J. H. Walther
S. Martin
ETH Zentrum, CLT
CH-8092 Zürich

Fall semester 2022

Set 4 - Monte Carlo, MPI

Issued: November 9, 2022
Hand in (optional): November 23, 2022 08:00

Question 1: Probabilities, Expectation, Variance (20 Points)

Probability Density Function: For a continuous random variable X with range $[a, b]$, the probability density function $p(x')$, is defined such that the probability P of the random variable X to take a value smaller than x is given by

$$P(X \leq x) = \int_{-\infty}^{x} p(x')\mathrm{d}x' \tag{1}$$

Usually this is denoted by $F_X(x) = P(X \leq x)$ and called **cumulative distribution function**.

Expectation Value: For a continuous random variable X with range $[a, b]$ and probability density function $p(x)$, the expected value of $h(X)$ is defined as

$$\mathbb{E}[h(X)] = \int_a^b h(x)p(x)dx \tag{2}$$

A special case is $h(X) = X$, which gives the mean $\mu = \mathbb{E}[X]$.

Variance: For a continuous random variable X the variance of X is:

$$\mathrm{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 \tag{3}$$

We realize that this is an expectation value with $h(X) = (X - \mu)^2$, i.e. the variance is the expected mean squared distance from the mean μ.

a) You arrive in a building and are about to take an elevator to your floor. Once you call the elevator, it will take between 0 and 40 seconds to arrive. The duration until the elevator arrives can be seen as a random variable T. For the following we assume the time of arrival $t \in \mathbb{R}$ to be uniformly distributed between 0 and 40 seconds, i.e. $t \sim \mathcal{U}([0, 40])$.

- Write down the probability density function $p(t)$ for the random variable T.
- Calculate the probability $P(T \leq t)$ that elevator takes up to 15 seconds to arrive.
- Calculate the mean value $\mathbb{E}[T]$ that the elevator takes to arrive.
- Calculate the variance $\mathrm{Var}[T]$ the elevator takes to arrive.

- We consider the probability distribution of the time of the arrival of the elevator to be uniform in the range $[a, b] = [0, 40]$ s. Therefore, the probability density of the random variable t is written as:

$$p(t) = \begin{cases} \frac{1}{b-a}, & \text{if } a \leq t \leq b \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

or, substituting a, b

$$p(t) = \begin{cases} \frac{1}{40}, & \text{if } 0 \leq t \leq 40 \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

- For any continuous PDF, we can calculate probabilities using integration. Specifically for a uniform distribution the probability of RV t being between $[c, d]$ is:

$$\mathrm{P}(c \leq t \leq d) = \int_c^d p(t)dt = \int_c^d \frac{1}{b-a}dt = \frac{d-c}{b-a} \tag{6}$$

In our specific case, the probability of time of arrival to be less than 15 s is given as:

$$\mathrm{P}(0 \leq t \leq 15) = \frac{15-0}{40-0} = \frac{15}{40} = \frac{3}{8} = 0.375 \tag{7}$$

- The expectation value of the above uniform distribution is:

$$\mathrm{E[T]} = \int_a^b tp(t)dt = \int_a^b \frac{t}{b-a}dt = \frac{b-a}{2} = \frac{40-0}{2} = 20\,\text{seconds} \tag{8}$$

- The variance of the above uniform distribution is:

$$\mathrm{Var[T]} = \mathrm{E[T^2]} - \mathrm{E^2[T]} = \int_a^b \frac{t^2}{b-a}dt - \left(\frac{b-a}{2}\right)^2 = \frac{(b-a)^2}{12} = \frac{(40-0)^2}{12} = \frac{400}{3} \tag{9}$$

Points:

- **5 points for the correct PDF** $p(t)$
- **5 points for correct** $P(0 \leq t \leq 15)$
- **5 points for** $\mathrm{E[T]}$
- **5 points for** $\mathrm{Var[T]}$

Question 2: Parallel Monte Carlo using MPI (38 points)

Monte Carlo integration is a method to estimate the value of an integral over a domain Ω by taking samples $x_i \sim \mathcal{U}(\Omega)$ from a uniform distribution on the domain Ω. First, note that the integral can be expressed as an expectation value over the uniform distribution

$$\frac{1}{|\Omega|}\int_{\Omega} f(x)\,\mathrm{d}x = \mathbb{E}_{x\sim\mathcal{U}(\Omega)}[f(x)]\,, \tag{10}$$

where $|\Omega|$ is the volume of the domain Ω. The central limit theorem states that we can estimate the expectation value using the average

$$\mathbb{E}_{x\sim\mathcal{U}(\Omega)}[f(x)] \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)\,, \quad \text{for } x_i \sim \mathcal{U}(\Omega) \tag{11}$$

