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

Voronin Victor J4132c

(Surname, initials) Group

Saint-Petersburg, 2020

1 GOAL OF LABORATORY WORK

Calculate the value of a definite integral with precision ε:

$$J(A,B) = \int_A^B f(x)\,dx$$

2 TASK DEFINITION

Calculate the value of a definite integral with precision ε:

$$J(A,B) = \int_A^B f(x)\,dx = \int\limits_A^B \frac{1}{x^2} sin^2\left(\frac{1}{x}\right) dx = -\frac{1}{2x} + \frac{1}{4} sin\left(\frac{2}{x}\right)\Bigg|_A^B$$

$$= \frac{1}{4}\left(2\frac{B-A}{AB} + sin\left(\frac{2}{B}\right) - sin\left(\frac{2}{A}\right)\right)$$

Consider there is uniform grid on [A,B] region with *n+1* points:

$$x_i = A + \frac{B-A}{n} i, \quad i = 0, \cdots, n$$

The trapezoidal rule is a technique for approximating the definite integral by approximating the region under the graph of the function as a trapezoid and calculating its area:

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

3 BRIEF THEORY

Calculate the value of definite integral means to calculate area under the curve of integrand. Divide the area along X axes with uniform grid on [A,B] region with *n+1* points:

$$x_i = A + \frac{B-A}{n} i, \quad i = 0, \cdots, n$$

And then form n trapezes with height of x_i. Approximating value of the definite integral can be get as sum of areas of trapezes. Namely:

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

To solve this problem it is used serial and parallel implementations with OpenMP.

Time of execution is measured with getTimeExecution method. See Appendix 1. Method return average time spent in microseconds for 10 attemps. In this method pointers to function and library <chrono> were used to calculate target function with given arguments and time spent.

Function showed above was realized with method f(x). It required to use additional library <cmath> to calculate sin(x).

Serial version of program to calculate was realized with integrateSerial(int a, int b) method. See appendix 2 for more details.

Parallel implementation were made with four kind of technics:

1. Atomic section
   a. The atomic directive ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads. Such the memory location must be value of resulting sum (variable “temp”). Method that implementation this directive is integrateAtomic (int a, int b) which was presented in appendix 3. Directive Atomic is very similar to critical section which will consider next.
2. Critical section.
   a. The critical directive identifies a construct that restricts execution of the associated structured block to a single thread at a time. A thread waits at the beginning of a critical region until no other thread is executing a critical region (anywhere in the program) with the same name.

   b. After analyzing the objective function, it is clear that the sum operation should be placed in the critical section. Each thread calculates its own function value (it was previously set as private) and then sends it to the critical section, where it is added with values from other threads. See code's detail in appendix 4
3. Lock function
   a. The functions described in this section manipulate locks used for synchronization. For the following functions, the lock variable must have type omp_lock_t. This variable must only be accessed through these functions. All lock functions require an argument that has a pointer to omp_lock_t type.
      i. The omp_init_lock function initializes a simple lock.
      ii. The omp_destroy_lock function removes a simple lock.
      iii. The omp_set_lock function waits until a simple lock is available.
      iv. The omp_unset_lock function releases a simple lock.
   b. The value of sum is made object of lock. Code was presented in appendix 5.
4. Reduction
   a. This clause performs a reduction on the scalar variables that appear in variable-list, with the operator op. A private copy of each variable in variable-list is created, one for each thread, as if the private clause had been used. The private copy is initialized with 0 for operators of sum or 1 for multiplicate operators. At the end of the region for which the reduction clause was specified, the original object is updated to reflect the result of combining its original value with the final value of each of the private copies using the operator specified.
   b. integrateReduction method was implemented using reduction clause. Code is presented in appendix 6.

4 RESULT AND EXPERIMENTS

The use of all of the above methods has been summarized in the method main() presented in appendix 7. Following tables were received (see table 1– 4).

