OpenMP

Parallel Computing
SummerTerm 2022, April 28th
Sven Bingert
sven.bingert@gwdg.de

Objectives

- What is OpenMP
- What can it be used for
  – and where not
- Learning the basic usage
- Conduct some simple examples
- Answer the question: can you benefit by OpenMP parallelization

Motivation

- Problems exist where Shared-Memory is required or beneficial
- Development of dedicated share-memory architectures is still ongoing
- Number of processor for such systems continuously increases
- But hardware specific code is not portable
- MPI might be to difficult

Open Specifications for Multi Processing (OpenMP)

- an API to hint the compiler about parallelizable sections
- to be implemented by the compiler
- therefor everywhere a bit different
- supported by gcc, icc, VisualC++, pgc, clang…

specified for Fortran and C (and works great with C++ as well!)

List of compilers:
https://www.openmp.org/resources/openmp-compilers-tools/

OpenMP

- meant for Shared Memory Systems
- can be combined with MPI
- does no magic!
  - You have to sync IO access on your own
  - You have to lock memory on your own
  - You have to avoid deadlocks on your own

Latest specification **OpenMP 5.2 from Nov 9th 2021**

OpenMP building block

- The C-API consists of 3 parts
  - compiler Directives

```
#pragma omp parallel default(shared) private(beta,pi)
```

  - a library

```
#include <omp.h>
int omp_get_num_threads(void)
```

  - environment variables

```
export OMP_NUM_THREADS=8
```

- compile with:
  - gcc -fopenmp foobar.c
  - icc -no-multibyte-chars -qopenmp foobar.c

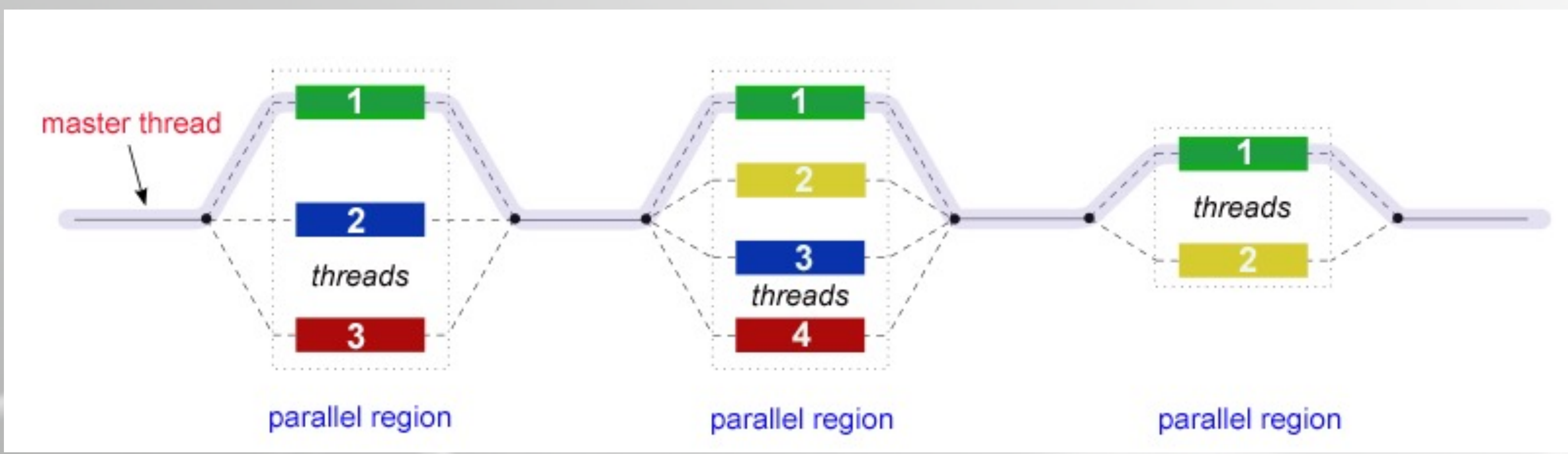
Simple Example

```
#include <omp.h>

main () {
#pragma omp parallel
    printf("Hello World");
}
```

- compile and execute
- output:

```
>./hello

Hello World
Hello World
Hello World
Hello World
```

