EPOKA UNIVERSITY

CEN330 Parallel Programming

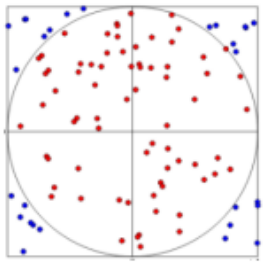
2021-2022

HOMEWORK 2

Deadline: 28.02.2022

PROBLEM 1 (25 pts)

In this problem you are required to write a parallel program which will estimate the value of the constant Π (pi) using a Monte Carlo method. The program will generate a large number of random points inside a square of dimensions 2x2 and then will count the number of these points which are located inside the circle (of radius 1) as shown in the figure. Then the area of the circle is estimated as:



$$Area_{circle} \approx \frac{N_0}{N} \cdot Area_{square}$$

Where N is the total number of generated random points and N_0 is the number of random points which are located inside the circle (i.e. the points which are depicted in red color in the given figure).

The area of the circle is $Area_{circle} = \pi r^2 = \pi \cdot 1^2 = \pi$, so when estimating the area of the circle, we are practically estimating the value of the constant Π (pi).

a. Write a program in OpenMP which reads as input the total number of random points (N) that will be generated and the number of threads that will be used, and will estimate the value of the constant Π (pi) using the Monte Carlo method described above.

b. Write a program in MPI which reads as input the total number of random points (N) that will be generated and will estimate the value of the constant Π (pi) using the Monte Carlo method described above (using the number of processed as specified in the mpiexec or mpirun command).

Solution:

Source code is provided in "Exercise 1 - calculatePi" folder. I wrote a program that will take as input the number of points to generate inside the {-1,-1,1,1} rectangle and how many thread/processes to use. In the OpenMP solution, I declare a variable insideCircle to keep track of how many points have an distance less or equal to 1 from the center. Using reduction in the pragma omp parallel for, the addition to the insideCircle is done in a synchronized way. However in the MPI solution, each process should have its own work calculated for it. It should be nrOfTries/size + (nrOfTries%size>rank). The second part takes care when nrOfTries%size!=0. In C, Boolean expressions return 1 and 0 so it will make processes work 1 more if their rank is lower than the remainder from nrOfTires/size. Then a reduce operation is done on each thread giving their individual insideCircle and having the result variable to print. Only rank 0, which stores the variable after reduce operation, prints the final result.

PROBLEM 2 (25 pts)

A famous Google interview question has been: "*Assume that you flip a fair coin until one of the following patterns of consecutive outcomes appears: HHT (head head tail) or HTH (head tail head). Find the probability that HHT pattern appears first and the probability that the HTT pattern appears first.*" This question may be solved using advanced probability theory and the precise answers will be 2/3 and 1/3.

Inspired by the above question, in this problem you are required to design a multi-threaded program named ProbabilityEstimation to estimate the probabilities that each among two given patterns (of H-s and T-s) appears first. The program will estimate the probabilities by repeating the random experiment many times and counting how many times each pattern appears first. So if the random experiment in the case of the given Google interview question is repeated 1000000 times and the pattern HHT appears first in 665000 of these experiments (and the pattern HTT appears first in 33500 of the experiments), then the estimated values of the probabilities are 0.665 (for HHT) and 0.335 (for HTH).

The program receives as command line parameters the two patterns (of H-s and T-s) and optionally the number of trials (i.e. times to repeat the random experiment) and the number of threads that will be used to solve the problem. If the optional arguments are not provided then the default number of trials is 1000000 and the default number of threads is 10.

A sample execution (if you are using Java) may be:

```
java ProbabilityEstimation HHT HTT 2000000 8
```

Or if you are using POSIX threads may be:

```
./ProbabilityEstimation HHT HTT 2000000 8
```

The expected result will look like:

```
Estimated values with 2000000 random experiments with 8 threads:
The probability that HHT appears first is 0.66671
The probability that HTT appears first is 0.33329
```

Another sample execution (if you are using Java) may be:

```
java ProbabilityEstimation THT HTH
```

Or if you are using POSIX threads may be:

```
./ProbabilityEstimation THT HTH
```

The expected result will look like:

```
Estimated values with 1000000 random experiments with 10 threads:
The probability that HTH appears first is 0.49998
The probability that THT appears first is 0.50002
```

a. Solve the given problem using Java multithreading.
b. Solve the given problem using POSIX threads.

NOTE: your solution may be tested with patterns of different lengths, for example HTT and TH, but it will not be tested with patterns where one is suffix of another for example HTT and TT, in order to avoid ambiguity (both patterns appearing in the same time).

