High Performance Computing
Assignment 6

Jahnavi Suthar(201301414) Deeksha Koul(201301435)

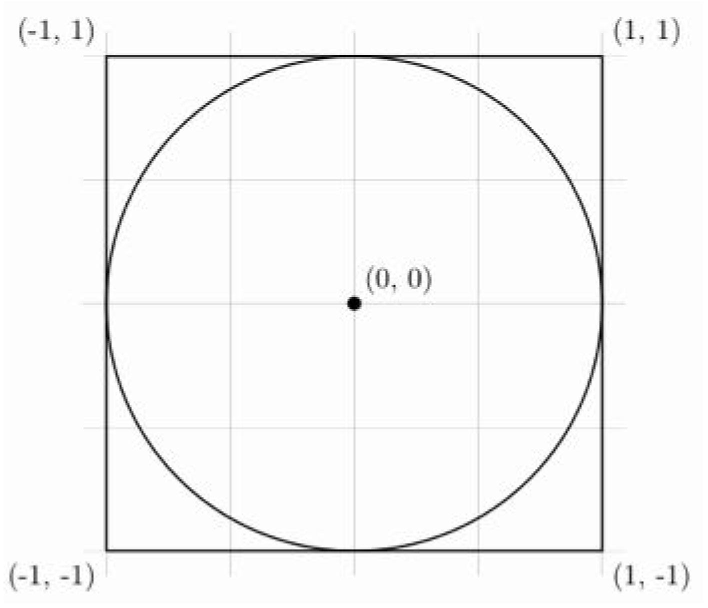
Description of the Problem

A simple Monte Carlo estimate for the value of π can be found by generating random points on a square and counting the proportion that lie inside an inscribed unit circle. The probability of a point landing in the circle is proportional to the relative areas of the circle and square.

$$Probability\ of\ points\ in\ the\ circle\ =\ \frac{Area\ of\ Circle}{Area\ of\ the\ Square}$$

$$\frac{c}{n} = \frac{\pi}{4}$$

Where n is the number of random points generated and c is number of points which fall in circle.



Complexity of the Serial Algorithm

Complexity of generating random numbers and checking if they fall in the circle or not is O(n), which is the complexity of the serial algorithm.

Possible Speedup(Theoretical)

We distribute number of points to be generated(n) into p threads. So the possible maximum speedup will be O($\frac{n}{p}$).

Parallelizing Strategy

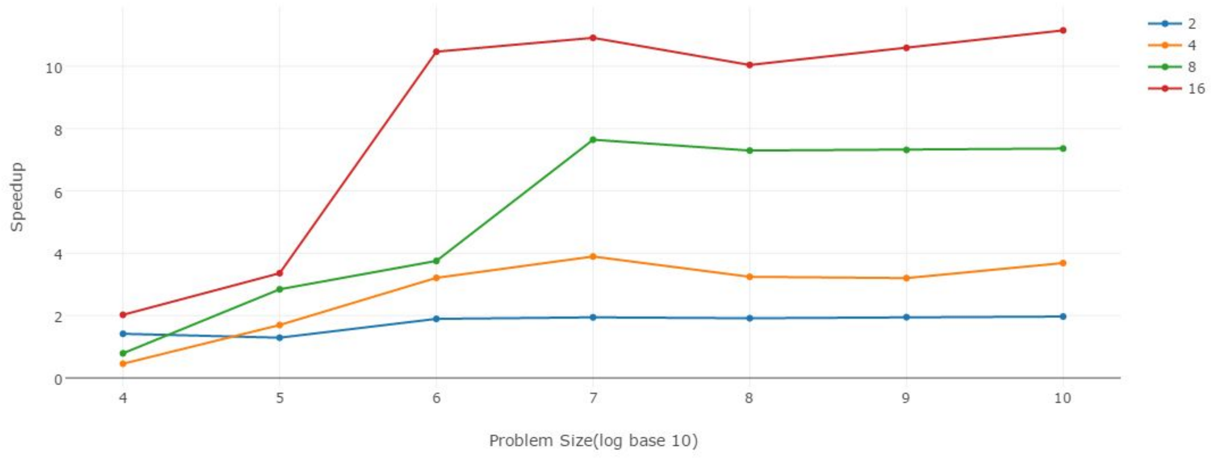
The program can be easily parallelized because there are no dependencies in the algorithm. All the threads generate the random numbers separately. At the end of the program we add number of points that fall in the circle for all the threads and calculate value of pi.

