Introduction to OpenMP

Matic Brank

LECAD laboratory, FS UL

Outline

What is OpenMP?

- Standard programming model for shared memory parallel programming
- Portable across all shared-memory architectures
- It allows incremental parallelization
- Compiler based extensions to existing programming languages
- Fortran and C/C++ binding

Outline

What is OpenMP?

OpenMP -> Open specifications for Multi Processing

- API for shared-memory parallel computing
- Open standard for portable and scalable parallel programming
- Flexible and easy to implement
- Specification for a set of compiler directives, library routines and environment variables
- Designed for Fortran and C/C++

Ownership and timeline

Ownership

- OpenMP ARB (Architecture review board)
  - Its mission is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable
  - Jointly defined by a group of major computer hardware and software vendors and major parallel computing user facilities (such as AMD, Intel, IBM, HP, Fujitsu, Microsoft,…)

Release history

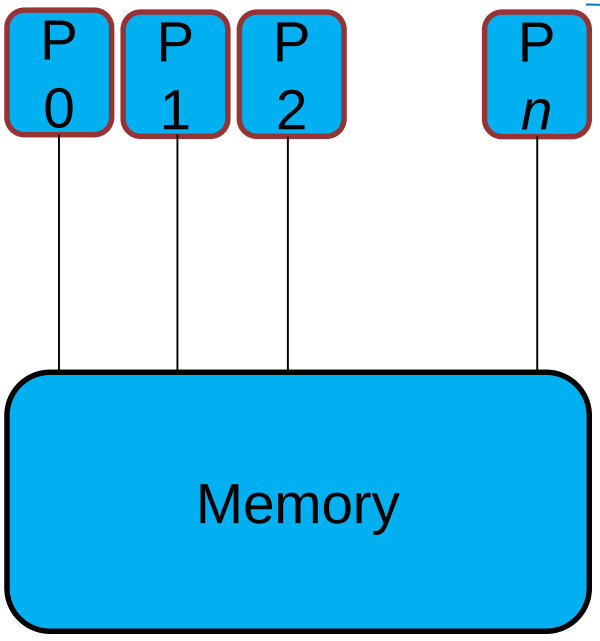
- 1997 OpenMP - Fortran 1.0
- 1998 OpenMP - C/C++ 1.0
- 1999 OpenMP - Fortran 1.1
- 2000 OpenMP - Fortran 2.0
- 2002 OpenMP - C/C++ 2.0
- 2005 OpenMP 2.5
- 2008 OpenMP 3.0
- 2013 OpenMP 4.0
- 2018 OpenMP 5.0 stable release
- 2020 OpenMP 5.1 release

OpenMP – Main terminology

- **OpenMP thread**: a running process specified by OpenMP
- **Thread team**: a set of threads which cooperate on a task
- **Master thread**: Main thread which coordinates the parallel jobs
- **Thread safety**: This term refers to correct execution of multiple threads
- **OpenMP directive**: OpenMP line of code for compilers with OpenMP
- **Construct**: an OpenMP executable directive
- **Clause**: controls the scoping of the variables during execution

OpenMP – Programming model

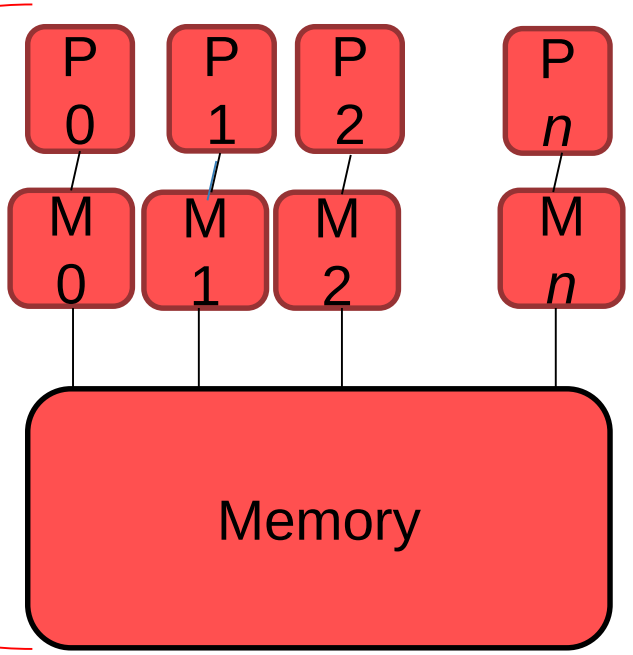
- Primarily designed for shared memory multiprocessors