Solution:

Source code is provided in "Exercise 2 - ProbabilityEstimation" folder. I wrote a program that will take two patterns containing H(Head) and T(Tail) and calculate the probability of one of them showing up before the other one. In the OpenMP, only 2 variables are needed to count how many times each pattern appears first. Synchronization is done by the reduction clause on these 2 variables. Until one pattern appears, the string is concatenated with a 'H' or 'T' each having a probability of 0.5. When one of these patterns appear, clear the string and add one to the respective counter. When all the tries are done, print the results for each pattern.

PROBLEM 3 (25 pts) - Random restart hill climbing

Hill climbing is a local search algorithm used in optimization problems (to determine an approximate maximum value of a function, which is known as the objective function). It is an iterative algorithm that starts with an arbitrary point (which may be given or may be generated randomly), then attempts to find a better solution by making an incremental change to the solution. If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found. The major drawback of this algorithm is that it may converge on a local maximum (thus providing a suboptimal solution). In order to avoid this drawback there is another version of this algorithm known as: random restart hill climbing.

In random restart hill climbing, the classical hill climbing is applied several times and among the found answers, the best one is picked as the final answer.

In this problem you are required to apply parallel random restart hill climbing in order to find the maximal value over a specified rectangular region for a two variable function, which will be already hard-coded in your source file (with signature):

double objectiveFunction(double x, double y).

The program will read as input the minX, maxX, minY, maxY as the bounds of the rectangular region and the stepSize parameter. So, starting at a random initial point (x_0, y_0), it will check the eight "neighboring" points: $(x_0 - stepSize, y_0)$, $(x_0 + stepSize, y_0)$, $(x_0, y_0 - stepSize)$, $(x_0, y_0 + stepSize)$, $(x_0 - stepSize, y_0 - stepSize)$, $(x_0 + stepSize, y_0 - stepSize)$, $(x_0 - stepSize, y_0 + stepSize)$ and $(x_0 + stepSize, y_0 + stepSize)$. The "neighboring" point which provides the largest value of the objective function is found and (if this value is larger than the current value), the "best neighbor" will become our current node and the procedure is repeated.

a. Write a program using Java multi-threading which reads as input the total number of times that hill climbing will be repeated, the number of threads that will be created, the values minX, maxX, minY, maxY and the stepSize. Finally it will print as result the largest value of the objective function (that is found) and the values x and y of the point for which it is obtained.

b. Write a program using POSIX threads which reads as input the total number of times that hill climbing will be repeated, the number of threads that will be created, the values minX, maxX, minY, maxY and the stepSize. Finally it will print as result the largest value of the objective function (that is found) and the values x and y of the point for which it is obtained.

Note: For testing purposes you may use the objective function defined as:

```
double objectiveFunction(double x, double y){
      return 50 + 20*x*y - 3x*x*y -  2*x*y*y;
}
```

And minx =0, maxX = 10, minY = 0, minY = 10, stepSize=0.01, so the final answer is expected to be approximately 99.4 obtained for x = 2.23 and y = 3.33

Solution:

Source code is provided in "Exercise 3 - RandomHill" folder. For each try, a random x and y is generated inside minx, miny, maxx and maxy. The program looks in 8 directions from this random point and if it finds a point with a bigger value, it goes in that direction and this is repeated until there isn't a point in these 8 directions with a higher value. In the OpenMP solution, we use the pragma omp critical when making comparisons with the global maximum value to make sure the comparison and the reassignment is done in synchronized way. In the MPI, the communication of the maximum value is done with the All_reduce. This is done because next if a process has the same value as the maximum value, it broadcasts its rank to say: "I have the maximum value. If you don't have my rank, don't print the x and y you found for the maximum value.". If the correctRank==rank, print the results.

PROBLEM 4 (25 points)

Formulate a problem or pick a known problem and firstly describe the serial solution to that problem. Then:

- Provide a parallel solution using one of the shared-memory parallel programming platforms (Java multithreading, Posix threads or OpenMP).
- Provide a parallel solution using distributed memory parallel programming in MPI.

The problem I chose was Matrix Multiplication. Source code is provided "Exercise 4 – MatrixMul". The problem is simple to solve in serial:

-Take as input a matrix mxn and a matrix nxp. N should match for the multiplication of matrix to make sense.

- For each row m, multiplied by each column p, summing for every column in n of matrix 1 multiplying each matching row in n of matrix 2. $O(n^3)$

-Print the result.

Ideally, we want to parallelize the loop which has the most computation from m, n and p. Since we can't be sure, by convention we parallelize the outer loops as much as possible. No synchronization is needed since each calculation is done in a different cell of the resulting matrix. In the MPI solution, the matrix is flattened to make the sending operation simpler. Each process take a portion of matrix 1 and the whole matrix 2 using scatter. After doing the calculations on their portion, partial solutions are gathered in process 0 and printed.