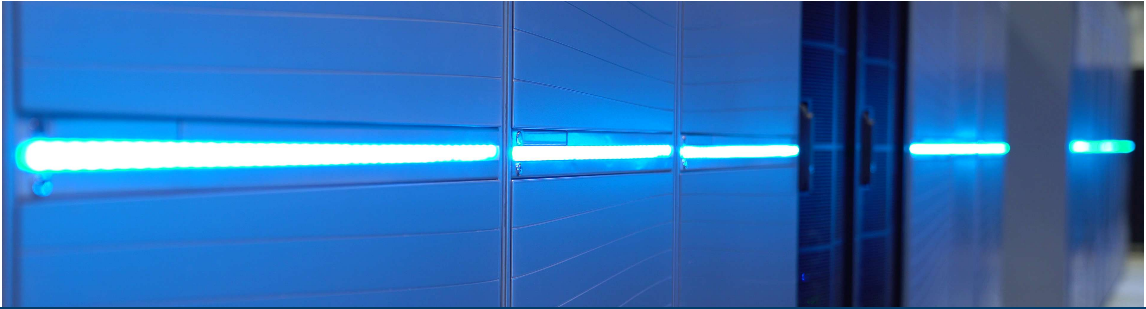


INTRODUCTION TO PARALLEL PROGRAMMING WITH MPI AND OPENMP

August 12-16 2024 | Junxian Chew, Michael Knobloch, Ilya Zhukov, Jolanta Zjupa | Jülich Supercomputing Centre

JÜLICH Forschungszentrum



Part I: First Steps with OpenMP

WHAT IS OPENMP?

> *OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs. (OpenMP FAQ[1])*

- Initially targeted SMP systems, now also DSPs, accelerators, etc.
- Provides specifications (not implementations)
- Portable across different platforms

Current version of the specification: 5.2 (November 2021)

[1] Matthijs van Waveren et al. OpenMP FAQ. Version 3.0. June 6, 2018. URL: https://www.openmp.org/about/openmp-faq/ (visited on 01/30/2019).

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

COVERAGE

- Overview of the OpenMP API
- Internal Control Variables
- Directive and Construct Syntax
- Base Language Formats and Restrictions
- Data Environment
- Memory Management
- Variant Directives
- Informational and Utility Directives
- Loop Transformation Constructs
- Parallelism Generation and Control
- Work-Distribution Constructs
- Tasking Constructs
- Device Directives and Clauses
- Interoperability
- Synchronization Constructs and Clauses
- Cancellation Constructs
- Composition of Contstructs
- Runtime Library Routines
- OMPT Interface
- OMPD Interface
- Environment Variables

COVERAGE

- Overview of the OpenMP API (✓)
- Internal Control Variables (✓)
- Directive and Construct Syntax (✓)
- Base Language Formats and Restrictions (✓)
- Data Environment (✓)
- Memory Management
- Variant Directives
- Informational and Utility Directives
- Loop Transformation Constructs
- Parallelism Generation and Control (✓)
- Work-Distribution Constructs (✓)
- Tasking Constructs (✓)
- Device Directives and Clauses
- Interoperability
- Synchronization Constructs and Clauses (✓)
- Cancellation Constructs
- Composition of Contstructs (✓)
- Runtime Library Routines (✓)
- OMPT Interface
- OMPD Interface
- Environment Variables (✓)

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

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

THREADS & TASKS

Thread

An execution entity with a stack and associated static memory, called threadprivate memory.

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.

LANGUAGE

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.

Directive

In C/C++, a `#pragma`, and in Fortran, a comment, that specifies OpenMP program behavior.

JÜLICH Forschungszentrum

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 Fortran Compiler OpenMP Command Line Switch

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

The name of this command line argument is not mandated by the specification and differs from one compiler to another.
Naturally, these arguments are then also accepted by the MPI compiler wrappers:

Compiling Programs with Hybrid Parallelization

```
$ mpicc -qopenmp ...
```

RUNTIME LIBRARY DEFINITIONS [OpenMP-5.2, 18.1]

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
```fortran
include "omp_lib.h"
```

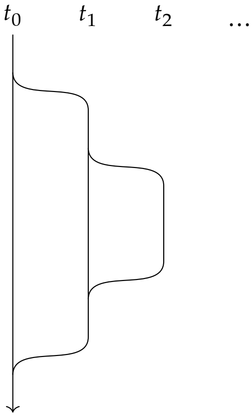
or a Fortran 90 module

F08
```fortran
use omp_lib
```


JÜLICH Forschungszentrum

WORLD ORDER IN OPENMP

- Program starts as one single-threaded process.
- Forks into teams of multiple threads when appropriate.
- Stream of instructions might be different for each thread.
- Information is exchanged via shared parts of memory.
- OpenMP threads may be nested inside MPI processes.



