NHR FAU

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Introduction to OpenMP Part 1

Markus Wittmann

based on work by
R. Bader (LRZ), G. Hager (RRZE), V. Weinberg (LRZ)

HPC High Performance Computing

Outline

- introduction
- basic overview
  - parallel construct
  - barrier
  - data sharing
  - worksharing loops
  - reductions
- more details on
  - parallel construct
  - worksharing loops
- worksharing constructs single & sections
- synchronization
  - critical, atomic, locks
- optional
  - thread private data
  - thread affinity
- API routines & environment variables

- 9:00 – 12:00 course
- 12:00 – 13:00 break
- 13:00 – 16:00 course

- interspersed breaks and hands-on

Introduction

OpenMP – What is it

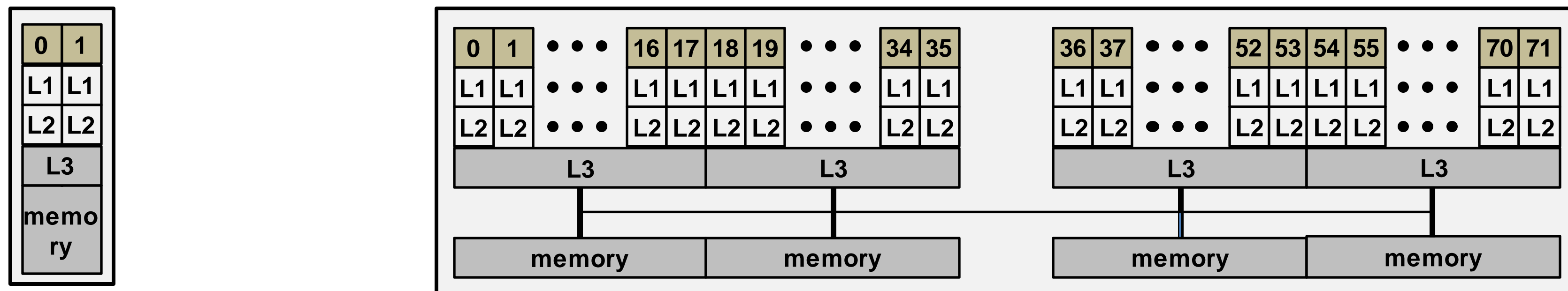
- directives and runtime functions for parallelization via threads
- supported languages: C, C++, Fortran
  - in this course: C also includes C++, except where mentioned differently
- requires compiler & runtime library support
- standard: https://www.openmp.org/
  - latest version: OpenMP 5.2 (Nov. 2021)
  - https://www.openmp.org/specifications/
    - documentation,
    - OpenMP API x.y Examples,
    - OpenMP API x.y Reference Guide
- Contains: parallelization, synchronization, tasking, accelerator offloading, SIMD support, loop transformations, …

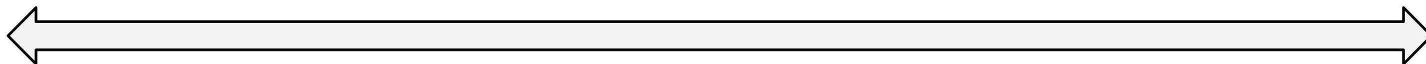
Compiling OpenMP applications sequentially

- OpenMP applications can be executed sequentially
  - directives are ignored
- if runtime functions are used
  - they need to be guarded off
  - a stubs library must be used (if available)

- it is no requirement that OpenMP applications can be compiled without OpenMP enabled

Shared Memory Architectures

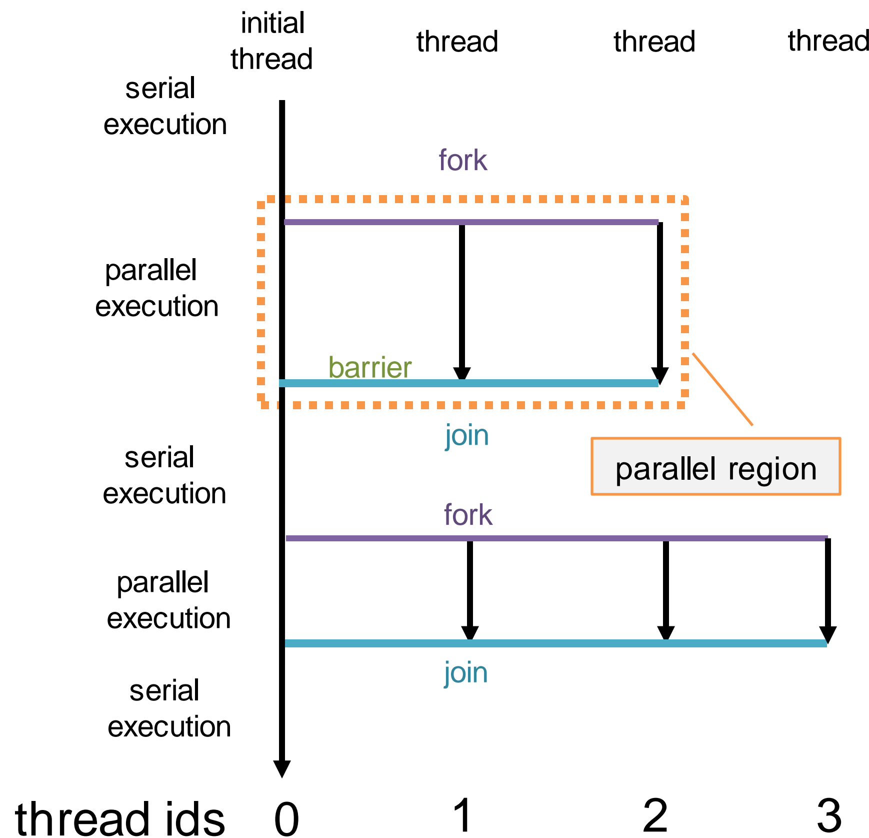




OpenMP targets all range of shared memory architectures

Fork-Join Model of Parallel Regions

- parallel execution happens in **parallel regions**
  - follow fork-join model
- initial thread
  - works serially, *sequential part*
- fork – begin of parallel region
  - executed by **team of threads**
- join – end of parallel region
  - threads wait in an **barrier** until all have arrived
- after join serial execution continues



OpenMP Directives – C/C++

- directives specify OpenMP behavior

C/C++

```
#pragma omp <directive> [<clause>[, <clause>[...]]]
```

ignored when OpenMP is not enabled

- depending on the directive the following statement or loop might be associated

```
statement;
#pragma omp barrier
statement;
```

`barrier` has no associated statements

```
#pragma omp parallel
{
  statement;
  statement;
}
```

`parallel` is associated with following statement

to associate multiple statements use a {} block

- `_Pragma("omp ...")` instead of `#pragma` works too

- C++: no directives in `constexpr` functions or constant expressions

OpenMP Directive with C++ Attribute Syntax

Requires ≥ v5.1

```
[[ omp :: <directive> [<clause> [<clause ...] ] ]]
```

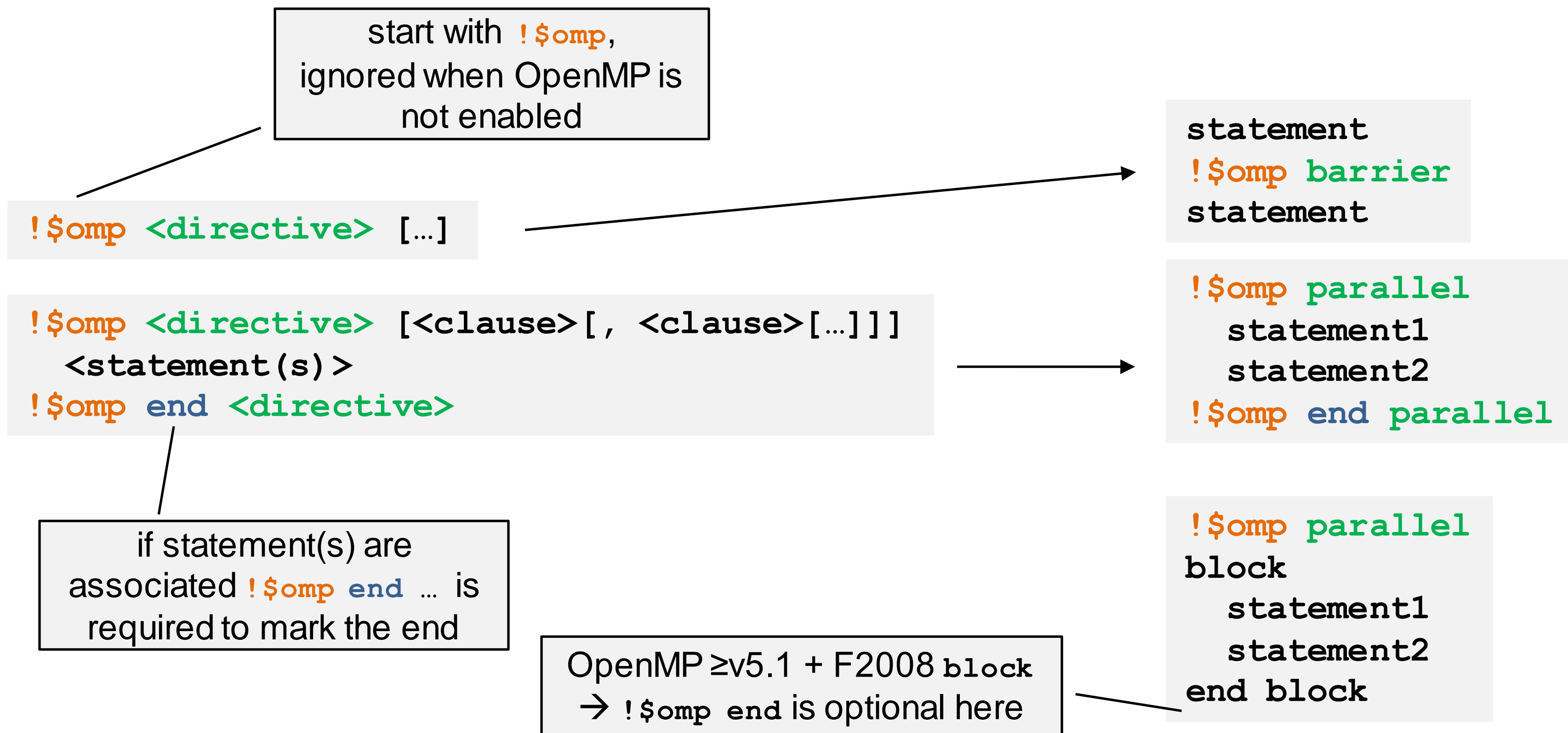
or

```
[[ using omp : <directive> [<clause> [<clause ...] ] ]]
```

- depending on directive binds to the following statement
  - use `{}` to associate a block
- no ordering of subsequent attributes, use

```
[[ omp :: sequence( [omp::<directive>...[, omp::<directive>...] ] ) ]]
```

OpenMP Directives – Fortran



Directives and Continuation Lines

continued directives on the next line

C/C++

```
#pragma omp parallel \
        num_threads(2)
```

\ at end of line

Fortran free form

```
!$omp parallel       &
!$omp& num_threads(2)
```

& at end of line

fixed form

```
!$omp parallel
!$omp+ num_threads(2)
```

non-whitespace

Example

C/C++:

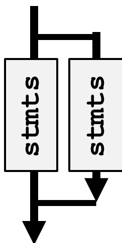
```
#pragma omp parallel
printf("hello\n");
```

Fortran:

```
!$omp parallel
  write(*,'(a)') 'hello'
!$omp end parallel
```

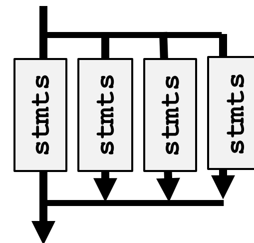
Output:

```
> OMP_NUM_THREADS=2 ./a.out
hello
hello
```



Output:

```
> OMP_NUM_THREADS=4 ./a.out
hello
hello
hello
hello
```



output might be interleaved

- control no. of threads by environment variable **`OMP_NUM_THREADS`**
  - see later for details
- without specifying **`OMP_NUM_THREADS`** the default is implementation defined
  - typically as many threads as cores are available are used

