**Project 5: MPI Programming (100 points)**

**Submission guidelines:**

You will submit **2 files** in Canvas.

* a PDF report file (filename: **Lastname\_Firstname\_Project\_5\_Report.pdf**)
* a ZIP file (filename: **Lastname\_Firstname\_Project\_5.zip**) containing:
  + Problem\_1/
    - pingpong\_MPI.c
    - Makefile
    - pingpong\_diffnode.sbatch
    - pingpong\_samenode.sbatch
  + Problem\_2/
    - dot\_product\_MPI.c
    - Makefile
    - dot\_product.sbatch
  + Problem\_3/
    - merge\_sort\_MPI.c
    - Makefile
    - merge\_sort\_diffnode.sbatch
    - merge\_sort\_samenode.sbatch
  + Problem\_4/ (If graduate)
    - pi\_MPI.c
    - Makefile
    - pi.sbatch

**Test data**:

The test data are provided on Schooner at ”/home/oucspdnta/current/test\_data/Project\_5\_Tests/”.

You may use the gpel machines to test your code, but please run the timings for your report on Schooner. Also please be aware that gpel may be unreliable for running your MPI programs.

**You may need to load these modules to compile on Schooner**

module load intel

module load GCC

We will deduct **5% of the points** if you don’t follow the code specifications AND submission guidelines and no matter whether the code contains the correct solution or not.

**Autograder Instructions**

**Where to get the Autograder:**

* The autograder and the test data is located on Schooner at:  
  /home/oucspdnta/current/test\_data/Project\_5\_Tests/
* To copy the autograder and the test data to your own home directory, run the command:  
  cp -r /home/oucspdnta/current/test\_data/Project\_5\_Tests/ .

**Required File Structure:**

* As noted within the python script, the file structure of the project is necessary:
  + test\_data/
    - Vectors/
    - Problem\_2/
    - Problem\_3/
    - Problem\_4/
  + LastName\_FirstName\_Project\_5/
    - Problem\_2/
      * dot\_product\_MPI.c
      * Makefile
    - Problem\_3/
      * merge\_sort\_MPI.c
      * Makefile
    - Problem\_4/
      * pi\_MPI.c
      * Makefile
    - autograder\_project\_5.sbatch
    - autograder\_project\_5.py
    - autograder\_problem\_5\_2.py
    - autograder\_problem\_5\_3.py
    - autograder\_problem\_5\_4.py
  + autograder\_base.py
* NOTE: There is no autograder for Problem 1

**To Run the Autograder:**

* **Running on Schooner:**
  + First, make sure to edit the sbatch file where indicated. You will have to edit your username, and the directories of where the script is located, and where to output the stdout and stderr files.
  + You can also edit the exact python script that the sbatch job will call, and thus indicate which problem to grade, or to grade all of them
  + To submit the job, run the command:  
    sbatch autograder\_individual.sbatch
  + To see the status of the job, run the command:  
    squeue –u oucspdn###
  + Once the job completes, it will return the output and error messages in two files:  
    autograding\_########\_stderr.txt  
    autograding\_########\_stdout.txt  
    (Where the #s are replaced with the job number)
* **Running Locally:**
  + For Windows users, we recommend using Windows Subsystem for Linux (WSL) with Ubuntu to run code from the command-line.
  + Make sure that your file structure is correct, otherwise the autograder will not be able to find your files.
  + Navigate to the location of your autograding files, and then run either:  
    python autograder\_problem\_5\_#.py  
    To run the autograder on just one individual problem   
    or  
    python autograding\_project\_5.py  
    To run the autograder on all problems
  + This will then test your program against the provided test data, and will output your score for each test, where a 1.0 for each test indicates that the test succeeded, and a 0.0 means the test failed.

**Problem 1. [CS4473 and CS5473] (30 points)**

Ping-Pong to estimate MPI communication time

Write an **MPI program** to calculate the communication time between two processes.

Given an array of random integers, Write MPI\_Send and MPI\_Recv to send an entire array back and forth between process 0 and process 1 (two processes) for 1000 times. After one process receives the array, it sends this array back to the other process. The array is bounced back and forth between the two processes like a ping-pong ball for 1000 times. Use MPI\_Wtime() to record the total time for arrays of 1 million integers, 2 million integers, 4 million integers, and 8 million integers.

