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
Pasala Krishna Chaitanya J4134c
(Surname, initials) Group

Saint-Petersburg, 2020

1. GOAL OF LABORATORY WORK

The main goal of the task is to understand how OpenMP API (Application Program Interface) is typically used for loop-level parallelism, and the implementation of synchronization and reduction methods.

2. TASK DEFINITION

Calculate the value of the definite integral of given function with any chosen precision for the given regions using trapezoidal rule. Find the execution time for the serial program and write a parallel program using ‘atomic’, ‘Critical section’, ‘Locks’, and ‘reduction’, and count speedup with different thread numbers.

$$f(x) = (1/x^2) * sin^2 (1/x)$$

3. BRIEF THEORY AND ALGORITHM

In numerical analysis, the trapezoidal rule is a technique for approximating the definite integral. The trapezoidal rule works by approximating the region under the graph of the function as a trapezoid and calculating its area. It may be viewed as the result obtained by averaging the left and right Riemann sums.

- Choose a precision, find out the step size (number of intervals). As the size of the interval is constant. Apply the for loop, to calculate the area of the trapezoid in each interval. Add them up to get the area of the region bounded by the function and x-axis.

Parallel computing is a type of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved at the same time at different processors.

The OpenMP is an Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism in C/C++ and Fortran. . The OpenMP API defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer..It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

An OpenMP application begins with a single thread, the master thread. As the program executes, the application may encounter parallel regions in which the master thread creates thread teams (which include the master thread). At the end of a parallel region, the thread teams are parked and the master thread continues execution. From within a parallel region there can be nested parallel regions where each thread of the original parallel region becomes the master of its own thread team. Nested parallelism can continue to further nest other parallel regions.

OpenMP is easy to use and consists of only two basic constructs: pragmas and runtime routines. The OpenMP pragmas typically direct the compiler to parallelize sections of code. All OpenMP pragmas begin with '#pragma omp'. OpenMP runtime routines are used primarily to set and retrieve information about the environment. There are also APIs for certain types of synchronization. In order to use the functions from the OpenMP runtime, the program must include the OpenMP header file omp.h. If the application is only using the pragmas, you can omit omp.h.

Parallel construct is used to specify the computations that should be executed in parallel. A team of threads are created to execute the associated parallel region. The work of the region is replicated for every thread. At the end of a parallel region, there is an implied barrier that forces all threads to wait until the work inside the region has been completed. Many applications can be parallelized by using just a parallel region and one or more of work-sharing constructs, possibly with clauses.

Workshare construct divides the execution of code region among the members of the team. They do not launch new threads, there is no implied barrier upon entry to a them, however there is an implied barrier at the end of a Workshare construct. They must be enclosed within a parallel region, and must be encountered by all members of a team or none at all

False sharing is a performance-degrading usage pattern that can arise in systems with distributed, coherent caches at the size of the smallest resource block managed by the caching mechanism. When two or more threads update different data elements in the same cache line simultaneously, they interfere with each other. It's the reason for false sharing.

With multiple threads running concurrently, there are often times when it's necessary to have one thread synchronize with another thread. OpenMP supplies multiple types of synchronization to help in many different situations. Synchronization is bringing one or more threads to a well defined and known point in their execution. A few synchronization clauses are:

- critical: 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: 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.

An OpenMP critical section is completely general - it can surround any arbitrary block of code. You pay for that generality, however, by incurring significant overhead every time a thread enters and exits the critical section . An atomic operation has much lower overhead. Where available, it takes advantage of the hardware providing an atomic increment operation; in that case there's no lock/unlock needed on entering/exiting the line of code, it just does the atomic increment.

OpenMP also has the traditional mechanism of a lock . A lock is somewhat similar to a critical section, it is a form of mutual exclusion. it guarantees that some instructions can only be performed by one process at a time. Locks make sure that some data elements can only be touched by one process at a time. Locks are a low synchronization method. A lock in OpenMP is an object (omp_lock_t) that can be held by at most one thread at a time.

The OpenMP reduction clause lets you specify one or more thread-private variables that are subject to a reduction operation at the end of the parallel region. OpenMP pre defines a set of reduction operators. Each reduction variable must be a scalar. OpenMP also defines several restrictions on how reduction variables are used in a parallel region.