C AND C++ DIRECTIVE FORMAT [OpenMP-5.2, 3.1]

In C and C++, OpenMP directives are written using the *#pragma* method:

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

- Directives are case-sensitive
- Applies to the next statement which must be a structured block

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 [OpenMP-5.2, 3.1.1, 3.1.2]

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

JÜLICH Forschungszentrum

QUIZ

Why were these formats chosen for OpenMP directives?

1. Syntax highlighting for pragmas and comments was already available in editors.
2. Pragmas and comments were already familiar to programmers so there was less new syntax to learn.
3. Compilers without support for OpenMP will just ignore the unknown pragmas and comments and thus degrade gracefully.

CONDITIONAL COMPILATION [OpenMP-5.2, 3.3]

C Preprocessor Macro

C
```c
#define _OPENMP yyyymm
```

`yyyy` and `mm` are the year and month the OpenMP specification supported by the compiler was published.

Fortran Fixed Form Sentinels

F08
```fortran
!$ | *$ | c$
```

- Must start in column 1
- Only numbers or white space in columns 3–5
- Column 6 marks continuation lines

Fortran Free Form Sentinel

F08
```fortran
!$
```

- Must only be preceded by white space
- Can be continued with ampersand

THE PARALLEL CONSTRUCT [OpenMP-5.2, 10.1]

C

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

F08

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

- Creates a team of threads to execute the `parallel` region
- Each thread executes the code contained in the structured block
- Inside the region threads are identified by consecutive numbers starting at zero
- Optional clauses (explained later) can be used to modify behavior and data environment of the `parallel` region

THREAD COORDINATES [OpenMP-5.2, 18.2.2, 18.2.4]

Team size

C
```c
int omp_get_num_threads(void);
```

F08
```fortran
integer function omp_get_num_threads()
```

Returns the number of threads in the current team

Thread number

C
```c
int omp_get_thread_num(void);
```

F08
```fortran
integer function omp_get_thread_num()
```

Returns the number that identifies the calling thread within the current team (between zero and `omp_get_num_threads()`)

A FIRST OPENMP PROGRAM

```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

```fortran
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.
```


JÜLICH Forschungszentrum

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.
```


JÜLICH Forschungszentrum

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.
```

EXERCISES

Exercise 1 – Warm Up

1.1 Generalized Vector Addition (axpy)

In the file `axpy.{c|c++|f90}`, fill in the missing body of the function/subroutine `axpy_serial(a, x, y, z[, n])` so that it implements the generalized vector addition (in serial, without making use of OpenMP):

$$\mathbf{z} = a\mathbf{x} + \mathbf{y}.$$

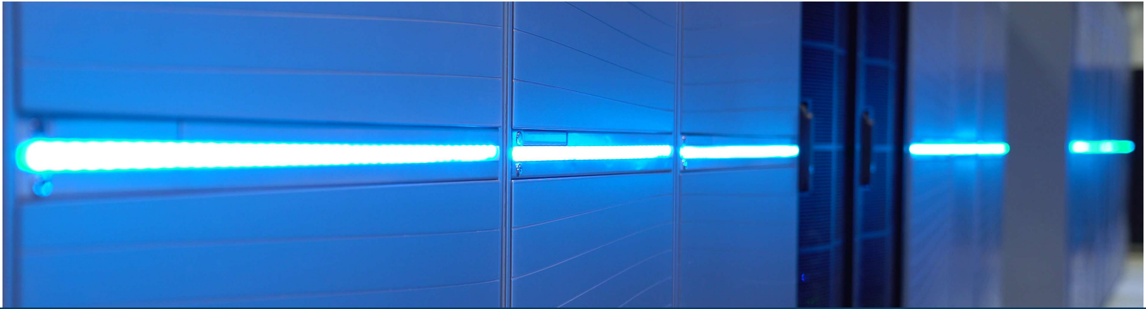
Compile the file into a program and run it to test your implementation.

1.2 Dot Product

In the file `dot.{c|c++|f90}`, fill in the missing body of the function/subroutine `dot_serial(x, y[, n])` so that it implements the dot product (in serial, without making use of OpenMP):

$$\texttt{dot}(\mathbf{x}, \mathbf{y}) = \sum_i x_i y_i.$$

Compile the file into a program and run it to test your implementation.



Part II: Low-Level OpenMP Concepts

MAGIC

Any sufficiently advanced technology is indistinguishable from magic. (Arthur C. Clarke[2])

[2] Arthur C. Clarke. Profiles of the future : an inquiry into the limits of the possible. London: Pan Books, 1973. ISBN: 9780330236195.

INTERNAL CONTROL VARIABLES [OpenMP-5.2, 2]

Internal Control Variable (ICV)

A conceptual variable that specifies runtime behavior of a set of threads or tasks in an OpenMP program.

- Set to an initial value by the OpenMP implementation
- Some can be modified through either environment variables (e.g. `OMP_NUM_THREADS`) or API routines (e.g. `omp_set_num_threads()`)
- Some can be read through API routines (e.g. `omp_get_max_threads()`)
- Some are inaccessible to the user
- Might have different values in different scopes (e.g. data environment, device, global)
- Some can be overridden by clauses (e.g. the `num_threads()` clause)
- Export `OMP_DISPLAY_ENV=TRUE` or call `omp_display_env(1)` to inspect the value of ICVs that correspond to environment variables [OpenMP-5.2, 18.15, 21.7]

PARALLELISM CLAUSES [OpenMP-5.2, 3.4, 10.1.2]

`if` Clause

C
```
if([parallel :] scalar-expression)
```

F08
```
if([parallel :] scalar-logical-expression)
```

If false, the region is executed only by the encountering thread(s) and no additional threads are forked.

`num_threads` Clause

C
```
num_threads(integer-expression)
```

F08
```
num_threads(scalar-integer-expression)
```

Requests a team size equal to the value of the expression (overrides the nthreads-var ICV)

EXAMPLE

A `parallel` directive with an `if` clause and associated structured block in C:

C
```c
#pragma omp parallel if( length > threshold )
{
  statement0;
  statement1;
  statement2;
}
```

A `parallel` directive with a `num_threads` clause and associated structured block in Fortran:

F08
```fortran
!$omp parallel num_threads( 64 )
statement1
statement2
statement3
!$omp end parallel
```

CONTROLLING THE nthreads-var ICV

omp_set_num_threads API Routine [OpenMP-5.2, 18.2.1]

C
```c
void omp_set_num_threads(int num_threads);
```

F08
```fortran
subroutine omp_set_num_threads(num_threads)
integer num_threads
```

Sets the ICV that controls the number of threads to fork for `parallel` regions (without `num_threads` clause) encountered subsequently.

omp_get_max_threads API Routine [OpenMP-5.2, 18.2.3]

C
```c
int omp_get_max_threads(void);
```

F08
```fortran
integer function omp_get_max_threads()
```

Queries the ICV that controls the number of threads to fork.

THREAD LIMIT & DYNAMIC ADJUSTMENT

omp_get_thread_limit API Routine [OpenMP-5.2, 18.2.13]

C