Combining eq. (10) and eq. (11) we conclude that we can compute the integral as

$$\int_{\Omega} f(x)\,\mathrm{d}x = \frac{|\Omega|}{N}\sum_{i=1}^{N} f(x_i)\,, \quad \text{for } x_i \sim \mathcal{U}(\Omega) \tag{12}$$

In the skeleton code you are given a serial implementation of this algorithm estimating the intersection of the two circles shown in Fig. 2. The goal of the exercise is to parallelize the code using MPI.

a) Implement all of the `TODO_a)` parts in the skeleton code to calculate the overlapping area of the two circles shown in Fig. 2 using Monte Carlo integration in serial mode!

Use the function `std::uniform_real_distribution<double>` which returns a uniformly distributed random number in the interval $[0, 1]$ and make use of it. Circle radius R_1, R_2 and circle R_2 x-coordinate x_2 are changeable variables in the system. Make use of them when generating random samples and checking if you are inside the circles area. With $R_1 = 5$, $R_2 = 10$ and $x_2 = 12$ you should get an area of ≈ 17.01.

You can ignore all the `TODO_b` parts and try only to recreate the same approximated area of ≈ 17.01.

Instructions on how to run your code:

- **module load open_mpi** to load the MPI libraries.
- **make** to compile the code.
- **bsub -n 1 -W 01:00 -Is bash** to have an interactive session.
- **mpiexec -n 1 main** to execute your code on one core.

We choose to sample for our integration from an orthogonal domain, containing both circles. Therefore, with (x_1, y_1) = (0,0) and R_1=5 and (x_2, y_2) = (12,0) and R_2=10:

- x should be in the interval [-5,22]: We have to transform the random values taken by `random()` to this interval. Sampling in the x-direction will be:
  $x = \texttt{random()} * 27 - 5$
- y should be in the interval [-10,10]: We transform sampling in the y-direction:
  $y = \texttt{random()} * 20 - 10$

Points:

Algorithm 1 Monte Carlo integration for overlapping area

Require: circle center x_2
circle radii R_1, R_2
number of Monte Carlo samples M

Ensure: overlap area of 2 circles

$R_1 = 5$
$R_2 = 10$
$x_2 = 12$
$M = 10000$

function OVERLAPMC(x_2, R_1, R_2, M)
  double area_rectangle= $(R_1 + x_2 + R_2) * 2. * R_2$
  int pts_inside = 0
  for int i:=0; i < M; i++ **do**
    double $x_i = (x_2 + R_2 + R_1) *$ `random()` $- R_1$
    double $y_i = 2 * R_2 *$ `random()` $- R_2$
    if $(x_i * x_i + y_i * y_i \leq R_1 * R_1)$ and $((x_i - x_2) * (x_i - x_2) + y_i * y_i \leq R_2 * R_2)$ **then**
      pts_inside ++
    end if
  end for
  return (double) pts_inside / M * area_rectangle
end function

- **3 points for the right** `area_rectangle`
- **3 points for the right** x_i
- **3 points for the right** y_i
- **5 points for the right if-statement in the for-loop**
- **5 points for replicating the same area** $\approx 17.01 \pm 0.01$
- **-5 points for not using** R_1, R_2 **&** x_2

b) Implement all of the `TODO_b)` parts in the skeleton code to parallelize it with MPI. You are given a fixed amount of MC samples $n = 1e9 + 1$. You should split the number of samples as equally as possible without leaving a sample out! One of the ranks will have more or less if the number of samples is not divisible by the number of processors. NOTE: if you are not able to do question a), you can still get full points in this question.

Instructions on how to run your code:

- **module load open_mpi** to load the MPI libraries.
- **make** to compile the code.
- **sbatch launch.sh** to run the code on 48 cores. This should not take more than 5 minutes if correctly implemented! It will run the `run.sh` script. Which will do multiple MPI runs with different numbers of cores. It outputs them in a separate folder called `out/`. `launch.sh` will create two different files. `slurm_error.txt` which consists of any errors and `slurm_output.txt` which has all of the execution runs as well as the information of which CPU used. It can happen, that you get Warnings in the

