Program 4 Report in CS6376 Parallel Processing Course

1. Problem Statement:

This program mainly focuses on optimizing the matrix multipication serial code by using combination of MPI, openMP and openACC. And implement cannon’s algorithm with MPI.

1. Approach to Solution:

First, I built a serial version of the matrix muliplication and test it to check if it gives me the correct result. Next, I started to implement the openMP parallel pragma for the first test case; implement the openACC parallel pragma for the second test case, and so on.

1. Solution Description:

For the openMP approach, I added omp pragma before the matrix multiplication nested loop.

The performance had the maximum Gflops when N, the input size, is around at 1600. But it went down as N increased.

Command line: pgcc -O4 -mcmodel=medium -mp -fast mm\_omp.c

export OMP\_NUM\_THREADS=28

|  |  |
| --- | --- |
|  | openMP (n = 28) |
| N=1600 | 0.06sec 136.4Gflops |
| N=3200 | 0.724sec 90.45Gflops |
| N=4800 | 3.55sec 62.21Gflops |
| N=6400 | 8.66sec 60.48Gflops |
| N=8000 | 17.21sec 59.46Gflops |
| N = 16000 | 297.52sec 27.53 Gflops |

**Gustafson-Barsis's Law**:



The original serial code running time: 0.000240sec (n= 32)

The running time result with oponMP: 0.000021sec (n = 32) 0.000021

Speedup = 240/21 = 11.43.

11.43 ≤ 28 + (-27)\*s

16.6 ≥ 27\*s

s ≤ 0.61

Based on the formula, the serial fraction is 61%. The serial fraction is this high because of the small n. Compared to the actual parallel processing, the serial communication time is longer. As we raise the matrix size N larger, the serial fraction will decrease significantly.