```c
int omp_get_thread_limit(void);
```

F08

```fortran
integer function omp_get_thread_limit()
```

Upper bound on the number of threads used in a program.

omp_get_dynamic and omp_set_dynamic API Routines [OpenMP-5.2, 18.2.6, 18.2.7]

C

```c
int omp_get_dynamic(void);
void omp_set_dynamic(int dynamic);
```

F08

```fortran
logical function omp_get_dynamic()
subroutine omp_set_dynamic(dynamic)
logical dynamic
```

Enable or disable dynamic adjustment of the number of threads.

JÜLICH Forschungszentrum

INSIDE OF A PARALLEL REGION?

`omp_in_parallel` API Routine [OpenMP-5.2, 18.2.5]

C
```c
int omp_in_parallel(void);
```

F08
```fortran
logical function omp_in_parallel()
```

Is this code being executed as part of a `parallel` region?

EXERCISES

Exercise 2 – Controlling parallel

2.1 Controlling the Number of Threads

Use hello_openmp.{c|c++|f90} to play around with the various ways to set the number of threads forked for a parallel region:

- The OMP_NUM_THREADS environment variable
- The omp_set_num_threads API routine
- The num_threads clause
- The if clause

Inspect the number of threads that are actually forked using omp_get_num_threads.

2.2 Limits of the OpenMP Implementation

Determine the maximum number of threads allowed by the OpenMP implementation you are using and check whether it supports dynamic adjustment of the number of threads.

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.

CONSTRUCTS & REGIONS

Construct
An OpenMP executable directive (and for Fortran, the paired end directive, if any) and the associated statement, loop or structured block, if any, not including the code in any called routines. That is, the lexical extent of an executable directive.

Region
All code encountered during a specific instance of the execution of a given construct or of an OpenMP library routine.

Executable Directive
An OpenMP directive that is not declarative. That is, it may be placed in an executable context.

CONSTRUCTS & REGIONS EXAMPLE

```c
int main(void) {
  #pragma omp parallel
  {
    f();
    if (true) {
      statement;
    } else {
      statement;
    }
  }
}
```