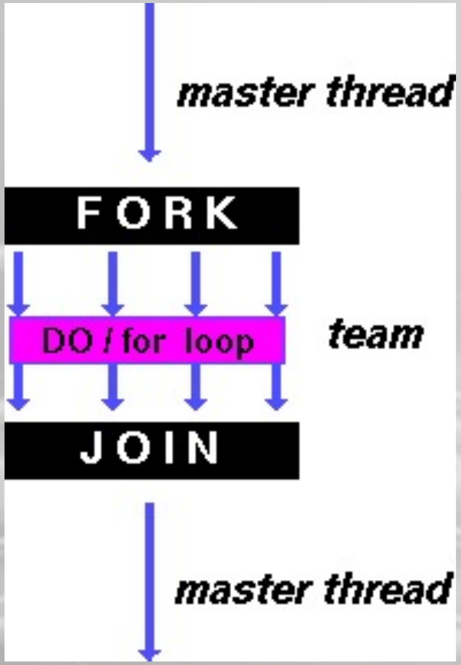
Fork – Join Model



- Starts with one thread (master thread or thread 0)
- Building teams with more threads during runtime
- Barrier at the end of parallel part and cleaning up the threads, master thread continues

Work sharing I

SIMD

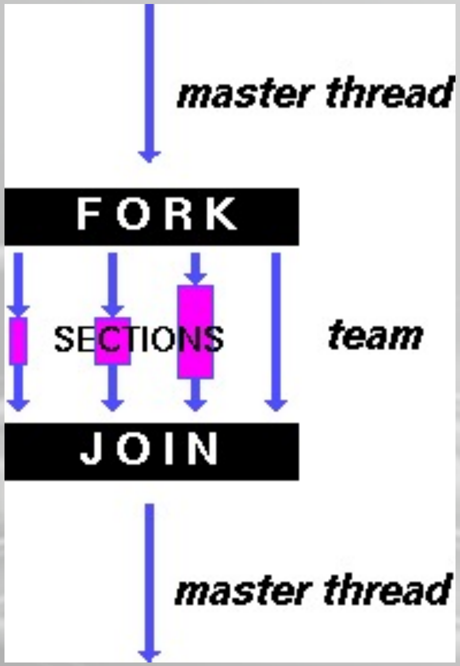


- Single Instruction Multiple Data (SIMD)
- Example:
  - add fixed number to a vector
- easy to parallelise

figures from https://computing.llnl.gov/tutorials/openMP

Work sharing II

MIMD



- Multiple Instructions Multiple Data
- Different tasks (and code!) for different threads in the parallel section
- hard to parallelise

figures from https://computing.llnl.gov/tutorials/openMP

Communication and data space

- Communication via shared variables
- Master Thread
  - – Execution context exists during runtime
- Worker Threads
  - – Execution context only during parallel section

Communication and data space II

- Variables are categorized in
  - *shared*
  - *private*
- status should explicitly specified
  - otherwise *shared* by default
- to simplify coding special attribute
  - *reduction*

Communication and data space III

- When using shared variables
  - – all threads access the same memory address
  - – the way to “communicate”

- private variables
  - – copies for each thread are created
  - – value **undefined** at the beginning and end of the parallel section

Synchronisation

- When using shared variables
  - avoid concurrent writes !
  - one thread might read while another writes
  - leads to unclear state end the end of the parallel section
  - Memory cache can be used to avoid conflicts
    - flush-directive enforces consistency

Simple Example II

```
#include <omp.h>

main () {
int nthreads, tid;
/* do something in parallel: */
#pragma omp parallel private(tid)
  {
  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and terminate */
}
```

Omp for directive #1

```
/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i *  1.0;

#pragma omp parallel shared(a,b,c) private(i)
  {

  #pragma omp for schedule(dynamic)
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
  }  /* end of parallel section */

  /* only the master does printf */
  #pragma omp master
  {
      for(i=0;i<N;i++) {
          printf("c[%d] = %f\n",i,c[i]);
      }
  }
}
```

omp for directive #2

- schedule (static/dynamic/guided/runtime/auto)
- nowait – do not synchronize threads after the loop (c.f. flush)
- ordered – iterations must be done in same order like in a serial program

```
#pragma omp parallel for ordered
for (i=0; i < N; i++)
  // do heavy stuff

  #pragma omp ordered
  c[i] = a[i] + b[i];

  // more heavy stuff
} /* end of parallel section */
```

OpenMP directives

- Parallelization
  - *for, parallel, sections, single, task, ...*
- Synchronization
  - *barrier, critical, master, atomic, ...*
- Data space
  - *threadprivate*