A canonical shared memory architecture

All of the processors are able to directly access all of the memory in the machine through a logically direct connection

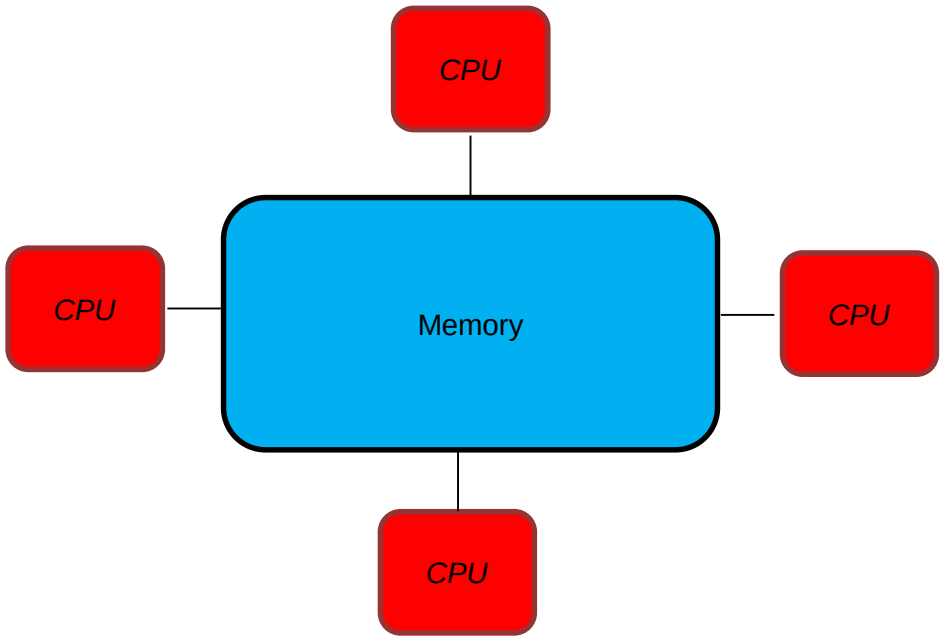
Each processor in the system is only capable of directly addressing memory (M0-M*n*) that is physically associated with him



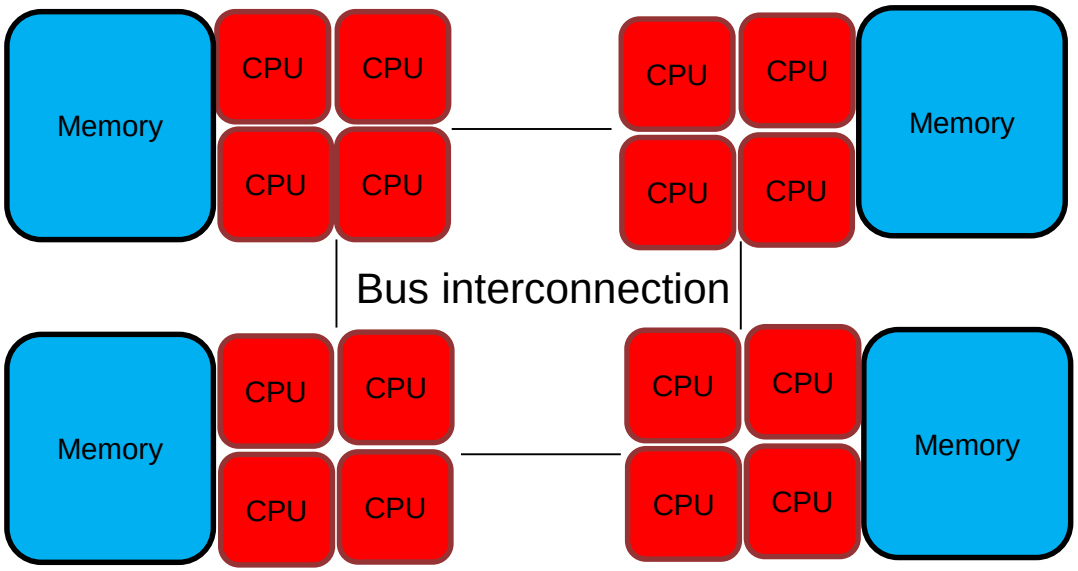
A canonical message passing (non-shared memory) architecture