```c
void f(void) {
  statement;
  statement;
  statement;
}
```

DATA-SHARING ATTRIBUTE RULES I [OpenMP-5.2, 5.1.1]

The rules that determine the data-sharing attributes of variables referenced from the inside of a construct fall into one of the following categories:

Pre-determined

- Variables with automatic storage duration declared inside the construct are private (C and C++)
- Objects with dynamic storage duration are shared (C and C++)
- Variables with static storage duration declared in the construct are shared (C and C++)
- Static data members are shared (C++)
- Loop iteration variables are private (Fortran)
- Implied-do indices and **`forall`** indices are private (Fortran)
- Assumed-size arrays are shared (Fortran)

Explicit

Data-sharing attributes are determined by explicit clauses on the respective constructs.

Implicit

If the data-sharing attributes are neither pre-determined nor explicitly determined, they fall back to the attribute determined by the `default` clause, or `shared` if no `default` clause is present.

DATA-SHARING ATTRIBUTE RULES II [OpenMP-5.2, 5.1.2]

The data-sharing attributes of variables inside regions, not constructs, are governed by simpler rules:

- Static variables (C and C++) and variables with the **`save`** attribute (Fortran) are shared
- File-scope (C and C++) or namespace-scope (C++) variables and common blocks or variables accessed through use or host association (Fortran) are shared
- Objects with dynamic storage duration are shared (C and C++)
- Static data members are shared (C++)
- Arguments passed by reference have the same data-sharing attributes as the variable they are referencing (C++ and Fortran)
- Implied-do indices, **`forall`** indices are private (Fortran)
- Local variables are private

JÜLICH Forschungszentrum

THE SHARED CLAUSE [OpenMP-5.2, 5.4.2]

```
shared(list)
```

- Declares the listed variables to be shared.
- The programmer must ensure that shared variables are alive while they are shared.
- Shared variables must not be part of another variable (i.e. array or structure elements).

THE PRIVATE CLAUSE [OpenMP-5.2, 5.4.3]

```
* private(list)
```

- Declares the listed variables to be private.
- All threads have their own new versions of these variables.
- Private variables must not be part of another variable.
- If private variables are of class type, a default constructor must be accessible. (C++)
- The type of a private variable must not be **`const`**-qualified, incomplete or reference to incomplete. (C and C++)
- Private variables must either be definable or allocatable. (Fortran)
- Private variables must not appear in **`namelist`** statements, variable format expressions or expressions for statement function definitions. (Fortran)
- Private variables must not be pointers with **`intent`**`(in)`. (Fortran)

FIRSTPRIVATE CLAUSE [OpenMP-5.2, 5.4.4]

```
* firstprivate(list)
```

Like private, but initialize the new versions of the variables to have the same value as the variable that exists before the construct.

- Non-array variables are initialized by copy assignment (C and C++)
- Arrays are initialize by element-wise assignment (C and C++)
- Copy constructors are invoked if present (C++)
- Non-**`pointer`** variables are initialized by assignment or not associated if the original variable is not associated (Fortran)
- **`pointer`** variables are initialized by pointer assignment (Fortran)

DEFAULT CLAUSE [OpenMP-5.2, 5.4.1]

C and C++

C
```
default(shared | none)
```

Fortran

F08
```
default(private | firstprivate | shared | none)
```

Determines the data-sharing attributes for all variables referenced from inside of a region that have neither pre-determined nor explicit data-sharing attributes.

`default(none)` forces the programmer to make data-sharing attributes explicit if they are not pre-determined. This can help clarify the programmer's intentions to someone who does not have the implicit data-sharing rules in mind.

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

QUIZ

Which exercise represents a reduction?

1. None
2. Generalized vector addition (AXPY)
3. Dot product
4. Both

EXERCISES

Exercise 3 – Data-sharing Attributes

3.1 Generalized Vector Addition (axpy)

In the file `axpy.{c|c++|f90}` add a new function/subroutine `axpy_parallel(a, x, y, z[, n])` that uses multiple threads to perform a generalized vector addition. Modify the main part of the program to have your function/subroutine tested.

Hints:

- Use the `parallel` construct and the necessary clauses to define an appropriate data environment.
- Use `omp_get_thread_num()` and `omp_get_num_threads()` to decompose the work.

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
```c
#pragma omp barrier
```

F08
```fortran
!$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
```


JÜLICH Forschungszentrum

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

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

F08

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

LOCK ROUTINES [OpenMP-5.2, 18.9]

C

```c
void omp_init_lock(omp_lock_t* lock);
void omp_destroy_lock(omp_lock_t* lock);
void omp_set_lock(omp_lock_t* lock);
void omp_unset_lock(omp_lock_t* lock);
```

F08

```fortran
subroutine omp_init_lock(svar)
subroutine omp_destroy_lock(svar)
subroutine omp_set_lock(svar)
subroutine omp_unset_lock(svar)
integer(kind = omp_lock_kind) :: svar
```

- Like `critical` sections, but identified by runtime value rather than global name
- Locks must be `shared` between threads
- Initialize a lock before first use
- Destroy a lock when it is no longer needed
- Lock and unlock using the `set` and `unset` routines
- `set` blocks if lock is already set

LOCK CONTROL FLOW

Thread 0

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
end program
```

