PART 1 LAB 3 (Songul Yasin):

1. Task

1. Introduction:

OpenMP is an application programming interface (API) defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for developers of shared-memory parallel applications. It's API supports parallel shared memory programming for C, C++ and Fortran in multiplatform.

2. Compiler/library;

Most OpenMP parallelisms are specified through the use of compiler directives embedded in C/C++ or Fortran source code.
Since we use the C++ compiler g++, the "-fopenmp"-flag is used and the library is supplemented with "omp.h".

3. Examples:

OpenMP is easy to use. The directives, also called pragmas, instruct the compiler to parallelize certain sections of code. All pragmas relevant to OpenMP start with #pragma omp. OpenMP pragmas generally have the form:

#pragma omp <Directive > [clause [[,] clause] ...]

All statements of the form #pragma omp <Directive> [clause [[,] clause] ...] must end with a line break. In particular, they must not end with the opening bracket of the following code block:

```
// misplaced bracket

#pragma omp parallel {

// does not compile

}

// correctly placed brackets

#pragma omp parallel {

// Code

}
```

OpenMP is composed of a set of compiler directives, library functions and environment variables.

Omp_directives:

The directives extend the underlying programming language with constructs for dividing work between threads running in parallel and constructs for synchronizing those threads and allowing them to access data together or separately. Below is a short explanation of different directives:

- *#pragma omp for*: the work-splitting pragma for splits loop runs across multiple threads within the team executing the parallel section

- *#pragma omp single:* The work-dividing directive single ensures that the code block enclosed by it is traversed by exactly one thread from the team executing the parallel section:

- *#pragma omp flush [(list of variables)]* : flush represents a synchronization point for memory areas.

- *#pragma omp section*: Work-dividing directive section to have independent code blocks executed in parallel by one thread from a team.

Omp_clauses

Clauses are optional and affect the behavior of the directive they refer to. Each directive has a different set of valid clauses. For some directives, this set is empty; so no clauses are allowed.

The following section introduces data access clauses that decide the scope of variables:

shared and *private*: They explicitly mark a variable as shared by all threads or privately used by one thread.

firstprivate and *lastprivate*: The contents of private variables are undefined when entering and leaving their scope. The firstprivate and lastprivate clauses allow initialization and finalization of their values, respectively.

default: Variables that are not explicitly listed with a shared or private clause are shared by all threads. This standard behavior can be changed using the default clause.

reduction: The reduce clause identifies special shared variables in which multiple threads can accumulate values.

Omp_shedule

The different flow charts in OpenMP differ in how the iterations of a loop are broken into pieces. chunks) and how they are assigned to a team's threads. The syntax of the schedule clause is: schedule(type[, chunk])
Here type stands for one of the four specifications *static, dynamic, guided or runtime and auto.*

Omp_routines and environment variables

The library functions (omp_routines) and environment variables control the parameters of the runtime environment in which the parallel program runs.

The functions of the runtime library are mainly used to query or set parameters of the OpenMP runtime environment. In addition, the library contains functions for synchronizing threads. If you want to use functions from the runtime library in a program, the header file omp.h must be included.

The OpenMP runtime library has various functions, three of which are described below:

- *omp_get_num_threads()*: returns the number of threads in the currently active team, i.e. the number of threads that are currently executing the code block in which the function call is located.

- *omp_get_max_threads()*: returns the maximum number of threads possible according to the above rules, which is not tied to the current team.

- *omp_get_num_procs()*: returns the number of processors on which the program can be executed in parallel.

The library functions and environment variables control the parameters of the runtime environment in which the parallel program runs.

Example:

The following minimal code example in C++ where a vector of integers is initialized in parallel by multiple threads.

```
1. const int size = 500;
2. int arr[size];
3. #pragma omp parallel for
4.  for(inti=0;i<size;++i)
5.          arr[i] = i;
```

The first two lines define a vector of integers called arr of size 500. In line 3 we encounter the first OpenMP expression: The compiler directive #pragma omp parallel for (also called pragma) causes the for loop following line 4 to run in parallel by multiple threads. The loop body of this loop consists of the single statement on line 5, where each vector element is assigned its index as a value. It is this instruction that is executed in parallel.