Directives Syntax

C/C++

```
#pragma omp directive [clause [[,] clause …]….]
        followed by a structured block
```

Fortran

```
!$OMP directive [clause[[,] clause] ...]
        followed by a structured block
!$OMP END
```

Important clauses for the data space

#pragma omp parallel …

- private (*var1,var2,var2*)
- shared (*var1,var2,var3*)
- default (*shared/none*) – Warning: private is not allowed here!
- reduction (*operator: var1*) – makes var1 (implicitly) thread private and puts them together via operator in the end

```
#pragma omp parallel default(shared) private(i) \\
                  reduction(+:result)
{
#pragma omp for schedule(static,chunk)
  for (i=0; i < n; i++)
    result = result + (a[i] *  b[i]);

} // end omp parallel
printf("Final result= %f\n",result);
```

Undefined Variables

- Variables declared with *private* are undefined at start and end of the section
- With *firstprivate(list of variables)* each copy will be initialized with the value prior the section and are set to *private*

Important Library Functions

- omp_in_parallel ()
- omp_get_num_threads ()
- omp_get_thread_num ()
- omp_set_num_threads ()
- omp_get_num_procs ()
- omp_get_wtime (), omp_get_wtick ()
- omp_init_lock (), omp_set_lock (), omp_unset_lock (), omp_test_lock (), omp_destroy_lock ()

Time measurement

- *double omp_get_wtime(void);*
- returns the time (in seconds) elapsed since a fixed point in time in the past
- the temporal resolution is limited depending on the underlying architecture OS
- elapsed time is the difference between the first call and the current time

Time measurement II

```
#pragma omp parallel
{
// ...
#pragma omp single nowait
        start = omp_get_wtime();
        // ... code of interest
#pragma omp single nowait
        end = omp_get_wtime();
        // ...
} // end of parallel section
printf(“time in seconds: %lf\n”, end - start);
```

OpenMP loop parallelization

- the strength of OpenMP!
- each thread will work on other subset of iterations
- should be SIMD but be aware of dependencies
- clauses: schedule, ordered, if, copyin

```
#pragma omp for (+clauses)
            for (….)
```

- Only the directly subsequent loop is parallelised

OpenMP loop parallelization II

- omp parallel: create parallel section
- omp for: use existing threads to process loop
  – a omp parallel has to be before
  – only the first loop is parallelized
- omp parallel for: do both in one go

