CS 426/525: Project 1

Spring 2025

Introduction

The first project consists of two parts, in which you can explore MPI and parallel programming concepts. In both parts you will write a serial program, then parallelize it using MPI library with C programming language. Part A is simpler to start with, whereas Part B presents a more complicated problem. After your implementations, you should experiment with serial and parallel programs and see if you are getting any speed-up in parallel implementations. Next, you should write a short report of your findings. Submission will be on Moodle.

Part A: Find Average

The objective of the problem is to write a serial, then a parallel program that takes as input an array of integers, stored in an ASCII file with one integer per line and the number of the elements on the top, and prints out the average of the elements into an output file, that is find the sum of all elements in the array, divide by the number of elements.

Example

Input:

```
4 //number of elements
13
27
35
18
```

Output:

```
(13 + 27 + 35 + 18) / 4
= 23.25
```

- **Serial**: Write a serial program to solve this problem. Name the source code `average-serial.c`. Takes the name of the input file as a single argument from the command line.

- **MPIv1**: Write an MPI implementation of the above program in which a master process reads in the entire input file and then dispatches pieces of it to workers, which these pieces being of as equal size as possible. **The master must also perform computation.** Each processor computes a local sum, and results are collected by the master. The master processor then finds the overall sum of the elements and computes the average from them. This implementation must not include any collective communication routines, only point-to-point communication is allowed. Name the source file `average-mpi-ppv1.c`. Note that this program should take the name of the input file as a single argument from the command line.
- **MPIv2**: This is similar to MPIv1, however in this part you are expected to use the MPI collective communication routines. Specifically, you should use the MPI_Scatter and MPI_Allreduce functions. Name the source file `average-mpi-ppv2.c`. As before, the program should take the name of the input file as an argument in the command line. Note that only the master process should output the result.

Part B: Find Average in Buckets

The objective of this problem is to separate pairs of non-negative integers into buckets according to the first key, then take the average of the second keys with respect to buckets. First, group the pairs by their first key into buckets. In each bucket, take the sum of the second keys for each bucket. Then, find the average of each bucket by dividing it into a number of the second keys for each bucket. For example, if we have the following array of pairs:

- [(2, 40), (3, 56), (2, 42), (2, 20), (4, 5), (3, 14), (3, 10), (4, 20), (4, 41)]

The first operation should be to separate the values according to their first key and put them into buckets:

- [(**2**, 40), (**2**, 42), (**2**, 20)]
- [(**3**, 56), (**3**, 14), (**3**, 10)]
- [(**4**, 5), (**4**, 20), (**4**, 41)]

Then, find the average of each bucket according to the second key.

- **Bucket 0:** [(2,40), (2,42), (2,20)] Average value of the second keys = 34
- **Bucket 1:** [(3,56), (3,14), (3,10)] Average value of the second keys = 26.6
- **Bucket 2:** [(4,5), (4,20), (4,41)] Average value of the second keys = 22

So, we create the following array: [(2, 34.0), (3, 26.6), (4, 22.0)]
And the final output is: 34.0, 26.6, 22.0

The file for the input is stored in an ASCII file where each line consists of two integers separated by space.

Example

Input:

```
9 //number of elements
2 40
3 56
2 42
2 20
4 5
3 14
3 10
4 20
4 41
```

Output:

```
34.0
26.6
22.0
```

- **Serial**: Write a serial program called `bucket_average-serial.c`. The serial implementation first finds the range of the buckets by the first key value in pairs, then finds the average of each bucket with respect to their second key value. In the first step, you should find the range on the first values of pairs. If the range is r, you will have $r + 1$ buckets, and bucket i will store the pairs whose first values are equal to i. After finding the r, scan the pairs and assign each pair to their corresponding bucket. After assigning all of the buckets, you can sum up the second key for each bucket and divide by the number of elements in the bucket. The final output is stored in those buckets in order (You may need to normalize the bucket indices, i.e., if the first key's min value is 2, the max value is 6, then the range is 4. Then we have 5 buckets; bucket 0 will store the pairs with the first value equal to 2, bucket 1 will store the pairs with the first value equal to 3, etc.). Your program should take two arguments from the command line:
  - the name of the input file
  - the name of the output file.