Table 1 – Serial version

A	B	Npoints	time, ms
0.00001	0.0001	1000000	97614
0.0001	0.001	1000000	99221
0.001	0.01	1000000	94396
0.01	0.1	1000000	95799
0.1	1	1000000	95542
1	10	1000000	93688
10	100	1000000	95207

Table 2 – Atomic version (4 threads)

A	B	Npoints	time, ms
0.00001	0.0001	1000000	39884
0.0001	0.001	1000000	38402
0.001	0.01	1000000	37503
0.01	0.1	1000000	36199
0.1	1	1000000	36402

1	10	1000000	34203
10	100	1000000	34002

Table 3 – Critical section version (4 threads)

A	B	Npoints	time, ns
0.00001	0.0001	1000000	82895
0.0001	0.001	1000000	83006
0.001	0.01	1000000	88868
0.01	0.1	1000000	84037
0.1	1	1000000	81309
1	10	1000000	82522
10	100	1000000	85115

Table 4 – Lock version (4 threads)

A	B	Npoints	time, ns
0.00001	0.0001	1000000	3130547
0.0001	0.001	1000000	3149981
0.001	0.01	1000000	3094402
0.01	0.1	1000000	3140854
0.1	1	1000000	3106169
1	10	1000000	3168419
10	100	1000000	3177140

Table 5 – Reduction version (4 threads)

A	B	Npoints	time, ns
0.00001	0.0001	1000000	95705
0.0001	0.001	1000000	95010
0.001	0.01	1000000	92398
0.01	0.1	1000000	92507
0.1	1	1000000	87059
1	10	1000000	92011
10	100	1000000	91128

Speedup that obtained were collected in table 6

Table 6 – Speedup relative serial version (4 threads)

Interval	Serial	Atomic	Critical	Lock	Reduction
0.00001--0.0001	1.00000	2.44745	1.17756	0.03118	1.01995
0.0001--0.001	1.00000	2.58375	1.19535	0.03150	1.04432
0.001--0.01	1.00000	2.51703	1.06220	0.03051	1.02162

0.01--0.1	1.00000	2.64645	1.13996	0.03050	1.03559
0.1--1	1.00000	2.62464	1.17505	0.03076	1.09744
1--10	1.00000	2.73917	1.13531	0.02957	1.01823
10--100	1.00000	2.80004	1.11857	0.02997	1.04476
Average	1.00000	2.62265	1.14343	0.03057	1.04027

Also calculate the speedup for a different number of threads (see table 7 and 8 and fig.1)

Table 7 – Speedup relative serial version (2 threads)

Interval	Serial	Atomic	Critical	Lock	Reduction
0.00001--0.0001	1.00000	1.62410	1.32268	0.03107	0.99329
0.0001--0.001	1.00000	1.61997	1.25092	0.03217	0.88416
0.001--0.01	1.00000	1.60757	1.38438	0.03111	0.03111
0.01--0.1	1.00000	1.77442	1.23640	0.02630	1.06246
0.1--1	1.00000	1.61629	1.23789	0.02441	1.05173
1--10	1.00000	1.67251	1.47440	0.02547	0.98929
10--100	1.00000	1.69492	1.54545	0.03200	1.04596
Average	1.00000	1.65854	1.35030	0.02893	0.86543

Table 8 – Speedup relative serial version (8 threads)

Interval	Serial	Atomic	Critical	Lock	Reduction
0.00001--0.0001	1.00000	1.73727	0.85990	0.03194	1.03982
0.0001--0.001	1.00000	1.67188	0.82708	0.02849	1.01274
0.001--0.01	1.00000	1.64101	0.85071	0.02803	0.93081
0.01--0.1	1.00000	1.67529	0.92655	0.03155	1.14248
0.1--1	1.00000	1.44982	0.81482	0.02884	1.06213
1--10	1.00000	1.52518	0.82067	0.02824	1.03080
10--100	1.00000	1.65522	0.86997	0.02823	0.90138
Average	1.00000	1.62224	0.85282	0.02933	1.01716