OpenMP – Programming model

- Shared memory architecture



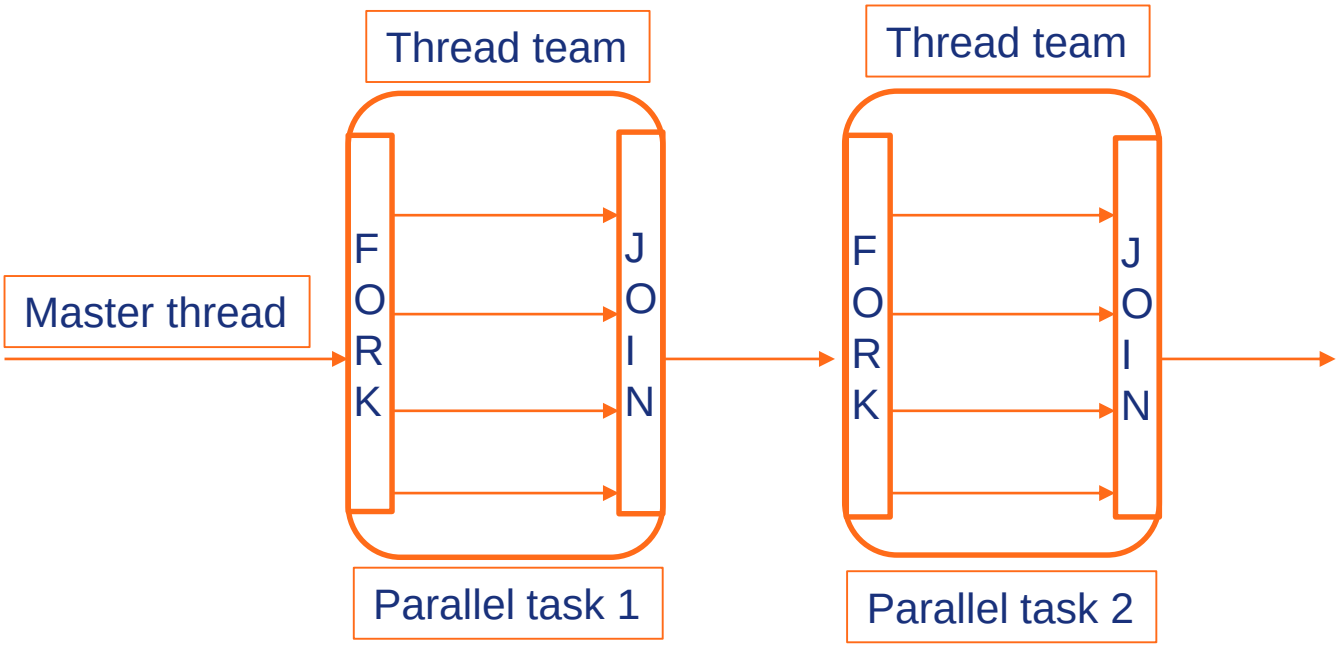
UMA



NUMA

OpenMP – Execution model

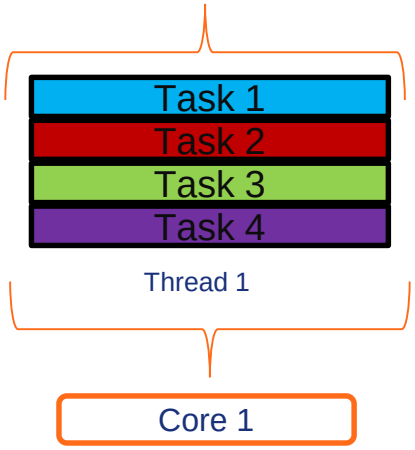
- Thread based parallelism
- Compiler directive based parallelism
- Explicit parallelism
- Fork-Join model
- Dynamic Threads
- Nested parallelism