Compilation and Linking

- enable OpenMP through flags

- for GCC (gcc, g++, gfortran), LLVM (clang, clang++, flang) add `-fopenmp`
  - `gcc -fopenmp example.c -o example`
  - best practice for `gcc`:
    also add `-Wall` (includes `-Wunknown-pragmas`),
    generates warnings when directives are mistyped or not supported

required for compiling and linking

- for Intel Classic (icc, icpc, ifort), Intel oneAPI (icx, icpx, ifx) add `-qopenmp`
  - `ifort -qopenmp example.F90 -o example`
  - for Intel oneAPI don't use `-fopenmp`,
    it uses LLVM OpenMP RT, might miss Intel extensions

Conditional Compilation

- enabling OpenMP defines `_OPENMP` in preprocessor
  - do not define/undefine `_OPENMP`
- Using `_OPENMP` define for conditional compilation:

C/C++:

```
int thread_id = 0;
#ifdef _OPENMP
thread_id = omp_get_thread_num();
#endif
```

Fortran:

```
integer :: thread_id
thread_id = 0
#ifdef _OPENMP
thread_id = omp_get_thread_num();
#endif
```

  - OpenMP specific API call is guarded off
- for Fortran use sentinels if no preprocessor support is available:
  - free source form: `!$`
  - fixed source form: `*$ !$ c$`

```
!$ thread_id = omp_get_thread_num()
```

`_OPENMP` define

- enabling OpenMP defines `_OPENMP`
  - set to year and month when the supported OpenMP standard was released
  - format `yyyymm` → `yyyy` = year, `mm` = month
- do not define/undefine `_OPENMP`
  - causes undefined behavior
- Fortran: `_OPENMP` requires preprocessor support
  - typically `.F90` files are preprocessed
  - typically `.f90` files are not
    - except corresponding flags are specified

`_OPENMP`	version	date
200805	3.0	May 2008
201107	3.1	July 2011
201311	4.0	Nov. 2013
201511	4.5	Nov. 2015
201611	5.0 preview 1	Nov. 2016
201711	5.0 preview 2	Nov. 2017
201811	5.0	Nov. 2018
202011	5.1	Nov. 2020
202111	5.2	Nov. 2021

Fortran alternatively:

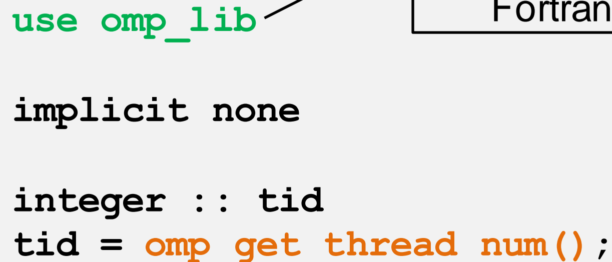
`omp.h` or `omp_lib` module define integer constant `openmp_version` that has the same value as `_OPENMP`

Using Runtime Function and Types

C/C++

```c
#include <omp.h>


int tid = omp_get_thread_num();
```

Fortran




```fortran
use omp_lib

implicit none

integer :: tid
tid = omp_get_thread_num();
```

```fortran
implicit none

include "omp_lib.h"

integer :: tid
tid = omp_get_thread_num();
```

Useful Runtime Functions

- **`int/integer omp_get_thread_num()`**
  - get id of current thread
- **`int/integer omp_get_num_threads()`**
  - get number of threads in current region (sequential or parallel)
- **`int/integer omp_get_max_threads()`**
  - get maximum number of threads in the **next** parallel region without a **`num_threads`** clause

to use API functions:
C/C++: **`#include <omp.h>`**
Fortran: **`use omp_lib`** or **`include "omp.h"`**

- **`double (precision) omp_get_wtime()`**
  - get elapsed time in seconds since some point in time
  - mostly useful for measuring durations
  - might not be synchronized between threads

OpenMP Example – C

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

int main(int argc, char *argv[])
{
  printf("sequential part\n");

  #pragma omp parallel
  {
    printf("thread %d of %d\n",
           omp_get_thread_num(),
           omp_get_num_threads());
  }
  printf("sequential part again\n");
  return 0;
}
```

```
$ gcc -fopenmp -o a.out omp.c
$ OMP_NUM_THREADS=<n> ./a.out
```

include OpenMP header for using runtime functions, **`omp_get_thread_num()`** and **`omp_get_num_threads()`**

`parallel` construct, each created thread executes the associated block

each thread prints its thread id and the total no. of threads inside the parallel region

at the end all threads wait in an implicit barrier until all have arrived, then execution continues sequentially

without **`OMP_NUM_THREADS`**, number of threads used is implementation specific

OpenMP Example – Fortran

```
program omp_application
  use omp_lib

  implicit none

  print *, "sequential part"

  !$omp parallel
    print *, "thread ", &
          omp_get_thread_num(), &
          " of ", &
          omp_get_num_threads()
  !$omp end parallel

  print *, "sequential part again"
end program
```

```
$ gfortran -fopenmp -o a.out omp.F90
$ OMP_NUM_THREADS=<n> ./a.out
```

use OpenMP module `omp_lib` for using runtime functions

`parallel` construct, each created thread executes the associated block

each thread prints its thread id and the total no. of threads inside the parallel region

at the end all threads wait in an implicit barrier until all have arrived, then execution continues sequentially

without `OMP_NUM_THREADS`, number of threads used is implementation specific

Exercise

- in directory `00-preparation`
  - ensure C or F90 OpenMP source compiles correctly
  - which OpenMP standard is supported
  - run with different no. of OpenMP threads

`parallel` Construct

`parallel` Construct

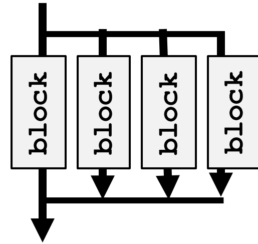
each thread executes the same associated statement/structured block

C/C++:

```
#pragma omp parallel [clauses...]
structured block
```

Fortran:

```
!$omp parallel [clauses...]
structured block
!$omp end parallel
```



- *associated* structured block is executed in a fork-join fashion
  - when a thread encounters a parallel region, all threads start executing the associated statement/block
- afterwards threads wait in **implicit barrier at the end** until all threads have arrived

Structured Block

- statement or multiple statements
  - C/C++: statements grouped together in { } block
  - Fortran: blocks require `!$omp … / !$omp end …`
    - or Fortran 2008: `BLOCK / END BLOCK` ≥v5.1
- one entry at the top, one exit at the bottom
  - C/C++:
    - calling `exit()`, `_Exit()`, `quick_exit()`, `abort()` OK
    - `throw`, `co_await`, `co_yield`, `co_return` OK, if the entry/exit criteria is not violated
  - Fortran: `STOP` or `ERROR STOP` OK
- no branch into or branch out

Clauses for `parallel` Construct

syntax:
```
parallel [clauses...]
structured block
```

- `if (expression)`
  - parallel execution of the region depends on value of expression, e.g.

    ```
    #pragma omp parallel if (!omp_in_parallel())
    ```
    avoids nested parallel regions
  - if `omp_in_parallel()` evaluates to
    - true: parallel region is executed in parallel,
    - false: region is executed serially
  - **active** parallel region: executed by > 1 thread
  - **inactive** parallel region: executed by one thread

Clauses for `parallel` Construct

- `num_threads(int-expr)`
  - no. of threads to execute parallel region with
  - integer must be > 0
  - overrides env. var. `OMP_NUM_THTREADS`
- `proc_bind(keyword)`
  - bind threads to certain places
  - keyword: can be `close`, `master` (deprecated), `primary`, `spread`
  - see later at Runtime Functions and Environment Variables
- clauses:
  - `shared`, `private`, `firstprivate`, `default`, `reduction`,
  - `copyin` → see later thread private memory

`barrier` Construct

NHR FAU

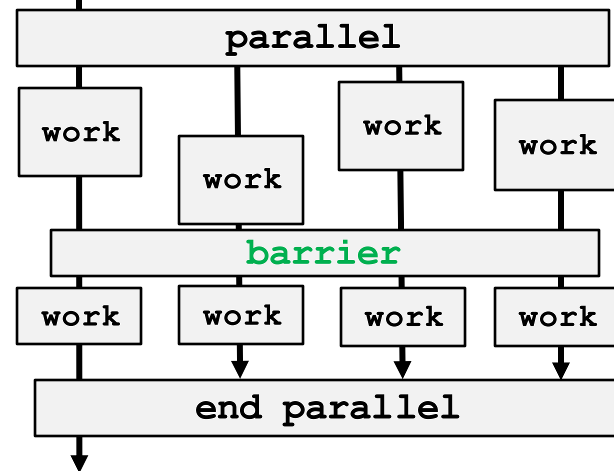
FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

HPC High Performance Computing

barrier construct

- all threads in current parallel region must enter the barrier before they can continue
- explicit barrier
  - in contrast to implicit barrier at the end of some constructs (parallel, …)
- used for
  - synchronization
  - debugging
- try to avoid

```
#pragma omp parallel
{
  /* work */

  #pragma omp barrier

  /* work */
}
```

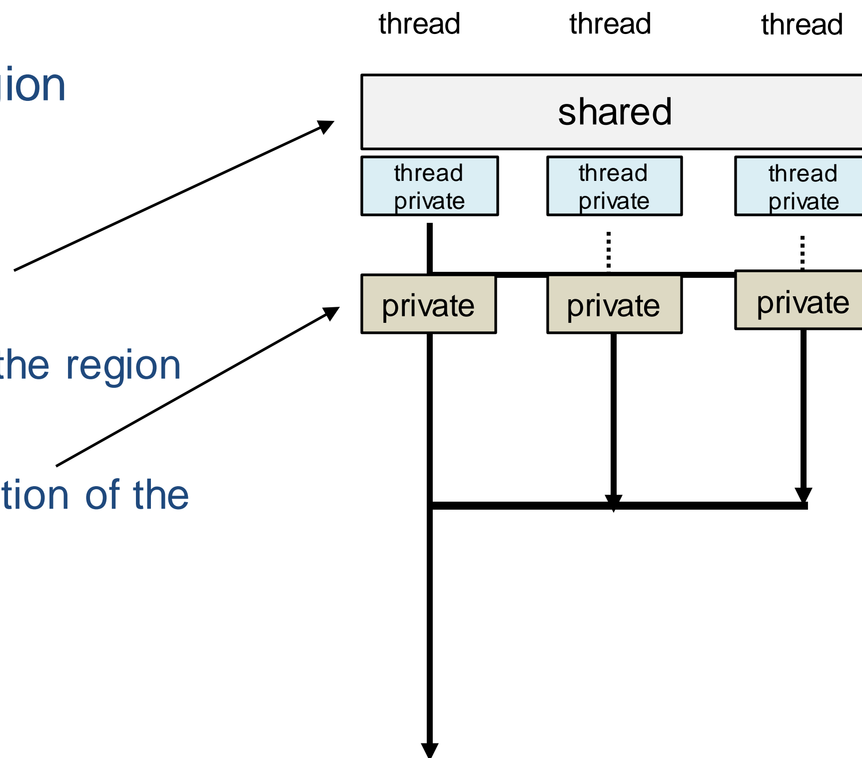


Data Sharing

NHR FAU
FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
HPC High Performance Computing

Data Environment

- **data environment**
  - the variables associated with a region
- variables there can be
  - shared
    - accessible by all threads executing the region
  - private
    - each thread has its own for the duration of the region
  - threadprivate
    - see later



Data Sharing Attributes

- by default most variables are shared
  - static/global (C/C++) or save/common (Fortran) variables
  - local variables outside the scope of construct
- except
  - variables* defined inside the construct are private
    - i.e. declared inside {}-block or BLOCK/END BLOCK
  - variables* local to functions/routines called from within the region are private
  - loop iteration variables of worksharing loops are private
    - see later for do/for construct

```
int g = 1;

void fn() {
  static int a = 0;
  int s = 1;

  #pragma omp parallel
  {
    static int b = 1;
    int p = omp_get_thread_num();
    printf("s=%d p=%d g=%d\n",
           s, p, g);
  }
}
```

* non-static (C/C++) or without save attribute (Fortran)

Example – 1

