ASSIGNMENT

Performance Evaluation for Estimation of value of pi:

Methodology:

Monte Carlo estimation:

Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. One of the basic examples of getting started with the Monte Carlo algorithm is the estimation of Pi.

Estimation of Pi:

The idea is to simulate random (x, y) points in a 2-D plane with domain as a square of side 1 unit. Imagine a circle inside the same domain with same diameter and inscribed into the square. We then calculate the ratio of number points that lied inside the circle and total number of generated points. Refer to the image below:

We know that area of the square is 1 unit sq while that of circle is . $\prod * \left(\frac{1}{2}\right)^2 = \frac{\prod}{4}$ Now

for a very large number of generated points,

$$\frac{\text{area of the circle}}{\text{area of the square}} = \frac{\text{no. of points generated inside the circle}}{\text{total no. of points generated or no. of points generated inside the square}}$$

that is,

$$\pi = 4 * \frac{\text{no. of points generated inside the circle}}{\text{no. of points generated inside the square}}$$

The beauty of this algorithm is that we don't need any graphics or simulation to display the generated points. We simply generate random (x, y) pairs and then check if $x^2 + y^2 \leq 1$. If yes, we increment the number of points that appears inside the circle. In randomized and simulation algorithms like Monte Carlo, the more the number of iterations, the more accurate the result is. Thus, the title is "**Estimating** the value of Pi" and not "Calculating the value of Pi.

We have done two openmp programs and one in mpi and there scalability is analyzed.

CPU specs: CPU Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz

Base speed:	2.50 GHz
Sockets:	1
Cores:	4
Logical processors:	4
Virtualisation:	Enabled
L1 cache:	256 KB
L2 cache:	1.0 MB
L3 cache:	6.0 MB

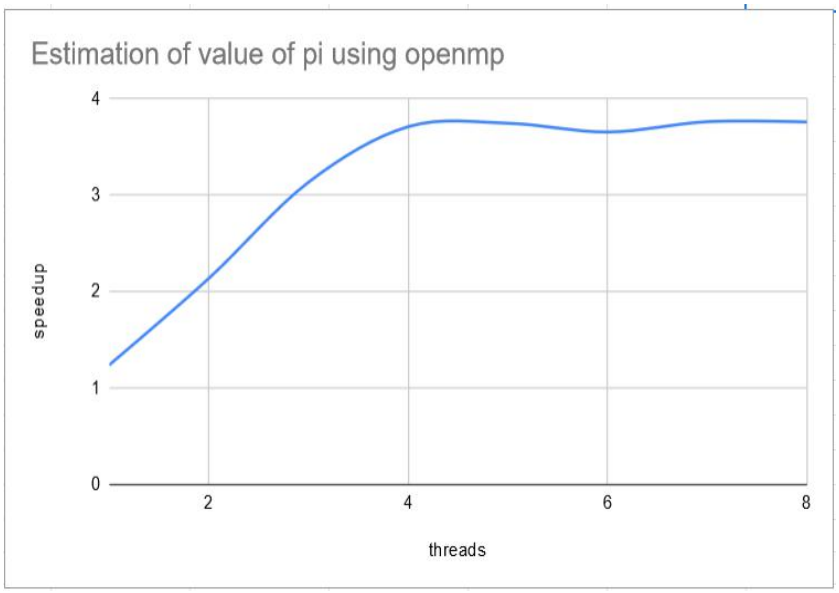
Method 1:

Code openmp:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>
#define MAX_THREADS 8

int main(int argc, char* argv[])
{
	int niter = 100000000;
	double x,y;
	int i,j;
	int count=0;
	double z,start,end,delta;
	double pi;
	for (j=1; j<= MAX_THREADS; j++) {
		printf(" running on %d threads: ", j);
		omp_set_num_threads(j);
		start = omp_get_wtime();