0. Task:

0.1 OpenMP pre-training:

0.1.1: What is the chunk variable used for?

The “Chunk” value determines how many loop iterations are allocated to each thread at a time.

0.1.2: Fully explain the pragma: pragma omp parallel shared(a,b,c,chunk) private(i)

Why and what is „shared(a,b,c,chunk)“ used for in this program?

The “shared(a,b,c,chunk)” clause indicates that the variables “a”, “b”, “c”, and “chunk” are shared among all threads. This means that each thread has its own copy of the variable.

Why is the variable „i“ labeled as private in the pragma?

The “private(i)” clause specifies that each thread should have its own private copy of the loop variable “i”. Without this clause, different threads could interfere with other threads' loop variables, resulting in incorrect results

0.1.3: What is the schedule for? What other possibilities are there?

The “schedule(dynamic, chunk)” clause specifies the scheduling policy to be used for distributing loop iterations among the threads. In this case, a dynamic scheduling policy is used.Other possible scheduling policies include:

- Static scheduling (“schedule(static, chunk)”): iterations are divided into blocks of size chunk_size and the blocks are distributed to the threads in the team in the order of thread numbers.

- Guided scheduling (“schedule(guided, chunk)”): Each thread executes an iteration block until there are no more blocks to allocate.

- Automatic scheduling (“schedule(auto)”): Scheduling decisions are delegated to the compiler and/or the runtime system.

0.1.4: What times and other performance metrics can we measure in parallelized code sections with OpenMP?

With OpenMP you can measure and analyze the following parameters:

- Execution time: The total time required to execute the parallelized section of the code.

- Speedup: The ratio of execution time to how much faster the parallelized code is compared to the serial version.

- Efficiency: It measures how efficiently the parallelized code uses the available processors.

- Load balancing: Analysis of how evenly the workload is distributed across threads.

- Overhead: Any additional time or resources consumed by parallelization

0.2 Task Pre-training std::async:

0.2.1 What is the „std::launch::async“ parameter used for?

As soon as the “std::async” call is made, the function starts execution in a new thread (asynchronous)

0.2.2 Calculate the time the program takes with „std::launch::async“ and „std::launch::deferred“. What is the reason for the time difference?

With “std::launch::async” the tasks are executed in separate threads at the same time. The program waits for “task1” to complete with “task1.wait()”, but “task2” continues to execute at the same time. The total time required will therefore roughly correspond to the time required by the longest task “Task2”.

With “std::launch::deferred” the tasks are executed with a delay. In this case, both tasks are executed one after the other because task2 does not start until get() is called on task1. Therefore, the total time is the sum of the individual task times.

0.2.3 What is the difference between the „wait“ and „get“ methods of „std::future“?

- “wait()”: It blocks the current thread and waits until the associated asynchronous task is ready.

- “get()”: It also blocks the current thread, but additionally returns the result of the associated asynchronous task when it is ready.

0.2.4 What advantages does „std::async“ offer over „std::thread“?

- Simplified syntax: It manages the thread automatically and simplifies the process of starting and managing asynchronous operations.

- Automatic management of threads: “std::async” automatically decides whether the task is executed asynchronously or postponed.

- Exception handling: “std::async” provides a convenient way to handle exceptions thrown by the asynchronously executed function.

0.3 Task: Pre-training std::vector :

0.3.1 Which of the two ways of initializing the vector and filling it is more efficient? Why?

The second option is more efficient: This is because it avoids the need for dynamic reallocation and resizing that occurs when using push_back in the first approach (“v1”). In contrast, the second approach uses “std::vector<float> v2(10000)”. This avoids the dynamic resizing overhead, resulting in better performance.

0.3.2 Could a problem occur when parallelizing the two for loops? Why?

Yes, parallelizing the two for loops could cause a problem if you try to change elements of the vector at the same time. If the two loops run in parallel, a data race could occur.
A data race occurs when multiple threads access the same memory location at the same time and at least one of them is a write.