**Karp-Flatt Metric**:

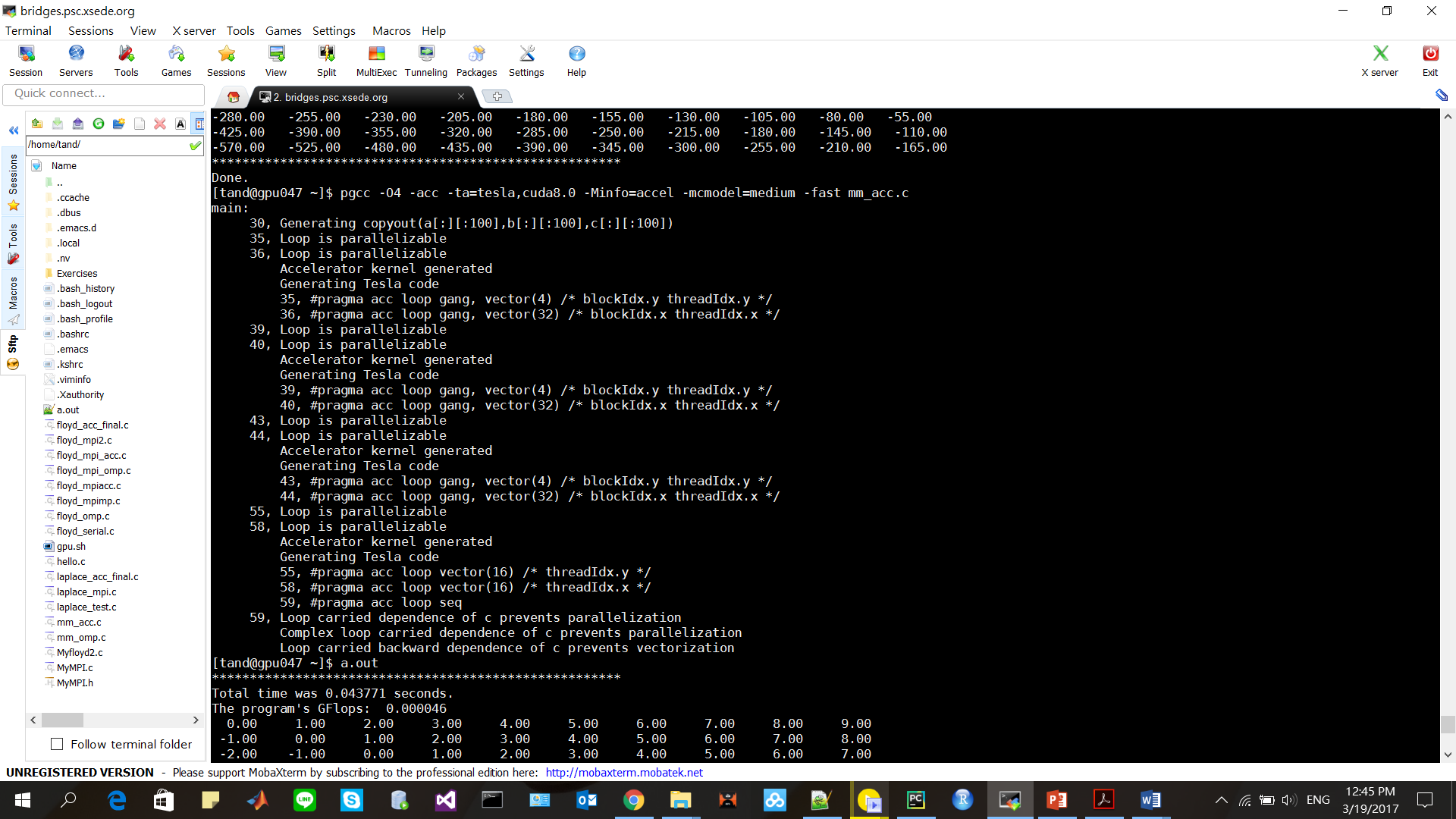


e = (1/11.43 – 1/28) / (1 -1/28) = 0.053

The matric is small, indicating that the program may not have large overhead, imbalanced workload that would decrease the speedup.

For the openACC portion. Unlike openMP, the Gflops went up as the N increased.

Command line: pgcc -O4 -acc -ta=tesla,cuda8.0 -Minfo=accel -mcmodel=medium -fast mm\_acc.c



Compile with open acc.

|  |  |
| --- | --- |
|  | openACC |
| N=1600 | 0.08sec 101.88Gflops |
| N=3200 | 0.29sec 224.99Gflops |
| N=4800 | 0.81sec 270.80Gflops |
| N=6400 | 1.81sec 288.36Gflops |
| N=8000 | 3.38sec 302.329Gflops |
| N=16000 | 25.43sec 322.08Gflops |