Please modify the two provided **sbatch files** to time your program using two processes (a) on two different compute nodes and (b) on one compute node.

Use the timing results from (a) two nodes and (b) one node to make **two plots** with the array size in the x-axis and the average transmission time for a one-way trip from one process to the other process in the y-axis in the report

Suppose that the average one-way-trip transmission time = latency + data\_amount / bandwidth. Please use your plot and **a linear regression function** for each curve in R, Matlab or any other software to estimate the latency and bandwidth for message passing (a) between two processes in two different nodes and (b) between two processes on the same node on the Schooner supercomputer.

For this project you'll have to have your program run on separate nodes and in order to do this, you should pay attention to the ntasks and ntasks-per-node variables in your sbatch file. The ntasks variable should be equal to the number of your processes, such as 2 for your ping-pong program. You should then alter ntasks-per-node the variable to delegate either all of these tasks to one node or only one task to one node.

Your program should be run like this:

pingpong\_MPI n\_items time\_prob1\_MPI.csv

n\_items: number of items in array

time\_prob1\_MPI.csv: time after back and forth transmission for 1000 times

Learning outcome:

Practice MPI send and receive operation and learn the communication cost in distributed-memory programs.

What to submit:

Please submit your MPI code, Makefile and sbatch files and in the report: plot, fitting and estimated values.

Grading rubrics

CS4473/5473

* 18 points for MPI code and Makefile running correctly
* 2 points for sbatch files
* 5 points for plot
* 5 points for linear function parameters

**Problem 2. [40 points for CS4473 and 20 points CS5473]**

Dot product is a commonly used operation in linear algebra. Please read more about this operation at <https://en.wikipedia.org/wiki/Dot_product>. Please parallelize the dot product algorithm using MPI. Process 0 will read in the two input arrays and print out the result.

A serial implementation is provided and it can be run like this:

dot\_product\_serial vector\_size vec\_1.csv vec\_2.csv result.csv time.csv

Your program should be run like this:

dot\_product\_MPI n\_items vec\_1.csv vec\_2.csv result\_prob2\_MPI.csv time\_prob2\_MPI.csv

* n\_items: number of items in array (262144, 524288 or 1048576)
* vec\_1.csv, vec\_2.csv: The input csv files with the input vectors
* result\_prob2\_MPI.csv: the dot product (a scalar)
* time\_prob2\_MPI.csv: Elapsed time. Start the timer after reading in the two arrays and stop the timer after the dot product is computed.

Learning outcome:

This problem is designed for you to practice MPI collective communication and implement it using the embarrassingly parallel pattern in MPI.

What to submit:

Please submit your MPI code, makefile and sbatch file(s) in the ZIP file.

Please report the wall-clock time table, the speedup table and the efficiency table using 2 processes, 4 processes and 8 processes on the “vec1” and “vec2” arrays of length 262144 (2^18), 524288 (2^19) and 1048576 (2^20). The processes can be created on the same node or across multiple nodes.

|  |  |  |  |
| --- | --- | --- | --- |
| Array size | 262144 | 524288 | 1048576 |
| 2 processes |  |  |  |
| 4 processes |  |  |  |
| 8 processes |  |  |  |

Please discuss the scalability of the MPI program.

Grading rubrics

CS4473

* 23 points for logical MPI code and Makefile running correctly
* 2 points for sbatch file(s)
* 5 points for correct results with respect to serial results (**within 1% range**)
* 10 points for table

CS5473

* 9 points for MPI code and Makefile running correctly
* 1 point for sbatch file(s)
* 5 points for correct results with respect to serial results (**within 1% range**)
* 5 points for table

**Problem 3. [CS4473 and CS5473] (30 points)**

You have learned a parallel sorting algorithm based on the odd-even transposition sort. Let’s implement another simple parallel sorting algorithm based on the merge sort:

<https://en.wikipedia.org/wiki/Merge_sort>

Below is a pseudo-code for the MPI parallel merge sort.

1. Read a global array from an input file using process 0
2. Scatter the global array across all processes
3. Sort the local arrays by every process using the build-in quick sort function (qsort)
4. Reduce the local sorted arrays on all processes to a global sorted array on process 0
5. Write the global sorted array to an output file using process 0

The Step 4 is the most interesting step. It uses a tree reduction pattern as illustrated below.