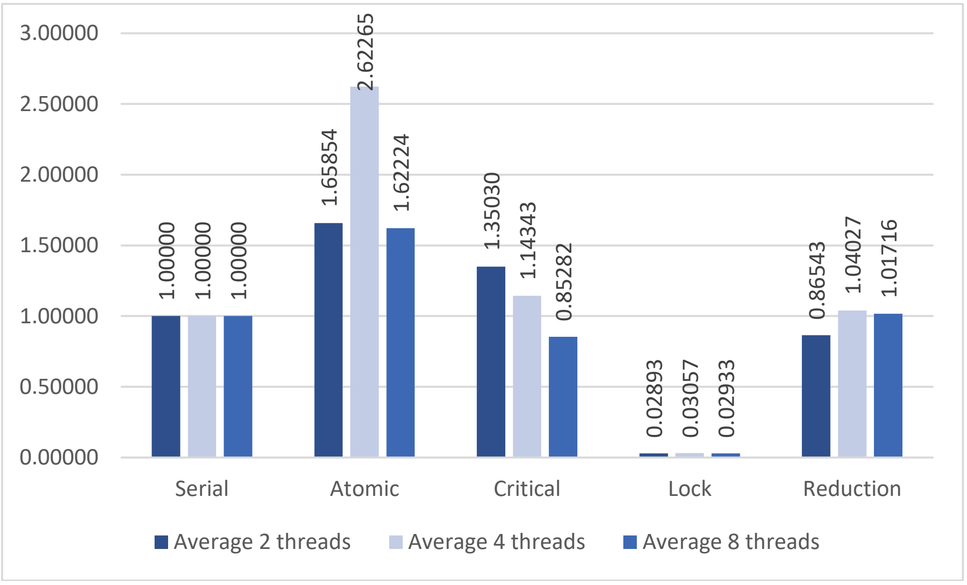

Figure 1 Average time spent versus count of threads

As seen from figure 1, it has threshold in 4 threads after that increase the number of threads is heading for increase performance. Almost in all cases it happened save for critical section version where with each increase performance makes poorer. It can be explained with this fact that each threads finishes its part of tasks in about the same time after that begins to try to get access to critical section. Obviously, the more flows we have, the greater the crush at the entrance to the critical section arises.

General decrease of speedup after 4 threads may be explained with fact that create threads also requires time and if you have not strongly hard task speedup may be not arise because increase overhead cost for create of threads. That can be seen in given task. To this reason it is important to remember that parallel techniques is not nostrum and must be applied with realize.

5 CONCLUSION

During the execution of the task, parallel directive such as Atomic, critical section, Lock and reduction were applied to calculate the definite integral in difference limit. The speed up obtained from the use of concurrency is determined. The results obtained were analyzed.

APPENDIX 1

```
int getTimeExecution(double(*)(double, double), double, double);
int getTimeExecution(double(*op)(double, double), double a, double b){
    int i;
    double cntIteration = 5;
    double totalTime = 0.0;
    for (i=0; i<cntIteration; i++)
    {
        auto t1 = std::chrono::high_resolution_clock::now();
        op(a, b);
        auto t2 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double, std::micro> fp_ms = t2 - t1;
        totalTime += fp_ms.count();
    }
    return totalTime/cntIteration;
}
```

APPENDIX 2

```
double integrateSerial(double a, double b){
  double x, temp = 0.0;
  int i;
  double step=(double)(b-a)/cntSteps;
  for (i=1; i<cntSteps; i++)
  {
    x = a+step*i;
   temp += f(x);
  }
  return step * (f(a)/2.0 + temp + f(b)/2.0);
}
```

Appendix 3

```
double integrateAtomic(double a, double b){
    double x, temp = 0.0;
    int i;
    double step=(double)(b-a)/cntSteps;
    #pragma omp parallel for private(i, x) num_threads(threadsCount)
        for (i=1; i<cntSteps; i++) {
            x = a+step*i;
            #pragma omp atomic
            temp += f(x);
        }
    return step * (f(a)/2.0 + temp + f(b)/2.0);
}
```

APPENDIX 4

```
    double integrateCritical(double a, double b){
        double x, temp = 0.0;
        int i;
        double step=(double)(b-a)/cntSteps;
        #pragma omp parallel for private(i, x) num_threads(threadsCount)
            for (i=1; i<cntSteps; i++) {
                x = f(a+step*i);
                #pragma omp critical
                temp += x;
            }
        return step * (f(a)/2.0 + temp + f(b)/2.0);
    }
```

