A tutorial of Solving the defined integral problem with OpenMp

André Furlan - UNESP - Universidade Estadual Paulista "Júlio de Mesquita Filho"

***Abstract*—The Trapezoid rule algorithm used in this tutorial is not the most fast method to calculate an interval defined integral but is a good example of how to make programs runs quicker by using OpenMp for parallelism. Hopefully it will help the understanding of how this library works and how to use it. All sources can be accessed at https://github.com/ensismoebius/ComputacaoDeAltoDesempenho**

***Index Terms*—OpenMp, Paralelism, C++, C, Tutorial**

I. Introduction

In High Performance Computation (HPC) the goal (other than solve a problem) is to use most of the computation power of the machines involved. Most of the time this means use all processors an its cores at maximum, avoiding idle intervals.

By using the OpenMp [2] libraries it is possible, using **#pragma** compiler directives, turn a sequential program into a parallel one. The problem in focus here is the defined interval integral calculation. To solve it the program must calculate the area bellow the plot by summing up the area of several trapezoids [1]. The more trapezoids fitted, the more precise are the values obtained. And it is here that the parallelism enters: By dividing the work of area calculation in lots of threads its possible to diminish the effective time of execution.

II. The trapezoid rule

Be a a point that marks the start of a interval in which the calculation must be done and b its end, than an **approximation** of the integral value is given by equation 1. In figure 1 the basic idea is illustrated.

$$\int_b^a f(x)dx \approx (b-a).\frac{1}{2}(f(a)+f(b)) \tag{1}$$

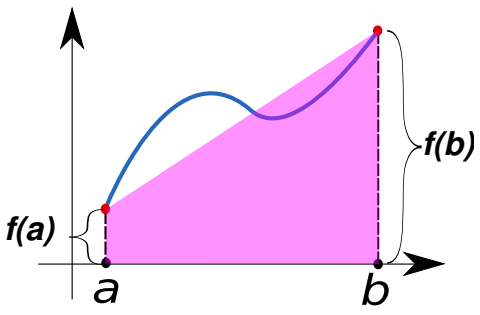

Fig. 1. In pink the approximation(trapezoid area), in blue the actual function curve. $f(a)$ and $f(b)$ are the trapezoid bases and the interval a, b its height.

The figure 1 and the equation 1 shows a pretty coarse approximation! What if the $[a, b]$ interval get subdivided (partitioned) like in figure 2? Well, then it is possible to get a value closer to the real one!

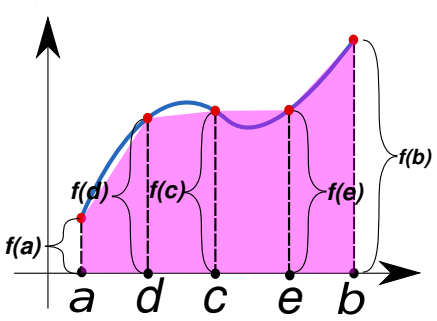

Fig. 2. In pink the **enhanced** approximation(trapezoid area), in blue the actual function curve. $f(a), f(b), f(c), f(d), f(e)$ are the trapezoid bases and the intervals $[a, d], [d, c], [c, e], [e, b]$ its heights. Note that now there are 4 trapezoids i.e. 4 areas that must be calculated.

To obtain the result all areas must be summed up, therefore the equation 1 turns into 2. So, as many partitions are made, the more sums are added.

$$\begin{aligned}\int_b^a f(x)dx \approx &(d-a).\frac{1}{2}(f(d)+f(a))+\\ &(c-d).\frac{1}{2}(f(c)+f(d))+\\ &(e-c).\frac{1}{2}(f(e)+f(c))+\\ &(b-e).\frac{1}{2}(f(b)+f(e))\end{aligned} \tag{2}$$

Let $x = [a, b]$ and $n = partitions + 1 = points$ of $[a, b]$ than it is true to say that $x_0 = a < x_1 < ... < x_{n-1} < x_n = b$. Being the interval between two points $\Delta_{x_k} = x_k - x_{k-1}$ than the equation 2 can be rewrote as in the equation 3.

$$\int_b^a f(x)dx \approx \sum_{k=1}^{n} \frac{f(x_{k-1})+f(x_k)}{2}.\Delta_{x_k} \tag{3}$$

And this is (finally) the idea that will be implemented ahead!

III. Implementation

First of all it is necessary to define a function to integrate, the one used here is defined in equation 4 and the integration

interval begins in **0 until 100**. The performance will be tested using **1, 2, 4 and 8 threads**. Note that this method supports any function, this one was chosen for example purposes only.

