CS301: HIGH PERFORMANCE COMPUTING
Assignment 6: MONTE CARLO

Submitted By:
Shaleen Kumar Gupta (201301429)
Visharad Bansal (201301438)

PROBLEM STATEMENT

Calculate the value of pi by the Monte Carlo method.

METHOD

If a circle of radius R is inscribed inside a square with side length 2R, then the area of the circle will be pi*R^2 and the area of the square will be (2R)^2. So the ratio of the area of the circle to the area of the square will be pi/4.

This means that, if you pick N points at random inside the square, approximately N*pi/4 of those points should fall inside the circle.

This program picks points at random inside the square. It then checks to see if the point is inside the circle (it knows it's inside the circle if x^2 + y^2 < R^2, where x and y are the coordinates of the point and R is the radius of the circle). The program keeps track of how many points it's picked so far (N) and how many of those points fell inside the circle (M).

Pi is then approximated as follows:

$$\Pi = \frac{4 * M}{N}$$

COMPLEXITY

The serial code runs in O(n) time, where n is the number of iterations.

OpenMP is used to divide the serial code among p threads, giving a theoretical speedup of p times

MPI is used to run the same problem on a distributed memory architecture, running on p processors.

OPTIMIZATION STRATEGY

A number of processes/threads are used to divide the task of allocating random numbers.
In OpenMp, reduction is used in order to count the number of points inside the circle.
In MPI, MPI_Reduce is used for the same purpose.

HARDWARE DETAILS

Architecture	: Genuine Intel x86_64
Cores	: 16
CPU Model Name	: Intel(R) Xeon(R) CPU E5-2640 v2
CPU MHz	: 1200.000
cache size	: 20480 KB

INPUT PARAMETERS

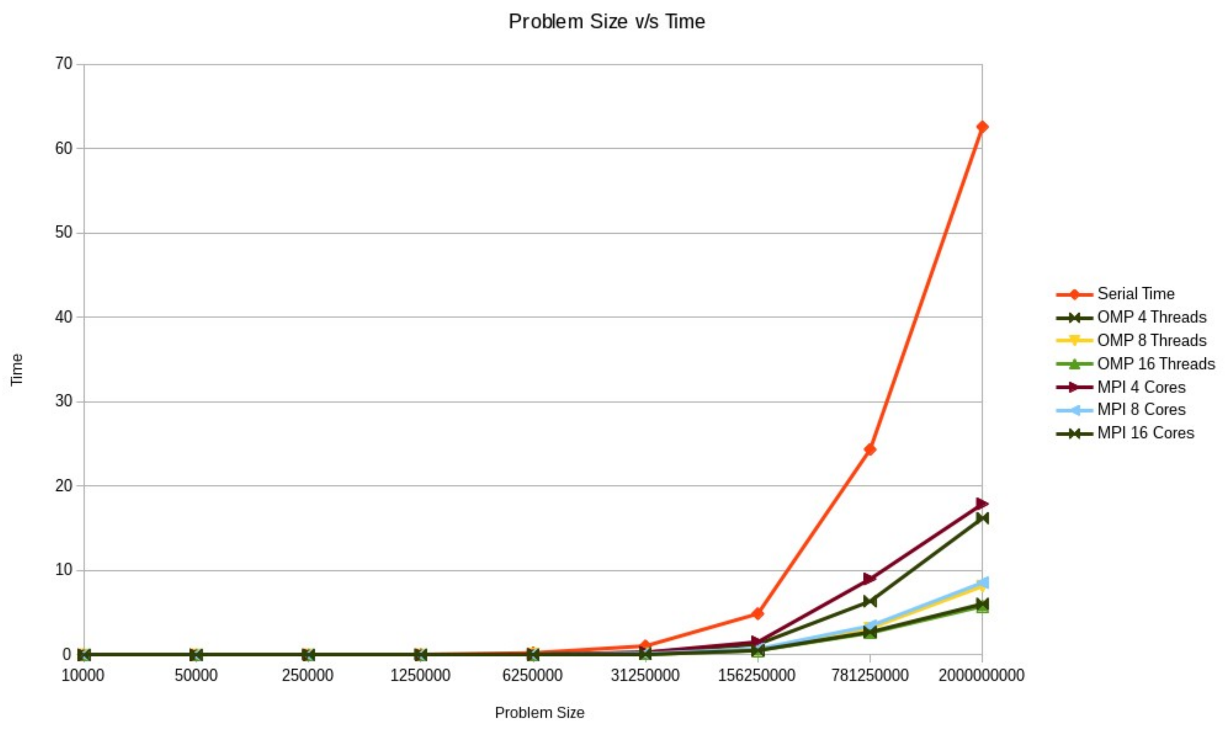
Number of steps (n).
n was taken from 10^4 to $2*10^9$

Also, the number of threads is taken as input for the OMP version of the Monte Carlo Program.
We have run the OMP code for 4, 8 and 16 threads.
Similarly, the MPI version was executed over 4, 8 and 16 processes.