Console

LOCK CONTROL FLOW

Thread 0

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
end program
```

Thread 1

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
end program
```

Console

LOCK CONTROL FLOW

Thread 0

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
end program
```

Thread 1

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
end program
```

Console

LOCK CONTROL FLOW

Thread 0

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
end program
```

Thread 1

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
end program
```

Console

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

LOCK CONTROL FLOW

Thread 0

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
end program
```

Thread 1

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
end program
```

Console

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

LOCK CONTROL FLOW

Thread 0

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
end program
```

Thread 1

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
end program
```

Console

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

LOCK CONTROL FLOW

Thread 0

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
end program
```

Thread 1

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
end program
```

Console

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

LOCK CONTROL FLOW

Thread 0

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
end program
```

Thread 1

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
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.
```

LOCK CONTROL FLOW

Thread 0

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
end program
```

Thread 1

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
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.
```

LOCK CONTROL FLOW

Thread 0

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
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.
```

LOCK CONTROL FLOW

Thread 0

```
program hello_critical
  call omp_init_lock(lock)
  !$omp parallel
  call omp_set_lock(lock)
  print *, "Hello..."
  print *, "Again..."
  call omp_unset_lock(lock)
  !$omp end parallel
  call omp_destroy_lock(lock)
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.
```

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

EXERCISES

Exercise 4 – Thread Synchronization

4.1 Dot Product

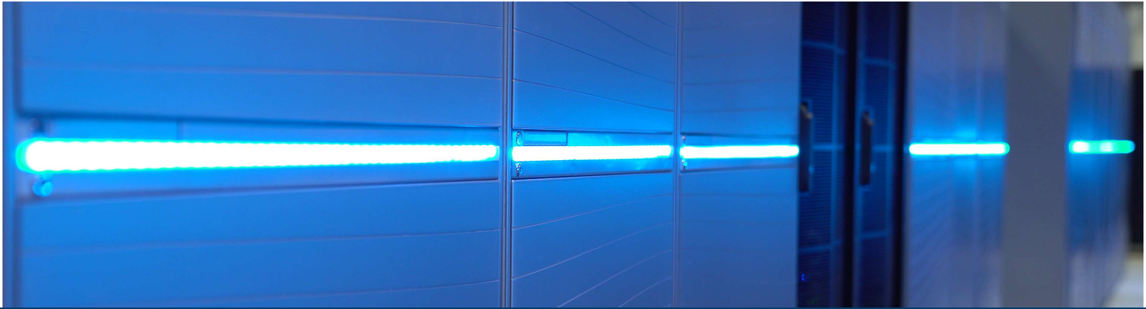
In the file `dot.{c|c++|f90}` add a new function/subroutine `dot_parallel(x, y[, n])` that uses multiple threads to perform the dot product. Do not use the `reduction` clause. Modify the main part of the program to have your function/subroutine tested.

Hint:

- Decomposition of the work load should be similar to the last exercise
- Partial results of different threads should be combined in a shared variable
- Use a suitable synchronization mechanism to coordinate access

Bonus

Use the `reduction` clause to simplify your program.



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

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

F08

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


JÜLICH Forschungszentrum

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.
```

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`

JÜLICH Forschungszentrum

WORKSHARING-LOOP CONTROL FLOW

Thread 0

```fortran
...
!$omp parallel
!$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
```

CANONICAL NEST LOOP FORM [OpenMP-5.2, 4.4.1]

In C and C++ the `for-loops` must have the following form:

C
```
for ([type] var = lb; var relational-op b; incr-expr) structured-block
```

C++
```
for (range-decl: range-expr) structured-block
```

- `var` can be an integer, a pointer, or a random access iterator
- `incr-expr` increments (or decrements) `var`, e.g. `var = var + incr`
- The increment `incr` must not change during execution of the loop
- For nested loops, the bounds of an inner loop (b and `lb`) may depend at most linearly on the iteration variable of an outer loop, i.e. `a0 + a1 * var-outer`
- `var` must not be modified by the loop body
- The beginning of the range has to be a random access iterator
- The number of iterations of the loop must be known beforehand

JÜLICH Forschungszentrum

CANONICAL NEST LOOP FORM [OpenMP-5.2, 4.4.1]

In Fortran the `do-loops` must have the following form:

F08
```
do [label] var = lb, b[, incr]
```

- `var` must be of integer type
- `incr` must be invariant with respect to the outermost loop
- The loop bounds b and `lb` of an inner loop may depend at most linearly on the iteration variable of an outer loop, i.e. `a0 + a1 * var-outer`
- The number of iterations of the loop must be known beforehand

