**Report: Exercises 1**

Sevde Yanik & Sima Esmaeili

**Problem 1**

As explained in the homework sheet, one of the methods to estimate the value of Pi (3.141592...) is using the Monte Carlo method. To do so, we can select random points in the unit square and keep a count of how many of them fall into the unit circle (that is if they meet the condition: x\*x = y\*y <= 1), let’s call these points circle\_points and the ones that fall into the unit square (equals to the total number of points generated) are called square\_points, Any point consists of two coordinates namely, x and y. The number of randomly generated points here is what really makes the difference.

If we have a square with 2r side length, the area of this square is 4r\*r. The circle that is in this square has the radius r, the area of the circle is Pi\*r\*r. The ratio of these two areas would be then Pi/4. For a large number of randomly generated points circle\_points/square\_points = Pi/4, which means Pi = 4\*(circle\_points/square\_points).

Now what are algorithm should do is:

1. Initialize circle\_points, square\_points, interval, i. (integers)
2. Generate random x. (floating number with value between 0 and 1)
3. Generate random y. (floating number with value between 0 and 1)
4. Calculate a value = x\*x = y\*y.
5. If value <= 1 increment the number of circle\_points.
6. Increment number of square\_points.
7. Increment interval.
8. If interval < i (number of iterations), repeat from step 2.
9. Calculate Pi = 4\*(circle\_points/square\_points).
10. Print out the estimated Pi.

**Problem 2**

Since the program we wrote is an implementation of the Monte Carlo method to estimate Pi using OpenMP, the calculation of each point is independent of others thus the work can be easily divided into separate threads and each thread can calculate a subset of the points. After using OpenMP to parallelize our program, our program can use multiple threads to perform the calculation in parallel, which reduces the overall running time.

1. The program uses omp\_set\_num\_threads() function to set the number of threads to be used.
2. #pragma omp parallel is written to indicate the beginning of the section to be run parallelized.
3. #pragma omp for distributes the loop iterations among threads.
4. The reduction function is used to calculate the sum of all circle\_points and square\_points among the threads.
5. Pi is calculated using the sum of all circle\_points and square\_points among the threads.
6. Print out Pi and running time.

*Speedup* = normal runtime / parallel runtime

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i | 50000 | 100000 | 200000 | 500000 | 800000 | 1000000 |
| Normal  runtime | 1 ms | 2 ms | 4 ms | 9 ms | 15 ms | 19 ms |
| Parallel runtime | 0,4 ms | 0,657 ms | 1,211 ms | 2,974 ms | 4,967 ms | 5,801 ms |
| Speedup | 2,5 | 3,0441 | 3,303 | 3,026 | 3,019 | 3,275 |

*Efficiency* = Speedup / No. of threads

We have set the number of threads to 4 in our program.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i | 50000 | 100000 | 200000 | 500000 | 800000 | 1000000 |
| Speedup | 2,5 | 3,0441 | 3,303 | 3,026 | 3,019 | 3,275 |
| Efficiency | 0,625 | 0,761 | 0,825 | 0,756 | 0,754 | 0,818 |

Since the efficiency mostly increases depending on the number of points generated, it can be said that the parallelization becomes more effective with larger workload. This may be because as the workload gets larger the overhead becomes less significant.

**Problem 3**

Again we implemented the Monte Carlo method to estimate the value of Pi using MPI (Message Passing Interface).

1. The MPI environment is initialized and the rank and size for the current process are retrieved.
2. The number of points to be generated are divided equally among the processes.
3. Each process generates a set of random points and counts the number of points in the unit circle and the unit square.
4. The process reduces the counts of the circle\_points and square\_points found in each process using the MPI reduce operation.
5. The root process calculates the final estimation of Pi = 4\*(circle\_points/square\_points)
6. Print out Pi and running time.

*Speedup* = normal runtime / MPI runtime

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i | 50000 | 100000 | 200000 | 500000 | 800000 | 1000000 |
| Normal  runtime | 1 ms | 2 ms | 4 ms | 9 ms | 15 ms | 19 ms |
| MPI runtime | 0,29 ms | 0,56 ms | 1,178 ms | 2,734 ms | 4,495 ms | 5,509 ms |
| Speedup | 3,44 | 3,57 | 3,39 | 3,29 | 3,33 | 3,44 |

*Efficiency* = Speedup / No. of MPI processes

We run our program with 4 MPI processes. (mpirun -np 4)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i | 50000 | 100000 | 200000 | 500000 | 800000 | 1000000 |
| Speedup | 3,44 | 3,57 | 3,39 | 3,29 | 3,33 | 3,44 |
| Efficiency | 0,86 | 0,89 | 0,84 | 0,82 | 0,83 | 0,86 |

The efficiency around 0,86 seems alright however since it doesn’t seem to be improving maybe there is some overhead.

**Problem 4**

The program implements binomial tree broadcasting in MPI.

1. The MPI environment is initialized and the rank and size for the current process are retrieved.
2. Two binomial tree broadcasts are performed,

* One in the normal order (each process send the data to its parent until the root is reached)
* One in reversed order (each process starts from the root and sends data to its child)

1. The running times for the processes are computed using the MPI\_Wtime() function.
2. The results are printed for each process.

Runtime becomes Ω(log2p) ???

**Problem 5**

In the implementation, the function called scalar\_product() takes two vectors of doubles and calculates their scalar product. The computation of the scalar product is parallelized using OpenMP. The result is calculated using the reduction operation that sums up the partial products computed by each thread. To test it out two vectors are initialized in main() and their scalar product is computed using the scalar\_product() function. The results are printed out.

Discuss the runtime complexity on the EREW PRAM model. How many processors can be used??

**Problem 6**

The implementation from problem 5 is extended so that it computes a matrix-vector product.

The function matrix\_vector\_product() takes in one matrix and one vector and calculates their product using OpenMP. The computation of the matrix-vector product is parallelized. The number of threads are set to 4.

Again, discuss the runtime complexity on the EREW PRAM and state the number of processors that are used??