Saint Petersburg National Research University of Information Technologies, Mechanics and Optics (ITMO University)
Faculty of Informational Technologies and Programming

REPORT

about laboratory work № 1

«Definite integral calculation»

Student

<u>Miroshnikova A.D.</u> **j4133c**

(Surname, initials) Group

Saint-Petersburg, 2020

1. GOAL OF LABORATORY WORK

The goal of the work is to become acquainted with parallel programming using OpenMP C++ library.

2. TASK DEFINITION

The problem to be solved is definite integral calculation using trapezoidal rule. The function for which the integral should be calculated is

$$f(x)=\frac{1}{x^2}\sin\left(\frac{1}{x^2}\right) \quad .$$

The trapezoidal rule allows to calculate approximate definite integral value from A to B using the following formula:

$$J_n(A,B)=\frac{B-A}{n}\left(\frac{f(x_0)}{2}+\sum_{i=1}^{n-1} f(x_i)+\frac{f(x_n)}{2}\right) \quad .$$

The calculation of the sum can be parallelized in different ways. The task is to implement it in a sequential way and in different parallel ways using atomic directive, critical directive, semaphore variables (locks), and reduction clause. Then the performance of the methods should be compared.

3. BRIEF THEORY

OpenMP is an implementation of multithreading, a method of parallelizing whereby a primary thread (a series of instructions executed consecutively) forks a specified number of sub-threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a compiler directive that will cause the threads to form before the section is executed. Each thread has an id attached to it which can be obtained using a function (called `omp_get_thread_num()`). The thread id is an integer, and the primary thread has an id of 0. After the execution of the parallelized code, the threads join back into the primary thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

Since OpenMP is a shared memory programming model, most variables in OpenMP code are visible to all threads by default. But sometimes private variables are necessary to avoid race conditions and there is a need to pass values between the sequential part and the parallel region (the code block executed in parallel), so data environment management is introduced as data sharing attribute clauses by appending them to the OpenMP directive. Several examples of this clause are the following:

- *critical* – (synchronization clause) the enclosed code block will be executed by only one thread at a time, and not simultaneously executed by multiple threads. It is often used to protect shared data from race conditions.
- *atomic* – (synchronization clause) the memory update (write, or read-modify-write) in the next instruction will be performed atomically. It does not make the entire statement atomic; only the memory update is atomic. A compiler might use special hardware instructions for better performance than when using *critical*.
- *usage of semaphore variables* – a semaphore variable can have only two possible values: locked or unlocked. When a semaphore is unlocked by the authorized process, other processes are free to access and lock it. If a process tries to lock an unlocked semaphore then the semaphore gets locked until the process unlocks it. If a process tries to lock an already locked semaphore it has to wait until the previous process unlocks the semaphore.
- *reduction* – (reduction clause) the variable has a local copy in each thread, but the values of the local copies will be summarized (reduced) into a global shared variable. This is very useful if a particular operation (specified in operator for this particular clause) on a variable runs iteratively, so that its value at a particular iteration depends on its value at a prior iteration. The steps that lead up to the operational increment are parallelized, but the threads updates the global variable in a thread safe manner.

4. ALGORITHM (METHOD) OF IMPLEMENTATION

The code for sequential and parallel definite integral calculation was implemented using C++ and compiled with g++ compiler (version 7.5.0, Ubuntu 18.04). The processor used during measurements was Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz with 6 physical and 12 virtual cores. The data was analyzed using Python 3.6.9. Source code and data gathered are available no https://github.com/Miffka/parallel_algo_itmo/tree/master/task1.

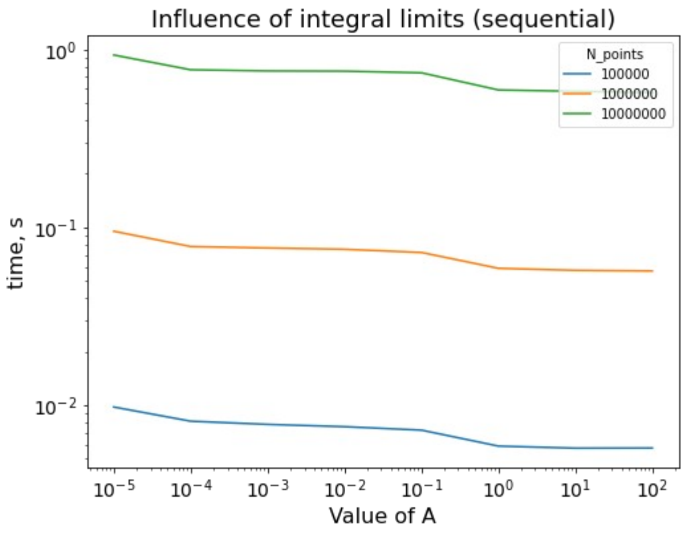
The experiments were designed as follows. For each implementation method in «sequential», «atomic», «critical», «lock», «reduction»:

- number of threads varied from 1 to 6;
- number of points varied in $\{10^5, 10^6, 10^7\}$;
- the values of A and B (limits of integration) varied in {(0.00001, 0.0001), (0.0001, 0.001), (0.001, 0.01), (0.01, 0.1), (0.1, 1), (1, 10), (10, 100), (100, 1000)}.