PROBLEMS FACED

Using rand function in OpenMp refused to give a speedup as the function is not thread safe. To counter that, the drand_48() is used to assign random numbers as it is thread safe.

OBSERVATIONS



Comparison of serial and parallel times

As can be seen from the graph above, the time taken by the serial code is massive when compared to the time taken by the parallel code, where the time taken has significantly got down.

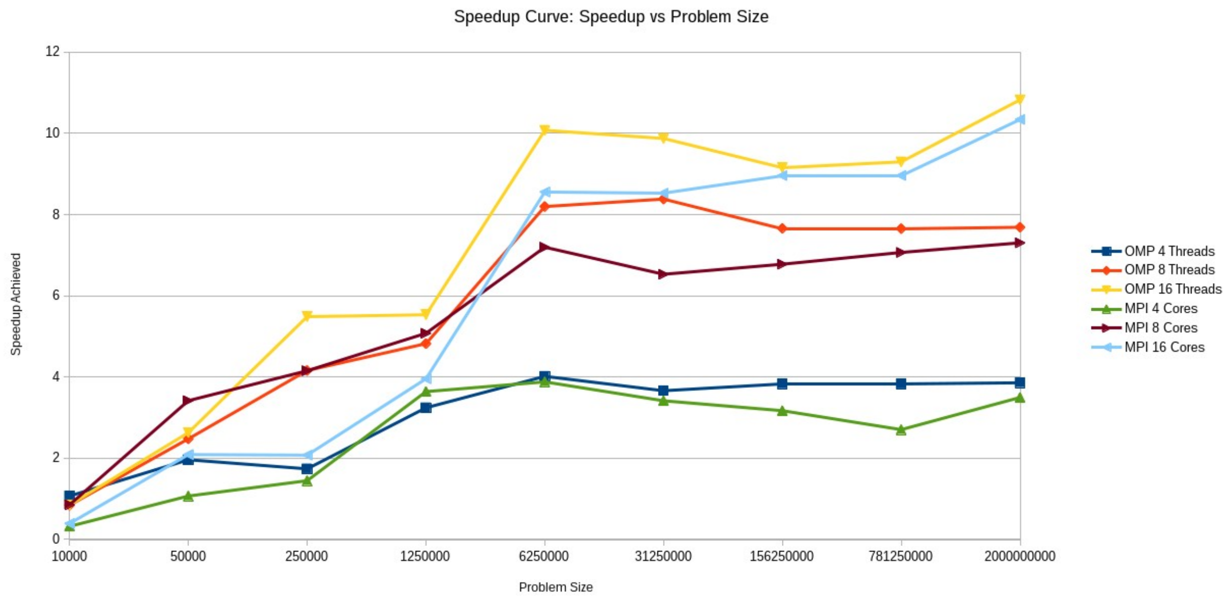
The reasons are clear. This problem falls under the category of Embarrassingly Parallel Problems, and hence, can be massively parallelized, which has been done. A finite number of observations need to be taken to calculate the value of pi. Since each observation is independent of any other observation, it can be done parallely by another thread or process. The same has been implemented, which thus, explains the reduction in the time taken.

Time was calculated using omp_get_wtime() library function include in the "omp.h" header file for OpenMp.

Time in MPI was recorded using MPI_.Wtime(), with the time being measured on the master process.

SPEED UP

Speed Up was calculated as:
Speed Up= (Time taken by serial code) / (Time taken by parallel code)



Speed Up v/s N

We observe from the speedup curve that the maximum speedup achieved within the limit of our range was almost equal to the number of threads or processes spawned in order to compute the solution of the problem. This is indeed a reflection of the fact that this being an Embarrassingly Parallel Problem, can be massively parallelized. The same has been implemented and thus, the speedup.

However, we do observe some kind of abnormalities in the speedup curve obtained for 16 threads on OpenMP and 16 processes in MPI. Possible reasons for this could be the increased amount of communication involved due to more than sufficient number of threads and processes. This must have increased the communication overhead, which might have then dominated the computation time.

Also, we concluded that random is a very expensive function, especially when it is to be used by a number of threads. In order to cater to this problem, we had to figure out a different set of methods, drand48_r and srand48_r, to be used instead of the srand function in C.