

The University of Texas at Dallas



CS 6376

PARALLEL PROCESSING

Programming Assignment 3

Anay Sonawane (ass151130)

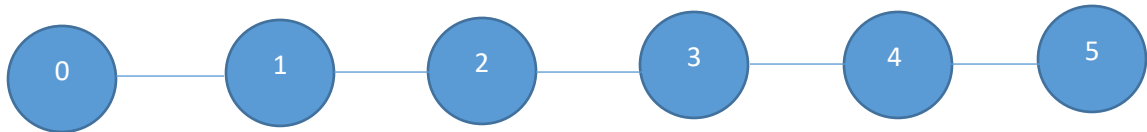
1. Problem Statement:

To implement the Floyd's algorithm in C/C++. Using OpenMP, OpenACC and MPI in combination to optimize the algorithm. Also design and implement test data generation code to be used with the algorithm. We have to implement 5 combinations such as OpenMP, MPI, OpenACC, OpenMP nad MPI, OpenACC and MPI. Also parameterize the code so that size can be scaled with the number of processors.

2. Implementation of Floyd algorithm in C/C++:

a. Approach to the solution:

I have implemented the Floyd algorithm first in C/C++. For that initially I have implemented the algorithm using 6 vertices.



Using this I have generated the 6X6 adjacency matrix as an input to the Floyd algorithm. The edge between the two nodes is treated as '1' and the distance between the nodes which are not connected to each other are is taken as 6 means infinity.

Adjacency matrix input:

0	1	6	6	6	6
1	0	1	6	6	6
6	1	0	1	6	6
6	6	1	0	1	6
6	6	6	1	0	1
6	6	6	6	1	0

Output:

0	1	2	3	4	5
1	0	1	2	3	4
2	1	0	1	2	3
3	2	1	0	1	2
4	3	2	1	0	1
5	4	3	2	1	0

I have implemented the algorithm in function:

```
compute_shortest_path() {  
    for (k = 0; k < n; k++)  
    {  
        for (i = 0; i < n; i++)  
            for (j = 0; j < n; j++)  
                a[i*n + j] = MIN(a[i*n + j], a[i*n + k] + a[k*n + j]);  
    }  
}
```

Also, I have written a program to generate the adjacency matrix for any given number of vertices. I have tested the floyd's algorithm for various dimensions of matrix from 3 to 8000.

I have tested all the other combinations with the matrix dimensions of 2000. The serial time required for the floyd's algorithm with the input of 2000 dimensions matrix is: **25.85606 seconds**

So to reduce this time of execution, we can parallelize the code using OpenMP, MPI and OpenACC. So I have evaluate the code using following combinations:

- i. OpenMp
- ii. MPI
- iii. OpenAcc
- iv. OpenMp and MPI
- v. OpenACC and MPI

3. Implementation of Floyd's algorithm using OpenMP:

a. Approach to Solution:

In order to do high performance computing I have used the 'bridges' environment provided by XSDE. It has shared memory computer system with 2 Intel Xeon processor with 14 cores each. So we are having up to 28 cores for our high performance computing. OpenMP uses the shared memory between these cores to execute the given code in parallel thread with each thread executed on each core of the system.

In the function `compute_shortest_path()` there are three nested for loops which can be parallelize using OpenMP. I have checked if there exist dependency across the loop and used the OpenMP pragmas accordingly.

b. Solution Description:

In the function '`compute_shortest_path()`' I have used the OpenMP pragmas to parallelize the nested for loops.

```
#pragma omp parallel for private(k)
for (k = 0; k < n; k++)
{
    #pragma omp parallel for private(i,j)
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            a[i*n + j] = MIN(a[i*n + j], a[i*n + k] + a[k*n + j]);
}
```

Using the OpenMP batch job script **job_openmp.sh** I have executed the code for threads 1 to 28 on the matrix dimensions of 2000x2000.

You can execute the batch script using '`sbatch job_floyd_openmp.sh`' on the bridges. I have saved the output in the '`floyd_openmp.csv`' file after execution of the script automatically.