```c
int p = 1;

#pragma omp parallel
{
  /* p shared or private */
}
```

```fortran
integer :: p
p = 1

!$omp parallel
  ! p shared or private
!$omp end parallel
```

Example – 2

```c
void foo() {
  static int i = 1;

  #pragma omp parallel
  {
    /* i shared or private? */
  }
}
```

```fortran
subroutine foo()
  integer, save :: i = 1

  !$omp parallel
    ! i shared or private?
  !$omp end parallel
end subroutine
```

Example – 3

```c
#pragma omp parallel
{
  int j = 2;
  /* j shared or private? */
}
```

```fortran
!$omp parallel
block
  integer :: j
  j = 2
  ! j shared or private?
end block
!$omp end parallel
```

Example – 4

```
#pragma omp parallel
{
  static int j = 2;
  /* j shared or private? */
}
```

```
!$omp parallel
block
  integer, save :: j = 2
  /* j shared or private? */
end block
```

Example – 5

```c
void foo() {
  int k = …
  /* k shared or private? */
}

void bar() {
  #pragma omp parallel
  {
    foo();
  }
}
```

```fortran
subroutine foo()
  integer :: k
  ! k shared or private?
end subroutine

subroutine bar()
  !$omp parallel
    call foo()
  !$omp end parallel
end subroutine
```

Example – 6

```c
void foo() {
   static int l = …
   /* l shared or private? */
}

void bar() {
   #pragma omp parallel
   foo();
}
```

```fortran
subroutine foo()
   integer, save l = …
   ! l shared or private?
end subroutine

subroutine bar()
   !$omp parallel
     call foo()
   !$omp end parallel
end subroutine
```

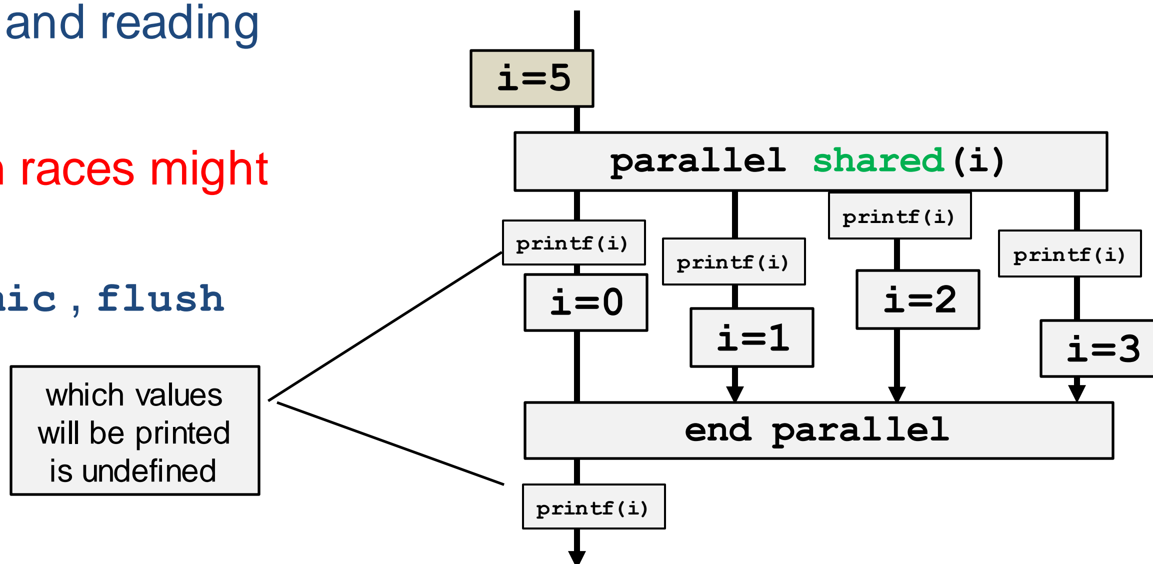
Data-Sharing Attribute Clauses

- clauses for explicitly specifying how a variable should be treated
  - supported by several directives, e.g., `parallel`, `do`/`for`, `single`, `sections`, `task`, …
- clauses:
  - `shared(var1, var2, …)`
  - `private(var3, var4, …)`
  - private + special operation
    - `firstprivate(var5, var6, …)`
    - `lastprivate`, see later `do`/`for` construct clauses
- change default: `default(shared|private|firstprivate|none)`
  - C/C++: `default(shared|none)` ≤ v5.0
  - best practice: `default(none)`
    - every variable referenced must appear in a `shared`/`private`/… clause
    - avoids incorrect assumptions about shared/private

`shared` clause

- treat listed variables as shared
- be careful when
  - concurrently writing to
  - concurrently writing and reading
- shared variables
  - without coordination races might occur
  - see `critical`, `atomic` , `flush`

```
int i = 5;

#pragma omp parallel shared(i)
{
  printf("%d\n", i);
  i = omp_get_thread_num();
}
printf("%d\n", i);
```

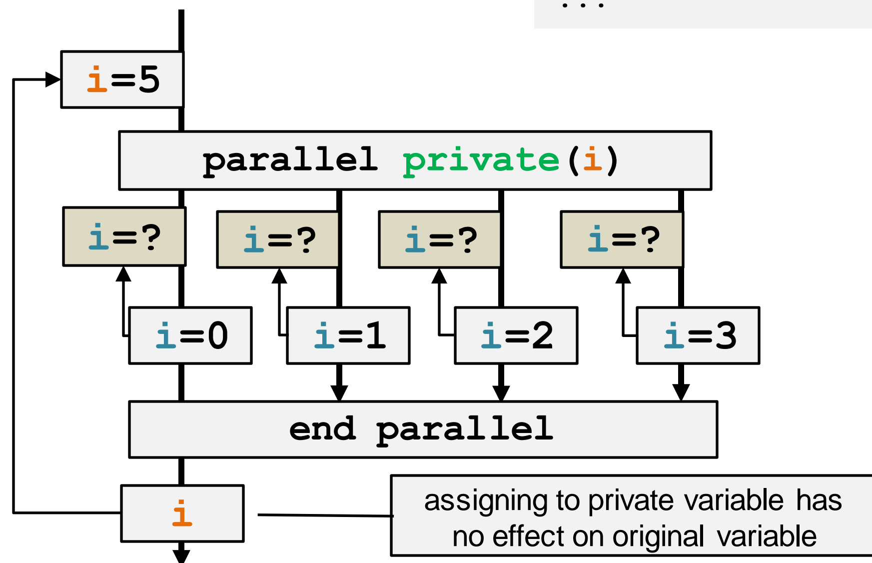


`private` clause

- **new uninitialized** variable of the same type and name as the original one
  - declared locally without initializer
- `private` variables disappear after the end of the region
- privatized variables **hide** original variables
- C++: privatized variable initialization depends on default ctor

```
int i = 5;

#pragma omp parallel private(i) \
                     num_threads(4)
{
  i = omp_get_thread_num();
}
printf("%d\n", i); // prints: 5
```

```
int i = 5;

#pragma omp parallel
{
  int i;
  ...
}
...
```

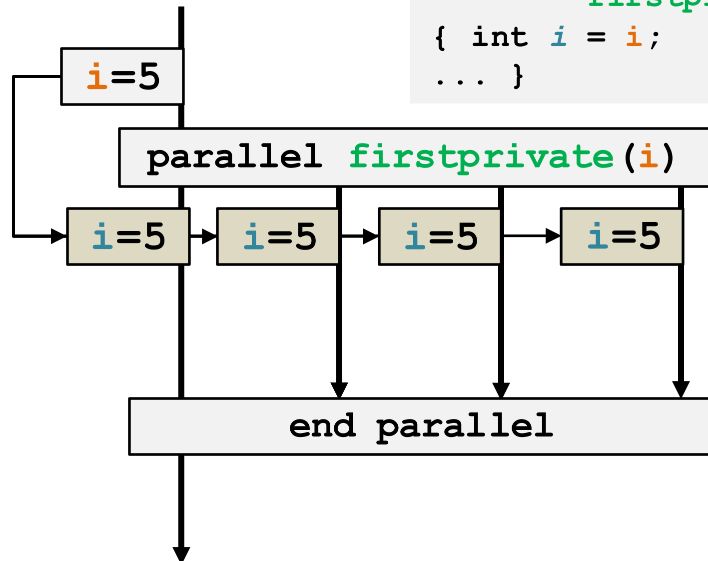


firstprivate clause

- listed variables become
  - private and
  - initialized with value from original variable

```
int i = 5;

#pragma omp parallel \
        firstprivate(i)
{
  printf("%d\n", i);
  /* prints 5 for each thread */
}
```

```
#pragma omp parallel \
        firstprivate(i)
{ int i = i;
... }
```



- C++: initialized via copy assignment (copy ctor)

```
struct X { };
X x;
#pragma omp parallel firstprivate(x)
{
  // X x = x
}
```

flush Construct

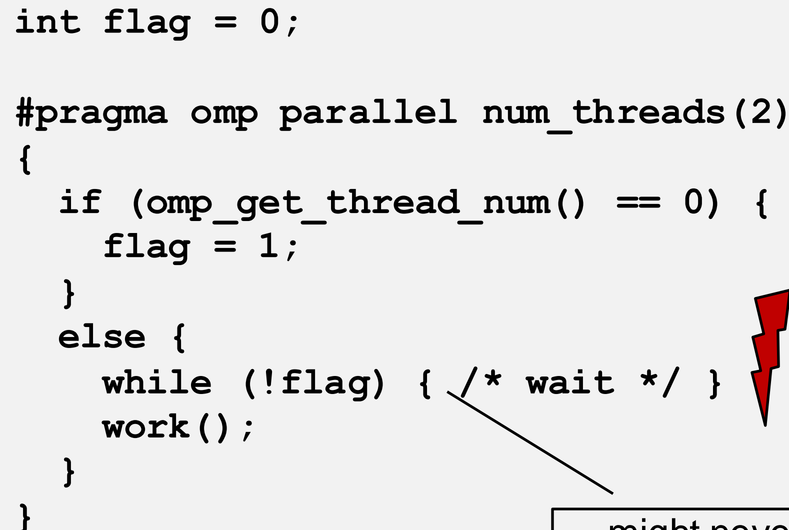
NHR FAU

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

HPC High Performance Computing

`flush` directive

- threads can have a temporary view to memory
- writes from one thread do not need to
  - visible to other threads immediately
  - reflected in memory
- `flush` in the encountering thread
  - writes changes to memory
  - updates variables from memory



```
int flag = 0;

#pragma omp parallel num_threads(2)
{
  if (omp_get_thread_num() == 0) {
    flag = 1;
  }
  else {
    while (!flag) { /* wait */ }
    work();
  }
}
```


- implicitly happens at
  - explicit/implicit barriers
  - entry/exit of critical sections and lock functions

`flush` directive

- can be limited to certain variables by providing a list
- `flush` is implied in all implicit/explicit barriers

```
int flag = 0;

#pragma omp parallel num_threads(2)
{
  if (omp_get_thread_num() == 0) {
    flag = 1;
    #pragma omp flush
  }
  else {
    #pragma omp flush
    while (!flag) {
      /* wait */
      #pragma omp flush
    }
    work();
  }
}
```

works but inefficient, see atomic construct later

NHR FAU

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Worksharing Constructs for Loop Parallelization

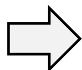
`for` / `do` construct

High Performance Computing

Manually Parallelize Loops

- sometimes it is necessary to manually distribute iterations of a loop over threads
- distribute iterations (nearly) equally across threads:

```
for (int i = 0; i < n; ++i) {
  /* work */
}
```



```
int n = ...;
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  int nt = omp_get_num_threads();
  int per_thread = n / nt;
  int rem = n % nt;
  int lb  = tid * per_thread;
      lb += tid < rem ? tid : rem;
  int ub  = lb + per_thread;
      ub += tid < rem ? 1 : 0;

  for (int i = lb; i < ub; ++i) {
    /* work */
  }
}
```

`for/do` Construct

C/C++ `for [clauses]`

```
#pragma omp for
for (i = 0; i < n; ++i) {
  <structured block>
} // impl. barrier
```

Fortran `do [clauses]`

