Department of Computer Engineering

Bilkent University

CS 426 - Parallel Computing

Project 1 Report

Name: Deniz Yüksel

Student ID Number: 21600880

Design Choices for Project Parts:

Min-serial: I used a simple modular pointer-based array implementation. I performed the computation in O(n) because it was intuitive and the main objective of this homework is to deal with parallel message passing. I will deep-dive to all the comparisons in the experiment part.

Min-mpi-v1: I improved my modular implementation from the serial part. I divided the work equally among all processors but the last one. The latter received the remainder processes. The reason I divided the work like this was because of the fact that I did not want to allocate a separate work array for the processes. This is, of course, a tradeoff but in this situation, I thought that invoking MPI_Send operation two times is not worth it because the communication gets blocked. Instead, one processor had to work more.

Min-mpi-v2: The instructions stated: *"This is similar to MPIv1 except that all processors should have the computed overall min."* It implies that a similar implementation of version 1 can be used, but in the end all the processors must have the computed overall min, so we need an MPI_Bcast. My design choice was that I did not want to broadcast all array to every processor. This would create an overhead that the same array is duplicated and dispatched to all processors. Instead, using MPI_Send and MPI_Receive, and to satisfy the requirement, invoking a broadcast operation at the end is reasonable.

Greyscale -serial: I implemented my own structs matrix and pixel. The initial design choice of struct instead of a 3D integer array was reasonable, and it paid off in the rest of the project as well, by means of ease of implementation. Again, the code works modular. The difficulty would arise while sending structs with MPI.

Greyscale -mpi-v1: Similar to min-mpi-v1, I used MPI_Send and MPI_Receive by fetching the rows of the text file. However this time, the input was an image that is naturally large. Therefore, I decided to send a work array that defines each processors' work amount. Here, I made a decision to distribute the amount of lines as equal as possible. I chose to invoke MPI_Send additionally, but I divided the work more equally than I previously did on min-mpi-v1.

Greyscale -mpi-v2: As the project instruction stated that we would send the processes with grids perspective, I did so. With complex for loops and needlessly tryhard algorithms, I managed to send and receive information as grids. However, I think the first version is more efficient since row-major operations in C utilize the cache more. On the other hand, if we were dealing with a rendering situation here, that is we would need the image to appear on the fly, then we could use gridwise sending and receiving.

Greyscale -mpi-v3: The assignment assumed that we would use MPI_Bcast for section 2. However, I used MPI_Send and MPI_Receive. I think an improvement to the implementation in stage 2 is simply not implementing it. However, real improvement can be not to send the fetched data back to the master. Instead, it would be wise for every slave processor to write to the same text file in a parallel fashion.

TESTS WITH VARIOUS SAMPLES

While doing tests for this project, I could not simulate more processors than I have. For Linux and Windows machines, the open-mpi implementation can simulate more processors than a system has. However, the Mac implementation of open-mpi would not let that. This is why I did not draw plots but rather showed data on tables. Moreover, for parallel execution tests, I measure the time with MPI_Wtime() as shown in lecture slides. However, I am also invoking MPI_Barrier() which stops all the processors until completion of the whole process.

Table of number of processors and elapsed time in milliseconds for Part A.

numProcs	min serial	min mpi-v1	min mpi-v2
1	0.69	0.39	1.18
2	NA	0.78	2.06
3	NA	1.48	0.88
4	NA	1.53	1.09

Interpretation: Surprisingly, the minimal time elapsed in min serial is less than that of other parallel tests. This may be because my input file had 5000 lines only. Although the individual parallel time measures for every processor is less than the serial execution, the bottleneck processor, which is the master took more time to finalize.

Table of processors versus elapsed time in milliseconds for Part B.

numprocs	serial	Row-based (mpiV1)	Grid-based (mpiV2)
1	83.23	245.92	226.34
2	NA	225.43	NA
3	NA	220.32	NA
4	NA	230.54	326.57

Interpretation: As I mentioned in my previous guesses, the amount of time when we invoke grid-based sending and receiving, is significantly much than that of other tests. Again, the time it takes to execute serial execution is measured to be less than its counterparts. However, each processor measures less than the serial, but the bottleneck processor measures more. This can be the case also because, in my implementation, I am doing too much dynamic array allocations and memory operations.