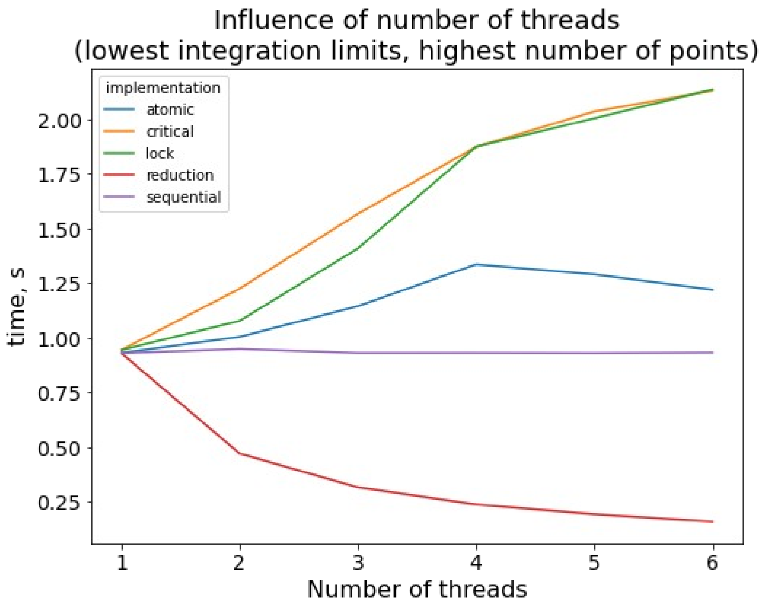
Each experiment was run 5 times and then average time was taken as final measurement. All measurements were written to a file (contains 720 rows, available on GitHub) and then analyzed in Python.

5. RESULT AND EXPERIMENTS

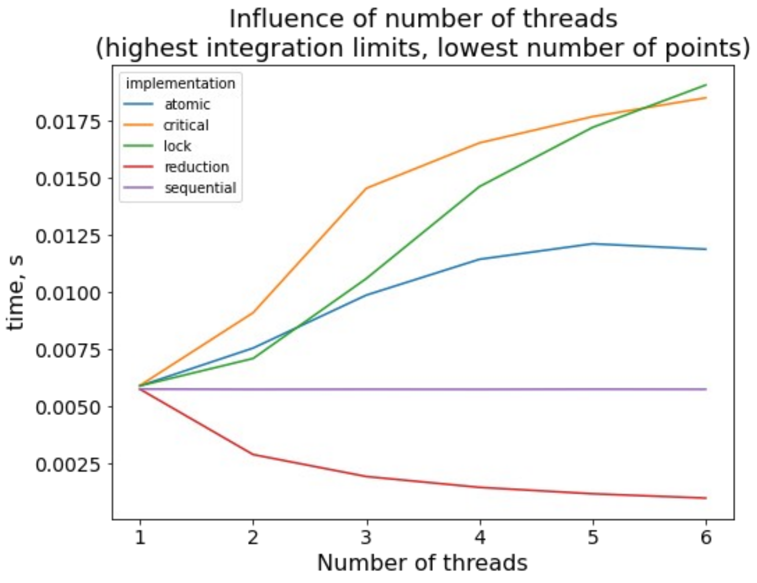
Influence of number of points and integral limits was analyzed on the sequential method of calculation. The higher the number of points was – the higher was time needed for calculation. Also the higher were values of A and B, the lower was the time.



The influence of implementation and number of threads was unexpected. When number of threads was equal to 1, both parallel and sequential approaches needed the same time to run. But as the number of threads increased, only code implemented with reduction clause became faster. The usage of all other parallel techniques even increased the time required for computation.



This effect was observed for any integration limits and number of points present in experiment. The possible explanation for it is that in «atomic», «critical», and «lock» cases each thread should wait until the slowest thread to finish working with their common resource – the variable “internal_sum” to which we add values of $f(x_i)$. In the «reduction» case each thread operates with its own copy of this variable.



In the case of the reduction clause the time spent on computation is decreasing as number of threads increases in a logarithmic way.

6. CONCLUSION

The concept of OpenMP techniques such as atomic directive, critical directive, semaphore variables (locks), and reduction clause was reviewed. The techniques were implemented in C++, and the results were discussed. The main takeaway is that if we have to use the same variable to keep the result in (like internal sum) it’s better to use reduction clause. For the other tasks which should e.g. just process some data the atomic clause would be better (probably).