OpenMP – Execution model

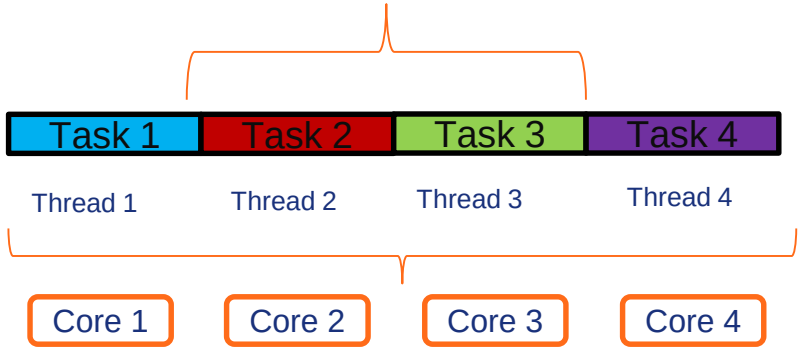
- Thread based parallelism

Single threading



Execution time = $\sum_{i=1}^{4} Task_i$

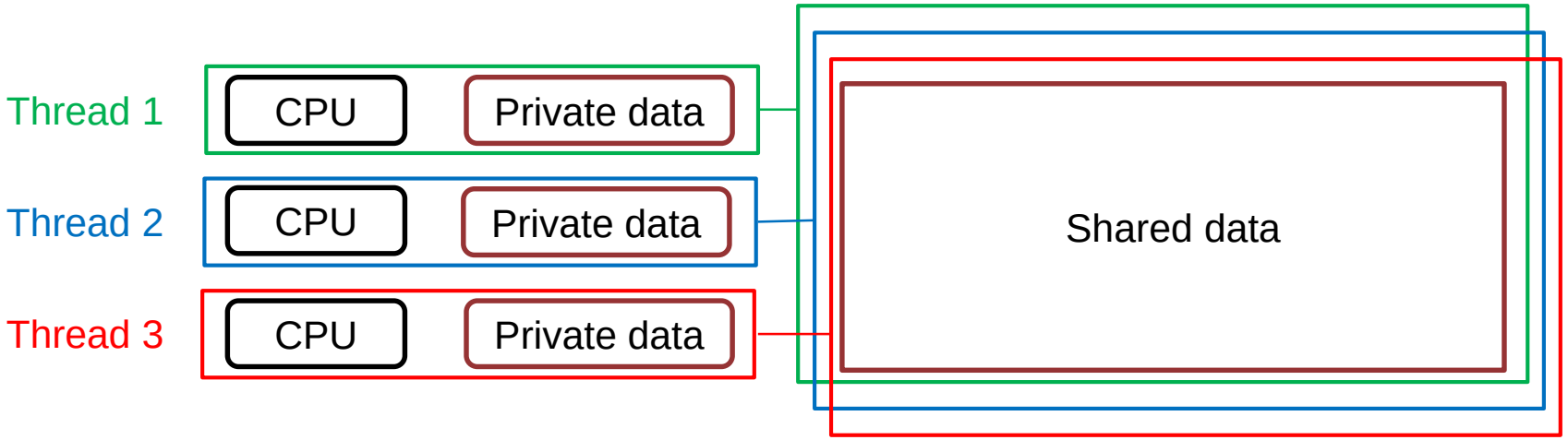
Thread based parallelism



Execution time = $MAX(Task_i)$

OpenMP – Memory model

- All threads have access to the same memory
- Threads can share data with other threads, but also have private data
- Threads can be synchronized
- Threads cache their data



OpenMP – Memory model

- Compiler directives and Clauses: appear as comments and execute when appropriate OpenMP flag is specified
  - Parallel construct
  - Work-sharing constructs
  - Synchronization constructs
  - Data attribute clauses
- C/C++ OpenMP comment:

```
#pragma omp directive-name [clause[clause]…]
```

OpenMP – Memory model

- Compiling

	Compiler	Flag
Intel	icc (C) icpc (C++) Ifort (Fortran)	**-openmp**
GNU	gcc (C) g++ (C++) g77/gfortran (Fortran)	**-fopenmp**
PGI	pgcc (C) pgCC (C++) pg77/pgfortran (Fortran)	**-mp**

- Example of command in terminal:

```
gcc -fopenmp -o executable foo.c
```

Compiler `gcc` compiles file `foo.c` with OpenMP flag into executable file named `executable`

- Notes for GCC compiler:
  - From GCC 6.1, OpenMP 4.5 is fully supported for C and C++.
  - From GCC 7.1, OpenMP 4.5 is partially supported for Fortran.
  - From GCC 9.1, OpenMP 5.0 is partially supported for C and C++

Note: For full list of vendors and compilers refer to https://www.openmp.org//resources/openmp-compilers-tools/

OpenMP – Main terminology

- Purpose of runtime functions is to manage the parallel processes
  - `omp_set_num_threads(n)` – set the desired number of threads
  - `omp_get_num_threads()` – returns the current number of threads
  - `omp_get_thread_num()` – returns the ID of this thread
  - `omp_in_parallel()` – return .true. if inside parallel region
- Correct usage in code:
  - For C/C++ add `#include<omp.h>` to the code
  - For Fortran add `use omp_lib`