$$f(x) = \sqrt{100^2 - x^2} \tag{4}$$

Then, for the impatient ones, here is the complete code that will be discussed:

```
1  #include <cmath>
2  #include <iostream>
3  #include <ostream>
4
5  #ifdef _OPENMP
6    #include <omp.h>
7  #endif
8
9  inline long double f(long double x)
10 {
11     return std::sqrt(std::pow(100, 2) - std::pow(x,
       2));
12 }
13
14 inline long double integrate(long double (*f)(long
       double),
15                              double lowerLimit,
16                              double upperLimit,
17                              int partitions)
18 {
19     double h = (upperLimit - lowerLimit) /
       partitions;
20     long double sum = 0;
21
22 #pragma default(none) shared(soma)
23     {
24 #pragma omp parallel
25         {
26 #pragma omp for reduction(+ : sum)
27             for (int k = 1; k < partitions - 1; k++)
        {
28                 sum += f(lowerLimit + k * h);
29             }
30         }
31 #pragma omp barrier
32
33         sum *= 2;
34         sum += (f(lowerLimit) + f(upperLimit));
35         return (h / 2) * sum;
36     }
37 }
38
39 int main()
40 {
41     double lowerLimit = 0, upperLimit = 100;
42
43     int partitions = 0;
44
45     partitions = upperLimit / 0.000001;
46     std::cout << "Interval: " << 0.000001 << "
       Partitions: " << partitions
47               << " - Result: " << integrate(f,
       lowerLimit, upperLimit, partitions) << std::endl
       ;
48
49     partitions = upperLimit / 0.00001;
50     std::cout << "Interval: " << 0.00001 << "
       Partitions: " << partitions
51               << " - Result: " << integrate(f,
       lowerLimit, upperLimit, partitions) << std::endl
       ;
52
53     partitions = upperLimit / 0.0001;
54     std::cout << "Interval: " << 0.0001 << "
       Partitions: " << partitions
55               << " - Result: " << integrate(f,
       lowerLimit, upperLimit, partitions) << std::endl
       ;
56
57     return 0;
58 }
```

IV. Discussion

This section focus only in the OpenMp directives, its assumed that the reader knows a minimum of C++.

The lines 5 to 7 includes OpenMp header files if the build system used detected such library.

```
1 #ifdef _OPENMP
2   #include <omp.h>
3 #endif
4
```

From line 9 to 12, the function to be integrated is defined. Note the *inline* statement: Inlinning functions **may** avoid the intrinsic overhead of calling a function but, since this it is not a command, the compiler may ignore this request.

```
1 inline long double f(long double x)
2 {
3   return std::sqrt(std::pow(100,2) - std::pow(x,2));
4 }
5
```

Line 22 states that the variable *sum* is shared by all threads all the other variables are not.

```
1 #pragma default(none) shared(soma)
2
```

At the 24 position OpenMp starts a parallel zone: From now on the instructions are replicated in several threads according to parallelization rules defined in the **operational system ambient variables** [3]. The zones are defined using curling braces (same as blocks in C++).

```
1 #pragma omp parallel
2
```

The most important part is declared here in line 26. This directive is the heart of the parallelization for this algorithm. For each thread *sum* is initialized to 0 then, after all calculations, the obtained values are summed together. For this directive it have to be possible to known beforehand the number of iterations in execution time i.e. the number of iterations cannot change while the loop are being executed. Also the k variable (used in for loop) must **not** be modified within the loop otherwise it can be changed like in k++ statement. Since, in this specific case, a summation is need OpenMp have a convenient *reduction* directive that enables it. The *reduction* directive can be used with other types of operations like *subtractions*, *maximums* and *minimums*.

```
1 #pragma omp for reduction(+ : sum)
2
```

Finally, at the line 31, a barrier to the execution of the program is created in order to ensure that all threads has been finished until this point. Now it is possible to do another operations with the outputted sum. This is optional in this **specific case** because there is an implicit barrier in *for* directive. It is important to note that **there is no code** associated with this directive i.e. it just marks a point where all threads must arrive. **Warning!** All or none of the threads must encounter the barrier otherwise a **deadlock** happens.

```
1 #pragma omp barrier
2
```

Among the aforementioned **operational system ambient variables** one is the most important for the sake of this program: OMP_NUM_THREADS. This variable sets the number of threads (an integer) that are going to be used when running the algorithm. If no value is assigned than the maximum of threads supported are used.

V. Performance tests

The tests were done using an *AMD Ryzen 5 5000 series* processor running an *Gnu/Linux operational system* with *30GB of RAM*.

The tables bellow shows the results according to the number of threads.

TABLE I
EXPORT OMP_NUM_THREADS=1

Interval Δ_{x_k}	Partitions	Time taken in sec	Result
0.000001	100000000	0.670965	7853.981634
0.000010	10000000	0.067101	7853.981633
0.000100	1000000	0.006715	7853.981617

TABLE II
EXPORT OMP_NUM_THREADS=2

Interval Δ_{x_k}	Partitions	Time taken in sec	Result
0.000001	100000000	0.341421	7853.981634
0.000010	10000000	0.033588	7853.981633
0.000100	1000000	0.003378	7853.981617

TABLE III
EXPORT OMP_NUM_THREADS=4

Interval Δ_{x_k}	Partitions	Time taken in sec	Result
0.000001	100000000	0.172951	7853.981634
0.000010	10000000	0.016810	7853.981633
0.000100	1000000	0.001684	7853.981617

VI. Conclusion

This is just a scratch on the surface of OpenMp please pay a visit to their website to learn more: https://www.openmp.org/.

That is all folks!

TABLE IV
EXPORT OMP_NUM_THREADS=8

Interval Δ_{x_k}	Partitions	Time taken in sec	Result
0.000001	100000000	0.144975	7853.981634
0.000010	10000000	0.014428	7853.981633
0.000100	1000000	0.001436	7853.981617

References

[1] Francis Begnaud Hildebrand. *Introduction to numerical analysis*. Courier Corporation, 1987.

[2] OpenMP Architecture Review Board. OpenMP application program interface version 5.2, May 2021.

[3] OpenMP Architecture Review Board. OpenMP environment variables, May 2023.