A tutorial of Solving the defined integral problem with OpenMp and Open MPI

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 or Open MPI for parallelism. Hopefully it will help the understanding of how these libraries works and how to use them. All sources can be accessed at https://github.com/ensismoebius/ComputacaoDeAltoDesempenho**

***Index Terms*—OpenMp, Open MPI, 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 [3] library it is possible by using **#pragma** compiler directives, turn a sequential program into a parallel one.

When using the Open MPI library, [2] it is necessary to use the "MPI_" prefixed commands. The "MPI_" prefix is a convention used in Open MPI implementations, but it is not required by the MPI standard.

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.

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

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!

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.

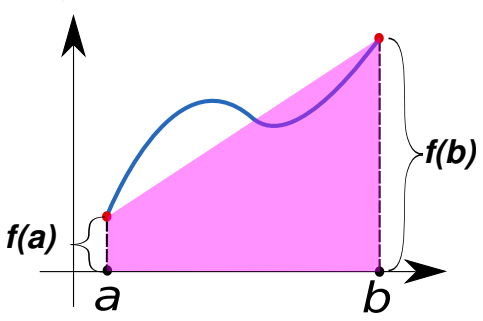

Fig. 1. In pink the approximation(trapezoid area), in blue the actual function curve. $f(a)$ and $f(b)$ are the trapezoid bases and the interval a, b its height.

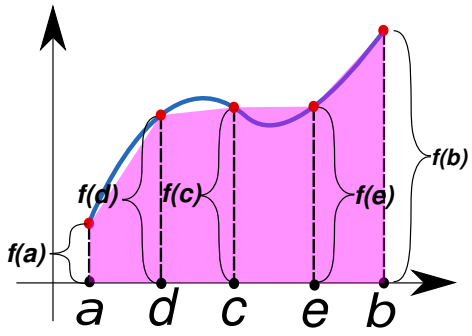

Fig. 2. In pink the **enhanced** approximation(trapezoid area), in blue the actual function curve. $f(a), f(b), f(c), f(d), f(e)$ are the trapezoid bases and the intervals $[a,d], [d,c], [c,e], [e,b]$ its heights. Note that now there are 4 trapezoids i.e. 4 areas that must be calculated.

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

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 the equation 3.

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

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.

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

A. OpenMp implementation

```cpp
#include <cmath>
#include <iostream>
#include <ostream>

#ifdef _OPENMP
  #include <omp.h>
#endif

inline long double f(long double x)
{
    return std::sqrt(std::pow(100, 2) - std::pow(x, 2));
}

inline long double integrate(long double (*f)(long double),
                             double lowerLimit,
                             double upperLimit,
                             int partitions)
{
    double h = (upperLimit - lowerLimit) / partitions;
    long double sum = 0;

#pragma default(none) shared(soma)
    {
#pragma omp parallel
        {
#pragma omp for reduction(+ : sum)
            for (int k = 1; k < partitions - 1; k++) {
                sum += f(lowerLimit + k * h);
            }
        }
#pragma omp barrier

        sum *= 2;
        sum += (f(lowerLimit) + f(upperLimit));
        return (h / 2) * sum;
    }
}

int main()
{
    double lowerLimit = 0, upperLimit = 100;

    int partitions = 0;

    partitions = upperLimit / 0.000001;
    std::cout << "Interval: " << 0.000001 << "
    Partitions: " << partitions
              << " - Result: " << integrate(f,
    lowerLimit, upperLimit, partitions) << std::endl
    ;

    partitions = upperLimit / 0.00001;
    std::cout << "Interval: " << 0.00001 << "
    Partitions: " << partitions
              << " - Result: " << integrate(f,
    lowerLimit, upperLimit, partitions) << std::endl
    ;

    partitions = upperLimit / 0.0001;
    std::cout << "Interval: " << 0.0001 << "
    Partitions: " << partitions
              << " - Result: " << integrate(f,
    lowerLimit, upperLimit, partitions) << std::endl
    ;

    return 0;
}
```

