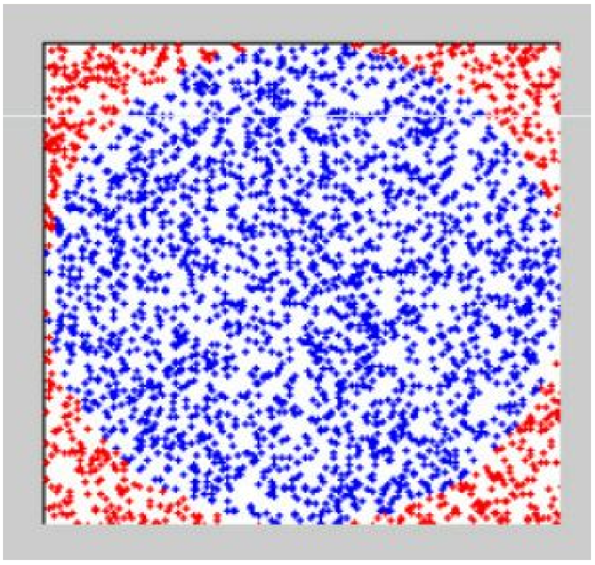
HPC Assignment 6

Calculating value of PI using Monte Carlo Method

Submitted by:
Saumya Bhadani 201301100
Shruti Singh 201301452

1. Problem statement : Calculate value of pi

Monte-Carlo method is a numerical method for statistical simulation which uses sequences of random numbers to perform the simulation.
To calculate value of pi, we generate a sequence of random numbers, that fall inside the square of edge length 2. We take the ratio of number of points that fall within the circle of radius 1 and that within the square.



$$\frac{\text{Number of points inside the circle}}{\text{Number of points inside the square}} = \frac{\text{Area of circle}}{\text{Area of square}} = \frac{pi*R*R}{2R*2R} = pi/4$$

- **Complexity :**
  Complexity of the serial algorithm is O(n), where n = number of divisions, the interval [0, 1] is divided into(i.e. problem size).
  Complexity of the parallel algorithm is O(n/p), where p = number of cores on the machine.

- **Possible speedup(theoretical) :**
  Speedup S = 1/ (P/n + s)
  n – number of cores
  P – percentage of code that can be parallelized
  s – percentage of serial code(which is not parallelized)
  For our code, P ~ 1 and s ~0
  n=4, So theoretical speedup = 4

- **Pseudo Code:**

```
n=0  (number of points inside the circle)
N (Total number of points)
Generate two sequences: Ri, Rj between [0, 1]
Xi= -1 + 2Ri
Yj= -1 + 2Rj
if( Xi^2 + Yj^2)<1
        n=n+1
pi = (4*n) / N
```

2. Hardware details:

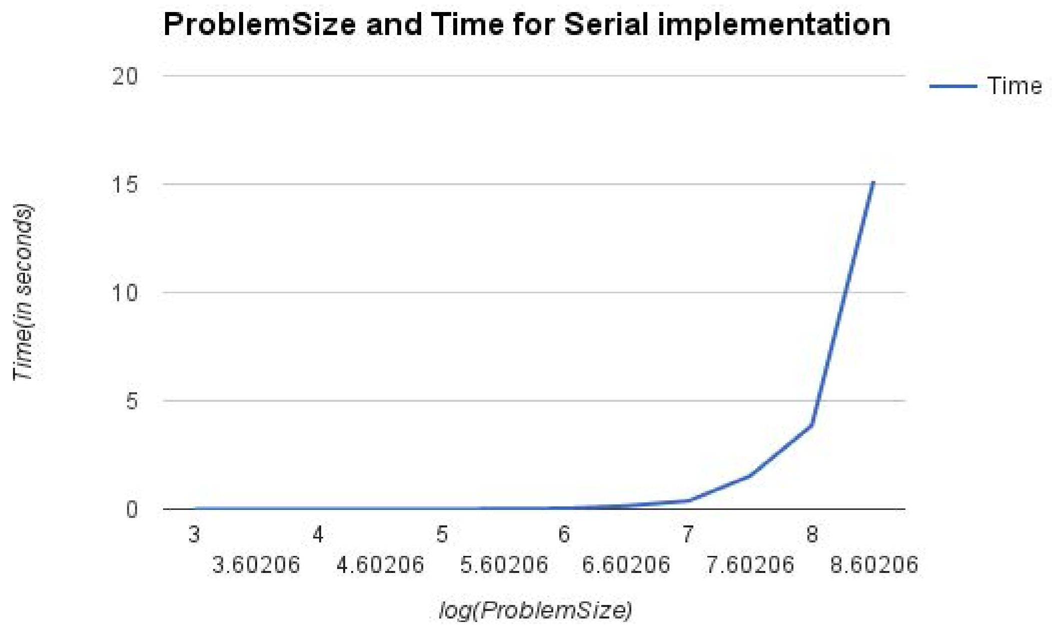
CPU : Intel® Core™ i5-4200U CPU @ 1.60GHz × 4
Compiler : gcc
Precision : Double
Peak performance = 4 FLOPs/cycle * 1.6GHz * 4 = 25.6 GFLOPS

