Alicja Wagner

# Individual work

## Task 0.1 OpenMP pre-training:

- **0.1.1**
  The “chunk” variable specifies what number of contiguous iterations is assigned to a thread. In dynamic scheduling, each thread executes the “chunk” of iterations and then requests another chunk until there are no iterations left.

- **0.1.2**
  *#pragma omp parallel shared(a,b,c,chunk) private(i)*

  The above pragma (*#pragma omp parallel*) is used to specify which region of the code is going to be executed in parallel.

  *Shared(a,b,c,chunk)* implies that variables “a”, “b”, “c” and “chunk’ are shared for all threads which means that if one thread changes the value of a variable, the rest will also see the change.

  *Private(i)* clause means that each thread has its own copy of the variable “i” (the changes of “i” in one thread are not visible in others). In a parallel loop, multiple threads are working on different iterations concurrently. Without marking “i” as private, all threads would share the same loop variable, which would lead to race conditions and incorrect results.

- **0.1.3**
  The *schedule* clause specifies how the chunks of iterations are being distributed among the threads. There are 5 types of scheduling.

  “Dynamic” scheduling (*schedule(dynamic, chunk_size)*), that is used in our example, distributes the loop iterations dynamically at runtime. Each thread requests a new chunk when it finishes its current work. It is especially useful when each iteration takes a different amount of time to complete.

  “Static” scheduling (*schedule(static, chunk_size)*) distributes the chunks to threads in a circular order, which happens at compile-time. It should be used when all iterations take more or less the same time to be completed.

  “Guided” scheduling (*schedule(guided, chunk_size)*) is similar to dynamic scheduling but with decreasing chunk sizes. Larger chunks are initially assigned, and the chunk size decreases as iterations are completed.

  “Auto” scheduling (*schedule(auto)*) allows the OpenMP runtime system or the compiler to automatically choose the scheduling strategy.

  “Runtime” scheduling (*schedule(runtime)*) defers the decision about the scheduling until the runtime.

- **0.1.4**
  In parallelized code sections with OpenMP we can measure efficiency and speedup. We can also measure several factors, such as: execution time, overhead time, the number of threads, each loop execution times.

## Task 0.2 Pre-training std::async:

- **0.2.1**
  The “std::launch::async” parameter is used for specifying how the task is run. “Async” indicates that the function will be executed on a new thread.

- **0.2.2**
  Time for “std::launch::async”: 3004 ms
  Time for “std::launch::deferred”: 5003 ms

  The reason for the time difference is that when the “std::launch::deferred” parameter is used, the execution of the task is deferred until the result is requested (either “wait()” or “get()” is called). Moreover, the task is executed on the calling thread. On the other hand, when using “std::launch::async”, the task is launched on a different thread, immediately after it is created.

- **0.2.3**
  “Wait()” is used to block until the “future” is ready (the result is available), while “get()” is used to retrieve the result and wait for its completion if necessary.

- **0.2.4**
  “Std::async” returns a “std::future” that represents the result of the asynchronous operation. “Future” is a wrapper around a value. On the other hand, “std::thread” represents an actual execution thread.

  “Std::async” manages the exceptions and return values automatically, while in “std::thread” it needs to be done manually. “Std::async” also gives the possibility to choose whether the task should be run asynchronously in a different thread or deferred (in the same thread). In contrast, std::thread always creates a new thread, which may be less efficient for small tasks. Moreover, “std::async” itself provides synchronization (there’s no need to use mutexes).

## Task 0.3 Pre-training std::vector:

- **0.3.1**
  Initializing with required size and adding elements with direct access is more efficient than the default initialization and adding elements with push_back. Direct access is more efficient in terms of time complexity because it doesn't involve dynamic memory allocation or resizing.

- **0.3.2**

  Parallelizing the first loop can lead to data races and undefined behavior. The “push_back” operation involves dynamic memory allocation and potential resizing. These operations are not thread-safe, and if multiple threads attempt to modify the vector concurrently, it can lead to data corruption or crashes. What’s more, if two threads attempt to push elements at the same time, they might interfere with each other, causing unpredictable behavior.

  The second loop however is generally safe to parallelize because it uses direct access to vector elements. There are no concurrent modifications to the same memory location. Each element in the vector is initialized independently, and the vector is not resized or modified in a way that could introduce concurrency issues.

## Task 1 Study of the OpenMP API:

### OpenMP Parallelism Recipe Book

### 1. Introduction

OpenMP (Open Multi-Processing) is an application programming interface (API) that simplifies parallel programming by providing a high-level interface. It is a set of compiler directives and library routines for parallel programming in C, C++, and Fortran.