Hardware Details

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             2
Vendor ID:             GenuineIntel
CPU MHz:               1200.000
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K
```

Results and Observations

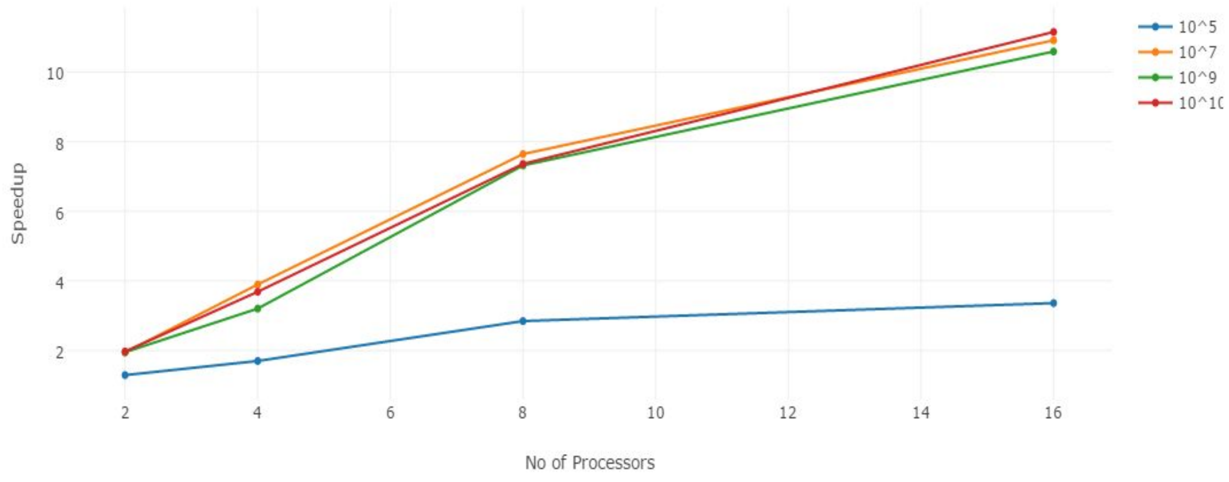
Using MPI



Problem Size vs Speedup for 2, 4, 8 and 16 processors using MPI

Observation:

For the small problem sizes, the speedup is less due to overhead for initializing of environment. But when we increase the problem size, the speedup becomes almost constant as the overhead is done by the parallelized computations.



No of Processors vs Speedup for Problem Sizes 10^5, 10^7, 10^9 and 10^{10} using MPI

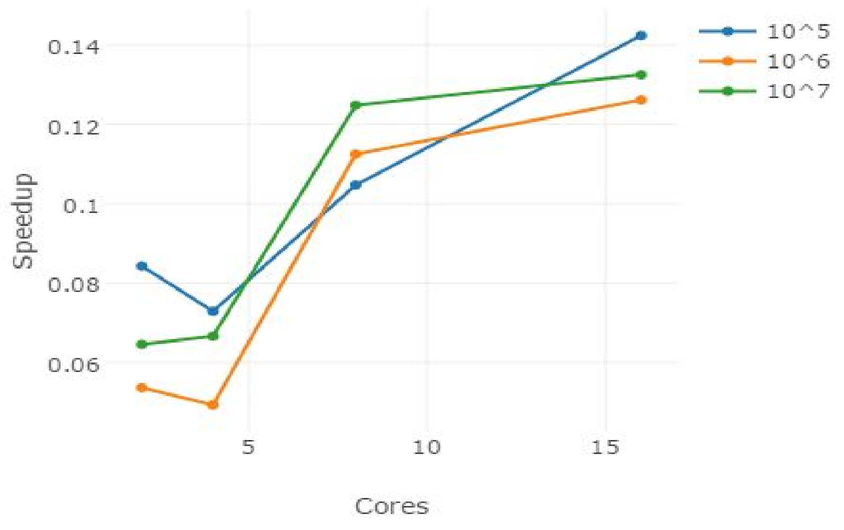
Observation:
As we increase the number of processors, the speedup increase. The slope of the graph decreases as we go to higher number of processors showing slower increase in speedup due to communication overhead. For the larger problem size we get more speedup for each number of processors.

Using OpenMP

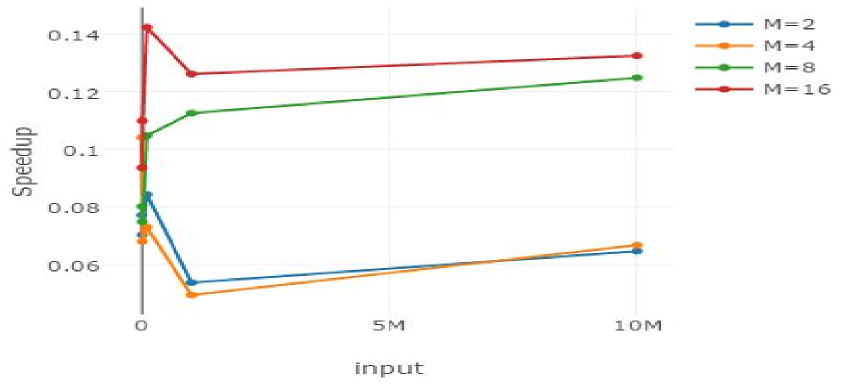
We generated random numbers in OpenMP using two methods:

a. Using built in rand() function in stdlib
b. Using our own random generator

Version 1



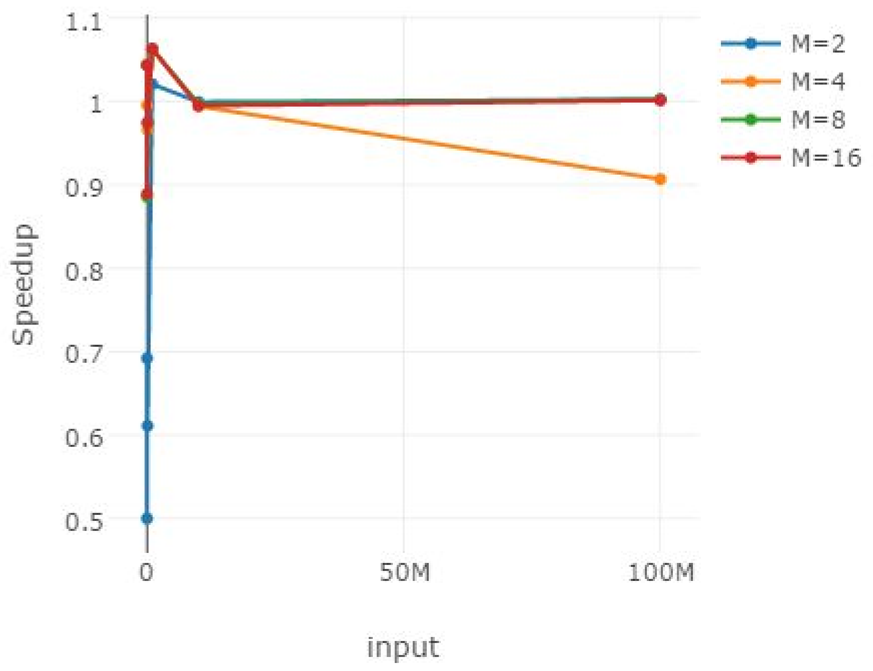
No of Threads vs Speedup



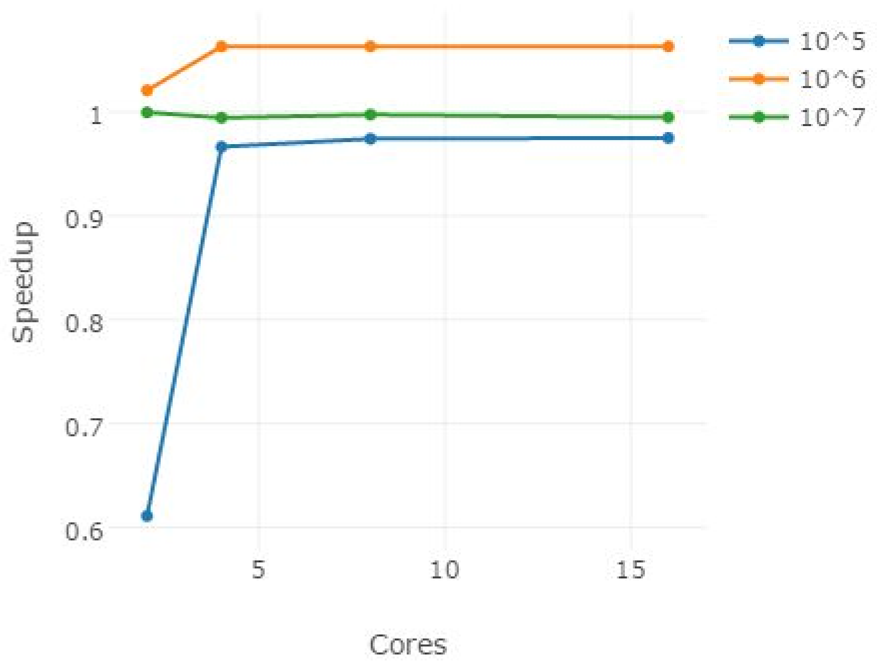
Problem Size vs Speedup

Observation
We do not get any speedup, when we use stdlib function rand(). Because the function rand() is not thread safe, i.e. there can be racing conditions on several global variables that it contains. This results in the poor performance as we start using multiple threads.

