Program 4 Report

CS6376

Parallel Processing

Submitted by –

Ayush Agrawal (ara150430)

1. Problem Statement

Main goal here is to implement the matrix multiplication algorithm. Implement it in a parallel environment and compare the performance among multiple techniques.

2. Approach to Solution

Implement the matrix multiplication algorithm in C programming language. To improve performance, I am using OpenACC, OpenMP, MPI with C to execute code in parallel. The main idea is to add constructs from OpenMP, OpenACC, MPI in the program wherever possible such that the code runs parallel and performance is improved. For the purpose of this assignment I am using the bridges environment provided by Pittsburgh Supercomputing Center.

3. Solution Description

3.1 General Assumptions

1. For all the programs the size of input matrices is 1000 * 1000.

2. Implementation of Matrix Multiplication algorithm has been modified and implemented using –

    1. OpenACC

    2. OpenMP

    3. MPI

3. For MPI cannon's algorithm is been implemented, the number of processors must be square.

3.2 Compiling and Building the solution

3.2.1 For OpenACC version

Step 1 – log into bridges using ssh username@bridges.psc.edu

Step 2 – enter password

Step 3 – use interact –gpu command to access and allocate resources to use.

Step 4 – Edit the source code using utilities like vi or emacs.

Step 5 – use module load cuda/8.0 command to load module for gpu and cuda functionalities

Step 6 – compile the program using

```
pgcc –acc Minfo=accel matrix_acc.c –o matrix_acc –ta=tesla,cuda8.0
```

and then run the program using ./matrix_acc

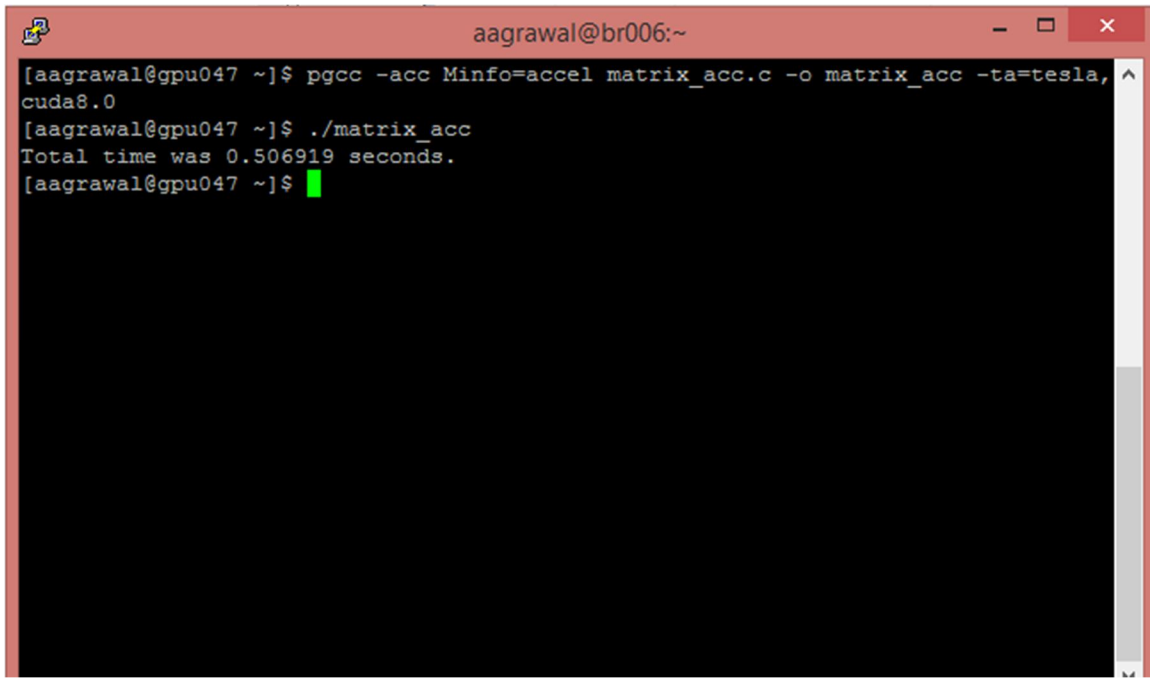


Fig 1 – Compiling and execution using OpenACC

3.2.2 For OpenMP version

Step 1 – log into bridges using ssh username@bridges.psc.edu

Step 2 – enter password

Step 3 - Edit the source code using utilities like vi or emacs.