Note: For full list of OpenMP runtime functions refer to https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-functions?view=vs-2019

OpenMP – Environment variables

- Purpose of environment variables is to control the execution of parallel program at runtime. These variables are not specified in the code itself but in the environment in which the parallel program is executed.
  - `OMP_NUM_THREADS` – specifies the number of threads to use
  - `OMP_PLACES` – specifies on which CPUs the threads should be placed
  - `OMP_DISPLAY_ENV` – show OpenMP version and environment
- Correct usage in code:
  - Environment csh/tcsh: `setenv OMP_NUM_THREADS n`
  - Environment ksh/sh/bash: `export OMP_NUM_THREADS=n`

Note: For full list of OpenMP environment variables for GCC compiler refer to https://gcc.gnu.org/onlinedocs/libgomp/Environment-Variables.html

OpenMP – Parallel construct

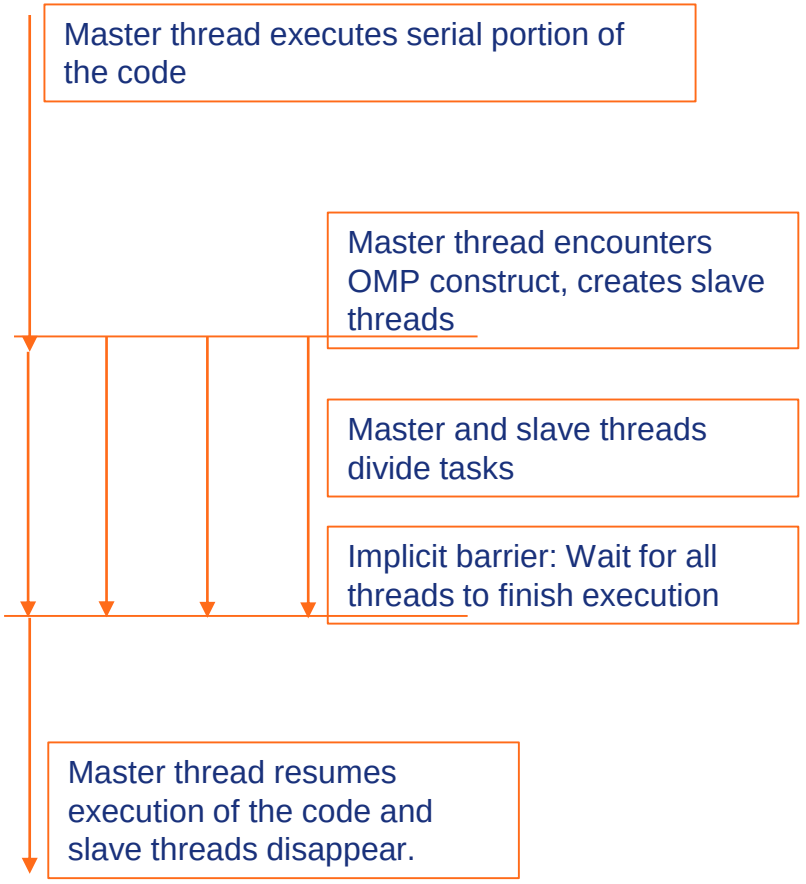
- Parallel construct is the fundamental construct in OpenMP
  - Every thread executes the same statements which are inside the parallel region simultaneously
  - At the end of the parallel region there is an implicit barrier for synchronization

C/C++:

```
#pragma omp parallel
{
…
}
```

Fortran:

```
!$omp parallel
…
!$omp end parallel
```



OpenMP – **Exercise 1**

- Goal: To write an OpenMP program which outputs „Hello world from thread 1-N“ by each MP process (help yourself with OpenMP runtime functions)
- Instructions:
  - Open *e1.c* file.
  - Include omp header file in the beginning of the file (should be above main function)
  - Define parallel region with pragma
  - Use printf function to print “Hello World from thread n of N”, where n is the ID number of thread and N is the number of all threads. Use runtime function that returns the ID of each thread.
  - Compile and run it in serial on a single processor (is there an error?)
  - Run it on several processors in parallel by using environment variable `export OMP_NUM_THREADS=n`
- Expected result:
  - The code outputs “Hello World from thread 1 of N” for each thread.