3. Optimization Strategy:

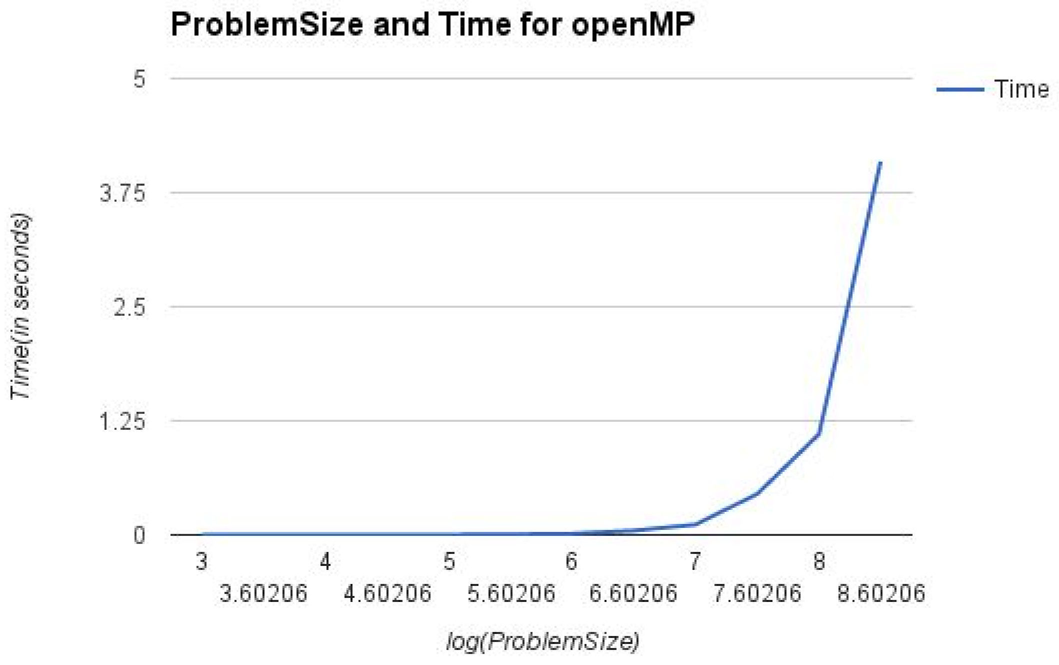
Initially we were using srand() and rand() to generate random numbers. For every different seed value used in a call to srand, the pseudo-random number generator can be expected to generate a different succession of results in the subsequent calls to rand. However, this took a lot of time and the we obtained a poor speedup using OpenMP. So instead of using srand() and rand(), we defined a random generator function, where we assign values using a global uninitialized variable which holds some garbage value.

3. Output:

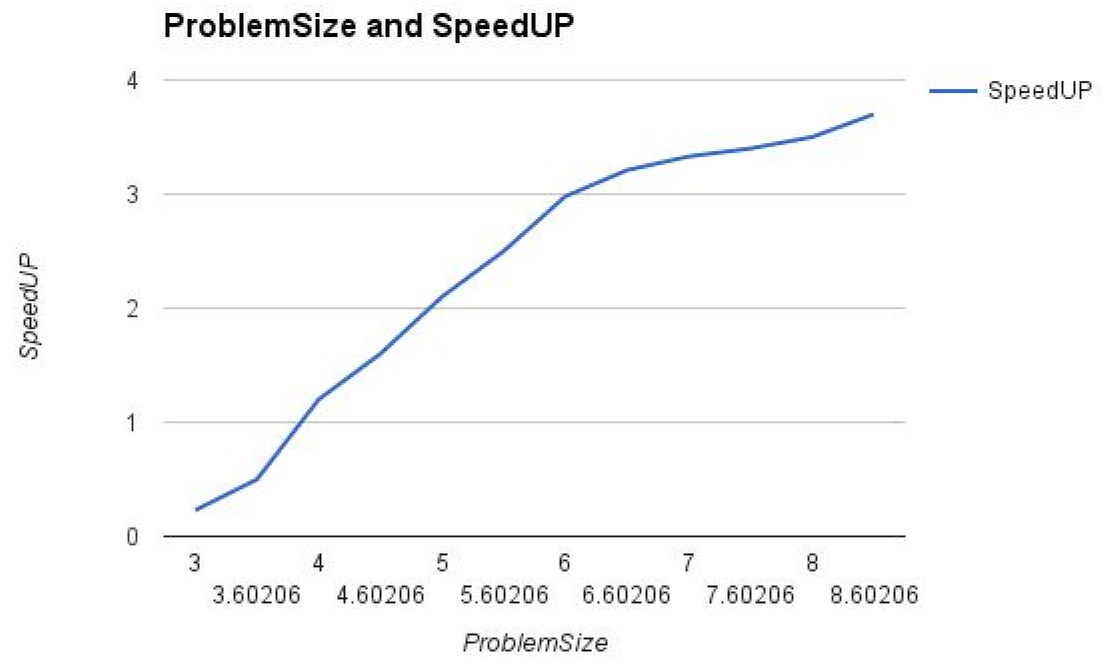
ProblemSize VS Time for Serial implementation