```
!$omp do
do i = 1, n
 <structured block>
end do
! impl. barrier
!$omp end do
```

- `for` and `do` distribute the iterations of one or more **associated** loops over threads
- loop counter will be private
- implicit barrier at the end
- require an enclosing `parallel` region, else *orphaned* construct

clauses:
`private`, `firstprivate`, `lastprivate`, `reduction`, `nowait`, `schedule`, `collapse`, …

- **ensure loop iterations are independent**

Example

C/C++

```
const int n = 1000; double a[n], b[n];
/* initialize b */
#pragma omp parallel
{
  #pragma omp for
  for (int i = 0; i < n; ++i) {
    a[i] = b[i]
  }
}
```

Fortran

```
integer, paramter :: n = 1000
integer :: i
real(8) :: a(n), b(n)
! initialize b
!$omp parallel
  !$omp do
  do i = 1, n
    a(i) = b(i)
  end do
  !$omp end do
!$omp end parallel
```

iteration space distributed over threads

implicit barrier at the end

- How are the iterations distributed over threads?
  - without specification it is implementation defined
  - specify via `schedule` clause

Fortran special

loop counter variable of associated loop is always private

C/C++

```
int i, j;
#pragma omp parallel
#pragma omp for
for (i = 0; i < N; ++i) {
   for (j = 0; j < N; ++j) {
   }
}
```

shared

Fortran

```
!$omp parallel
!$omp do
do i = 1, N
   do j = 1, N
   end do
end do
!$omp end do
!$omp end parallel
```

only Fortran: loop counter variables inside parallel region are always private (if not specified otherwise)

Combined `parallel do/for` Construct

- **`parallel`** and **`for/do`** construct can be merged

```
#pragma omp parallel for \
        [clauses parallel + for]
```

```
!$omp parallel do [clauses parallel + do]
...
!$omp end parallel do
```

- all clauses of **`parallel`** and **`for/do`** can be applied except **`nowait`**
- only one implicit barrier at the end

Reductions

NHR FAU
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
High Performance Computing

reduction clause

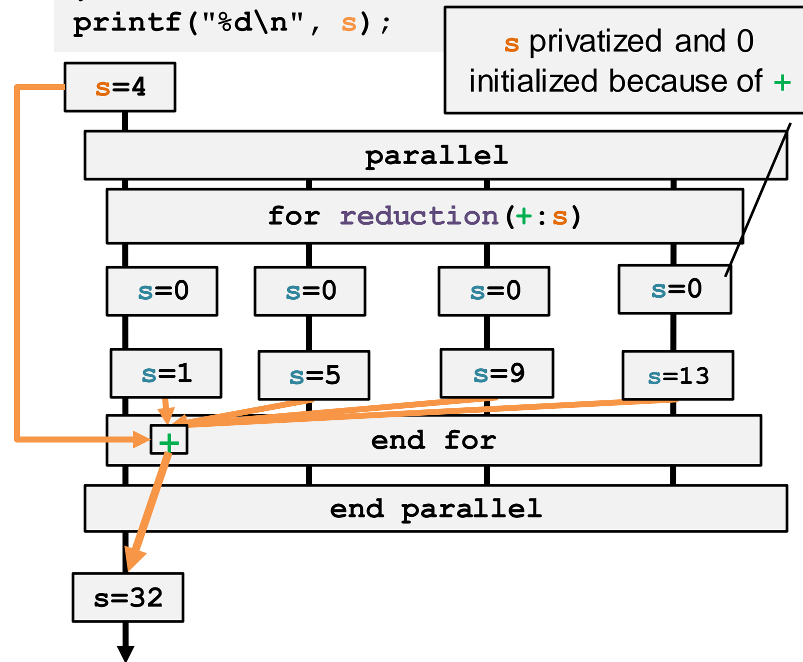
```
int s = 4, n = ...;
#pragma omp parallel
{
  #pragma omp for reduction(+:s)
  for (int i = 0; i < 8; ++i)
    s += i;
}
printf("%d\n", s);
```

reductions allow for aggregating values of private variables computed in parallel

```
reduction(rid:variables)
```

- rid: reduction identifier
  - operation to perform: +, -, *, …
- variables: variables to reduce
  - privatizes listed variables
  - listed variables must be shared
- according to rid variables are
  - initialized + combined at the end

supported by directives:
parallel, for/do, sections, simd, loop, scope, taskloop, teams (not handled here)



`reduction` clause

- predefined reductions for arithmetic types:

C/C++		Fortran	
rid	initializer	rid	initializer
+	0	+	0
-	0	-	0
*	1	*	1
&	~0	.and.	.true.
|	0	.or.	.false.
^	0	.eqv.	.true.
&&	1	.neqv.	.false.
||	0	max	min(type)
max	min(type)	min	max(type)
min	max(type)	iand	all bits one
		ior	0
		ieor	0

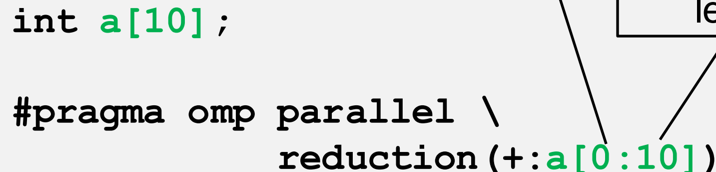
- multiple reductions possible:

```
reduction(+:a, b) reduction(*:c, d)
```

  - listed variable is only allowed to occur in one `reduction` clause

Reduction on Array (Sections)

- variables in reduction clauses can be arrays or sections of arrays
- requirements:
  - need to be contiguous
  - C/C++: requires array section syntax



- initialization/reduction is performed elementwise

```
integer :: a(10)

!$omp parallel reduction(+:a)
```

Exercise

- Pi computation

User-defined Reductions – Syntax

- extend predefined ***rid*** for different types ***types*** (comma separated)
- introduce own identifiers for ***rid***

```
declare reduction(rid : types : combiner) \
        initializer(init_expr)
```

initializer

how private variable **omp_priv** is initialized (optionally with **omp_orig**):

init_expr:

```
omp_priv = initializer (C/C++)
omp_priv initializer (C++)
function-name(argument-list) (C/C++)
omp_priv = expression (F)
subroutine-name(argument-list) (F)
```

combiner

describe how private instances are reduced:

```
omp_out <op>= omp_in
omp_out = omp_in <op> omp_out
fn(…, &omp_out, …, &omp_in, …)
```

special variables:

input: **omp_in**

input and output: **omp_out**

User-defined Reductions – Example

implement "+" reduction for `std::complex<double>`:

```
#pragma omp declare \
        reduction(+: std::complex<double>: omp_out += omp_in) \
        initializer(omp_priv std::complex<double>{})

…
std::complex<double> c;

#pragma omp parallel for reduction(+:c)
for (size_t i = 0; i < n; ++i) {
  c += i;
}
```

<rid>

type(s)

combiner expression
`omp_in`: input value(s)
`omp_out`: in-/output value

initialize private copies:
`omp_priv` represents copy

perform + reduction on `c`

User-defined Reductions

- more complex combiner expressions, e.g., for `std::vector<T>`:
  - must be specified for each type `T` used

<rid>

`type(s)`, fill in as needed

combiner expression is an STL algorithm

```
#pragma omp declare \
  reduction(+ : std::vector<double> : std::transform(omp_out.begin(), \
    omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<double>())) \
  initializer(omp_priv = decl_type(omp_orig)(omp_orig.size()))
…
std::vector<double> v(n, 0.0);

#pragma omp parallel for reduction(+:v)
for (size_t i = 0; i < n; ++i) {
  v[i] = static_cast<double>(i);
}
```

create a new `std::vector` with `omp_orig.size()` zeros,

perform + reduction on `v`

User-defined Reductions

- more complex combiner expressions for `std::vector<T>`:
  - must be specified for each type `T` used

combiner expression is a lambda invocation

```
#pragma omp declare \
  reduction(+ : std::vector<double> : \
    [](decltype(omp_out) & out, decltype(omp_in) & in) { \
      for (size_t i = 0ul; i < out.size(); ++i) { out[i] += in[i]; } \
    }(omp_out, omp_in)) ) \
  initializer(omp_priv decltype(omp_orig)(omp_orig.size()))
…
```

scope construct

≥ v5.0

- introduces a new “scope” inside a parallel region
- mostly used to perform reductions

```
scope [reduction(rid:var-list)] [allocate] \
      [firstprivate] [private] [nowait]
```

  - implicit barrier at the end

```
int sum = 0;
#pragma omp parallel
{
  ...
  #pragma omp scope reduction(+:sum)
  {
    sum += omp_get_thread_num();
  }
  ...
  /* use sum */
  ...
}
```

impl. barrier

Worksharing Constructs for Loop Parallelization

`for` / `do` construct clauses

NHR FAU

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

HPC High Performance Computing

Loop Schedules

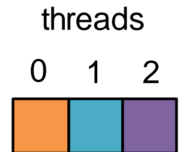
- `for`/`do` `schedule` clause supports different schedules
  - static
    - fixed size chunks, static chunk to thread mapping
  - dynamic
    - fixed size chunks, non-deterministic mapping of chunks to threads
  - guided
    - reduce chunk size over time, non-deterministic mapping of chunks to threads
  - auto
    - implementation defined
  - runtime
    - choose schedule at runtime, either programmatically or via environment variable

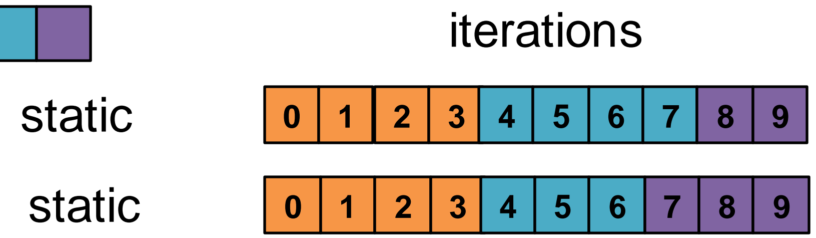
`schedule` clause: `static`

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

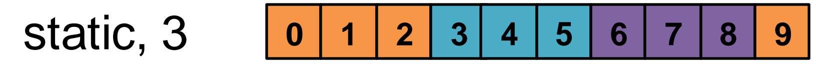
- effect
  - chunks are distributed round-robin over threads
  - mapping of chunk to thread is **deterministic**
- *`chunk_size`* is an integer expression:
  - chunks will have *`chunk_size`* iterations
    - last chunk can be smaller
  - without *`chunk_size`*:
    - each thread gets exactly **one** (nearly) equally sized chunk

```
#pragma omp parallel num_threads(3)
#pragma omp for schedule(static)
for (int i = 0; i < 10; ++i) {
  /* work */;
}
```






```
#pragma omp parallel num_threads(3)
#pragma omp for schedule(static, 3)
for (int i = 0; i < 10; ++i) {
  /* work */
}
```

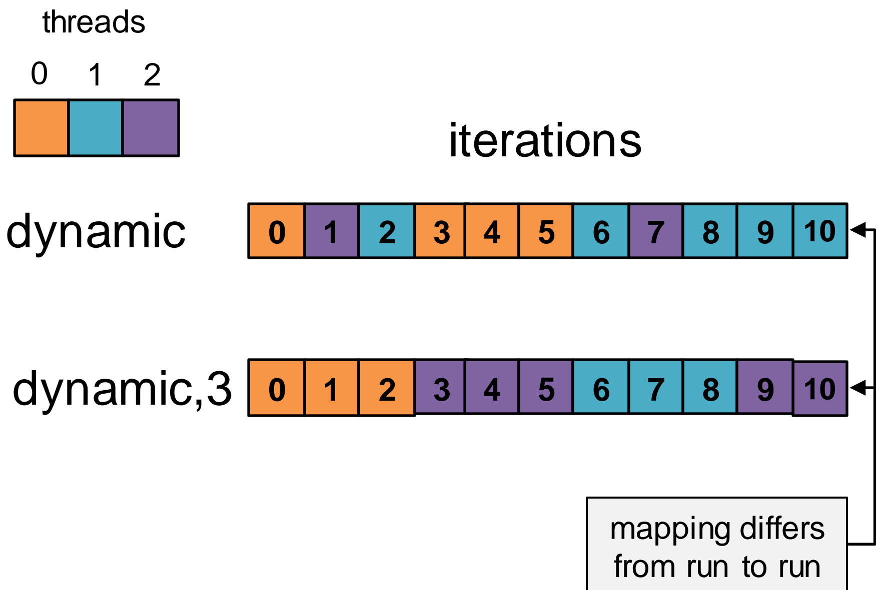


`schedule` clause: `dynamic`