  The final result should be written into the output file in order as well.

- **Parallel:** Write a parallel program called `bucket_average-mpi.c`. The parallel version takes the same arguments from the command line and writes the result to the output file as in the serial version. Write an MPI implementation of the program by parallel decomposition based on partitioning the input data. The master process reads the input file, finds the overall range, and then scatters the array and range to all other processes

in which workers work on inputs as equally as possible. Each worker is responsible for a specific portion of the input. **The master must perform computation as well.** Each worker calculates the sum and number of the elements for each bucket locally. After each worker finishes, they will send the local sum and number of the elements for each bucket to the master process. Then, the master process will calculate the average values per bucket and output the result to the file. For example with two processors:

```
P1 computes [(2,40), (3, 56), (2, 42), (2,20), (4,5)]
    bucket[0] => local_sum = 102, # of elements = 3
    bucket[1] => local_sum = 56, # of elements = 1
    bucket[2] => local_sum = 5, # of elements = 1
P2 computes [(3,14), (3,10), (4,20), (4,41)]
    bucket[0] => local_sum = 0, # of elements = 0
    bucket[1] => local_sum = 24, # of elements = 2
    bucket[2] => local_sum = 61, # of elements = 2
The master process (assuming p1) gathers the results and
computes the average value per bucket:
    bucket[0] => global_sum = 102, # of elements = 3
    bucket[1] => global_sum = 80, # of elements = 3
    bucket[2] => global_sum = 66, # of elements = 3
    bucket[0] => average = 34.0
    bucket[1] => average = 26.6
    bucket[2] => average = 22.0
```

You may assume that the division of the number of inputs to the number of processes is an exact integer value with no residual. And the first keys are consecutive positive numbers.

Submission Format

Submit a single zip file containing source files, a Makefile, a SLURM script file, outputs for various input sizes, and a report discussing your findings. **yourname_lastname_p1.zip** to Moodle that includes:

- Your implementation with source files and necessary inputs for the following:
  - average-serial.c
  - average-mpi-ppv1.c
  - average-mpi-ppv2.c
  - bucket_average-serial.c

  - bucket_average-mpi.c

- A makefile that generates the following 5 executables:
  - average-serial
  - average-mpi-ppv1
  - average-mpi-ppv2
  - bucket_average-serial
  - bucket_average-mpi
- Run your code with a various number of processors.
- Provide output generated by your implementation with various input sizes.
- Your report (3 pages at most):
  - **Part A:** Briefly explain your implementations. Plot graph(s) with various thread numbers for small, medium, and large input sizes indicating the performance of your implementation by comparing serial, MPIv1, and MPIv2. What are your observations? Does parallel implementation improve the performance of any inputs? If not, what might be the reasons?
  - **Part B:** Briefly explain your implementations. State and explain the reason for your choice of implementation for the parallel version. What are your expected results? Plot graph(s) with various thread numbers for small, medium, and large input sizes. Also include multiple range values indicating the performance of your implementation. What are your observations? Does the input size and range of the values impact the performance of your implementation?

Grading

- Part A: 40 points
  - Serial: 5 points
  - MPI-v1: 20 points
  - MPI-v2: 15 points
- Part B: 50 points
  - Serial: 10 points
  - Parallel: 40 points
- Report: 10 points

Important Notes

- **DUE DATE: March 23, 2025 23:59**
- **No Late Submission Allowed!**
- Please ask your questions via **Moodle forum** so that everybody can see the problem and the answer. (Will be open in a week)
- Your submission file **MUST BE** in a **ZIP** format. (not RAR, TARBALL etc.)
- In order for your project to be graded, it must be able to compile with the Makefile you provided and must run without any problem on the TRUBA/ARF system with the SLURM script file you provided.
- You can always reach out to me via email. (can.bagirgan@bilkent.edu.tr)