CX 4220/CSE 6220 High Performance Computing (Fall 2024)
Programming Assignment 1
Due: 18 Sept

1 Problem Statement

Write a parallel program in C/C++ to estimate the value of π using the Monte Carlo method specified below. The program should take n as input, which is the number of points to be used for the estimation. The program should then generate n random points in the unit square $[0,1] \times [0,1]$ and count the number of points that lie inside the first quarter of the unit circle.

Let n be a large integer, then the estimated value of π is

$$\pi \approx 4 \cdot \frac{\text{number of points in the circle}}{n}$$

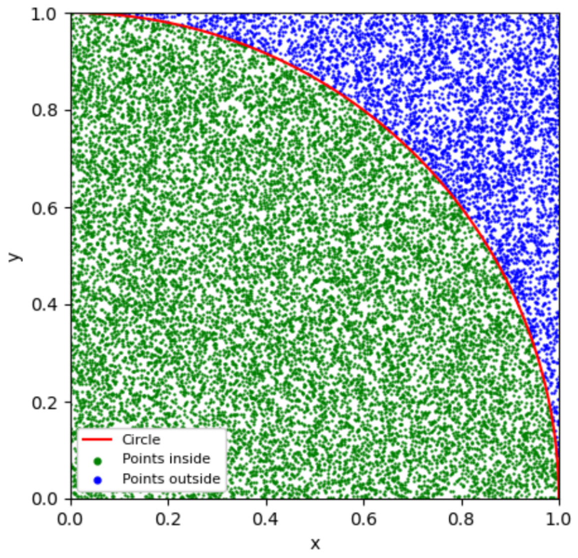

Figure 1: Estimating π using Monte Carlo method

2 Parallel Algorithm

The estimation can be easily parallelized by assigning each processor the responsibility for $\frac{n}{p}$ distinct points, where p is the total number of processors. You can make use of the following MPI functions:

- We pass the value of n (problem size) to the pi_calc function. You need to divide the problem size into the p processors. Make sure to take care of the situation where p does not perfectly divide n.

- Use **MPI_Reduce** function to sum the number of points in the circle across all processors.

For the random number generator, you can use the **rand()** function in C/C++. However, you should **make sure that the random number generator is seeded differently for each processor**. You can use the processor rank for this purpose. For example, if the rank of the processor is i, then you can use **srand(time(NULL) + i)** to seed the random number generator on that processor.

3 Code Framework

3.1 Input & Output Format

Your program takes n as the input using command line arguments and output the estimated value of π and the time taken to compute this value. The output will be in the below given format.
Estimated Pi: 3.14152
Time: 0.0777975 seconds

3.2 Download Files

Download the PA1 files from this link **(or)**
Use `git clone https://github.com/gtcse6220-fall24/pa1.git`

1. **Makefile**
2. **pi.h** - Develop your code in this file. This is your **submission file**. **Note:-** You need to only need to submit the pi.h file.
3. **pi.cpp** - The driver file for your code. You can make changes in the driver file for your own testing and to generate data for plot. But make sure that the submitted 'pi.h' file runs with the original 'pi.cpp' file
4. **autograder.sh** - This file can be used to check if your code passes the test cases. This script will request the resources, run the tests and display the output in the command line.

3.2.1 Instructions

1. After downloading all the files, go through each of the files and try to understand the structure.
2. Write your implementation in `pi.h` file.
3. Use `make` to compile the program. An executable **pi** will be created if there are no compilation errors.
4. Request the resources (interactive node) as discussed in the review session (using `salloc`) and use `module load openmpi` to load the MPI library (which will solve the include/command not found errors, if any).

5. Then use `mpirun -np <num_processors> ./pi -n <num_points>` to run the program.

6. After finalizing your code, you can run the test cases from the autograder.sh file. Use `sbatch autograder.sh` to run the autograding script. You can see the output in the file `results.out`.

3.3 Deliverables

1. **pi.h** - Make sure this file runs with the original 'pi.cpp' file.

2. **Report** - Make sure to list names of all your teammates at the very beginning of your report.

   - For $n = 10^9$, plot a graph of run-time of the program vs. the number of processors for values of $p = 1, 2, 4, 8, 16, 24$. This run-time should include all computations that contribute to the estimation, including any local computations and global reductions. Include your graph and observation regarding the speedup in a PDF file. Make sure to list names of all your teammates at the very beginning of your report.
   - Do the runtime analysis for T(n,1) (serial code), T(n,p) (parallel code) which includes both computation and communication cost and the speed-up in terms of n, p. This analysis will be on your entire code in "pi.h".
   - Answer the questions given below

3. **Questions :-**

   (a) Why is using MPI_Reduce better than MPI_Gather?

   (b) Why is it necessary to have a random generator function for generating points? Why is necessary to have different seeds for the random generator in every processor?

   (c) Explain the functioning of every MPI instruction that you used in your program (pi.h and pi.cpp), also including the MPI setup instructions (eg. MPI_Init, etc).

3.4 Grading

- Code

  NOTE: We will be running your submissions on a Intel(R) Xeon(R) Gold 6226 CPU @ 2.70GHz (24 cores) node.

  - Passing all code correctness and scalability tests - **10**
    - Correctness Test - The estimated value of Pi should lie between 3.13 and 3.15 for $n = 10^6$ and $p = 1, 16$.
    - Runtime Test - Runtime should be less than 5.5 seconds for $n = 10^9$ for $p = 8$.
    - Scalability Test - For $n = 10^9$, $\frac{T(p=1)}{T(p=16)}$ should be greater than 15.5
  - Random Generator and different seeds for all processors - **5**

- Report -

  - Plot - **4**
  - Runtime analysis - **2**
  - Questions - **9**

4 Resources

- What is a `Makefile` and how does it work?: `https://opensource.com/article/18/8/what-how-makefile`
- PACE ICE cluster guide: `https://docs.pace.gatech.edu/ice_cluster/ice-guide/`. Documentation for writing a PBS script: `https://docs.pace.gatech.edu/software/PBS_script_guide/`