### 2. Compilation

Since we are working with programs written in c/c++, we will focus on how to enable OpenMP with the g++ compiler. Firstly, in the code, the required library needs to be included.

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

To compile the code, it is needed to add a flag “-fopenmp”.

```
g++ file_name.cpp -o out_name -fopenmp
```

### 3. The basic directive

The basic OpenMP directive looks like this:

```
#pragma omp parallel
{
        // some code to be executed in parallel…
}
```

and tells the compiler to parallelize the chosen block of code taking into account available cores. There are many other directives, designed specifically to handle e.g. for loops, tasks or sections that can be used inside the block of code. There are also clauses which can be added to the directives. They allow to specify e.g. which variables should be shared between threads and which should be private. In the following points several directives and clauses will be discussed in more detail.

# 4. Examples

## 4.1. Directives

### 4.1.1. For loop

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

When the above directive is used, the iterations are (more or less) evenly divided among multiple threads and therefore executed concurrently. This directive can be further specified using clauses (e.g. schedule, shared, private, reduction).

### 4.1.2. Critical

```
#pragma omp critical
{
        // critical section…
}
```

This directive is used to create a critical section that is a part of the code that cannot be disturbed by any other thread. The critical section will be executed by only one thread at a time.

### 4.1.3. Sections

```
#pragma omp sections
{
        #pragma omp section
        {
                // Code for section 1
        }

        #pragma omp section
        {
                // Code for section 2
        }
}
```

The “sections” directive is used to parallelize sections of code. It allows different sections of code to be executed in parallel. Each section is executed by a different thread, and the sections can run concurrently.

### 4.1.4. Barrier

```
#pragma omp barrier
```

The “barrier” directive is used to synchronize threads, ensuring that all threads reach the same point in the code before any of them proceed beyond that point. It acts as a synchronization point, forcing all threads in a

parallel region to wait until all of them have reached the barrier before moving forward.

#### 4.1.5. Ordered

*#pragma omp ordered*

The “ordered” directive is used to ensure that the iterations of a loop or the execution of certain code blocks are executed in the order specified by the program.

### 4.2. Clauses

#### 4.2.1. Schedules

*#pragma omp for schedule(static, chunk_size)*

Scheduling allows us to control how the division between threads will be performed. There are 5 types of schedules: static, dynamic, guided, runtime and auto. The most popular ones are “static” and “dynamic”.

“Static” scheduling distributes the chunks to threads in a circular order. It means that, assuming we have 3 threads, the 1st thread will get the 1st chunk, the 2nd thread - the 2nd chunk, the 3rd, thread - the 3rd, chunk, and now the assignation will start again from the first thread, so the 1st thread will get the 4th chunk, the 2nd thread - the 5th chunk, and so on.

“Dynamic” scheduling distributes the loop iterations dynamically at runtime. Each thread requests a new chunk when it finishes its current work. It is especially useful when each iteration takes a different amount of time to complete.

#### 4.2.2. Shared

*#pragma omp parallel shared(variable1, variable2)*

The “shared” clause in OpenMP is used to specify that a variable should be shared among all threads in a parallel region. When you use shared(*variable1, variable2*), it means that the variables “variable1” and “variable2” will be shared among all threads, and any changes made by one thread will be visible to all other threads.

#### 4.2.3. Private

*#pragma omp parallel private(variable1, variable2)*

The “private” clause is used to specify that each thread should have its private copy of the specified variable(s). Each thread gets its own

independent copy of the variable(s), and modifications to these private copies do not affect the values seen by other threads.

### 4.2.4. Default

*#pragma omp parallel default(shared | private)*

The “default” clause is used to specify the default data-sharing attribute for variables in a parallel region. It allows you to define whether variables should be treated as shared or private by default if no explicit data-sharing clause is specified.

### 4.2.5. Reduction

*#pragma omp for reduction(operation:variable)*
*#pragma omp for reduction(+:sum_var)*

The “reduction” clause is used to perform a reduction operation on one or more variables. It is commonly employed in parallel loops where the result of a certain operation needs to be aggregated across multiple threads. The reduction clause specifies which variables should be treated as private to each thread and specifies the reduction operation to be performed on these private copies.

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; i++) {
        sum += array[i];
}
```

In the example above each thread has its private copy of the “sum” variable on which the summation (operation of the reduction) is performed. After the loop the final result is obtained by combining all the sums from the threads.

### 4.2.6. Nowait

*#pragma omp parallel for reduction(+:sum) nowait*

The “nowait’ clause is used to indicate that a thread executing a parallel region should not wait for other threads to complete before moving on to the next section of code.

Bibliography

https://lsi2.ugr.es/jmantas/ppr/ayuda/omp_ayuda.php?idioma=en
https://stackoverflow.com/questions/
https://learn.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170