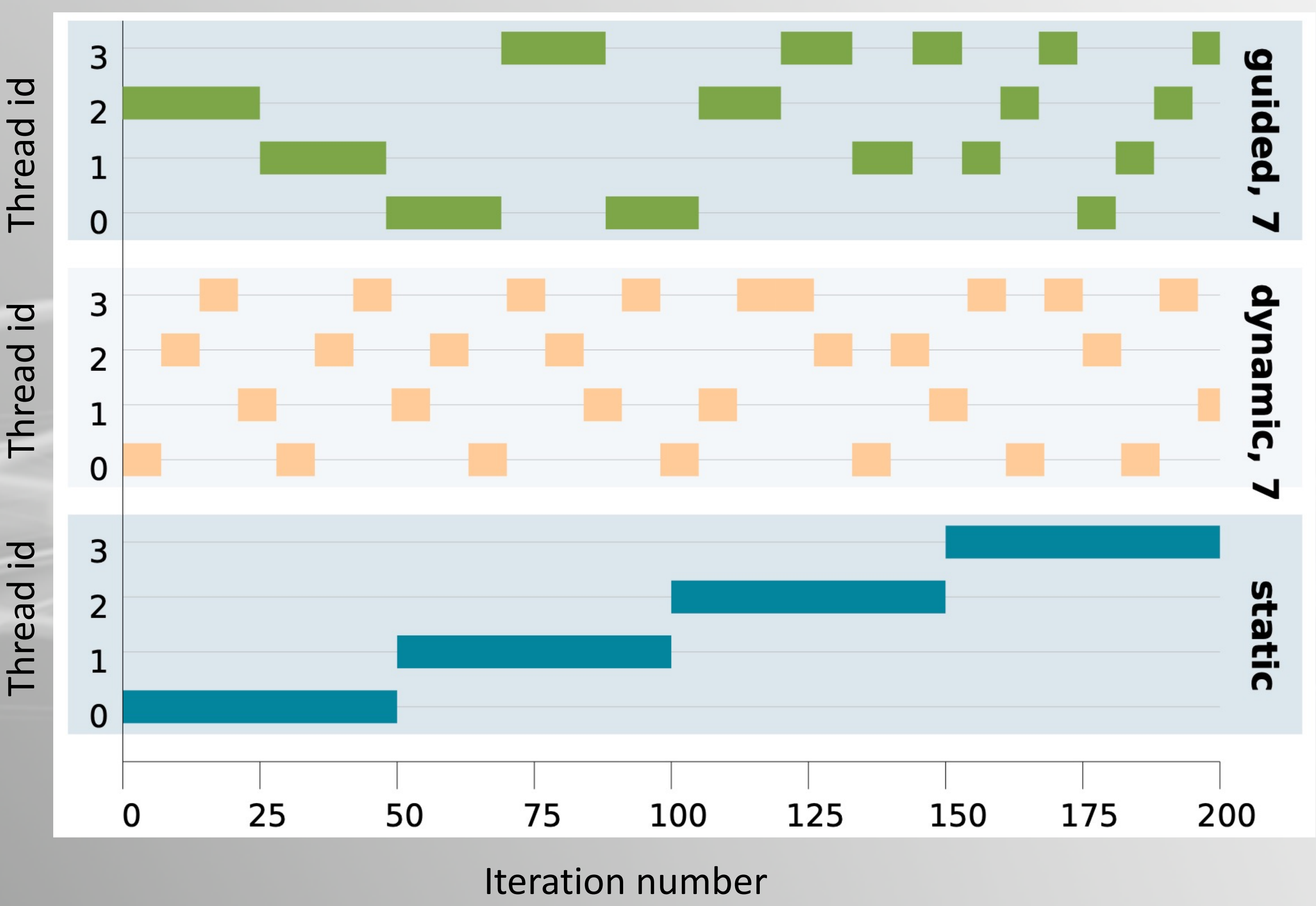
```
#pragma omp parallel for collapse(3)
  for(int l=0; l<10; ++l) {
    /* no code allowed here */
    for(int j=0; j<3; ++j) {
      /* no code allowed here */
      for(int k=0; k<7; ++k){
          foo[l][j][k] = 0;
      }
    }
  }
```

OpenMP loop parallelization II

- omp parallel for ordered: threads of the team execute the ordered region sequentially in the order of the loop iterations

- omp parallel for schedule: hint how iterations should be distributed among the threads
  - static: same chunk size (exeception: **chunk**)
  - dynamic: each requesting thread gets a chunk (controlled with **chunk**)
  - guided: chunk size decreases with iterations
  - runtime: using environment variables
  - auto: compiler and run time environemnt

Loop scheduling



Parallel sections

- Useful for MIMD operations
- omp parallel sections: to start several sections
  - omp section: for each section
- Each section is executed by one thread!
- good for small tasks
- order of execution is not defined

```
for (i=0; i < N; i++) {
 a[i] = i *  1.5;  b[i] = i + 22.35;
 }

#pragma omp parallel shared(a,b,c,d) private(i)
 {
 #pragma omp sections       // you might use "nowait"
  {

  #pragma omp section
  for (i=0; i < N; i++)
   c[i] = a[i] + b[i];

  #pragma omp section
  for (i=0; i < N; i++)
   d[i] = a[i] *  b[i];

  }  /* end of sections */

 }  /* end of parallel section */
```

Important directives

#pragma omp ...

- master: only executed by the master
- critical: only one thread allowed at a time
- barrier: A barrier for everybody to wait for
- flush: synchronize shared memory of all threads; implicitly done at barrier, for, critical, parallel...

Other workload distribution

- omp single: Useful for task within parallel section to be executed by one thread only, e.g. IO operations

- omp critical: to avoid data races, only one thread at a time executes this block