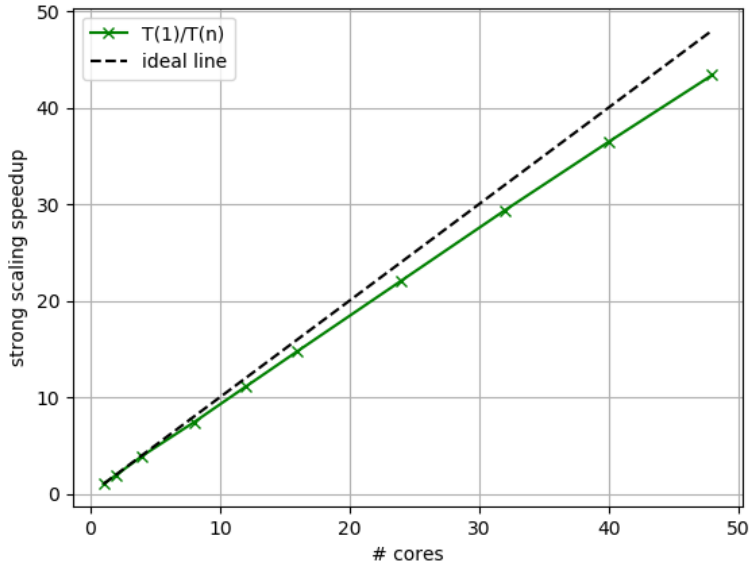

Figure 1: AMD EPYC 7763 64-Core Processor

`slurm_error.txt` "WARNING: No preset parameters were found for the device that Open MPI detected:". You can ignore those.

- You are free to modify the number of cores and which type of CPU to use.
- **python3 plot.py** to plot the results.png.
- Note: You could also run the code directly on your computer. In this case, you have to install MPI on your own and adapt the number of cores in `run.sh`. You would also have to run the `run.sh` script directly using **source run.sh**.

Points:

- **3 points for initializing MPI at the start of the `main()` function.**
- **3 points for splitting the for-loop in the `overlapMC()` function as equally as possible for each processor.**
- **3 points for the correct implementation of `MPI_Reduce()`**
- **1 point for `MPI_Finalize()`**

c) Plot the results with the given python script. Don't forget to state which type of cpu you used!

Points:

- **3 points for a reasonable strong scaling plot.**
- **-1 point for not stating the CPU type.**

d) What would you add to your code if you wanted to estimate the error of the Monte Carlo integration? Answer qualitatively, do not write any code.

In general, if you want to compute the MC error, you have to keep both function evaluations AND function evaluations squared. In this specific case, the function is the unit function, if the random point is inside the overlapping area.

Points:

- **3 points for the right statement about the error of the MC integration**

e) How does the error of the method change if you use 10 times more samples?

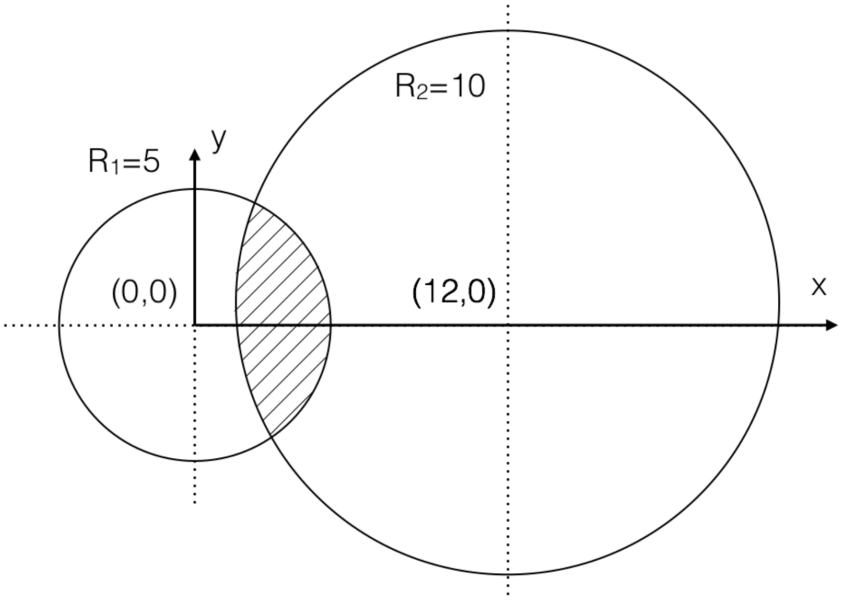

Figure 2: Sketch of two overlapping circles

We know that MC error behaves as: $\epsilon_{MC} \sim \sqrt{\frac{|Var|}{M}}$. Given $M_2 = 10 \cdot M_1$, we have $\epsilon_{MC-2} = \frac{\epsilon_{MC-1}}{\sqrt{10}}$

Points:

- **3 points MC error change**

Question 3: MPI Bug Hunt (32 points)

Find the bug(s) in the following MPI code snippets and find a way to fix the problem!