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.:

```
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.

WORKSHARE (FORTRAN ONLY) [OpenMP-5.2, 11.4]

F08

```
!$omp workshare
  structured-block
!$omp end workshare [nowait]
```

The structured block may contain:

- array assignments
- scalar assignments
- **`forall`** constructs
- **`where`** statements and constructs
- `atomic`, `critical` and `parallel` constructs

Where possible, these are decomposed into independent units of work and executed in parallel.

JÜLICH Forschungszentrum

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 5 – Loop Worksharing

5.1 Generalized Vector Addition (axpy)

In the file `axpy.{c|c++|f90}` add a new function/subroutine `axpy_parallel_for(a, x, y, z[, n])` that uses loop worksharing to perform the generalised vector addition.

5.2 Dot Product

In the file `dot.{c|c++|f90}` add a new function/subroutine `dot_parallel_for(x, y[, n])` that uses loop worksharing to perform the dot product.

Caveat: Make sure to correctly synchronize access to the accumulator variable.

JÜLICH Forschungszentrum

EXERCISES

Exercise 6 – workshare Construct

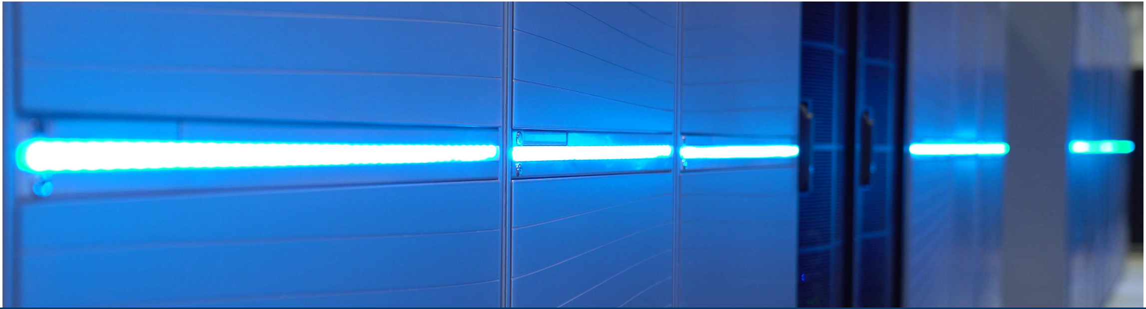
6.1 Generalized Vector Addition (axpy)

In the file `axpy.f90` add a new subroutine `axpy_parallel_workshare(a, x, y, z)` that uses the `workshare` construct to perform the generalized vector addition.

6.2 Dot Product

In the file `dot.f90` add a new function `dot_parallel_workshare(x, y)` that uses the `workshare` construct to perform the dot product.

Caveat: Make sure to correctly synchronize access to the accumulator variable.



Part IV: Task Worksharing

TASK TERMINOLOGY

Task
A specific instance of executable code and its data environment, generated when a thread encounters a `task`, `taskloop`, `parallel`, `target` or `teams` construct.

Child Task
A task is a child task of its generating task region. A child task region is not part of its generating task region.

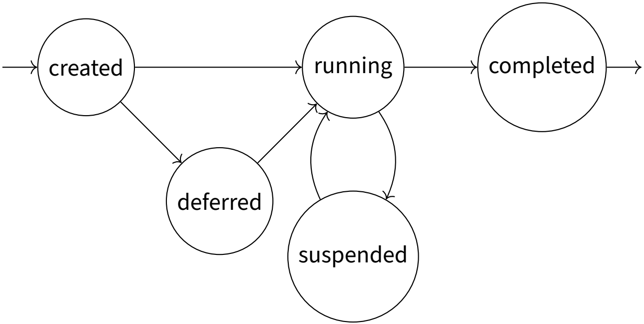
Descendent Task
A task that is the child task of a task region or of one of its descendent task regions.

Sibling Task
Tasks that are child tasks of the same task region.

JÜLICH Forschungszentrum

TASK LIFE-CYCLE

- Execution of tasks can be deferred and suspended
- Scheduling is done by the OpenMP runtime system at scheduling points
- Scheduling decisions can be influenced by e.g. task dependencies and task priorities



THE TASK CONSTRUCT [OpenMP-5.2, 12.5]

C

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

F08

```fortran
!$omp task [clause[[,] clause]...]
  structured-block
!$omp end task
```

Creates a task. Execution of the task may commence immediately or be deferred.

Data-environment clauses

- private
- firstprivate
- shared

Task-specific clauses

- if
- final
- untied
- mergeable
- depend
- priority

TASK DATA-ENVIRONMENT [OpenMP-5.2, 5.1.1]

The rules for implicitly determined data-sharing attributes of variables referenced in task generating constructs are slightly different from other constructs:
If no `default` clause is present and

- the variable is `shared` by all implicit tasks in the enclosing context, it is also `shared` by the generated task,
- otherwise, the variable is `firstprivate`.

THE IF CLAUSE [OpenMP-5.2, 3.4, 12.5]