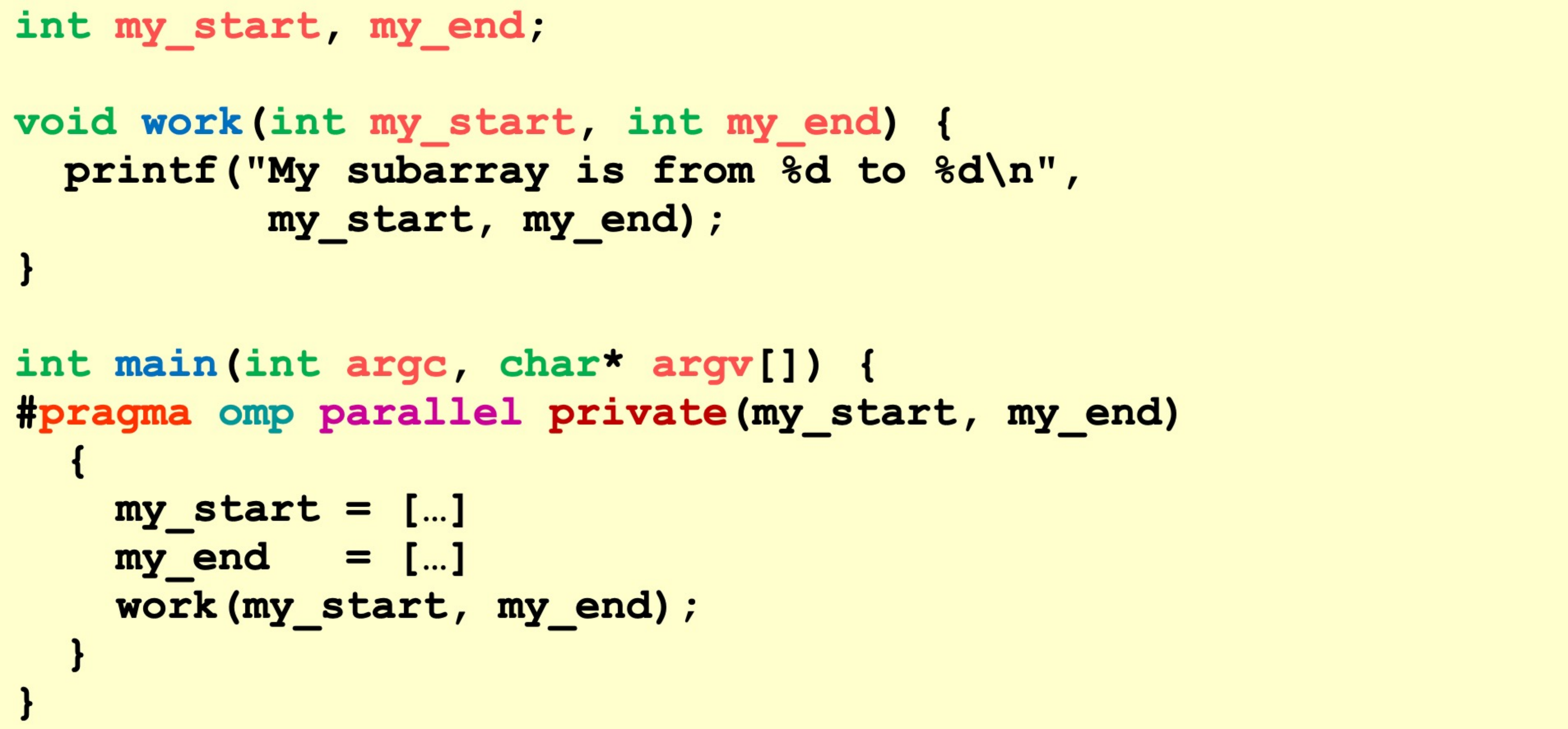
- Make use of nowait for some directives/clauses to allow threads passing by

Code within/without parallel sections

```
int my_start, my_end;

void work() {    /* my_start and my_end are undefined */
  printf("My subarray is from %d to %d\n",
          my_start, my_end);
}

int main(int argc, char* argv[]) {
#pragma omp parallel private(my_start, my_end)
  {
    /* get subarray indices */
    my_start = get_my_start(omp_get_thread_num(),
                            omp_get_num_threads());
    my_end   = get_my_end(omp_get_thread_num(),
                          omp_get_num_threads());
    work();
  }
}
```