Syntaxes of the constructs and the C++ code is mentioned below in the APPENDIX.

4. RESULT AND EXPERIMENTS

The results show the efficiency of the use of OpenMP API and how the execution can be speed up through the controlled use of synchronization and reduction methods. The execution time is calculated for the step size of 10000000, and the number of threads are 2, 4, and 16. The average of 5 executions is used to calculate the execution time.

a	b	Integral	Serial	Atomic	Critical	Locks	Reduction
0.00001	0.0001	45000.2	1.44582	0.886676	0.77605	2.37892	0.807796
0.0001	0.001	4500.09	1.49531	0.807975	0.824674	2.28834	0.795983
0.001	0.01	449.549	1.49658	0.765806	0.815965	0.22235	0.762358
0.01	0.1	45.4466	1.4963	0.757254	0.775833	2.14984	0.796498
0.1	1	4.49909	1.36122	0.755371	0.740679	2.18774	0.748265
1	10	0.272343	1.30007	0.703417	0.69887	2.14904	0.708285
10	100	0.000332	1.35578	0.746672	0.703241	2.10321	0.75509

Table 1: Execution time using serial and parallel programs for number of threads 2, and step size is 10000000.

a	b	Integral	Serial	Atomic	Critical	Locks	Reduction
0.00001	0.0001	45000.2	1.53124	0.54109	0.59793	2.47998	0.48258
0.0001	0.001	4500.09	1.47676	0.606402	0.59448	2.64273	0.60098
0.001	0.01	449.549	1.47024	0.56054	0.58874	2.6309	0.58195
0.01	0.1	45.4466	1.50825	0.4587	0.531261	2.30614	0.4854
0.1	1	4.49909	1.33221	0.46658	0.55147	2.5712	0.5684
1	10	0.272343	1.46558	0.5468	0.50477	2.3984	0.4275
10	100	0.000332	1.34758	0.4987	0.5334	2.69522	0.527166

Table 2: Execution time using serial and parallel programs for number of threads 4, and step size is 10000000.

a	b	Integral	Serial	Atomic	Critical	Locks	Reduction
0.00001	0.0001	45000.2	1.5134	0.48521	0.52513	2.5324	0.46521
0.0001	0.001	4500.09	1.4724	0.53455	0.47954	2.6354	0.52104
0.001	0.01	449.549	1.5021	0.48954	0.51458	2.5954	0.47726
0.01	0.1	45.4466	0.5074	0.51347	0.55368	2.5547	0.48806
0.1	1	4.49909	1.4358	0.5984	0.49985	2.54697	0.538415
1	10	0.272343	1.3781	0.46254	0.50874	2.4824	0.4658
10	100	0.000332	1.3774	0.4854	0.49247	2.4851	0.4679

Table 2: Execution time using serial and parallel programs for number of threads 16, and step size is 10000000.

5. CONCLUSION

It is observed that using parallel synchronization and reduction methods had provided speed up for the definite integral calculation problem using the trapezoidal method. Using more threads has increased the speed of the algorithm, but the speed up gradually becomes constant using more and more threads. As it is observed that 8 and 16 threads, both provide similar speed up. A speed up of 2 is provided using 2 threads and a speed of 3 using 8 and 16 threads. As Locks are low level synchronization methods, it does not provide better results. But locking is exception safe and it is allowed to leave the locked region with jumps, which is forbidden in regions protected by the critical-directive.

As an atomic variable comes with synchronization. In order to ensure that there are no race conditions, threads must synchronize which effectively means that parallelism is lost. The serialization of memory accesses disables parallelism, but all memory accesses are done by atomic operations. These operations require at least 20--50 cycles on modern computers and will dramatically slow down your performances if used intensively. Reduction on the other hand is a general operation that can be carried out in parallel using parallel reduction algorithms.