Output for input matrix of size 10:

```
sonawane@br005:~/program3/openmp
floyd_openmp.csv slurm-893923.out slurm-895152.out
[sonawane@r001 openmp]$ vi floyd_openmp.c
[sonawane@r001 openmp]$ ls
a.out          job_openmp.job  slurm-894901.out  slurm-895165.out
floyd_openmp.c  pgm1.sh         slurm-894903.out
floyd_openmp.csv  slurm-893923.out  slurm-895152.out
[sonawane@r001 openmp]$ module load pgi
[sonawane@r001 openmp]$ pgcc -mp floyd_openmp.c
[sonawane@r001 openmp]$ ./a.out
Enter the dimensions of matrix:
10
Threads=28 Total_time=0.002800 Performance=0.000714
output:
0 1 2 3 4 5 6 7 8 9
1 0 1 2 3 4 5 6 7 8
2 1 0 1 2 3 4 5 6 7
3 2 1 0 1 2 3 4 5 6
4 3 2 1 0 1 2 3 4 5
5 4 3 2 1 0 1 2 3 4
6 5 4 3 2 1 0 1 2 3
7 6 5 4 3 2 1 0 1 2
8 7 6 5 4 3 2 1 0 1
9 8 7 6 5 4 3 2 1 0
[sonawane@r001 openmp]$
```

c. Results:

i. Execution time and performance:

We have measured the performance by calculating speedup and GFLOPS required for the execution of the code for the given number threads and input size matrix.

$$\text{Speedup} = \text{Serial execution time} / \text{Parallel Execution time}$$

$$\text{GFLOPS} = 2 * N^3 / T$$

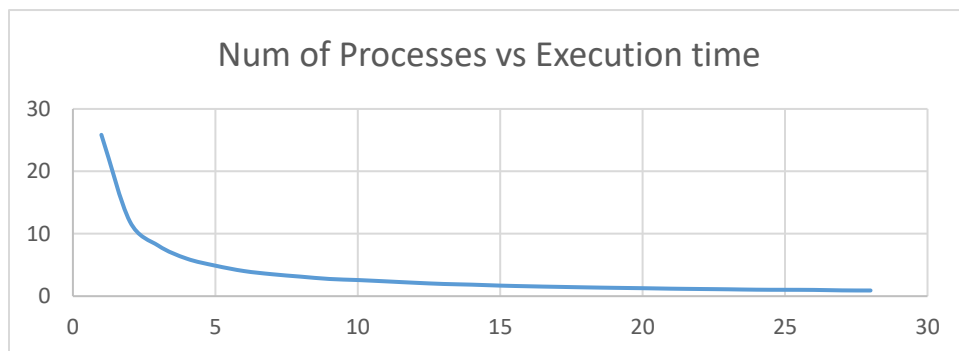
Where N = Matrix size and T = execution time

This is the table for OpenMP output for 1 to 28 threads for matrix size of 2000:

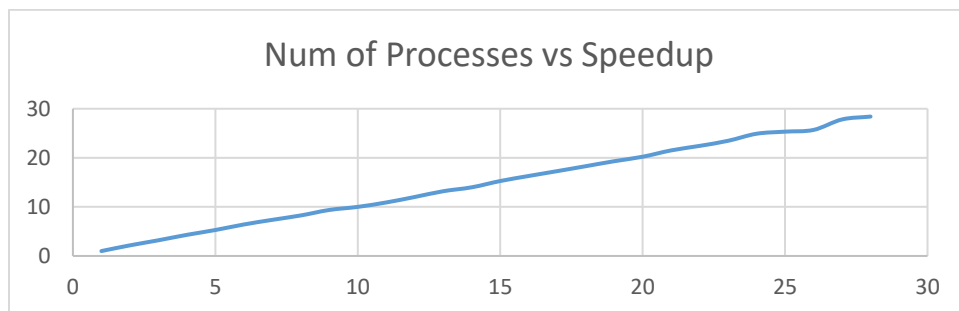
No of processes	Execution Time	Speedup	Performance
1	25.850606	1	0.618941
2	11.962874	2.160902639	1.337471
3	8.090748	3.195082334	1.977567
4	6.010443	4.300948532	2.662033
5	4.890169	5.286239801	3.271871
6	4.030612	6.413568461	3.96962

7	3.51158	7.361531277	4.556354
8	3.130989	8.256370751	5.110206
9	2.758176	9.372355499	5.800935
10	2.585202	9.999453041	6.189071
11	2.370715	10.90413905	6.749019
12	2.151186	12.01690881	7.437758
13	1.959246	13.19416041	8.166407
14	1.847936	13.98890762	8.658309
15	1.693085	15.26834506	9.450205
16	1.585832	16.30097387	10.089341
17	1.495588	17.28457704	10.698133
18	1.413548	18.28774545	11.319036
19	1.33892	19.30705793	11.94993
20	1.278664	20.21688731	12.513061
21	1.201679	21.51207269	13.314704
22	1.152467	22.43066916	13.883261
23	1.101627	23.46584279	14.523972
24	1.037914	24.90630823	15.415535
25	1.020553	25.32999854	15.677775
26	1.006469	25.68445327	15.897161
27	0.929287	27.81767742	17.217501
28	0.910198	28.40107977	17.578593

