CPEN 512 Assignment 1 Report

Jiaqi Zhang (63174551)

1. Parallel Implementation

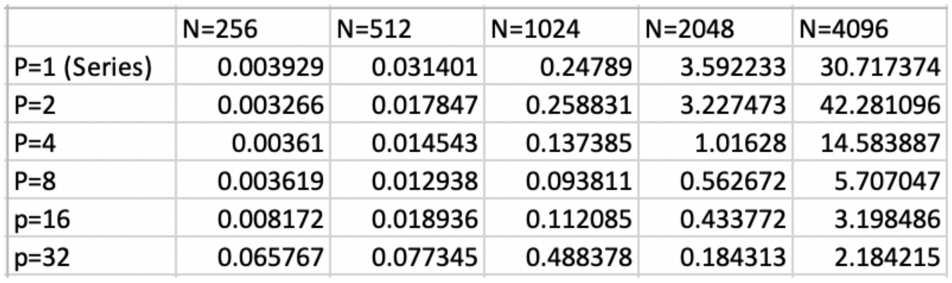
The main idea is to divide the matrix such that rows of different sub matrixes could be processed at the same time. I used MPI_Isend to send rows and MPI_Ireceive to receive rows. During the MPI_Isend, the program can do normalization of rows at the same time.

Cyclic striped mapping was implemented and greatly improved load-balancing issue.

2. Performance

2.1 Summary

From the table, when the matrix size is small (less than 1024), we observe a worse performance when increasing the number of processors. This is because if matrix is small, the time spent on sending and receiving rows from different processors greatly exceeds that of calculating values. But when matrix is getting bigger, we can see a huge improvement of runtime between serial and parallel implementations.



	N=256	N=512	N=1024	N=2048	N=4096
P=1 (Series)	0.003929	0.031401	0.24789	3.592233	30.717374
P=2	0.003266	0.017847	0.258831	3.227473	42.281096
P=4	0.00361	0.014543	0.137385	1.01628	14.583887
P=8	0.003619	0.012938	0.093811	0.562672	5.707047
p=16	0.008172	0.018936	0.112085	0.433772	3.198486
p=32	0.065767	0.077345	0.488378	0.184313	2.184215

Figure 1. Performance Summary

2.2 Observations from the line chart:

1) As the number of processors increases, the performance gradually improves (except p=1 and p=2). This is because in parallel programming, different processors can do different calculations at the same time.
2) When p=2, the programs performs even worse than p=1 (sequential vanilla implementation without MPI). I think the reason is that in case of p=2 we have one submatrix sending rows, after which the other submatrix will send the rows, and there is no parallellism involved in this scenario. Compared to sequential, however, there will be other overheads such as memory copy. But when p>2, the MPI solution performs much better than sequential.

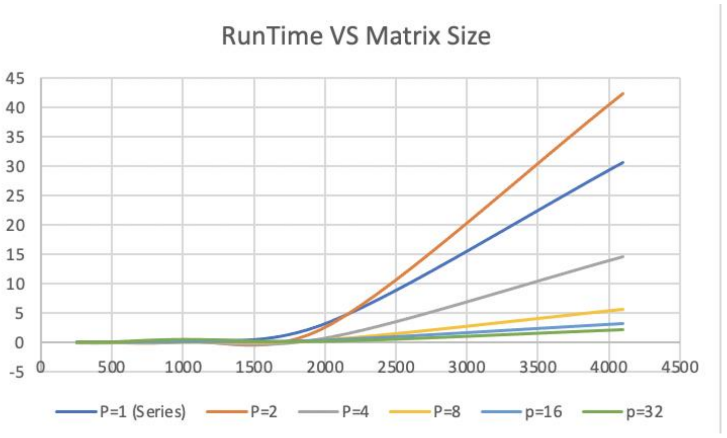


Figure 2. Runtime vs. Matrix Size
Custom flag: -Ofast -finline-functions

2.3 Observations from Optimization Settings:

1) The performance for parallel implementation of Gaussian Elimination improves a lot by using custom flags.
2) Custom Optimization Setting is about 8 times faster than the one without optimization setting.

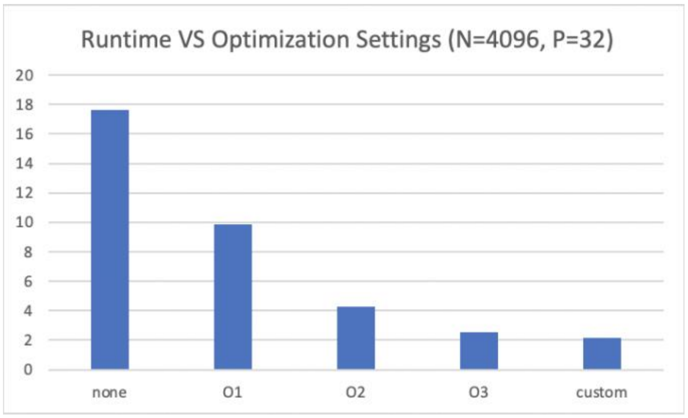


Figure 3. Optimization Settings

2.4 Observations from processor count:

1) When number of processors is smaller or equal to 32, there is a decent speedup as processors increase. However the trend is opposite when we further double the processors. One reason is that the server (i.e. cpen512.ece.ubc.ca) only has 32 cores so trying to set more processors when running the program doesn't help. Another reason is probably due to excessive communications between processors.
2) Open MPI with 32 processors is about 14x speed up than the serial one when the matrix size is 4096.

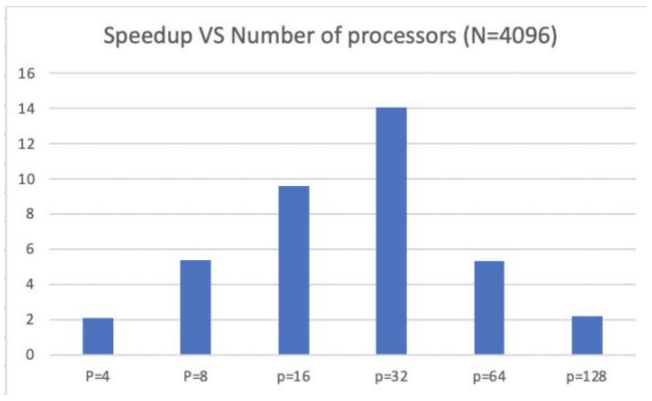


Figure 4. Speedup vs. Processor Count

3. Compile and Run

Parallel Compile: *mpicc -Ofast -finline-functions -o gm gaussian_mpi.c*
Parallel Run: *mpirun -np 32 ./gm 4096*

Serial Compile: *gcc -Ofast -finline-functions -o gs gaussian_serial.c*
Serial Run: *./gs 4096*

Best result for parallel version: 2.184215 seconds.
Best result for sequential version: 14.711683 seconds.