Version 2



Observation:
As the input increases the speedup per core increases but after the input 10^4 the speedup remains nearly constant . Overall for input >=10^4 , the speedup obtained for all the cores remained nearly same and didn’t increase much.



Observation:
As the core increases the speedup remains nearly constant regardless of the input.Even when the input is increased for a particular core,the efficiency doesn't increases much and speedup is nearly same.

KARP-FLATT METRIC(experimentally determined serial fraction)

	VERSION 1				VERSION 2			
	M=2	M=4	M=8	M=16	M=2	M=4	M=8	M=16
10^6	36.27	26.68	10.01	8.38	0.95	0.92	0.93	0.93
10^7	29.96	19.66	9.01	7.98	1.00	1.00	1.00	1.00

Problems faced in Parallelization :

a. In version 2,the 'e' nearly remains same that is the parallel overhead doesn't affect the efficiency much.
b. In version 1,we used built in random number generator, because the function is not thread safe, performance decreases.
c. In MPI, synchronization and system overhead(the time spent when copying the message data from sender's message buffer to network and from network to the

receiver’s message buffer) plays an important role but still we are able to get some speedup. This is because in MPI, we create processes and not threads. Each process uses its own variables for generating random numbers, and hence there is no overhead due to racing conditions on the shared variables, as in the case of OpenMP.