		#pragma omp parallel for reduction(+:count) private(x,y,z)
		for (i = 0; i < niter; ++i)
		{
			x = (double)random()/RAND_MAX;
			y = (double)random()/RAND_MAX;
			z = sqrt((x*x)+(y*y));

			if (z<=1)
			{
			   ++count;
			}
		}
		pi = ((double)count/(double)(niter*j))*4.0;
		delta = omp_get_wtime() - start;
		printf("PI = %.16g computed in %.4g seconds\n", pi, delta);
	}
	return 0;
}
```

Now the analyses of the above code is shown below.

threads	speedup
1	1.241702
2	2.140536
3	3.137217
4	3.711937
5	3.746661
6	3.655844
7	3.763088
8	3.759948



Observation:

In above code we have done same thing which is explained earlier here the count denotes the number of point inside the circle or area of circle similarly no of iteration denotes area of square or number of points inside the square.

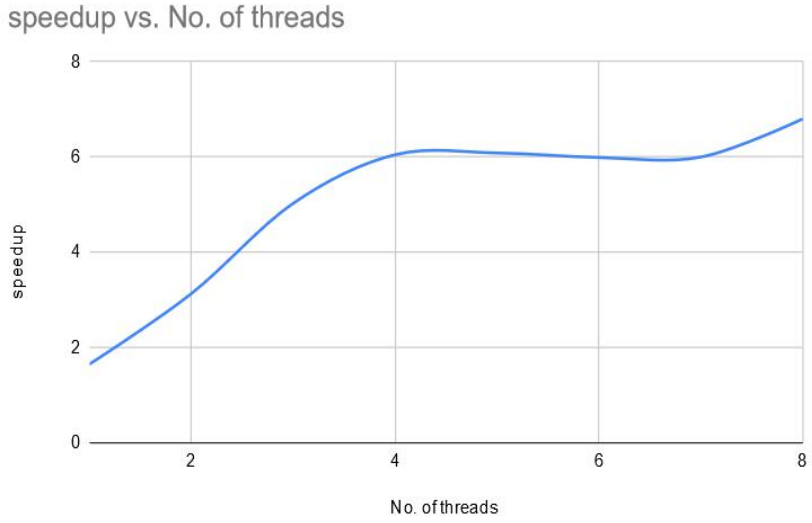
We can observe from the graph that for large number of iteration when we increase threads the speedup is increasing. But after number of threads equals to 4 then speedup almost becomes constant it is mainly because we have four core CPU .

Method 2:

Code Openmp:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_THREADS 8

static long steps = 1000000000;
double step;

int main (int argc, const char *argv[]) {

	int i,j;
	double x;
	double pi, sum = 0.0;
	double start, delta;

	step = 1.0/(double) steps;

	for (j=1; j<= MAX_THREADS; j++) {

		printf(" running on %d threads: ", j);
		omp_set_num_threads(j);

		sum = 0.0;
		double start = omp_get_wtime();


		#pragma omp parallel for reduction(+:sum) private(x)
		for (i=0; i < steps; i++) {
			x = (i+0.5)*step;
			sum += 4.0 / (1.0+x*x);
		}

		pi = step * sum;
		delta = omp_get_wtime() - start;
		printf("PI = %.16g computed in %.4g seconds\n", pi, delta);

	}
	    return 0;
}
```

Analysis of the above code:



Number of threads	speedup
1	1.652757
2	3.134489
3	5.030931
4	6.049106
5	6.084845
6	5.989801
7	5.999641
8	6.802533

Observation:

From observing the above graph we can achieve superlinear speedup. In the above code we are using Openmp 'sfor reduction for sum in which each thread will have its own copy of sum and at the end each thread will copy their respective sum to global sum.

By using the openmp's for reduction speedup achieved is very good. But from the graph we can also observe that even though when we increase the number of threads beyond four speedup seems to be constant. So the high speedup achieved is not only because of increasing number of threads but also openmp does something inside in for reduction such that the communication overhead decreases between the threads and superlinear speedup is observed.

Method 3:

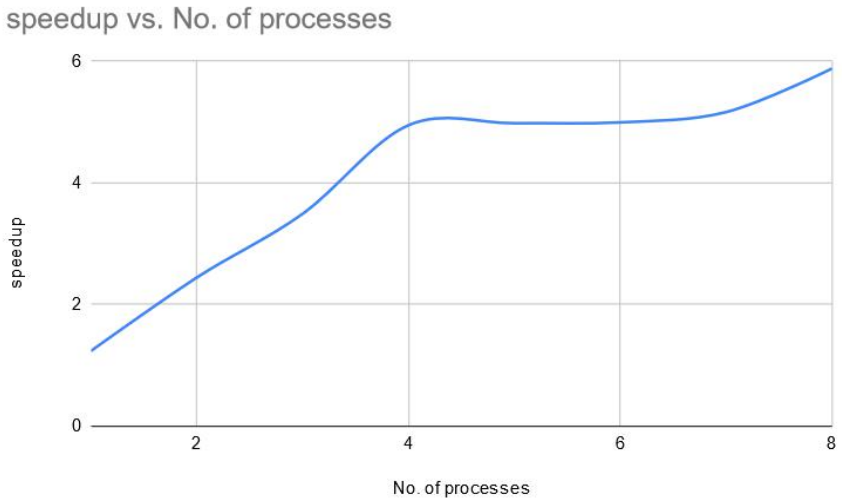
CPU specs:

Memory	5.8 GiB
Processor	Intel® Core™ i3-4130 CPU @ 3.40GHz × 4
Graphics	Intel® Haswell x86/MMX/SSE2

Code MPI:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

#define N 1E9
#define d 1E-9

int main (int argc, char* argv[])
{
    int rank, size, i, result=0, sum=0;
    double pi=0.0, begin=0.0, end=0.0, x, y;
    MPI_Init (&argc, &argv);
     MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Barrier(MPI_COMM_WORLD);
    begin = MPI_Wtime();
     srand((int)time(0));
    for (i=rank; i<N; i+=size)
    {
        x=rand()/(RAND_MAX+1.0);
        y=rand()/(RAND_MAX+1.0);
        if(x*x+y*y<1.0)
            result++;
    }
    MPI_Reduce(&result, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Wtime();

    if (rank==0)
    {
        pi=4*d*sum;
        printf("np=%2d; Time=%fs; PI=%0.4f\n", size, end-begin, pi);
    }
    MPI_Finalize();
return 0;
}
```

Analysis of above code:



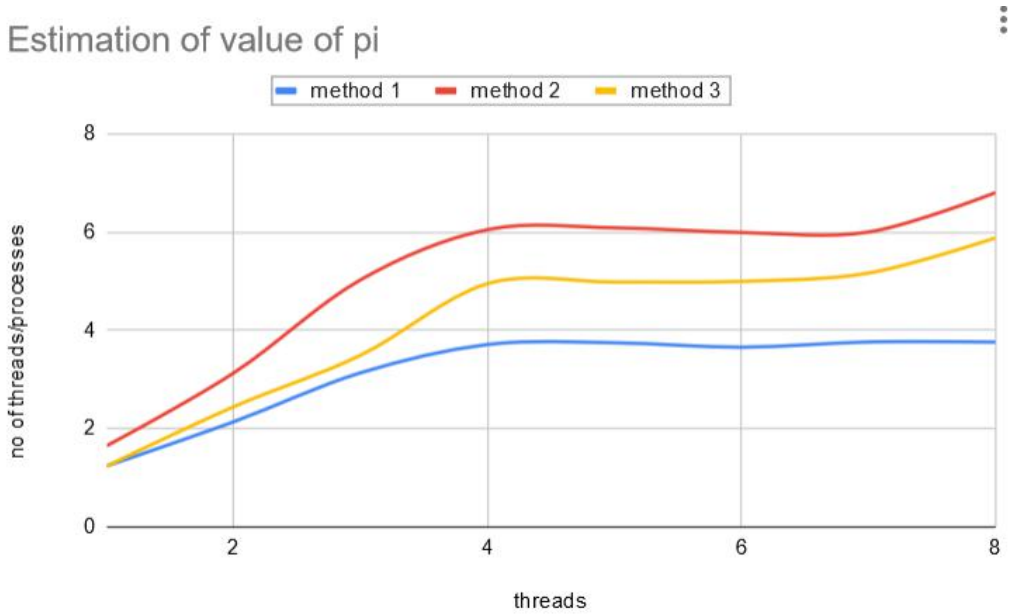
#processes	speedup
1	1.239458
2	2.446816
3	3.494744
4	4.950495
5	4.981482
6	4.995373
7	5.165281
8	5.882045

Observation:

In above code each process perform the iteration for counting number of points inside the circle and finally all count of number of points of each process is added using MPI_Reduce and finally the master process finds the value of pi.

So for analysis of the above code number of processes are varied and their speedup is calculated respectively.

Conclusion:



By observing all the methods mentioned above and analyzing them individually and together we can finally conclude that method 2 is best among all the three method mentioned. Even though Method 3 is using distributed memory Method 2 gives better scalability and it is using only one CPU/One node so it is cost efficient too to setup in comparison to Setup a cluster.