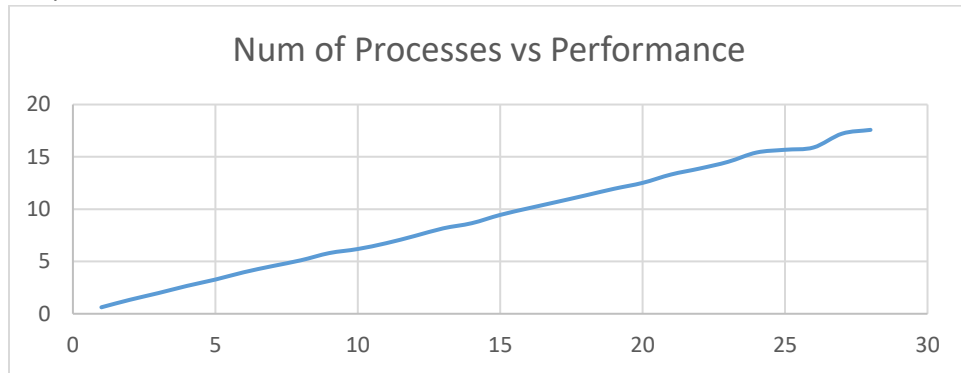
Graph for Number of threads vs Execution Time:



Graph for Number of Threads vs Speedup:



Graph for Number of threads vs Performance:



From the graph and the tables you can see that speedup as well as performance i.e GFLOPS is increased as we increased number of processes with 2000 as input matrix size:

The maximum speedup we could achieve: 28.40

The maximum Performance we could achieve: 17.61 for matrix size 2000

Output:

```
sonawane@br005:~/program3/openmp
srun: job 896473 queued and waiting for resources
srun: job 896473 has been allocated resources
[sonawane@r001 openmp]$ ./a.out
Enter the dimensions of matrix:
20^C
[sonawane@r001 openmp]$ ^C
[sonawane@r001 openmp]$ ^C
[sonawane@r001 openmp]$ export OMP_NUM_THREADS=28
[sonawane@r001 openmp]$ ./a.out
Enter the dimensions of matrix:
2000
Threads=28 Total_time=0.908513 Performance=17.611195
output:
[sonawane@r001 openmp]$ ./a.out
Enter the dimensions of matrix:
4000
Threads=28 Total_time=7.293440 Performance=17.550018
output:
[sonawane@r001 openmp]$ ./a.out
Enter the dimensions of matrix:
8000
Threads=28 Total_time=59.989048 Performance=17.069782
output:
[sonawane@r001 openmp]$
```

ii. Performance measure using Gustafson – Barsis’s law and Karp – Flatt metric:

Gustafson Barsis’s law:

Karp_Flatt metric:

$$\psi \leq p + (1 - p)s$$

$$e = \frac{1/\psi - 1/p}{1 - 1/p}$$

Here we have considered that s= 10% i.e 10% of the code executes serially. P is number of total processes used for the execution. In our case it is from 1 to 28.

Table for Gustafson-Barsis’s law and Karp-Flatt metric:

P	Ψ	E
2	1.9	0.052632
3	2.8	0.035714
4	3.7	0.027027
5	4.6	0.021739
6	5.5	0.018182
7	6.4	0.015625
8	7.3	0.013699
9	8.2	0.012195
10	9.1	0.010989
11	10	0.01
12	10.9	0.009174
13	11.8	0.008475
14	12.7	0.007874
15	13.6	0.007353
16	14.5	0.006897
17	15.4	0.006494
18	16.3	0.006135
19	17.2	0.005814
20	18.1	0.005525
21	19	0.005263
22	19.9	0.005025
23	20.8	0.004808
24	21.7	0.004608
25	22.6	0.004425
26	23.5	0.004255
27	24.4	0.004098
28	25.3	0.003953

4. Implementation of Floyd's using OpenACC:

a. Approach to the solution:

In order to do high performance computing I have used the 'bridges' environment provided by XSDE. Bridges GPU nodes have the 16 GPU nodes.