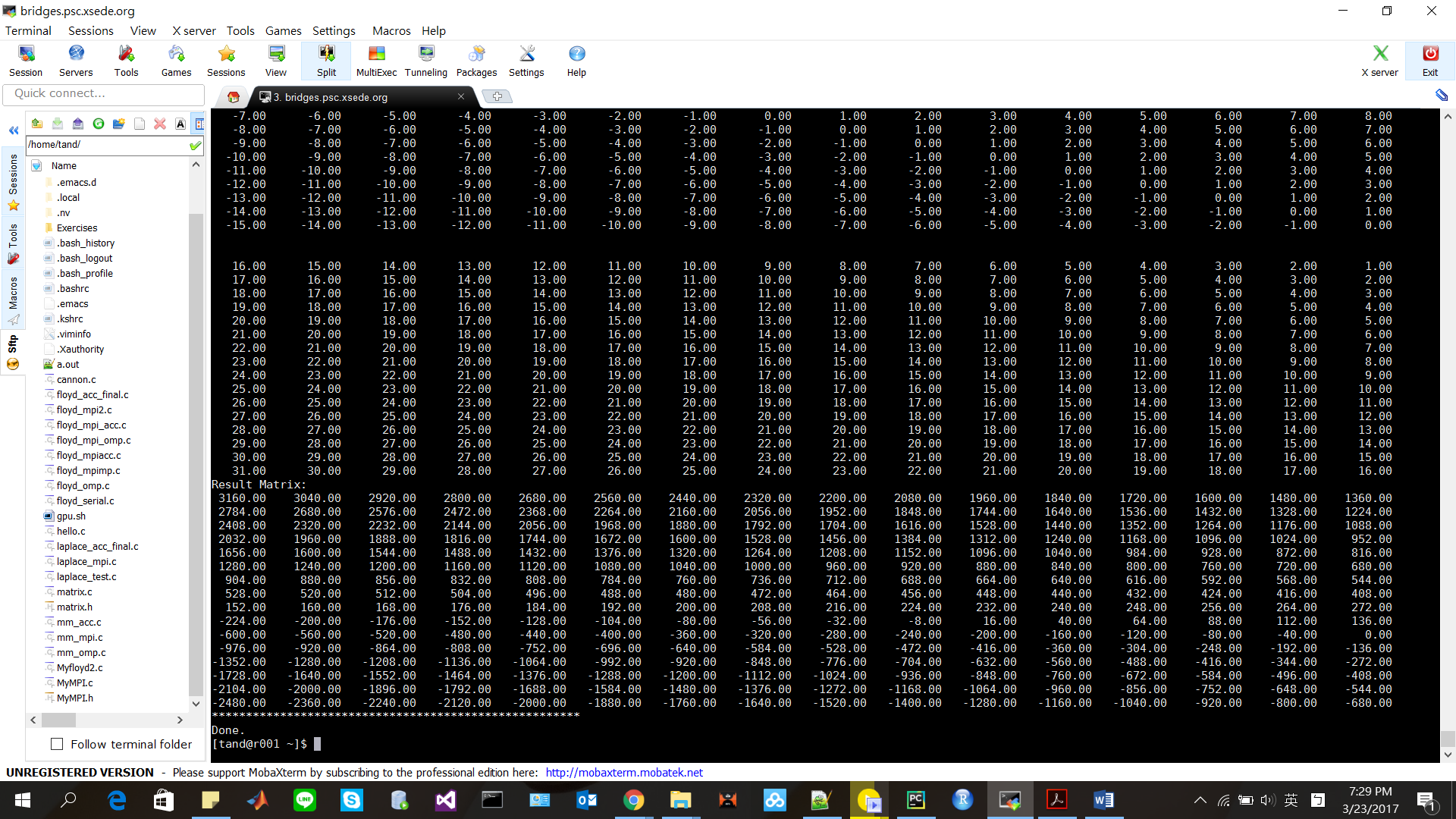
For the MPI implementation, I used MPI library and its function to help speed up the program. I initialized an array of size N\*N, and scatter it to 16(processes) blocks. I also used MPI\_Cart\_create to change the original communicator to a 2D communicator.

Command line: module load mpi/pgi\_openmpi

mpicc -O4 -fast cannon.c

mpirun -np 16 a.out

|  |  |
| --- | --- |
|  | MPI(n 16) |
| N=1600 | 0.8569sec 9.56Gflops |
| N=3200 | 5.2742 sec 12.43 Gflops |
| N=4800 | 20.1741sec 10.96 Gflops |
| N=6400 | 59.1605 sec 8.86 Gflops |
| N=8000 | 97.4763 sec 10.51 Gflops |
| N=16000 | 851.79sec 9.62 Gflops |



16 by 16 matrix testing MPI implementation with result matrix C.

For the MPI openMP hybrid coding, I still having trouble to optimize the code with MPI plus OMP. Though I passed the compilation. The code has poor performance.

Command line: module load mpi/pgi\_openmpi

mpicc -O4 -mp -mcmodel=medium -fast mm\_mpi\_omp.c

export OMP\_NUM\_THREADS=28

mpirun -np 16 a.out

In this program, I spent most of the time building the Cannon’s algorithm with the MPI. The algorithm itself is very interesting. Fist, we swap the matrix’s row and column to the right place and scatter the blocks of matrix A, B to each process, and do the multiply and shift every time. Then, rearrange the blocks to the original place.