Appendix:

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

using namespace std;
double f(double x);
double integral(double a , double b);
double integral_critical(double a, double b);
double integral_atomic(double a, double b);
double integral_lock(double a, double b);
double integral_reduction(double a, double b);
static int thread_count = 4;
static int n=10000000;

int main()
{
  double I, start, end, I_C, start_c, end_c, I_A, start_a, end_a, \
  I_L, start_l, end_l, I_R, start_r, end_r ;

  double a[]={0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10};
  double b[]={0.0001, 0.001, 0.01, 0.1, 1, 10, 100};

  int size= sizeof(a)/sizeof(a[0]);
  cout<< "Number of steps: "<< n<< "\n Number of threads: "<< thread_count<< endl;
  for (int i=0; i<size; i++)
  {
    start=omp_get_wtime();
    for(int j=0; j<5; j++)
    {
      I=integral(a[i],b[i]);
    }
    end=omp_get_wtime();
    cout<< "a="<<a[i]<<" :b="<<b[i]<<"\n";
    cout<<"Integral: "<< I<<"\n"<<"Execution time: "<<(end-start)/5<<endl;

    start_c=omp_get_wtime();
    for(int j=0; j<5; j++)
    {
      I_C=integral_critical(a[i],b[i]);
    }

    end_c=omp_get_wtime();
    cout<<"Integral_critical: "<< I_C<<"\n"<<"Execution time: "<<(end_c-start_c)/5<<endl;

    start_a=omp_get_wtime();
    for(int j=0; j<5; j++)
    {
      I_A=integral_atomic(a[i],b[i]);
    }
    end_a=omp_get_wtime();
    cout<<"Integral_atomic: "<< I_A<<"\n"<<"Execution time: "<<(end_a-start_a)/5<<endl;

    start_l=omp_get_wtime();
    for(int j=0; j<5; j++)
    {
      I_L=integral_lock(a[i],b[i]);
    }
    end_l=omp_get_wtime();
    cout<<"Integral_lock: "<< I_L<<"\n"<<"Execution time: "<<(end_l-start_l)/5<<endl;
```

```
    start_r=omp_get_wtime();
    for(int j=0; j<5; j++)
    {
      I_R=integral_reduction(a[i],b[i]);
    }
    end_r=omp_get_wtime();
    cout<<"Integral_reduction: "<< I_R<<"\n"<<"Execution time: "<<(end_r-start_r)/5<<endl;
    cout<< "\n";
  }
  return 0;

}

double f(double x)
{
  return (1/pow(x,2))*pow(sin(1/x),2);
}

double integral(double a , double b)
{
  double step=(b-a)/n ;
  double I= (f(a)+f(b))/2;
  double x;
  for (int i=1; i<n; i++ )
  {
    I=I+f(a+step*i);
  }
  I=I*step;
  return I;
}


double integral_critical(double a, double b)
{
  double step= (b-a)/n;
  double I=0;
  double J=0;

  #pragma omp parallel num_threads(thread_count)
  {
    double I=0;
  #pragma omp for
    for (int i=1; i<n; i++)
        I+=f(a+step*i);
  #pragma omp critical
    {
        J+=I;
    }
  }
  J=(J+0.5*(f(a)+f(b)))*step;
  return J;
}

double integral_atomic(double a, double b)
{
  double step= (b-a)/n;
  double I=0;
  double J=0;

  #pragma omp parallel num_threads(thread_count)
  {
    double I=0;
```

```
  #pragma omp for
    for (int i=1; i<n; i++)
        I+=f(a+step*i);
  #pragma omp atomic
        J+=I;

  }
  J=(J+0.5*(f(a)+f(b)))*step;
  return J;
}

double integral_lock(double a, double b)
{
  double step= (b-a)/n;
  double I=0;
  double J=0;
  omp_lock_t lock;
  omp_init_lock(&lock);

  #pragma omp parallel num_threads(thread_count)
  {
  #pragma omp for
    for (int i=1; i<n; i++)
    {
        omp_set_lock(&lock);
        I+=f(a+step*i);
        omp_unset_lock(&lock);
    }
  }
  omp_destroy_lock(&lock);

  J=(I+0.5*(f(a)+f(b)))*step;
  return J;
}

double integral_reduction(double a, double b)
{
  double step= (b-a)/n;
  double I=0;
  double J=0;
  #pragma omp parallel num_threads(thread_count)
  {
    #pragma omp for reduction(+: I)
    for (int i=1; i<n; i++)
    {
      I=I+f(a+step*i);
    }
  }
  J=(I+0.5*(f(a)+f(b)))*step;
  return J;
}
```