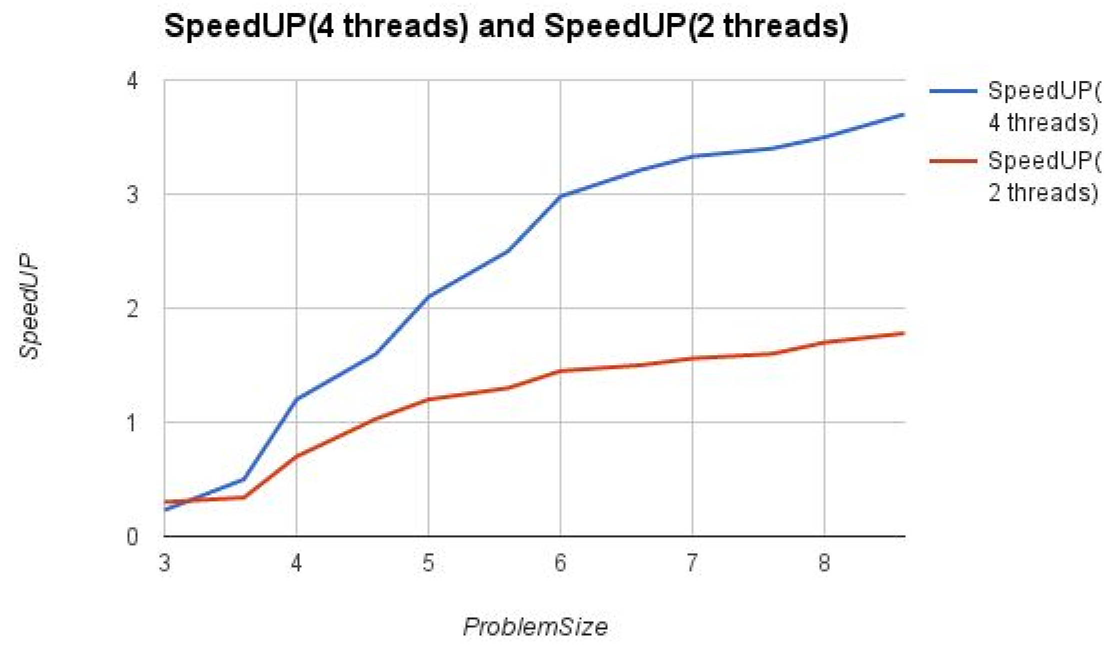
ProblemSize VS time for OpenMP for 4 threads