a)

```
1  const int N = 10000;
2  double* result = new double[N];
3  // do a very computationally expensive calculation
4  // ...
5
6  // write the result to a file
7  std::ofstream file("result.txt");
8
9  for(int i = 0; i <= N; ++i){
10     file << result[i] << std::endl;
11 }
12
13 delete[] result;
```

- There is a segfault hidden in line 9.
  It can be fixed by changing `i <= N` to `i < N`.
- All ranks write simultaneously to the same output file! This is a problem for many reasons:
  On one hand, it leads to an incorrect output file because of many concurrent writes that overwrite each other.
  On the other hand, the same work is done many times. One remedy would be to let only the root rank write the output. This can be implemented as shown in the code below.

```
1  const int N = 10000;
2  double* result = new double[N];
3  // do a very computationally expensive calculation
4  // ...
5
6  // write the result to a file
7  if(rank == 0){
8      std::ofstream file("result.txt");
9
10     for(int i = 0; i <= N; ++i){
11         file << result[i] << std::endl;
12     }
13 }
14
15 delete[] result;
```

  Of course, this assumes that every rank has the same data. If that is not the case, one would have to send all the information to the root rank.
  Later in the course, you will learn to to use MPI to perform I/O (input/output) operations involving many ranks in parallel.

Points:

- **2.5 points for each identified bug**
- **2.5 points per fixed bug**
- **-5.0 points if another bug is introduced through the "improvement"**
- **you cannot have less than zero points in this subquestion**

b)

```
// only 2 ranks: 0, 1
double important_value;

// obtain the important value
// ...

// exchange the value
if(rank == 0)
    MPI_Send(&important_value, 1, MPI_DOUBLE, 1, 123,
      MPI_COMM_WORLD);
else
    MPI_Send(&important_value, 1, MPI_DOUBLE, 0, 123,
      MPI_COMM_WORLD);

MPI_Recv(
    &important_value, 1, MPI_INT, MPI_ANY_SOURCE,
    MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE
);

// do other work
```

- The MPI type in the receive call (`MPI_INT`) does not match the MPI type that is sent (`MPI_DOUBLE`). Change it to `MPI_DOUBLE` to ensure defined behaviour.
- This code will deadlock: Both ranks send first and receive later. Neither of the two ranks can return from the send call because the call blocks until the sending is completed. But the sending cannot complete because the corresponding receive is not called yet. Deadlock!
  Possible solution:

```
// only 2 ranks: 0, 1
double important_value;

// obtain the important value
// ...

// exchange the value
if(rank == 0){
    MPI_Send(&important_value, 1, MPI_DOUBLE, 1, 123,
      MPI_COMM_WORLD);
    MPI_Recv(
```

```
11          &important_value, 1, MPI_DOUBLE,
              MPI_ANY_SOURCE,
12          MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE
13      );
14  }else{
15      MPI_Recv(
16          &important_value, 1, MPI_DOUBLE,
              MPI_ANY_SOURCE,
17          MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE
18      );
19      MPI_Send(&important_value, 1, MPI_DOUBLE, 0, 123,
          MPI_COMM_WORLD);
20  }
21      // do other work
```

Points:

- 2.5 points for feach identified bug
- 2.5 points per fixed bug
- -5.0 points if another bug is introduced through the "improvement"
- you cannot have less than zero points in this subquestion

c) What is the output of the following program when run with 1 rank? What if there are 2 ranks? Will the program complete for any number of ranks?

```
1  MPI_Init(&argc, &argv);
2
3  int rank, size;
4  MPI_Comm_size(MPI_COMM_WORLD, &size);
5  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6
7  int bval;
8  if (0 == rank)
9  {
10     bval = rank;
11     MPI_Bcast(&bval, 1, MPI_INT, 0, MPI_COMM_WORLD);
12 }
13 else
14 {
15     MPI_Status stat;
16     MPI_Recv(&bval, 1, MPI_INT, 0, rank, MPI_COMM_WORLD,
         &stat);
17 }
18
19 cout << "[" << rank << "] " << bval << endl;
20
21 MPI_Finalize();
22 return 0;
```

For only 1 rank, everything is fine and the output of the program is:

```
[0] 0
```

With 2 ranks, however, there will be a deadlock:
Rank 0 arrives at the broadcast. This is a blocking *collective* operation, which means rank 0 waits until all other ranks in the communicator `MPI_COMM_WORLD` have reached this point and performed the collective operation. But rank 1 never arrives at this point! Instead, it gets stuck in the receive operation.

Points:

- **4 points for predicting correct behaviour with one rank**
- **4 point for predicting correct behaviour with two ranks**
- **4 points for justifying the behaviour of two ranks**