Step 4 – Use interact command to allocate resources.

Step 5 – Use pgcc –mp matrix_omp.c –o matrix_omp command to compile the program.

Step 6 – Submit the shell script covering various cases for different number of processors using sbatch

sbatch ./momp.sh

Step 7 – Look for the results in slurm.out file.

3.2.3 For MPI version

Step 1 – log into bridges using ssh username@bridges.psc.edu

Step 2 – enter password

Step 3 - use interact –gpu –N 4 command to access and allocate resources to use.

Step 4 – Edit the source code using utilities like vi or emacs.

Step 5 – use module load mpi/pgi_openmpi to load module for mpi.

Step 6 – Compile using

mpicc –O matrix_mpi.c

Step 7 – Execute using

mpirun –np 4 ./a.out

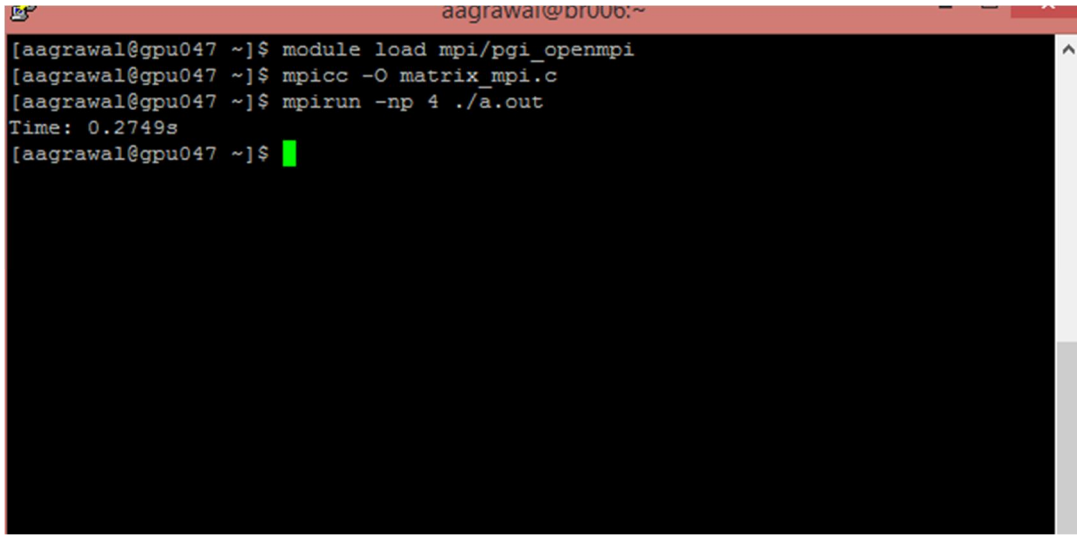
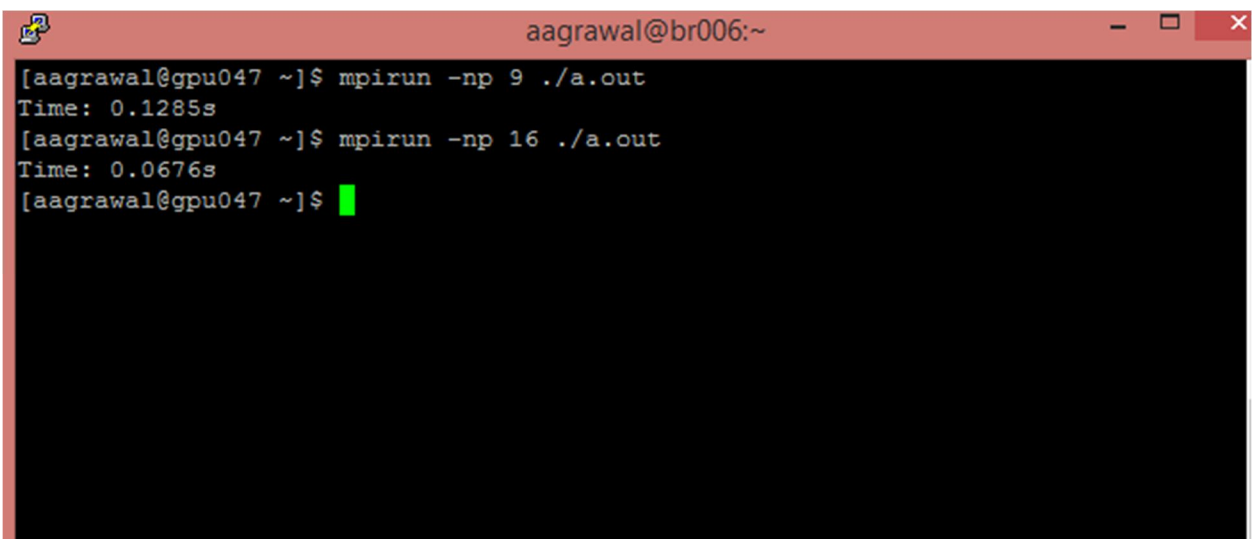