```
schedule(dynamic[,<chunk_size>])
```

- iterations are divided into chunks of size `chunk_size`
  - `chunk_size` is 1 if not provided
  - `chunk_size` is an integer expression
- threads request next chunk when they have finished one
  - no strict round-robin assignment of chunks to threads
  - no deterministic mapping of thread to chunk
- NOTE: if load balancing is an issue, dynamic is better suited than static

```
#pragma omp parallel num_threads(3)
#pragma omp for schedule(dynamic, 3)
for (int i = 0; i < 11; ++i) {
  /* work */
}
```

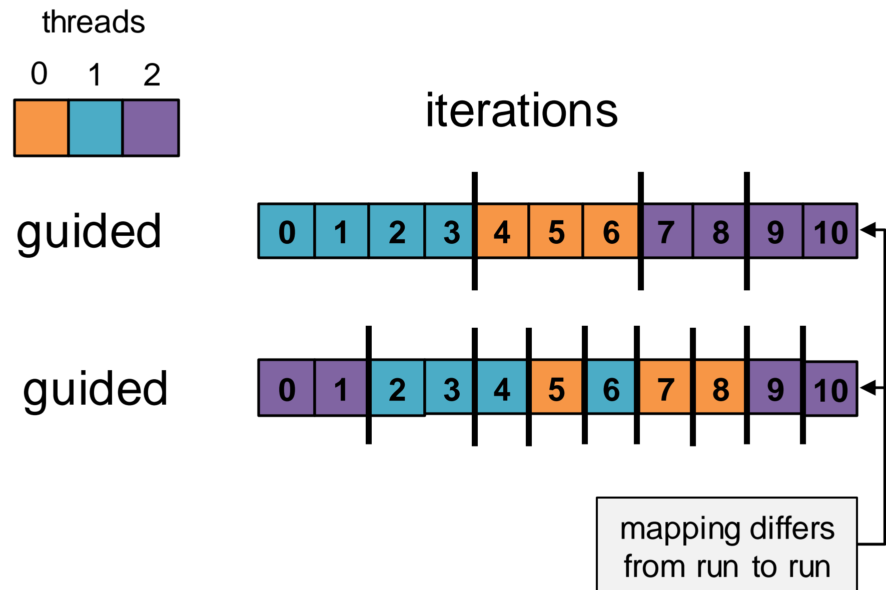


schedule clause: guided

```
schedule(guided[,<chunk_size>])
```

- larger chunks at the beginning, getting smaller to the end of the iteration space
- `chunk_size` = 1 by default
- `chunk_size` = k, k > 1
  - smallest `chunk_size` will be k except for maybe the last iteration
- chunk to thread mapping non-deterministic
- NOTE: useful when load imbalances exist, less overhead than dynamic

```
#pragma omp parallel num_threads(3)
#pragma omp for schedule(guided)
for (int i = 0; i < 11; ++i) {
  /* work */;
}
```



`schedule` clause: `auto` and `runtime`

- **`auto`**:
  - implementation defined scheduling will be used

- **`runtime`**: chosen at runtime
  - initial value is implementation defined
  - set via environment variable
    `OMP_SCHEDULE=[modifier:]kind[,chunk_size]`
    - `kind`: `static`, `dynamic`, `guided`, or `auto`
    - `chunk_size`: same meaning as if specified at schedule
    - run binary with: `OMP_SCHEDULE=static,512 ./openmp-binary`
  - set via runtime call `omp_set_schedule(omp_sched_t kind, int chunk_size)`
  - retrieve runtime scheduling:
    - `omp_get_schedule(omp_sched_t *kind, int *chunk_size)`

order of chunks

```
schedule([monotonic    ] : <schedule>[,<chunk_size>])
         [nonmonotonic ]
```

- `monotonic`
  - thread processes assigned chunk in increasing logical order
- `nonmonotonic`
  - chunks are assigned to threads in any order
- not allowed: `nonmonotonic` and `ordered` clause
- do not depend on the order of the execution of the chunks (except for `ordered`)
- defaults
  - ≤ v4.5: `monotonic`
  - ≥ v5.0: `nonmonotonic` (except for static and ordered)

`collapse(n)` clause

- merge iteration space of n nested loops into one *logical iteration space*
  - distribute resulting iterations over threads
- these `n` loops become *associated loops*
- typically more evenly distributed no. of iterations

```
#pragma omp parallel
#pragma omp for collapse(2) \
             schedule(static) \
            num_threads(3)
for (int x = 0; x < 5; ++x) {
  for (int y = 0; y < 5; ++y) {
    /* work on (x, y) */
} }
```

- distribute `5*5` instead of `5` iterations
- distribution of iterations over 3 threads:
  - without collapse: 2, 2, 1
  - with collapse: 9, 8, 8 or 9, 9, 7

`collapse(n)` clause

- NOTE: only as many loops are associated as specified
  - through `collapse(2)` only loop over `x` and `y` is associated
  - loop over `z` is not associated

```
!$omp parallel &
!$omp for collapse(2) &
!$omp     schedule(static)
do x = 1, nx
  do y = 1, ny
    do z = 1, nz
      ! work on (x, y, z)
    end do
  end do
end do
!$omp end do
!$omp end parallel
```

`lastprivate` clause

```
lastprivate([conditional:]var1[,...]])
```

- listed variables are privatized
- value from sequentially last iteration is assigned to original variable
  - if sequentially last iteration is not performed or no assignment takes place
    - value of variable after construct is undefined
  - use `conditional` modifier
    - original variable gets value of sequentially last assignment

```
int i, s = 1000;
int u = -2, n = 100;
#pragma omp parallel
#pragma omp for shared(n) \
            lastprivate(s, u)
for (i = 0; i < n; ++i) {
  if (i < n - 1) u = i;
  s = i + 3;
}

printf("s=%d u=%d n=%d\n",
       s, u, n);

// output:
// s=102 u=??? n=100
```

no assignment in last iteration

assignment in every iteration

undefined value of `u`, use `conditional` modifier

- also supported: `private`, `firstprivate`

ordered construct and clause

execute parts of a loop's body in its original sequential order, e.g. for output, loop dependencies

```
#pragma omp parallel
#pragma omp for ordered
for (int i = 0; i < ni; ++i) {
  /* P1 */
  #pragma omp ordered
  { /* P2 */ }
  /* P3 */
}
```

1. requires `for`/`do` loop* with clause:

```
for … ordered[(n)]
```

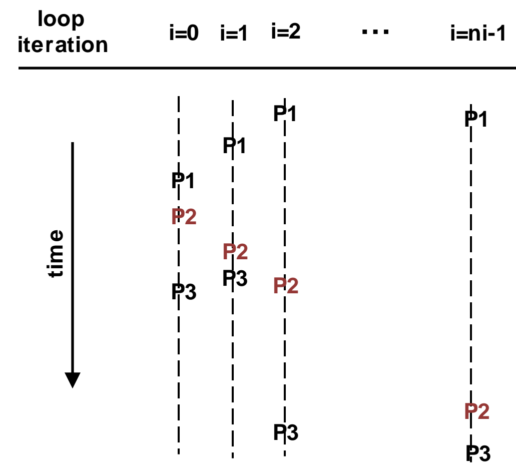
```
do … ordered[(n)]
```

2. requires construct:

```
ordered
{ structed-block }
```

```
ordered
structued-block
end ordered
```

- construct must be nested within region that has an `ordered` clause



* or simd construct, see part 2 of this course

`ordered` clause restrictions

- clause

```
for … ordered[(n)]
```

```
do … ordered[(n)]
```

  - **n** specifies how many **perfectly** nested loops are associated
  - **n** must be ≤ no. of associated loops (by `collapse`)
  - if **n** is unspecified, the same no. as specified for `collapse` is used
  - if **n** is specified linear clause must not be specified
  - C++: if **n** is specified, assoc. loops must not be range-based loops
  - associated loops must be rectangular

`nowait` clause

- no implicit barrier at the end of the loop
- threads finished early do not have to wait and can proceed
  - might lead to better utilization of resources

WARNING: use `nowait` only if no race-conditions between the loop and the following code exist

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (...) { /* first loop */ }

  /* work without race between
     the two loops */

  #pragma omp for nowait
  for (...) { /* second loop */ }

  /* more work without race */

} /* all threads will wait here */
```

```
!$omp do
do i = 1, n
...
end do
!$omp end do nowait
```

for Fortran place at the end directive

Canonical Loop Forms

Canonical Loop Forms

- `do`/`for` (and other constructs) require a well formed loop

C/C++:

```
for (init-expr; test-expr; incr-expr)
```

```
var = lb
int-type var = lb
ptr-type var = lb
random-access-it it = lb
```

only one variable allowed

```
var op ub
ub  op var
op: <, <=, >, >=, !=
```

```
++var, var++,
--var, var--,
var += incr,
var -= incr,
var  = var + incr,
var  = incr + var,
var  = var - incr
```

loop iteration variable `var` **must not be modified** by intervening code or loop body

`lb & ub`
loop invariant to outermost associated loop

`incr`:
- integer expression
- loop invariant to associated outermost loop

C++:

```
for (range-decl : range-expr)
```

decl/expr as defined by C++

- range-expr must be invariant to outer loops
- derived iterator must be a random-access-iterator

Canonical Loop Forms

- **do**/**for** (and other constructs) require a well formed loop

Fortran:

```
DO var = lb, ub[, incr]
…
END DO
```

- integer expression
- loop invariant to associated outermost loop

integer type

lb & ub loop invariant to outermost associated loop

loop iteration variable **var** **must not be modified** by intervening code or loop body

Loop Nests

- `collapse(n)` allows for associating loops to `for`/`do` construct
- associated nested loops must meet certain requirements
- `intervening-code` optional
  - if not present: *perfectly nested* loop nest
  - **must not** contain OpenMP directive/API call in region

C/C++:

- no iteration statement
- no `break`, no `continue` for enclosing loop

Fortran:

- no loops
- no array expression
- no `exit`, no `cycle`

```
for (...)
{
  [intervening-code]
  for (...) {
    ...
  }
  [intervening-code]
}
```

```
do ...
  [intervening-code]
  do ...
    ...
  end do
  [intervening-code]
end do
```

must be of the form as described on previous slides

more loops possible

Exercise

- parallelize ray tracer

Worksharing Constructs: `single`, `sections`

NHR FAU

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

HPC High Performance Computing

`single` Construct

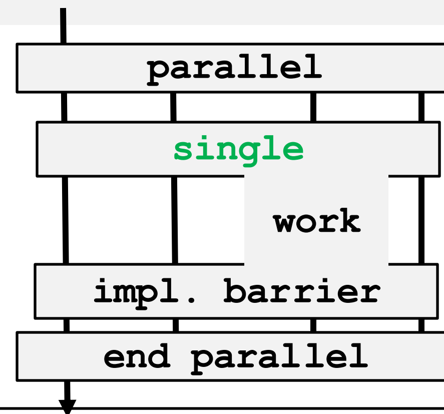
- execute structured block by only one thread

```
single [clauses]
```

```
private,
firstprivate,
copyprivate,
nowait
```

- all other threads wait in impl. barrier at the end
  - remove barrier with `nowait`
- which thread executes the block is implementation defined
  - might change from run to run
- requires surrounding `parallel` region