B. Open MPI implementation

```cpp
#include <cmath>
#include <iomanip>
#include <iostream>
#include <mpi.h>

inline long double f(long double x)
{
  return std::sqrt(std::pow(100, 2) - std::pow(x, 2));
}

inline long double integrate(long double (*f)(long double),
double lowerLimit,
double upperLimit,
int partitions)
{
  double h = (upperLimit - lowerLimit) / partitions;
  long double sum = 0;

  int slotId, slotsAvailable;
  MPI_Comm_rank(MPI_COMM_WORLD, &slotId);
  MPI_Comm_size(MPI_COMM_WORLD, &slotsAvailable);

  int localIterations = (partitions - 2) /
    slotsAvailable;
  int remainingIterations = (partitions - 2) %
    slotsAvailable;

  int start = slotId * localIterations + std::min(
    slotId, remainingIterations) + 1;
  int end = start + localIterations + (slotId <
    remainingIterations ? 1 : 0);

  long double localSum = 0;

  for (int k = start; k < end; k++) {
    localSum += f(lowerLimit + k * h);
  }

  MPI_Reduce(&localSum, &sum, 1, MPI_LONG_DOUBLE,
    MPI_SUM, 0, MPI_COMM_WORLD);

  if (slotId == 0) {
    sum *= 2;
    sum += (f(lowerLimit) + f(upperLimit));
    sum *= (h / 2);
  }

  return sum;
}
```

```
45
46 inline void experiment(double lowerLimit, double
       upperLimit, double interval)
47 {
48   long double result = 0;
49   int partitions = upperLimit / interval;
50
51   double start_time = MPI_Wtime();
52
53   result = integrate(f, lowerLimit, upperLimit,
       partitions);
54
55   double end_time = MPI_Wtime();
56   double elapsed_time = end_time - start_time;
57
58   int rank;
59   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
60
61   if (rank == 0) {
62     std::cout << std::fixed << std::setprecision(6)
       << interval << " \t " << partitions
63     << " \t " << elapsed_time << " \t " << result <<
        std::endl;
64   }
65 }
66
67 int main(int argc, char *argv[])
68 {
69   MPI_Init(&argc, &argv);
70
71   double lowerLimit = 0, upperLimit = 100;
72
73   int slotId;
74   MPI_Comm_rank(MPI_COMM_WORLD, &slotId);
75
76   if (slotId == 0) {
77     std::cout << "Interval \t Partitions \t Time
       taken \t Result" << std::endl;
78   }
79
80   experiment(lowerLimit, upperLimit, 0.000001);
81   experiment(lowerLimit, upperLimit, 0.00001);
82   experiment(lowerLimit, upperLimit, 0.0001);
83
84   MPI_Finalize();
85   return 0;
86 }
```

IV. Discussion

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

A. OpenMp discussion

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** [4]. 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.

B. Open MPI discussion

On line 4 there is a **required** include for the Open MPI library.

```
1 #include <mpi.h>
2
```

From line 6 to 9, 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.

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

```

On the nice 69 line there is a command that initializes the MPI environment, including communication channels and resources necessary for running parallel Open MPI programs. The "MPI_Init" command should be called exactly once per MPI process and has the same arguments as the C++ main function i.e. the program arguments count and the arguments.

```
MPI_Init(&argc, &argv);

```

At line 73 and 74 there is a slot identificantion retrieval. A slot is a resource to make computation (usually an CPU core) locally or remotely available. This slot information will be used in the next lines.

```
int slotId;
MPI_Comm_rank(MPI_COMM_WORLD, &slotId);

```

On line 76 to 78 the intended message is only shown if the process is from slot 0 i.e. the main process.

```
 if (slotId == 0) {
  std::cout << "Interval \t Partitions \t Time taken
     \t Result" << std::endl;
}

```

The line 51 begins the time measurement.