OpenMP – Clauses and directives format

- Directive format
  - Format:

    `#pragma omp` *directive_name* [clause[clause]…]

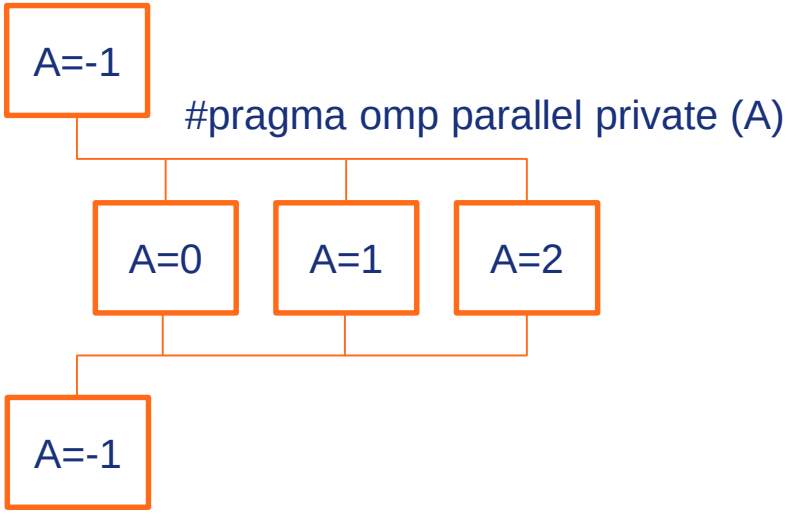
  - Conditionals:

```
#ifdef _OPENMP
        block of code to be executed if code was compiled with OpenMP, for example
        printf("Number of threads: %d", omp_get_num_threads);
#else
        block of code to be executed if code was compiled without OpenMP
#endif
```

OpenMP – Clauses and directives format

- Clauses
  - In C/C++ clauses can be:
    - **private** (list) – in this case the variable is private to each thread
    - **shared** (list) – in this case the variable is shared between threads
  - C/C++:

```
int A;
#pragma omp parallel private(A)
{
A=omp_get_thread_num();
…
}
```



OpenMP – **Exercise 2**

- Goal: To use if/else clauses
- Instructions
  - Open file *e2.c* and move to comment `/* Begin of SPACE for second exercise */` and add an `if` clause if omp is used. Inside add a parallel region that prints the ID of each thread (with `omp_get_thread_num()`) and the number of all threads (with `omp_get_num_threads()`).
  - Compile and run on 4 CPUs and then on 12 CPUs. Observe output.
  - Add an `else` clause if program is not compiled with OpenMP. Add a print statement that says: „The program is not compiled with OpenMP“. Compile the program without OpenMP and run it. Observe the output.

OpenMP – **Exercise 2**

- Define a new variable and assign `omp_get_thread_num()` to it
- Add a `private` clause to `omp parallel` directive for the variable associated with `omp_get_thread_num()` and observe the difference in the output.
- Check if you get race condition:
  - Two threads access the same shared variable and at least one thread modifies the variable and the accesses are unsynchronized
  - Outcome of the program depends on timing of the threads in the team
  - This is caused by unintended shared of data

```
thread 0 of 4
thread 0 of 4
thread 0 of 4
thread 0 of 4
```

Don‘t worry if you always get correct output, because the compiler may use a private register on each thread instead of writing into shared memory

OpenMP – **Exercise 2**

- Expected output:
  - If compiled with OpenMP, the program should output and ID of each thread and number of all threads
  - If not compiled with OpenMP, the program should output “The program was not compiled with OpenMP“

OpenMP – **Exercise shared vs. private**

- Goal: To see difference between private and shared variable.
- Instructions:
  - Open file *exercise_shared_vs_private.c*
  - Follow the instruction on the comments:
    - After variable `int thr_num`, set number of used threads to 4
    - Define a new variable named `list_of_thread_ids`
    - Initialize all 4 elements in the array to 0
    - For parallel pragma, define private variables `thr_num` and `list_of_thread_ids`
    - Inside parallel pragma, get ID number of each thread and assign it to `thr_num`
  - Run the code:
    - What is the output?
    - What are the elements inside `list_of_thread_ids` after execution of parallel pragma?
  - Modify line `#pragma omp parallel…` and change the type of `list_of_thread_ids` from private to shared