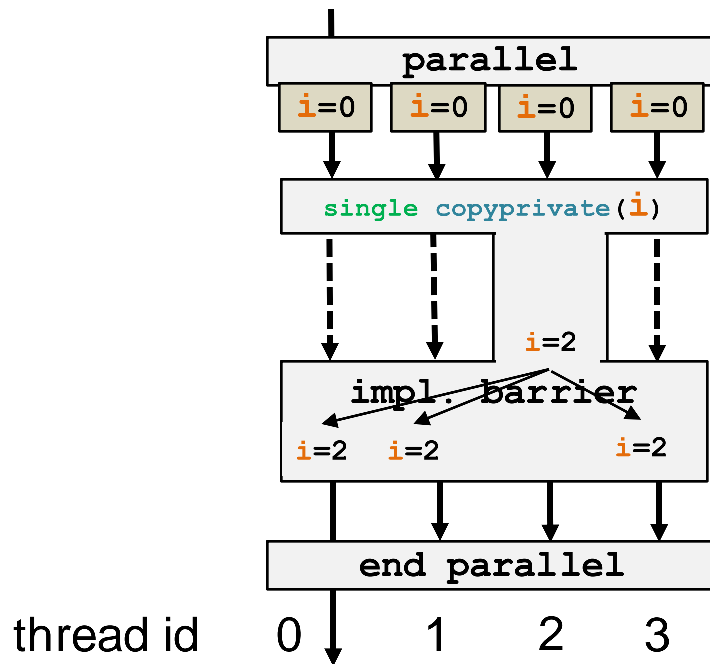
```
#pragma omp parallel
{
  /* work done by every thread */

  #pragma omp single
  {
    /* executed by only one thread */
  } /* impl. barrier */

 /* work done by every thread */
}
```



`copyprivate` clause

- at the end of a single clause:
  broadcast a private variable's value to the other thread’s variable with the same name

```
#pragma omp parallel
{
  int i = 0;
  #pragma omp single copyprivate(i)
  {
    i = omp_get_thread_num();
  }
  /* for all threads i=<thread id> of
     thread that executed the single
     region */
}
```



`sections` Construct

- distribute different structured blocks to threads
- syntax:

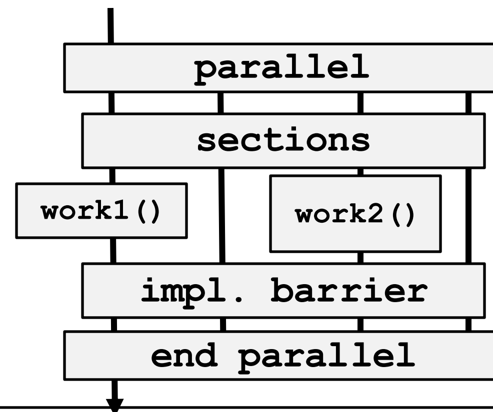
```
sections [clauses]
section
```

- requires a `parallel` region
- each `section` marks a structed block to be distributed
- implicit barrier at the end of `sections` construct
  - remove with `nowait`
- mapping of structured blocks to threads is implementation defined

```
#pragma omp parallel
#pragma omp sections
{
  #pragma omp section
  work1();

  #pragma omp section
  work2();
} /* impl. barrier */
```

first `section` directive is optional

`private, firstprivate, lastprivate, reduction, nowait`



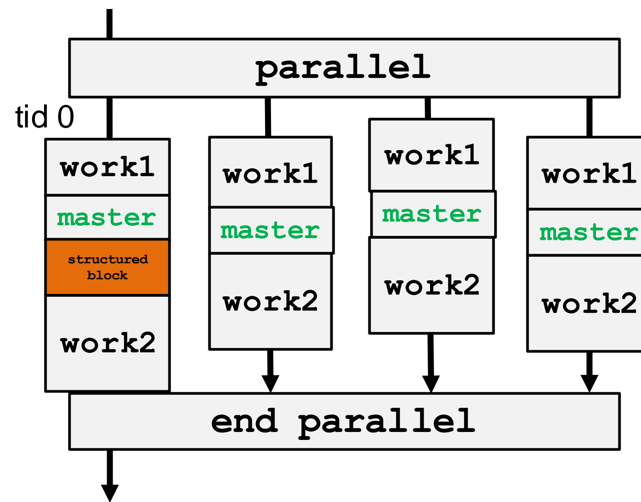
Constructs: `master`, `masked`

NHR FAU
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
High Performance Computing

`master` construct – deprecated

- syntax:
```
master
structured block
```
- equivalent to
```
if (omp_get_thread_num() == 0) {
  /* structured block */
}
```
- execute structured block by thread with ID 0
- no implicit barrier at the end
- deprecated since v5.1 in favor of **`masked`**

```
#pragma omp parallel
{
  /* work1 */

  #pragma omp master
  /* executed by thread 0 */

  /* work2 */
}
```



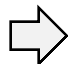
`masked` construct

≥ v5.1

- execute structured block only by certain threads
- syntax: `masked [filter(expr)]`
  - without `filter`, thread with id 0 is used
  - `expr` in filter returns the thread id(s) to use
  - executing thread compares value of *`expr`* with its ID
- no barrier on entry or end of construct
- replaces **deprecated** `master` construct since v5.1

```
#pragma omp parallel num_threads(2)
{
  #pragma omp masked filter(1)
  {
    /* executed by thread 1 */
  } /* no barrier here */
  ...
}
```

```
#pragma omp parallel
{
  #pragma omp master
  { /* tid 0 */ }
  ...
}
```



```
#pragma omp parallel
{
  #pragma omp masked
  { /* tid 0 */ }
  ...
}
```

Synchronization: `critical` construct

`critical` construct

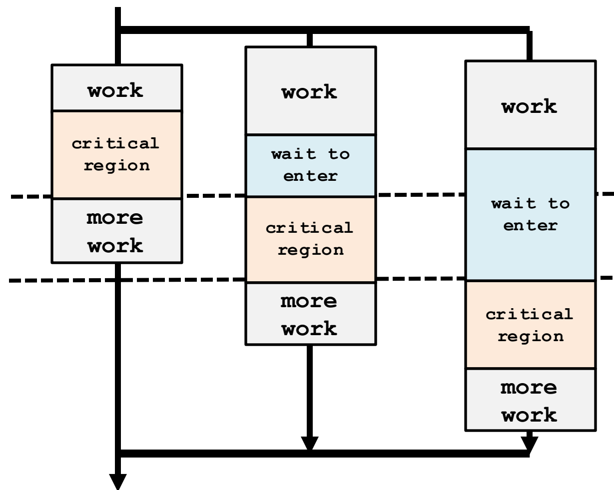
- useful for coordinating access to shared resources

syntax:
```
critical [(name)]
structured-block
```

- associated block can only be entered by **one** thread at a time

- all `critical` regions with the same name belong together
- all unnamed `critical` regions belong together

```
#pragma omp parallel
{
  /* work */
  #pragma omp critical
  { … }
  /* more work */
}
```

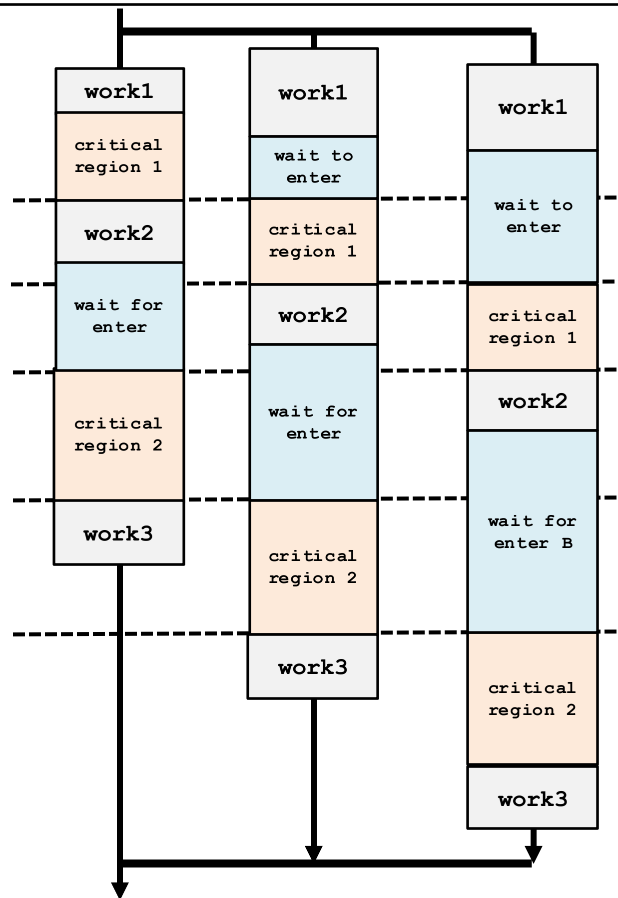


`critical` construct

C/C++:

```
#pragma omp parallel
{
  /* work1 */
  #pragma omp critical
  { ... } /* 1 */
  /* work2 */
  #pragma omp critical
  { ... } /* 2 */
  /* work3 */
}
```

- if the two critical regions don’t have to be exclusive to each other the **`name`** argument provides a way to lift this restriction by giving each region a different name

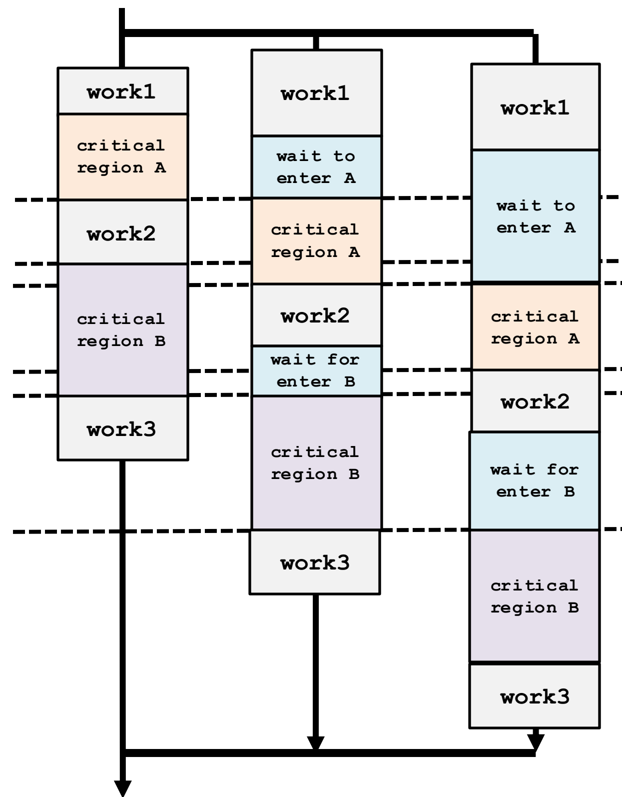


`critical` construct

C/C++:

```
#pragma omp parallel
{
  /* work1 */
  #pragma omp critical(A)
  { ... }
  /* work2 */
  #pragma omp critical(B)
  { ... }
  /* work3 */
}
```

- if the two critical regions don't have to be exclusive to each other the **`name`** argument provides a way to lift this restriction by giving each region a different name



critical construct

Fortran:

```
!$omp parallel
  ! work1
  !$omp critical(A)
  ...
  !$omp end critical(A)
  ! work2
  !$omp critical(B)
  ...
  !$omp end critical(B)
  ! work3
!$omp end parallel
```

end **critical** requires the same name as the corresponding **critical** directive

`critical` construct

- the exclusive access to `critical` regions
  - without a name
  - or with the same name

  works throughout the application
- is not restricted to the lexically surrounding code

```
void work1(var) {
  #pragma omp critical
  { /* work on var */ }
}

void work2(var) {
  #pragma omp critical
  { /* work on var */ }
}
```

critical regions in `work1` and `work2` will never be entered by two threads at the same time

`critical` construct

Performance:

- keep amount of code inside `critical` regions as short as possible
  - reduces amount of waiting time for other threads
- for coordinating access to **single variables** `atomic` construct might be better suited

NHR FAU

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Synchronization: Atomics

High Performance Computing

atomic update

- update a storage location **atomically**
  - read,
  - compute,
  - write

  } performed as if it is one operation and other threads cannot interfere

- mutually exclusive to all other threads using atomic updates
- uses hardware instructions, if available
  - faster than a critical region
  - requirements regarding alignment have to be met

