Program 3 Report

CS6376

Parallel Processing

Submitted by –

Ayush Agrawal (ara150430)

1. Problem Statement

Main goal here is to implement the floyd's algorithm for computing the al pair shortest paths in a graph. Implement it in a parallel environment and compare the performance among multiple techniques.

2. Approach to Solution

Implement the floyd's 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 number of nodes in graph are 2000. In all the cases the graph is a fully connected graph with edge length no longer than 2 for any edge.

2. Implementation of floyds algorithm for computing all pair shortest path in a graph has been modified and implemented using –

    1. OpenACC
    2. OpenMP
    3. MPI

3. For MPI version the code works only if the number of processes involved are divisible by number of nodes in graph.

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.

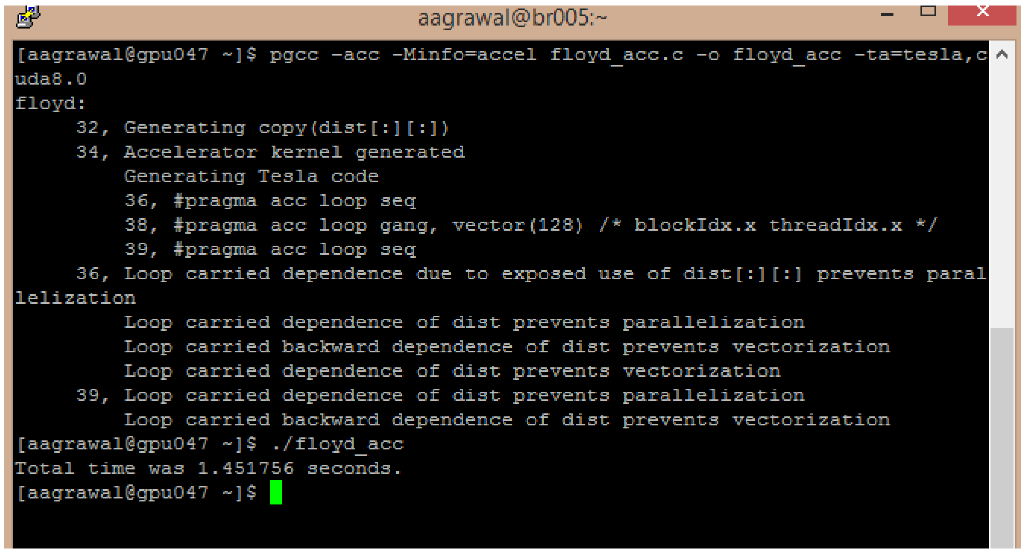
Step 4 – Edit the source code using utilities like vi or emacs.

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 floyd_acc.c –o floyd_acc –ta=tesla,cuda8.0
```

and then run the program using ./floyd_acc



aagrawal@br005:~

```
[aagrawal@gpu047 ~]$ pgcc -acc -Minfo=accel floyd_acc.c -o floyd_acc -ta=tesla,c
uda8.0
floyd:
     32, Generating copy(dist[:][:])
     34, Accelerator kernel generated
         Generating Tesla code
         36, #pragma acc loop seq
         38, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         39, #pragma acc loop seq
     36, Loop carried dependence due to exposed use of dist[:][:] prevents paral
lelization
         Loop carried dependence of dist prevents parallelization
         Loop carried backward dependence of dist prevents vectorization
         Loop carried dependence of dist prevents vectorization
     39, Loop carried dependence of dist prevents parallelization
         Loop carried backward dependence of dist prevents vectorization
[aagrawal@gpu047 ~]$ ./floyd_acc
Total time was 1.451756 seconds.
[aagrawal@gpu047 ~]$
```

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 floyd_omp.c –o Floyd_omp command to compile the program.

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

sbatch ./fomp.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 - Edit the source code using utilities like vi or emacs.

Step 4 – Submit the shell script which compiles the mpi based program and executes it various number of processes using sbatch

sbatch ./fmpi.sh

Step 5 – Look for the results in slurm.out file.

3.3 Performance Measures

For computing all the measures or metrics number of nodes in the graph used are 2000.

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

$$\text{Performance} = 2 * (\text{Number of nodes}^{3}) / \text{execution time} * 10^{9}$$

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

Scenario	Execution time	GFlops/Sec
Sequentially	12.733	1.256
Using OpenACC	1.451	11.03

This table shows in case of OpenMP

Number of threads	Execution time	GFlops/Sec
1	12.733	1.256
2	8.378243	1.90
3	5.759652	2.777
4	4.542805	3.522
5	3.573658	4.477
6	3.064816	5.22
8	2.291172	6.98
12	1.593170	10.04
16	1.185650	13.49
20	0.885004	18.07
24	0.707764	22.60
28	0.574075	27.87

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	1.51
3	2.21
4	2.80
5	3.56
6	4.15
8	5.55
12	7.99
16	10.73
20	14.38

24	17.99
28	22.18

The speedup achieved in case of OpenACC is 8.775.

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	1.51	0.662
3	2.21	0.452
4	2.80	0.357
5	3.56	0.280
6	4.15	0.240
8	5.55	0.180
12	7.99	0.125
16	10.73	0.093
20	14.38	0.069
24	17.99	0.055
28	22.18	0.045

Note – Unfortunately I was not able to generate results for MPI on time and was getting segmentation faults while running it on interact. I have included the scripts for both MPI and OpenMP and slurm output for OpenMP in the submission folder.