```
double start_time = MPI_Wtime();

```

Line 55 and 56 measure time again and calculates the elapsed time respectively.

```
double end_time = MPI_Wtime();
double elapsed_time = end_time - start_time;

```

As already explained from lines 58 to 64 it guarantees that the elapsed times outputs only runs on main slot.

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  std::cout << std::fixed << std::setprecision(6) <<
     interval << " \t " << partitions
  << " \t " << elapsed_time << " \t " << result <<
    std::endl;
}

```

Line 19 creates the slots variables (*slotId* as the current slot/process identification and *slotsAvailable* as the number of available slots). Lines 20 and 21 retrieves these values.

```
int slotId, slotsAvailable;
MPI_Comm_rank(MPI_COMM_WORLD, &slotId);
MPI_Comm_size(MPI_COMM_WORLD, &slotsAvailable);

```

The line 35 aggregates all sums from *localSum* variables across the slots in the *sum* variable.

```
MPI_Reduce(&localSum, &sum, 1, MPI_LONG_DOUBLE,
    MPI_SUM, 0, MPI_COMM_WORLD);

```

Finally at 84 line all resources allocated are freed.

```
MPI_Finalize();

```

Note that most of the processing is done in lines 31 to 33, and the problem **must** be partitioned manually in Open MPI, which is different from OpenMP.

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*.

A. OpenMp performance tests

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

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

VI. Performance tests

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

A. Open MPI performance tests

In order to change the number of parallel processing units in an Open MPI process, first you need to create a text file that contains the "hosts" and their respective available slots that will be used by Open MPI. The hosts can be specified using their DNS name or IP address. In this example a file named "hosts.txt" was created.

```
localhost slots=8
```

But it could be like the one bellow:

```
localhost slots=4
192.168.0.56 slots=10
192.168.0.101 slots=7
```

When the aforementioned file is created, you can start the processing using the command bellow. Note the use of the "-np" parameter, which represents the number of slots to be used and can be interpreted as the number of threads in Open MPI. If the specified "-np" value is greater than the number of available slots, an error will be raised.

```
mpirun --hostfile hosts.txt -np 1 ./executable
```

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

TABLE V
MPIRUN –HOSTFILE HOSTS.TXT -NP 1

Interval	Partitions	Time taken	Result
0.000001	100000000	0.470636	7853.981634
0.000010	10000000	0.044703	7853.981633
0.000100	1000000	0.004453	7853.981617

TABLE VI
MPIRUN –HOSTFILE HOSTS.TXT -NP 2

Interval	Partitions	Time taken	Result
0.000001	100000000	0.240130	7853.981634
0.000010	10000000	0.022352	7853.981633
0.000100	1000000	0.002229	7853.981617

TABLE VII
MPIRUN –HOSTFILE HOSTS.TXT -NP 4

Interval	Partitions	Time taken	Result
0.000001	100000000	0.136689	7853.981634
0.000010	10000000	0.011185	7853.981633
0.000100	1000000	0.001120	7853.981617

TABLE VIII
LMPIRUN –HOSTFILE HOSTS.TXT -NP 8

Interval	Partitions	Time taken	Result
0.000001	100000000	0.116957	7853.981634
0.000010	10000000	0.009895	7853.981633
0.000100	1000000	0.000946	7853.981617

VII. Conclusion

The demonstrated example **does not imply** the superiority of OpenMP over Open MPI or the other way around because, depending on the problem to be solved and the environment involved, one or the other may be more suitable for the task.

OpenMP is used when there is a need to utilize most of the local computer's resources for processing. OpenMP programs are simpler to code as well, thanks to the use of **#pragma** compiler directives.

On the other hand, Open MPI is better suited when the problem needs to be divided into larger pieces and processed across multiple machines. However, it requires the programmer to split the problem, resulting in a more complex and error-prone code.

Hope it helps and, for more information, please consult the references.

That is all folks!

References

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

[2] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

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

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