OpenMP – **Exercise shared vs. private**

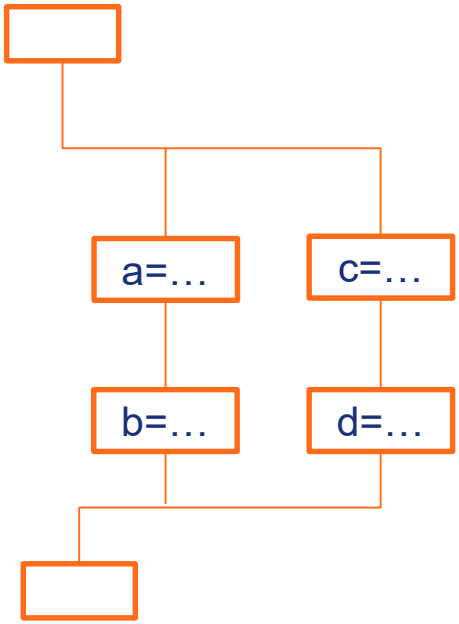
- What is the new output?
- What is the difference between `list_of_thread_ids` being private or shared?

OpenMP – Worksharing constructs

- They divide the execution of code region among the members of the team
- Constructs do not launch new threads
- They are enclosed dynamically within the parallel region
- Examples:
  - `sections`
  - `for`
  - `task`
  - `single`
- No barrier defined on entry

OpenMP – Sections construct

- When using `sections` construct, multiple blocks of code are executed in parallel



Sections in parallel

- C/C++:

```
#pragma omp parallel
{
#pragma omp sections
{{a=…; b=…;}
#pragma omp section
{
        c=…;
        d=…;
}
}// end of sections
}// end of parallel
```

OpenMP – For construct

- C/C++:

```
#pragma omp for [clause[[,]clause]…]
```

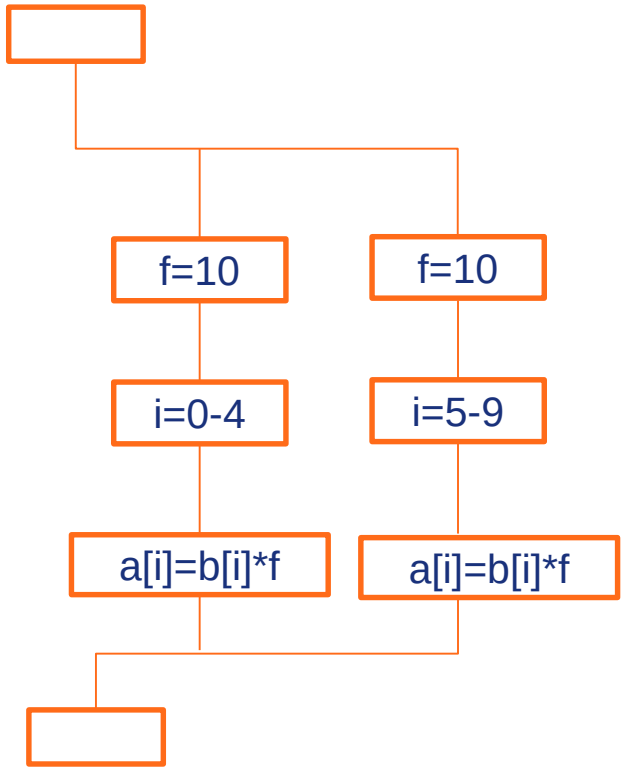
- Corresponding for loop must have a *canonical* shape, where an iterator (`i`) must not be modified in the loop body:

```
for (int i=it; i<M; i++)
```

- Clauses:
  - `private (list)`
  - `schedule` -> classifies how the iterations of loops are divided among the threads
  - `collapse (n)` -> the iterations of n loops are collapsed into one larger iteration space

OpenMP – For construct

- Private variable f is fixed in each thread, list a is updated in parallel.



- C/C++:

```
#pragma omp parallel private(f)
{
f=10;
#pragma omp for
for (i=0; i<10; i++)
a[i] = b[i]*f;
} /* omp end parallel */
```