C/C++

```
#pragma omp atomic [update]
update-expr
```

default

update-expr:

```
x++;    x--;
++x;    --x;
x op= expr;
x = x op expr;
x = expr op x;
```

op: +, *, -, /, &, ^, |, <<, >>

lvalue of scalar type

type must be scalar

Fortran

```
!$omp atomic
update-expr
!$omp end atomic
```

op: +, *, -, /, .AND., .OR., .EQV., .NEQV.

intr: MAX, MIN, IAND, IOR, IEOR

update-expr:

```
x = x op expr
x = expr op x
x = intr(x, expr-list)
x = intr(expr-list, x)
```

scalar expression

scalar var of intrinsic type

atomic update Examples

- counting threads

```
int counter = 0;

#pragma omp parallel
{
  #pragma omp atomic
  counter += 1;
}
```

- counting even and odd numbers

```
int histogram[2] = {0, 0};

#pragma omp parallel for
for (int i = 0; i < N; ++i) {
  #pragma omp atomic
  ++histogram[i % 2];
}
```

WARNING: don't do this in real code, performance is poor
often providing each thread with its own data + reduction yields a better solution

`atomic` `read` and `write`

- atomically read/write value
- requires specification of clause:
  `read` / **`write`**

- further clauses:
  - **`capture`**: obtain original or updated value
  - **`compare`**: conditionally update variable ≥v5.0

C/C++

```
#pragma omp atomic read
v = x
```

```
#pragma omp atomic write
x = expr
```

ensure value of **`x`** is read/written atomically, no part of **`x`** can change until after the operation

Fortran

```
!$omp atomic read
v = x
!$omp atomic read
```

```
!$omp atomic write
x = expr
!$omp atomic write
```

atomic capture

- atomically update a value and keep the value before/after the update
  - before: fetch-and-*op*

```
#pragma omp atomic capture
{ v = x; x += counter; }
```

  - after: *op*-and-fetch

```
#pragma omp atomic capture
{ x += counter; v = x; }
```

C/C++

```
#pragma omp atomic capture
• v = update-expr;
• { v = x; update-expr; }
• { update-expr; v = x; }
```

captured value

updated value

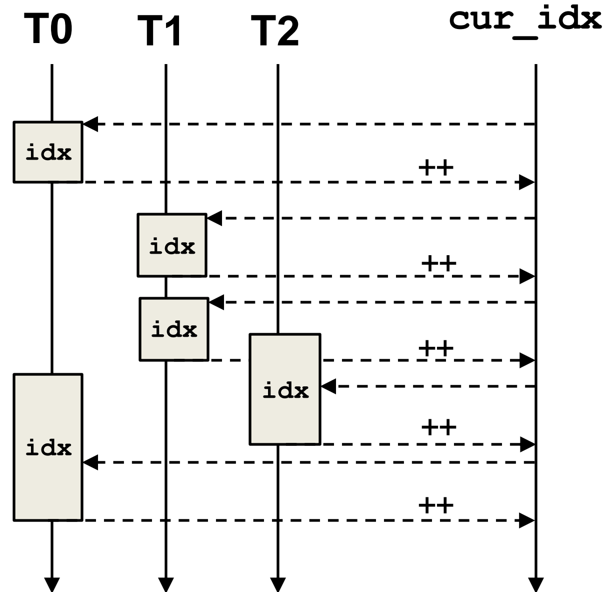
Fortran

```
!$omp atomic capture
statement
v = x
!$omp end atomic
```

```
!$omp atomic capture
v = x
statement
!$omp end atomic
```

atomic capture Example

```
struct items_t { ... };

int n_items = ...;
struct items_t * items = ...;
int cur_idx = 0;

#pragma omp parallel shared(n_items, items, cur_idx)
{
    int idx;
    do {
        #pragma omp atomic capture
        { idx = cur_idx; ++cur_idx; }

        if (idx < n_items) process(&items[idx]);
    } while (idx < n_items);
}
```



WARNING: other constructs might be suited better

Synchronization: Locks

NHR FAU

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

HPC High Performance Computing

Locks

- **runtime functions** for mutual exclusion
- locks are represented by **variables**
- task that *owns* a lock (successfully set a lock) can continue

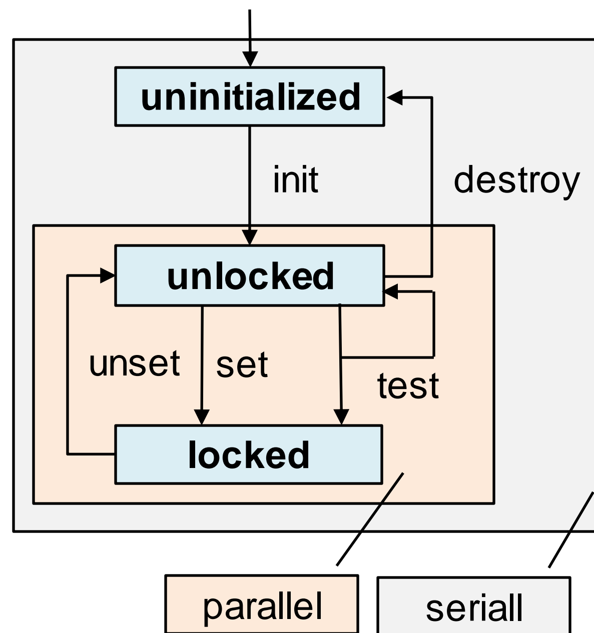
- two types
  - simple locks
    - can only be locked if unlocked
  - nested locks
    - owning task can lock **multiple** times
    - owning task **must unlock the lock the same number of times it locked it**

C/C++	`omp_lock_t`
Fortran	`integer (kind=omp_lock_kind)`

C/C++	`omp_nest_lock_t`
Fortran	`integer (kind=omp_nest_lock_kind)`

Simple Locks

- init: `omp_init_lock(omp_lock_t *)`
- lock
  - set (blocking lock): `omp_set_lock(omp_lock_t *)`
    - on return lock is locked, task owns the lock
    - if already locked, waits until lock becomes unset
    - dead lock if owning task tries to set the lock again
  - test (non-blocking lock) `omp_test_lock(omp_lock_t *)`
    - true: if lock was set
    - false: lock was not set, already set by another task
    - unspecified behavior if calling task already owns the lock
- unset (unlock): `omp_unset_lock(omp_lock_t *)`
  - lock must be in locked state
  - task unlocking must own the lock
- destroy: `omp_destroy_lock(omp_lock_t *)`

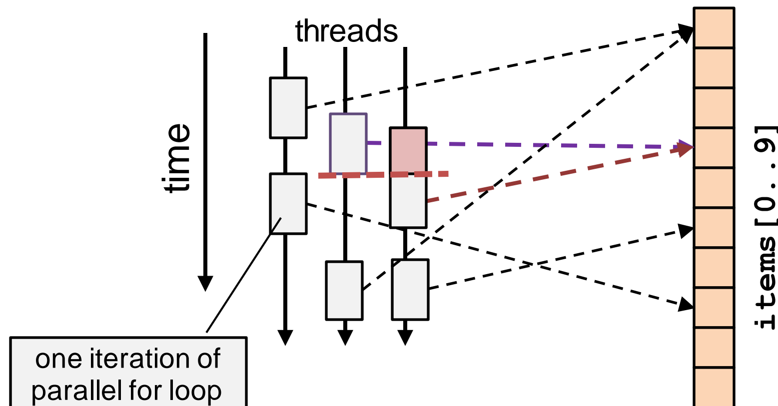


- every other state transition is **non-conforming**

Example

```
typedef struct
{
    omp_lock_t lock;
    size_t value;
} item;

...
size_t n_items = 10;
size_t n_indices = 1000;

item items[n_items];
for (size_t i = 0; i < n_items; ++i) {
  omp_init_lock(&items[i].lock);
}

size_t indices[n_indices];
for (size_t i = 0; i < n_indices; ++i) {
    indices[i] = rand() % n_items;
}
```

```
#pragma omp parallel for
for (size_t i = 0; i < n_indices; ++i) {
  item * it = &items[indices[i]];
  omp_set_lock(&it->lock);
  ++it->value;
  omp_unset_lock(&it->lock);
}

for (size_t i = 0; i < n_items; ++i) {
    omp_destroy_lock(&items[i].lock);
}
```

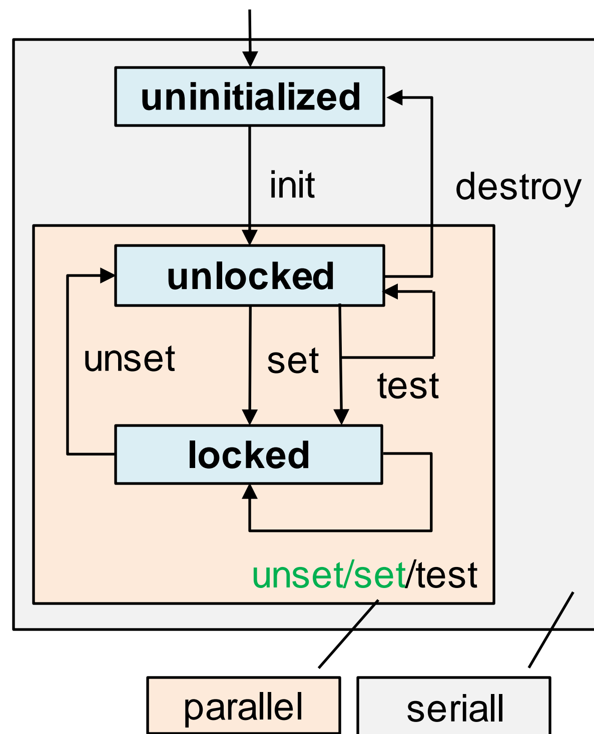


Nested Locks

- behave similar to simple locks except that a task owning a lock can lock it multiple times
- a lock reaches it unlocked state if it is unlocked as many times it was locked
  - must happen by the same task, as only the owning task can lock it multiple times
- similar routines like simple locks, named

```
omp_*_nest_lock(omp_nest_lock_t*)
```

- every other state transition is **non-conforming**



NHR FAU

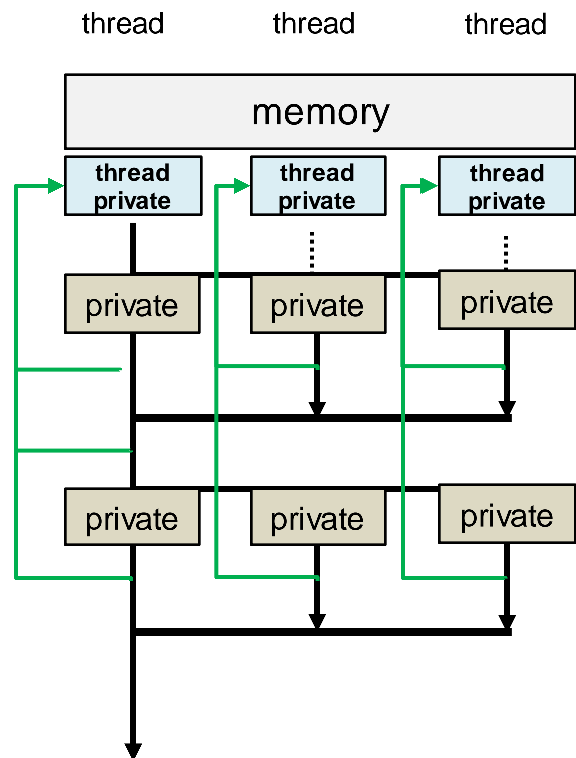
FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Thread Private Memory: `threadprivate` directive

HPC High Performance Computing

`threadprivate` directive

- `threadprivate` variables are
  - preserved over parallel regions (with restrictions)
  - tied to the thread
- global or static variables, replicated for each thread
- outside parallel regions only primary thread can access its `threadprivate` variables



`threadprivate` directive

- `threadprivate` directive declares a variable thread private
  - initialized for threads that encounter definition
  - uninitialized for remaining threads
- values of `threadprivate` variables are preserved between parallel regions for threads != initial thread, iff:
  - not nested parallel regions
  - both regions use the same no. of threads and affinity policy
  - no `order` construct that specifies `concurrent`
  - `OMP_DYNAMIC` is false at entry of both regions

declare `id` `threadprivate` initialized only for thread 0

```
static int id = 0;
#pragma omp threadprivate(id)

#pragma omp parallel
id = omp_get_thread_num();

printf("pt: %d\n", id)

#pragma omp parallel
printf("%d", id);
```