```
*  if([task: ] scalar-expression)
```

If the scalar expression evaluates to false:

- Execution of the current task
  - is suspended and
  - may only be resumed once the generated task is complete
- Execution of the generated task may commence immediately

Undeferred Task

A task for which execution is not deferred with respect to its generating task region. That is, its generating task region is suspended until execution of the undeferred task is completed.

THE FINAL CLAUSE [OpenMP-5.2, 12.3]

```
final(scalar-expression)
```

If the scalar expression evaluates to true all descendent tasks of the generated task are

- undeferred and
- executed immediately.

Final Task
A task that forces all of its child tasks to become final and included tasks.

Included Task
A task for which execution is sequentially included in the generating task region. That is, an included task is undeferred and executed immediately by the encountering thread.

THE UNTIED CLAUSE [OpenMP-5.2, 12.1]

```
* untied
```

- The generated task is untied meaning it can be suspended by one thread and resume execution on another.
- By default, tasks are generated as tied tasks.

Untied Task

A task that, when its task region is suspended, can be resumed by any thread in the team. That is, the task is not tied to any thread.

Tied Task

A task that, when its task region is suspended, can be resumed only by the same thread that suspended it. That is, the task is tied to that thread.

THE PRIORITY CLAUSE [OpenMP-5.2, 12.4]

```
* priority(priority-value)
```

- priority-value is a scalar non-negative numerical value
- Priority influences the order of task execution
- Among tasks that are ready for execution, those with a higher priority are more likely to be executed next

THE DEPEND CLAUSE [OpenMP-5.2, 15.9.5]

```
depend(in: list)
depend(out: list)
depend(inout: list)
```
*

- list contains storage locations
- A task with a dependence on x, `depend(in: x)`, has to wait for completion of previously generated sibling tasks with `depend(out: x)` or `depend(inout: x)`
- A task with a dependence `depend(out: x)` or `depend(inout: x)` has to wait for completion of previously generated sibling tasks with any kind of dependence on x
- `in`, `out` and `inout` correspond to intended read and/or write operations to the listed variables.

Dependent Task

A task that because of a task dependence cannot be executed until its predecessor tasks have completed.

TASK SCHEDULING POLICY [OpenMP-5.2, 12.9]

The task scheduler of the OpenMP runtime environment becomes active at task scheduling points. It may then

- begin execution of a task or
- resume execution of untied tasks or tasks tied to the current thread.

Task scheduling points

- generation of an explicit task
- task completion
- `taskyield` regions
- `taskwait` regions
- the end of `taskgroup` regions
- implicit and explicit `barrier` regions

THE TASKYIELD CONSTRUCT [OpenMP-5.2, 12.7]

C
```
#pragma omp taskyield
```

F08
```
!$omp taskyield
```

- Notifies the scheduler that execution of the current task may be suspended at this point in favor of another task
- Inserts an explicit scheduling point

THE TASKWAIT & TASKGROUP CONSTRUCTS [OpenMP-5.2, 15.4, 15.5]

C
```c
#pragma omp taskwait
```

F08
```fortran
!$omp taskwait
```

Suspends the current task until all child tasks are completed.

C
```c
#pragma omp taskgroup
  structured-block
```

F08
```fortran
!$omp taskgroup
  structured-block
!$omp end taskgroup
```

The current task is suspended at the end of the `taskgroup` region until all descendent tasks generated within the region are completed.

TASK CONTROL FLOW

```c
unsigned fib(unsigned n) {
  if (n < 2) return n;
  unsigned a, b;
  a = fib(n - 1);
  b = fib(n - 2);
  return a + b;
}

int main(int argc, char* argv[]) {
  printf("fib(3) = %u\n", fib(3));
}
```

TASK CONTROL FLOW

```
unsigned fib(unsigned n) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  #pragma omp task default(shared)
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}

int main(int argc, char* argv[]) {
  #pragma omp parallel
  #pragma omp single
  printf("fib(3) = %u\n", fib(3));
}
```

TASK CONTROL FLOW

Thread 0

```
unsigned fib(unsigned n = 3) {
   if (n < 2) return n;
   unsigned a, b;
   #pragma omp task default(shared)
   a = fib(n - 1);
   #pragma omp task default(shared)
   b = fib(n - 2);
   #pragma omp taskwait
   return a + b;
}
```

Tasks:

Thread 1

TASK CONTROL FLOW

Thread 0