Diagram

Description automatically generated

Instead of combining two partial sums, you need to combine two local sorted arrays to a larger sorted array in each reduction operation. Please follow the pseudo-code below to merge two sorted arrays into one sorted array.

<https://en.wikipedia.org/wiki/Merge_algorithm#Merging_two_lists>

Your program should be run like this:

merge\_sort\_MPI n\_items input\_array.csv output\_array.csv time\_prob3\_MPI.csv

* n\_items: number of items in the array (262144, 524288 or 1048576)
* input\_array.csv: the input array to be sorted
* output\_array.csv: the sorted output array
* time\_prob3\_MPI.csv: Elapsed time. Start the timer after reading in the input array and stop the timer after the array is sorted.

Learning outcome:

This problem is designed for you to practice MPI point-to-point communication. You need to carefully calculate the ranks of the senders and receivers in each iteration of the tree reduction. This also uses MPI to reinforce the tree reduction pattern that you first learned in CUDA.

What to submit:

Please submit your MPI code, makefile and sbatch file(s) in the ZIP file.

Please report the wall-clock time table, the speedup table and the efficiency table using 4 processes on the “vec1” arrays of length 262144 (2^18), 524288 (2^19) and 1048576 (2^20). Please benchmark using multiple processes on the same node and then using multiple processes on separate nodes.

You can use “#SBATCH --ntasks-per-node=4” to ask SLURM to run all 4 process in one node and use “#SBATCH --ntasks-per-node=1” to ask SLURM to run only one process per node.

|  |  |  |  |
| --- | --- | --- | --- |
| Array size | 262144 | 524288 | 1048576 |
| 4 processes on the same node |  |  |  |
| 4 processes on 4 different nodes |  |  |  |

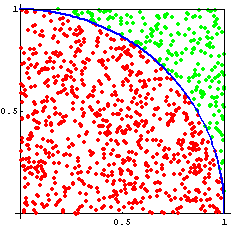
Please discuss the scalability of the MPI program under the 2 scenarios (same node vs. different nodes). The discussion should be based on the communication cost that you measured in Problem 1.

Grading rubrics

* 20 points for logical MPI code and Makefile running correctly
* 2 points for sbatch file(s)
* 5 points for correct results
* 3 points for table

**Problem 4. [CS5473] (20 points)**

Write an MPI program using Monte Carlo method to estimate the value of π. For this purpose, we randomly generate a large number of points (x, y), where 0 ≤ x, y ≤ 1. All these points lie in the square with each of its sides of length 1 with its south-west corner at origin.



Note that the area of the square is one and that of the arc of the circle of radius one in the first quadrant is π/4. The point (x, y) will lie in the part of the circle in the first quadrant if x2 + y2 ≤ 1. If the random numbers are distributed uniformly, then the fraction of the points so generated that lie within the circle is approximately π/4.

Your program should generate 2^16 random points (n = 2^16) to estimate π. Each process generates a subset of the random points and determines the number of points that lie within the circular part of the first quadrant. For different processes to generate a sequence of different random numbers, choose a different seed value for generating random numbers for each process. This seed value can be a very simple function of the process id.

Next, the process 0 should sum up the counts of all the points that lie in the sector (red points) calculated by all the processes. Finally, the process 0 should compute π and print out the value of π. The value of π is 4 times of the ratio of (the number of points within circular part)/(the total number of points generated).

Run the program for **n = 2^16**, with varying processors from **2, 4, 8, 16**. The processes can be created on the same node or across multiple nodes. Make a wall-clock time table, a speedup table and an efficiency table when you keep the number of points fixed but the number of processes increases.

pi\_MPI result\_prob4\_MPI.csv time\_prob4\_MPI.csv

* result\_prob4\_MPI.csv: the estimated value of pi
* time\_prob4\_MPI.csv: Elapsed time from the start until the estimation of pi.

What to submit:

Please submit your MPI code, Makefile and sbatch file(s) in the ZIP file. Please provide the three tables and discuss the scalability in the report.

Grading rubrics

CS5473

* 10 points for MPI code and Makefile running correctly
* 3 points for reasonable value for pi (3.14) within **5% range**
* 2 points for sbatch file(s)
* 5 points for table