Fig 2 – Compiling and execution using MPI

3.3 Screenshot –

Result of running matrix multiplication with p = 9 and 16 since number of processes had to be a square



3.4 Performance Measures

For computing all the measures or metrics the dimension of matrices is N * N where N is 1000.

The below shown tables shows performance measure in GFLOPs/sec for the code on various configurations. It is computed using formulae-

$$\text{Performance} = (2 * N^3 - N^2) / \text{execution time} * 10^9$$

Here, execution time is the amount of time it took to run under that configuration.

Scenario	Execution time	GFlops/Sec
Sequentially	3.437318	0.581558063583
Using OpenACC	0.506919	3.94343080453

This table shows in case of OpenMP

Number of threads	Execution time	GFlops/Sec
1	3.437318	0.581558063583
2	2.136791	0.935514984853
3	1.274138	1.5689038393
4	1.024149	1.95186442598
5	0.821581	2.43311371612
6	0.676980	2.95281987651
8	0.538941	3.70912585979
12	0.354227	5.6432739458
16	0.294950	6.77741990168
20	0.234281	8.53248876349
24	0.195202	10.2406737636
28	0.157411	12.6992395703

This table shows in case of MPI

Number of processes	Execution time	GFlops/Sec
4	0.2749	7.27173517643
9	0.1285	15.5564202335
16	0.0676	29.5710059172

Second measure is Speedup, which is computed using formulae -

Speedup = execution time on one core (serial execution time)/ execution time in parallel on multiple core

Below table shows the speedup value in case when OpenMP is used.

Number of cores used	Speedup
2	1.60863556614
3	2.69775958334
4	3.35626749623
5	4.18378467856
6	5.07742917073

8	6.37791149681
12	9.70371541413
16	11.6539006611
20	14.671774493
24	17.6090306452
28	21.836580671

The speedup achieved in case of OpenACC is 6.780.

Below table shows the speedup value in case when MPI is used.

Number of processes	Speedup
4	12.5038850491
9	26.7495564202
16	50.8478994083

Third measure is Karp Flatt metric which computes experimentally determined serial function e.

The formulae to compute e is –

$$e = (1/\text{ speedup} - 1/p) / (1 - 1/p)$$

Here p is the number of cores used and speedup is the speedup corresponding to that number of cores.

This table shows in case of OpenMP

Number of cores used	Speedup	e
2	1.60863556614	0.62164484054
3	2.69775958334	0.370677952986
4	3.35626749623	0.297950029646
5	4.18378467856	0.239018036737
6	5.07742917073	0.196950063974
8	6.37791149681	0.156791137742
12	9.70371541413	0.10305331075
16	11.6539006611	0.0858081795169
20	14.671774493	0.068158081388
24	17.6090306452	0.0567890430853
28	21.836580671	0.0457947155311

This table shows in case of MPI

Number of processes	Speedup	e
4	12.5038850491	0.0799751434113
9	26.7495564202	0.037383797484
16	50.8478994083	0.019666495797

Note – Unfortunately I was not able to generate results for MPI and OpenMP combination on time. I have included the program for it and also slurm output for OpenMP program.