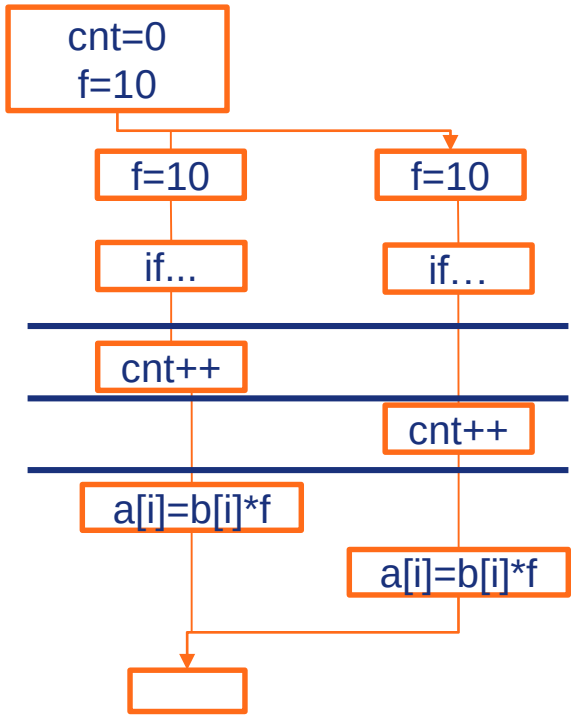
OpenMP – Synchronisation

- Implicit barrier
  - It represents a barrier for beginning and end of parallel constructs, as well as all other control constructs
  - Implicit synchronization can be removed with `nowait` clause
- Explicit barrier
  - Can be achieved by using `critical` clause
  - Code, enclosed in `critical` clause is executed by all threads, but is restricted to only one thread at the time
  - `critical` clause in C/C++:

```
#pragma omp critical [(name)]
```

OpenMP – Synchronisation - critical clause

- Private variable f is fixed in each thread, list a is updated in parallel.



Due to `critical` clause only one thread at a time is executed for `cnt` variable

- C/C++:

```
cnt = 0;
f=10;
#pragma omp parallel
{
#pragma omp for
      for (i=0; i<10; i++) {
         if (b[i] == 0) {
            #pragma omp critical
            cnt ++;
         } /* endif */
         a[i] = b[i]*f;
     } /* end for */
} /*omp end parallel */
```

OpenMP – **Exercise 3**

- Goal: To use a worksharing construct `for` and `critical` directive. To use runtime library calls, conditional compilations, environment variables and parallel regions by calculating pi constant.
- Explanation of algorithm in script *e3.c*
  - Value π is calculated by solving integral
    - $\pi = \int_0^1 \frac{4}{1+x^2} dx$
  - Integral is solved by Riemann sum
    - $\pi = 4 \sum_{i=0}^{n-1} f\left(x_i + \frac{w}{2}\right) w$

OpenMP – **Exercise 3**

- Instructions
  - Compile file *e3.c* and run it. What is the output. Is the value of pi correct?
  - Open file *e3.c* and move to `/* Begin of SPACE for third exercise */`.
  - Add parallel region and parallel `for` directive in *e3.c* and compile it
  - Set environment variable OMP_NUM_THREADS to 2 and run the file.
    - The result is wrong
  - Set environment variable OMP_NUM_THREADS to 12 and run again
    - The result is wrong
  - Add private (x) clause in *e3.c* and compile
  - Set environment variable OMP_NUM_THREADS to 2 and run
    - Should be still incorrect

OpenMP – **Exercise 3**

- Set environment variable `OMP_NUM_THREADS` to 12 and run
  - Still incorrect
- Set environment variable `OMP_NUM_THREADS` to 2 and add `critical` directive in *e2.c* before the `sum` variable in `for` loop and compile again
  - What is the value of pi? (should be correct)
  - What is the CPU time?
- Set environment variable `OMP_NUM_THREADS` to 12 and run
  - What is the CPU time? (it should take longer)
- How to optimize the code? Try to modify the code and move the `critical` directive outside `for` loop to decrease computational time.

OpenMP – **Exercise 4**

- Goal: To exercise the concepts explained in this lecture
- SAXPY stands for “Single-Precision A·X Plus Y”:
  - Equation: $ax + y$
- Instructions
  - Compile *saxpy.c* file without OpenMP and run it.
  - Parallelize the for loop in saxpy function
  - Add timer to the function call. In C language, timer can be added by using function `gettimeofday`. Refer to file *e2.c* and try to figure it out.
  - Compile the *saxpy.c* file. If there are any erorrs, try to solve them on your own.
  - Set number of threads to 1 and run the code. What is the time of execution for saxpy function?
  - Set the number of threads to 2 and run the code. What is the difference in time of execution?
  - Set the number of threads to 12 and run the code. How does the time of execution change with increasing number of threads?

THANK YOU FOR YOUR ATTENTION

www.prace-ri.eu