```
unsigned fib(unsigned n = 3) {
   if (n < 2) return n;
   unsigned a, b;
   #pragma omp task default(shared)
   a = fib(n - 1);
   #pragma omp task default(shared)
   b = fib(n - 2);
   #pragma omp taskwait
   return a + b;
}
```

Tasks: `fib(2)`

Thread 1

TASK CONTROL FLOW

Thread 0

```
unsigned fib(unsigned n = 3) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  #pragma omp task default(shared)
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}
```

Tasks: `fib(2)`, `fib(1)`

Thread 1

TASK CONTROL FLOW

Thread 0

```
unsigned fib(unsigned n = 3) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  #pragma omp task default(shared)
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}
```

Tasks: `fib(1)`, `fib(3)...`

Thread 1

```
unsigned fib(unsigned n = 2) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  #pragma omp task default(shared)
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}
```

TASK CONTROL FLOW

Thread 0

```
unsigned fib(unsigned n = 1) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  #pragma omp task default(shared)
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}
```

Tasks: fib(3)..., fib(1)

Thread 1

```
unsigned fib(unsigned n = 2) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  #pragma omp task default(shared)
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}
```

TASK CONTROL FLOW

Thread 0

```
unsigned fib(unsigned n = 1) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  #pragma omp task default(shared)
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}
```

Tasks: `fib(3)...`, `fib(1)`, `fib(0)`

Thread 1

```
unsigned fib(unsigned n = 2) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  #pragma omp task default(shared)
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}
```

TASK CONTROL FLOW

Thread 0

```
unsigned fib(unsigned n = 1) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  #pragma omp task default(shared)
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}
```

Tasks: `fib(3)...`, `fib(0)`, `fib(2)...`

Thread 1

```
unsigned fib(unsigned n = 2) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  #pragma omp task default(shared)
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}
```

TASK CONTROL FLOW

Thread 0

```
unsigned fib(unsigned n = 1) {
   if (n < 2) return n;
   unsigned a, b;
   #pragma omp task default(shared)
   a = fib(n - 1);
   #pragma omp task default(shared)
   b = fib(n - 2);
   #pragma omp taskwait
   return a + b;
}
```

Thread 1

```
unsigned fib(unsigned n = 0) {
   if (n < 2) return n;
   unsigned a, b;
   #pragma omp task default(shared)
   a = fib(n - 1);
   #pragma omp task default(shared)
   b = fib(n - 2);
   #pragma omp taskwait
   return a + b;
}
```

Tasks: `fib(3)...`, `fib(2)...`

TASK CONTROL FLOW

Thread 0

```
unsigned fib(unsigned n = 3) {
   if (n < 2) return n;
   unsigned a, b;
   #pragma omp task default(shared)
   a = fib(n - 1);
   #pragma omp task default(shared)
   b = fib(n - 2);
   #pragma omp taskwait
   return a + b;
}
```

Tasks: `fib(2)...`

Thread 1

```
unsigned fib(unsigned n = 0) {
   if (n < 2) return n;
   unsigned a, b;
   #pragma omp task default(shared)
   a = fib(n - 1);
   #pragma omp task default(shared)
   b = fib(n - 2);
   #pragma omp taskwait
   return a + b;
}
```

TASK CONTROL FLOW

Thread 0

```
unsigned fib(unsigned n = 3) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  #pragma omp task default(shared)
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}
```

Thread 1

```
unsigned fib(unsigned n = 2) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  #pragma omp task default(shared)
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}
```

Tasks:

TASK CONTROL FLOW

Thread 0

```
unsigned fib(unsigned n = 3) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  #pragma omp task default(shared)
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}
```

Thread 1

```
unsigned fib(unsigned n = 2) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  #pragma omp task default(shared)
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}
```

Tasks:

TASK CONTROL FLOW

Thread 0

```
unsigned fib(unsigned n = 3) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  #pragma omp task default(shared)
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}
```

Tasks:

Thread 1

TASK CONTROL FLOW

```
unsigned fib(unsigned n) {
  if (n < 2) return n;
  unsigned a, b;
  #pragma omp task default(shared)
  a = fib(n - 1);
  b = fib(n - 2);
  #pragma omp taskwait
  return a + b;
}

int main(int argc, char* argv[]) {
  #pragma omp parallel
  #pragma omp single
  printf("fib(3) = %u\n", fib(3));
}
```

EXERCISES

Exercise 7 – Task worksharing

7.1 Generalized Vector Addition (axpy)

In the file `axpy.{c|c++|f90}` add a new function/subroutine `axpy_parallel_task(a, x, y, z[, n])` that uses task worksharing to perform the generalized vector addition.

7.2 Dot Product

In the file `dot.{c|c++|f90}` add a new function/subroutine `dot_parallel_task(x, y[, n])` that uses task worksharing to perform the dot product.

Caveat: Make sure to correctly synchronize access to the accumulator variable.