APPENDIX 5

```
    double integrateLock(double a, double b){
        double x, temp = 0.0;
        int i;
        double step=(double)(b-a)/cntSteps;
        omp_lock_t mylock;
        omp_init_lock(&mylock);
        #pragma omp parallel for private(i, x) num_threads(threadsCount)
            for (i=1; i<cntSteps; i++) {
                x = a+step*i;
                omp_set_lock(&mylock);
                temp += f(x);
                omp_unset_lock(&mylock);
            }
        omp_destroy_lock(&mylock);
        return step * (f(a)/2.0 + temp + f(b)/2.0);
    }
```

APPENDIX 6

```
    double integrateReduction(double a, double b){
        double x, temp = 2.0;
        int i;
        double step=(double)(b-a)/cntSteps;
            #pragma opm parallel reduction (+ : temp)
            for (i=1; i<cntSteps; i++) {
                temp += f(a+step*i);
            }
        return step * (f(a)/2.0 + temp + f(b)/2.0);
    }
```

APPENDIX 7

```
int main() {
    double a, b, i = 0.0;
    for (i = 0.00001; i<11; i=i*10)
    {
        a = i;
        b = i*10;
        double integralValueSerial = integrateSerial(a,b);
        double timeExecutionSerial = getTimeExecution(integrateSerial, a, b);

        double integralValueAtomic = integrateAtomic(a,b);
        double timeExecutionAtomic = getTimeExecution(integrateAtomic, a, b);

        double integralValueCritical = integrateCritical(a,b);
        double timeExecutionCritical = getTimeExecution(integrateCritical, a, b);

        double integralValueLock = integrateLock(a,b);
        double timeExecutionLock = getTimeExecution(integrateLock, a, b);

        double integralValueReduction = integrateReduction(a,b);
        double timeExecutionReduction = getTimeExecution(integrateReduction, a, b
);

        double integralEntirely = (2.0*(b-
a)/(a*b) + sin(2.0/b) - sin(2.0/a)) / 4.0;
        double accurancy = abs(integralEntirely - integralValueSerial)*100.0/inte
gralEntirely;

        std::cout<<std::endl<< "A \t| B \t| Npoint|" << std::endl;
        std::cout<<" "<<a<<" \t|" <<b<<" \t| "<<cntSteps<<"\t|"<< std::endl;

        std::cout<<"----------------|-------|"<<std::endl;
        std::cout<< "Method \t\t|  Value \t| AvrTime  |" << std::endl;
        std::cout<< "Serial\t\t| "<<std::setw(10)<< std::setprecision(11) << inte
gralValueSerial << "|"
                  <<std::setw(10) << std::setprecision(10) << timeExecutionSerial
<<"| "<<std::endl;
        std::cout<< "Atomic\t\t| "<<std::setw(10)<< std::setprecision(10) << inte
gralValueAtomic << "|"
                  <<std::setw(10) << std::setprecision(10) << timeExecutionAtomic
<<"| "<< std::endl;
        std::cout<< "Critical\t| "<<std::setw(10)<< std::setprecision(10) << inte
gralValueCritical << "|"
                  <<std::setw(10) << std::setprecision(10) << timeExecutionCritica
l <<"| "<< std::endl;
        std::cout<< "Lock\t\t| "<<std::setw(10)<< std::setprecision(10) << integr
alValueLock << "|"
                  <<std::setw(10) << std::setprecision(10) << timeExecutionLock <<
"| "<< std::endl;
        std::cout<< "Reduction\t| "<<std::setw(10)<< std::setprecision(10) << int
egralValueReduction << "|"
                  <<std::setw(10) << std::setprecision(10) << timeExecutionReducti
on <<"| "<< std::endl;
```

```
        std::cout << "W/o steps \t| " << std::setw(10) << std::setprecision(10) <
< integralEntirely << "|"<<std::endl;
        std::cout << "Accurancy \t| "  << std::setw(10) << std::setprecision(11)
<< accurancy<< "\t |" << std::endl;
    }
}
```