Code within/without parallel sections II

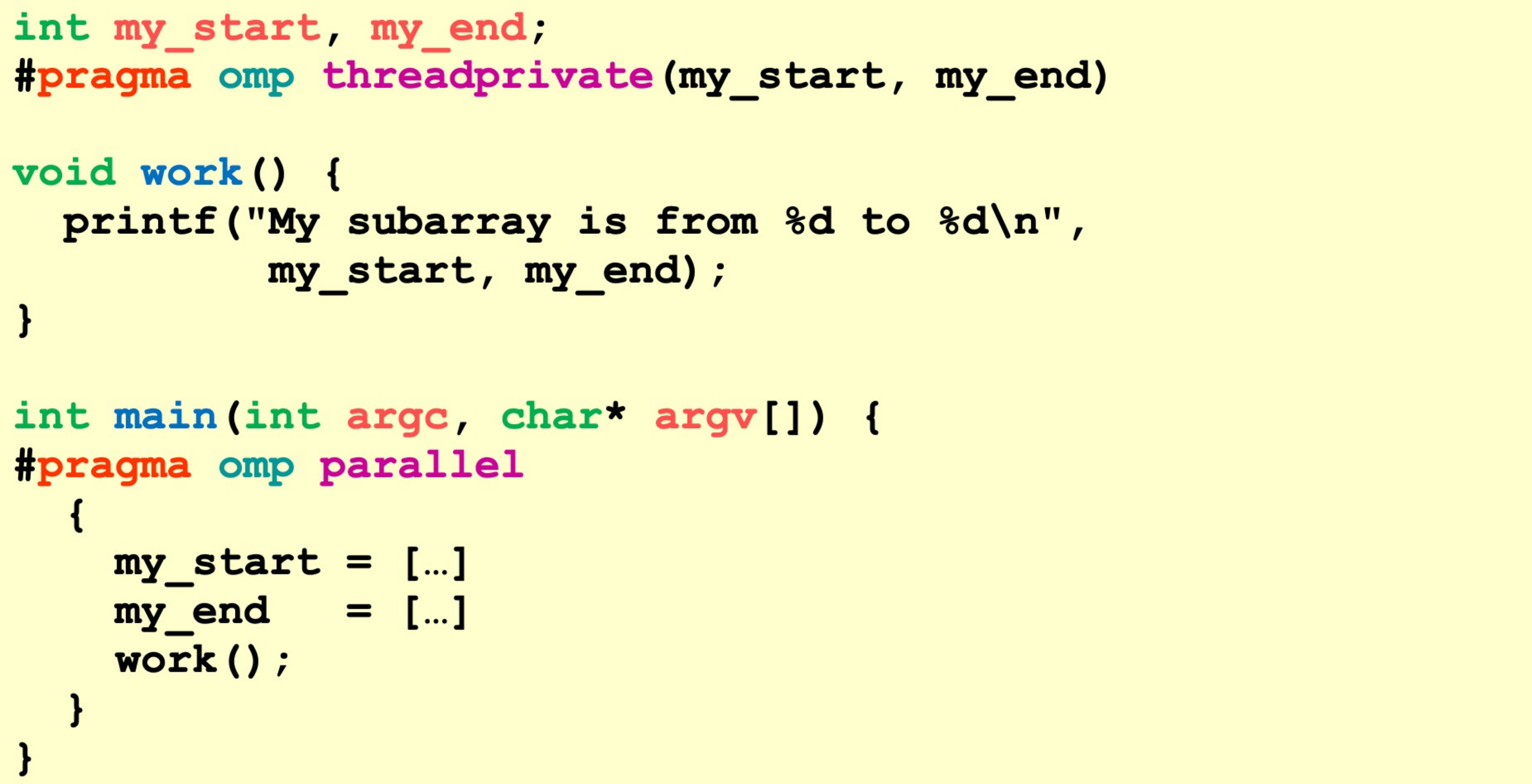
- solution 1: variables as parameters



```
int my_start, my_end;

void work(int my_start, int my_end) {
  printf("My subarray is from %d to %d\n",
          my_start, my_end);
}

int main(int argc, char* argv[]) {
#pragma omp parallel private(my_start, my_end)
  {
    my_start = […]
    my_end   = […]
    work(my_start, my_end);
  }
}
```

Code within/without parallel sections III

- Solution two: use **omp threadprivate**



```
int my_start, my_end;
#pragma omp threadprivate(my_start, my_end)

void work() {
  printf("My subarray is from %d to %d\n",
          my_start, my_end);
}

int main(int argc, char* argv[]) {
#pragma omp parallel
  {
    my_start = […]
    my_end   = […]
    work();
  }
}
```

Exercises

- Simple to more complex tasks
- Use the online OpenMP specification !
- Questions without coding are to test your understanding

- Open for new ideas/numerical problems
  - calculate Pi
  - estimate stock market
  - ...

References

- https://sourceware.org/gdb/current/onlinedocs/gdb/Threads.html
- https://www.openmp.org/spec-html/5.2/openmp.html
- https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- https://gcc.gnu.org/wiki/Graphite/Parallelization
- https://hpc-tutorials.llnl.gov/openmp/

THANKS