Tezan Sahu
170100035
tezansahu@iitb.ac.in

Homework 1
ME766: High Performance Scientific Computing

Due Date: 01/03/21

Background

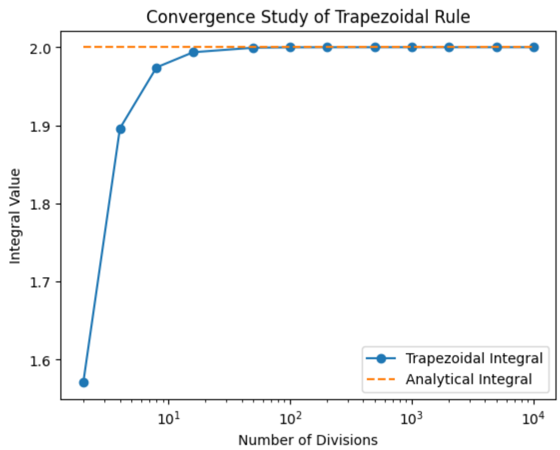
- **Function to be Integrated:** $f(x) = cos(x)$
- **Interval for Performing Integration:** $[-\frac{\pi}{2}, \frac{\pi}{2}]$
- **Analytical Integral of Function over Interval:** $\int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} cos(x) = sin(x) \mid_{-\frac{\pi}{2}}^{\frac{\pi}{2}} = 2$
- **Numerical Integration Methods:**
  1. Trapezoidal Rule [implemented in `trapezoidal_serial.cpp` & `trapezoidal_parallel.cpp`]
  2. Montecarlo Method [implemented in `montecarlo_serial.cpp` & `montecarlo_parallel.cpp`]

Trapezoidal Rule

Convergence Study

The following table & plot demonstrates the convergence study performed for the Trapezoidal Rule:

No. of Divisions	Numerical Integral	Error (%)
2	1.5708	21.46
4	1.89612	5.194
8	1.97423	1.2885
16	1.99357	0.3215
50	1.99934	0.033
100	1.99984	0.008
200	1.99996	0.002
500	1.99999	0.0005
1000	2	0
2000	2	0
5000	2	0
10000	2	0



It can be seen clearly that with the increase in number of divisions used for computing the integral, the value of the numerical integral slowly approaches the true integral value found using the analytical solution (i.e. 2). We also notice that all the values are ≤ 2 (due to the nature of the numerical method).

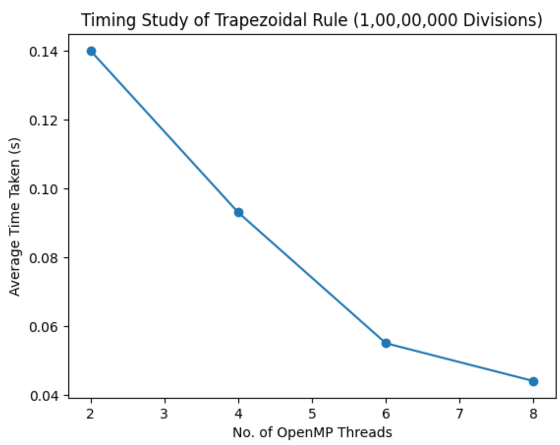
Timing Study

We perform the timing study for the Trapezoidal Rule implemented parallely using OpenMP threads by considering 10000000 divisions (to clearly understand the difference in times taken). An average of 5 different runs is taken while using a certain number of threads. The results can be found in the table & plot:

No. of Threads	2	4	6	8
T_1 (s)	0.137	0.069	0.073	0.043
T_2 (s)	0.129	0.071	0.05	0.043
T_3 (s)	0.129	0.071	0.05	0.049
T_4 (s)	0.144	0.071	0.052	0.045
T_5 (s)	0.159	0.183	0.051	0.041
Average Time (s)	0.140	0.093	0.055	0.044

The average time taken by serial code for 10000000 divisions is **0.213 s**.

Here, we note that although the speedup is not truly linear, yet the use of more than one OpenMP threads has made the execution considerably faster. It is fastest with the use of **8 threads**.

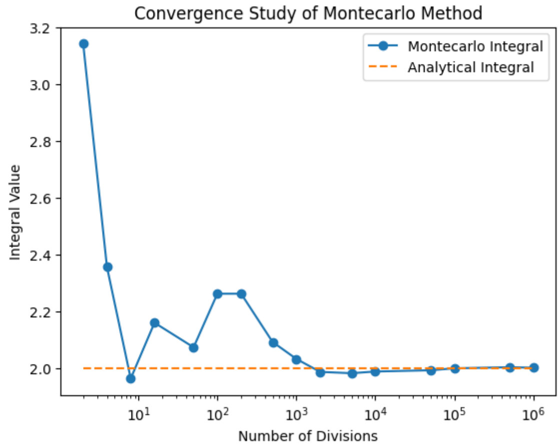


Montecarlo Method

Convergence Study

The following table & plot demonstrates the convergence study performed for the Montecarlo Method:

No. of Divisions	Numerical Integral	Error (%)
2	3.14159	-57.0795
4	2.35619	-17.8095
8	1.9635	1.825
16	2.15984	-7.992
50	2.07345	-3.6725
100	2.26195	-13.0975
200	2.26195	-13.0975
500	2.0923	-4.615
1000	2.03261	-1.6305
2000	1.98706	0.647
5000	1.98234	0.883
10000	1.98831	0.5845
50000	1.99271	0.3645
100000	1.99959	0.0205
500000	2.00344	-0.172
1000000	2.00167	-0.0835

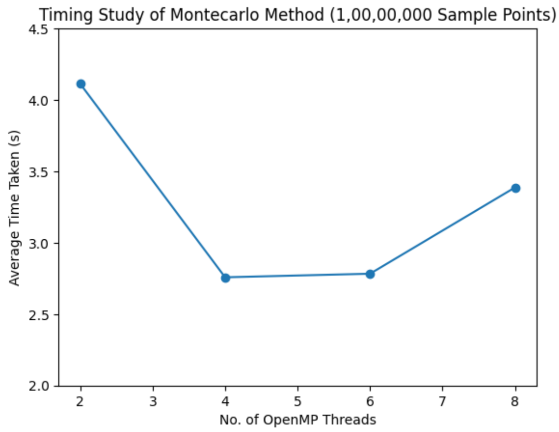


We can notice that in spite of the initial fluctuations due to the stochastic nature of this method, as we increase the number of sampling points, the value of the numerical integral converges to the actual value of the integral found using the analytical solution (i.e., 2).

Timing Study

We perform the timing study for the Montecarlo Method implemented parallely using OpenMP threads by considering 10000000 sampling points (to clearly understand the difference in times taken). An average of 5 different runs is taken while using a certain number of threads. The results can be found in the table & plot:

No. of Threads	2	4	6	8
T_1 (s)	4.194	2.831	2.847	3.267
T_2 (s)	4.409	2.625	2.658	3.482
T_3 (s)	3.516	2.788	2.790	3.533
T_4 (s)	4.288	2.936	2.824	3.366
T_5 (s)	4.157	2.629	2.798	3.258
Average Time (s)	4.113	2.758	2.783	3.387



It can be seen than the increase in number of threads has not typically offered the speedup that one would have expected. In fact, the fastest execution takes place with **4 threads**, while using 6 or 8 threads actually slows down the execution.

On comparing the times taken by the OpenMP parallel version with the time taken by the serial version of the Montecarlo method (**0.387 s**), it is evident that contrary to our expectations, the parallel code takes much longer to execute. This could be potentially due to the use of the `rand()` function, which needs to maintain the state information that is shared across all threads.