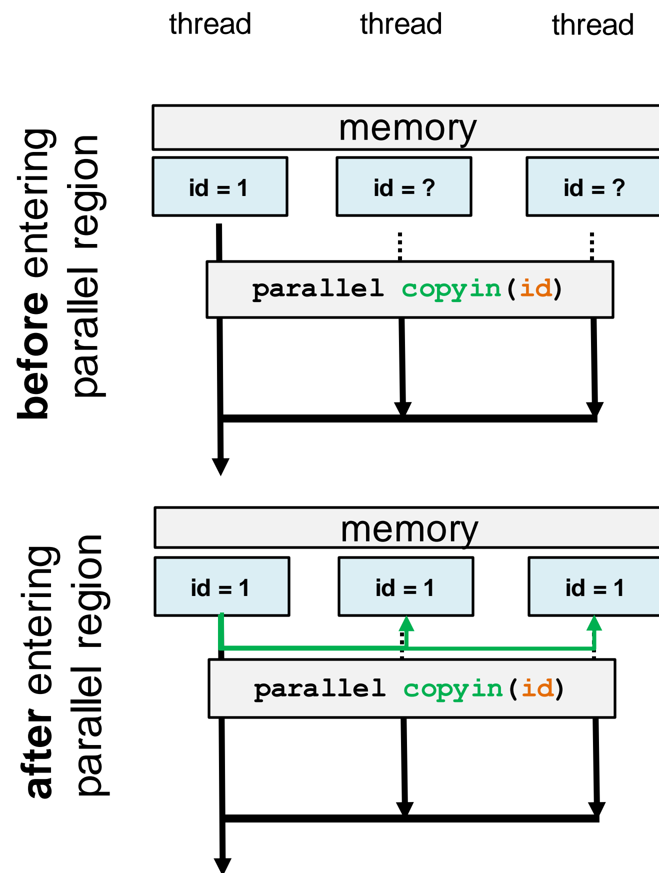
new threads get their own (uninitialized) `id` variable

`copyin` clause

- `copyin` clause of `parallel` construct
- propagate value of **primary thread** to other threads executing `parallel` region

```
static int id = 1;
#pragma omp threadprivate(id)

#pragma omp parallel copyin(id)
printf("id: %d\n", id);
```

only `threadprivate` variables are allowed

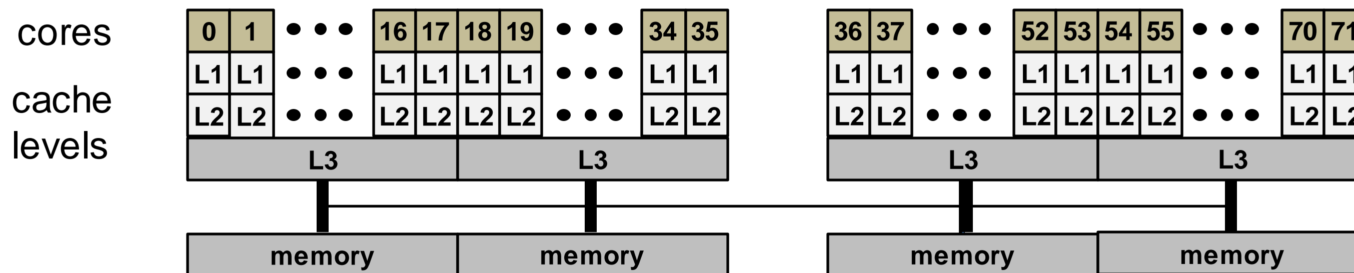


Thread Affinity

NHR FAU
FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
HPC High Performance Computing

Thread Affinity

- controls to which places threads are assigned
  - a.k.a. thread binding, thread pinning



- Why does it matter?
  - use shared/separate resources
  - avoid thread migration

Places – Where Threads Can Be Executed

- env. var. `OMP_PLACES`
  - values can be an abstract name

what is supported depends on the OpenMP version

abstract name	description
`threads`	HW threads, a.k.a. SMT-threads, virtual cores
`cores`	physical CPU cores
`ll_caches`	cores sharing a last level cache
`numa_domains`	cores belonging to the same NUMA domain
`sockets`	cores belonging to a socket

```
OMP_PLACES="cores"
OMP_PLACES="cores(4)"
OMP_PLACES="sockets"
OMP_PLACES="sockets(2)"
```

    - selected only a certain amount: **`abstract-name(count)`**
  - hardware ids of cores, format examples
    - `<id>[,<id>[,…]]`
    - `{<ids>}`
    - `{<ids>},{<ids>},…`
    - `{<ids>}[:<len>[:<stride>]]`

```
OMP_PLACES="0,2,4,6,8"
OMP_PLACES="{0,1},{2,3}"
OMP_PLACES="{0}:5:2"
```

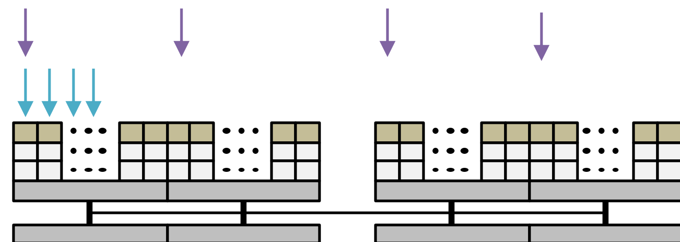
Control Affinity Policy

- env. var. `OMP_PROC_BIND` for setting affinity policy

value	description
`false`	disable affinity, `proc_bind` clause (`parallel` construct) is ignored
`true`	enable affinity, strategy is implementation defined
`close`	bind threads to adjacent places →typically used low latency
`spread`	distribute threads equally over available places → typically used for high bandwidth or separate resources
`primary`	bind all threads to the place of the initial thread

`OMP_PROC_BIND=` spread
close

`OMP_NUM_THREADS=4`
`OMP_PLACES=cores`



Show where threads are bound to

- env. var. `OMP_DISPLAY_AFFINITY=true`
  - print where threads are bound to

```
$ OMP_NUM_THREADS=6 OMP_PLACES=cores \
   OMP_PROC_BIND=true OMP_DISPLAY_AFFINITY=true ./a.out
level 1 thread 0x7f07a55e77c0 affinity 0-1
level 1 thread 0x7f07a51ff640 affinity 2-3
level 1 thread 0x7f07a49fe640 affinity 4-5
level 1 thread 0x7f07a41fd640 affinity 6-7
level 1 thread 0x7f07a39fc640 affinity 8-9
level 1 thread 0x7f07a31fb640 affinity 10-11
```

SMT enabled, i.e. one physical core houses two virtual cores

NHR FAU

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

OpenMP Runtime Functions and Environment Variables

High Performance Computing

Runtime Functions

- get id of current thread
  - `int omp_get_thread_num(void);`
  - `integer function omp_get_thread_num()`
- get number of threads in current region (sequential or parallel)
  - `int omp_get_num_threads();`
  - `integer function omp_get_num_threads()`
- get maximum number of threads in the **next** parallel region without a `num_threads` clause
  - `int omp_get_max_threads();`
  - `integer function omp_get_max_threads()`

to use API functions:
C/C++: `#include <omp.h>`
Fortran: `use omp_lib` or `include "omp.h"`

Runtime Functions

- get number of processors/cores
  - `int omp_get_num_procs();`
  - `integer function omp_get_num_procs()`

- get if inside a parallel region (true/false)
  - `int omp_in_parallel();`
  - `logical function omp_in_parallel()`

- print information about OpenMP
  - `void omp_display_env(int verbose);`
  - `subroutine omp_display_env(verbose)`
    `logical,intent(in) :: verbose`
  - if `verbose = true` → print vendor specific information too

Runtime Functions

- get elapsed time in seconds since some point in time
  - `double omp_get_wtime();`
  - `double precision function omp_get_wtime()`
  - mostly useful for measuring durations
  - might not be synchronized between threads

```
t = omp_get_wtime();

/* work */

dur = omp_get_wtime() - t;
```

- resolution in seconds of `omp_get_wtime`
  - `double omp_get_wtick();`
  - `double precision function omp_get_wtick()`

Environment Variables

- **`OMP_NUM_THREADS=n`**
  - use n threads for parallel regions
  - priority: `OMP_NUM_THREADS` < `omp_set_num_threads()` < `num_threads` clause
- **`OMP_DYNAMIC=true|false`**
  - if true the runtime may use a different number of threads for executing parallel regions
- **`OMP_THREAD_LIMIT=n`**
  - maximum number of threads to use
- **`OMP_STACKSIZE=n[BKMG]`**
  - stack size of OpenMP threads (not including the initial thread)
  - without unit, KiB (1024 B) are assumed, units K, M, G are base 2 based
  - often: increase for Fortran, if arrays don't fit onto the stack

Environment Variables

- **`OMP_WAIT_POLICY=active|passive`**
  - hint of how waiting threads should behave
  - active: actively check for work
  - passive: might sleep when waiting
- **`OMP_DISPLAY_AFFINITY=true|false`**
  - print to which cores OpenMP threads are bound when created

Environment Variables

- `OMP_SCHEDULE=[modifier:](static|dynamic|guided|auto)[,chunk]`
  - schedule to use for loops with clause schedule(runtime)
  - modifier: optional, can be monotonic or nonmonotonic
  - chunk: optional, chunk size

- `OMP_MAX_TASK_PRIORITY=n`
  - maximum task priority that can be used in `priority` clause of `task` construct

- `OMP_DISPLAY_ENV=true|false|vebose`
  - print information about OpenMP settings
  - verbose → print vendor specific information too
  - use env. var. `OMP_AFFINITY_FORMAT` to change output

OMP_DISPLAY_ENV example

```
> OMP_DISPLAY_ENV=true ./binary
OPENMP DISPLAY ENVIRONMENT BEGIN
   _OPENMP='201611'
  [host] OMP_AFFINITY_FORMAT='OMP: pid %P tid %i thread %n bound to OS proc set {%A}'
  [host] OMP_ALLOCATOR='omp_default_mem_alloc'
  [host] OMP_CANCELLATION='FALSE'
  [host] OMP_DEBUG='disabled'
  [host] OMP_DEFAULT_DEVICE='0'
  [host] OMP_DISPLAY_AFFINITY='FALSE'
  [host] OMP_DISPLAY_ENV='TRUE'
  [host] OMP_DYNAMIC='FALSE'
  [host] OMP_MAX_ACTIVE_LEVELS='1'
  [host] OMP_MAX_TASK_PRIORITY='0'
  [host] OMP_NESTED: deprecated; max-active-levels-var=1
  [host] OMP_NUM_TEAMS='0'
  [host] OMP_NUM_THREADS: value is not defined
  [host] OMP_PLACES: value is not defined
  [host] OMP_PROC_BIND='false'
  [host] OMP_SCHEDULE='static'
  [host] OMP_STACKSIZE='8M'
  [host] OMP_TARGET_OFFLOAD=DEFAULT
  [host] OMP_TEAMS_THREAD_LIMIT='0'
  [host] OMP_THREAD_LIMIT='2147483647'
  [host] OMP_TOOL='enabled'
  [host] OMP_TOOL_LIBRARIES: value is not defined
  [host] OMP_TOOL_VERBOSE_INIT: value is not defined
  [host] OMP_WAIT_POLICY='PASSIVE'
OPENMP DISPLAY ENVIRONMENT END
```

NHR FAU

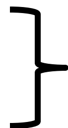
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Resources

High Performance Computing

Resources

- https://www.openmp.org/specifications/
  - OpenMP API x.y Specification
  - OpenMP API x.y Examples
  - OpenMP API x.y Reference Guide
    - some kind of cheat sheet



highly recommended

- Books:
  - B. Chapman, G. Jost, R. v. d. Pas: **Using OpenMP**. MIT Press, 2007, ISBN 978-0262533027
  - R. v. d. Pas, E. Stotzer, C. Terboven: **Using OpenMP – The Next Step**. MIT Press, 2017, ISBN 978-0-262-53478-9