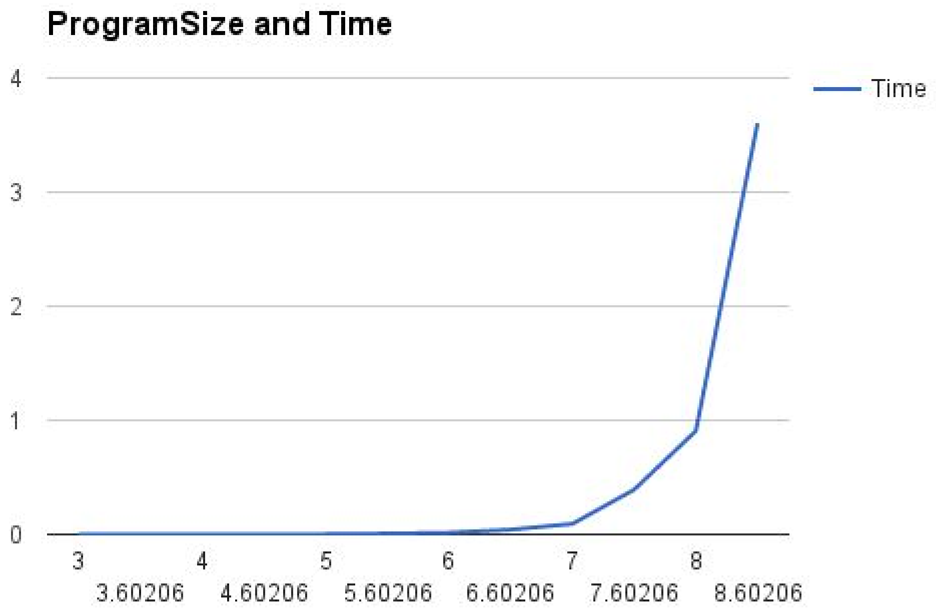
SpeedUP for openMP



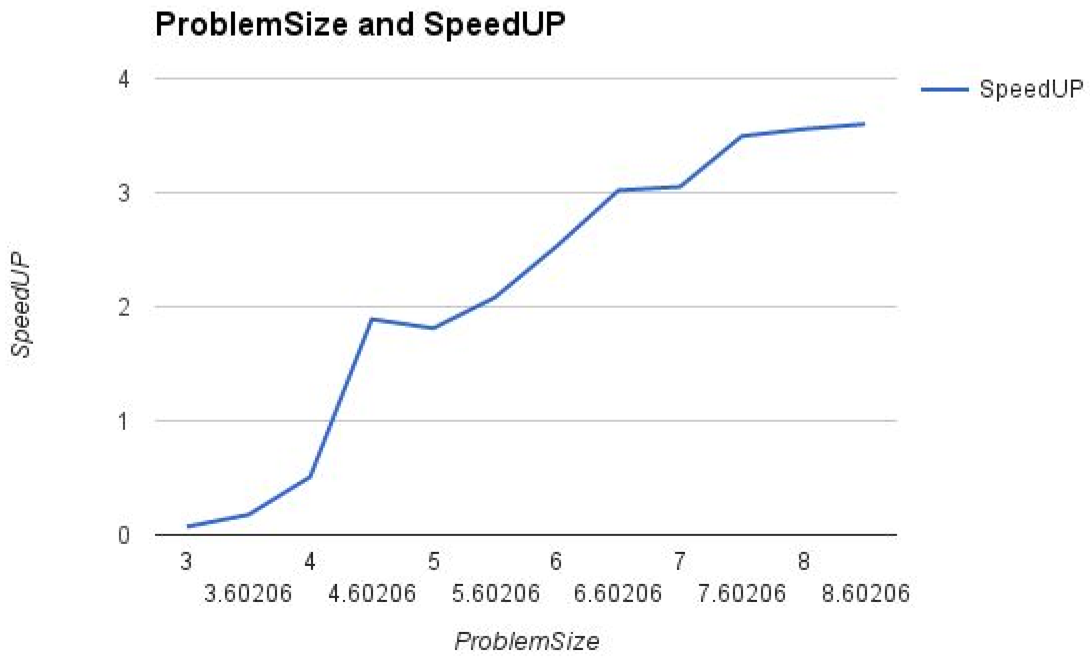
Problem size VS speedUP for 2 and 4 threads



ProblemSize VS Time for MPI implementation



ProblemSize Vs SpeedUP for MPI implementation



4. Observations and Conclusions:

For the serial code, the log(ProblemSize) VS Time is a log graph, which is expected as the complexity of algorithm is O(n).
This is a computation intensive, embarrassingly parallel problem, as it involves only generating two random numbers and checking if they lie within the circle. The speedup obtained using OpenMP is high, which is around 3.6 for 4 cores.

For OpenMP, better results were achieved using reduction than Naive Parallel approach. Because of memory hierarchy, it is possible that in naive parallel implementation, two or more threads try to write into variables (i.e. sum[i]) which belong to the same cache line. So in this situation, one write instruction hinders other write instructions. This might result in Reduction being faster than Naive parallel implementation.

Speedup obtained for MPI is less than OpenMP. In OpenMP, we used reduction to calculate the sum of all points inside the circle, as calculated by each thread. In MPI this is achieved using MPI_Send() and MPI_Recieve() which might be slow, however the speedups are comparable, i.e. 3.6 and 3.7 for 4 processes.