Each phase 1 GPU nodes contains

- a. 2 NVIDIA K80 GPUs
- b. 2 Intel Xeon E5-2695 v3 CPUs, each with 14 cores
- c. 128GB RAM

In the function `compute_shortest_path()` of the given code ww I can use the OpenACC pragmas in order to accelerate the execution of algorithm. In those identified sections I have checked if there exist any dependency across the loop if we parallelize the code. Then according to the scenario I have used the OpenACC pragmas for the given sections.

After using pragmas I compiled the given code with `-Minfo=accel` to see where I need to use more pragmas in order to parallelize its execution.

b. Solution Description:

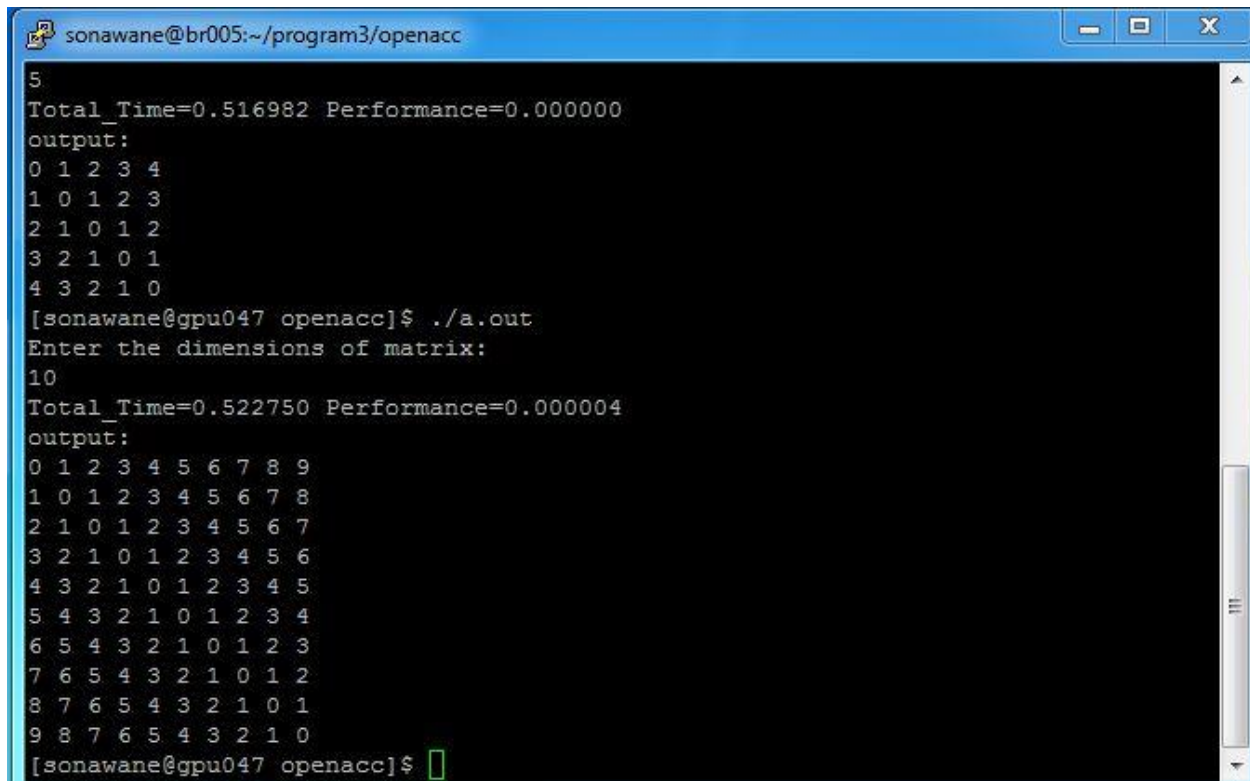
In the function `compute_shortest_path()` I have used the OpenACC pragmas in order to accelerate the execution of algorithm.

```
#pragma acc data copy(a[0:n*n])
{
  for (k = 0; k < n; k++)
  {
    #pragma acc parallel
    {
      #pragma acc loop independent
      for (i = 0; i < n; i++){
        #pragma acc loop independent
        for (j = 0; j < n; j++)
          a[i*n+j] = MIN(a[i*n+j], a[i*n+k] + a[k*n+j]);
      }
    }
  }
}
```

As we have three nested for loops here, we have to use '`#pragma acc loop auto`' in order to correct execution of all the loops with '`#pragma acc kernels`'. Also we have to use the '`#pragma acc data`' clause in order to minimize the data transfer between host and device.

I have ran the code in batch mode for 1 GPU using the job script for 'GPU Partition'. You can run the script as 'sbatch job_openacc_part.sh' in batch mode and you will get output in slurm-xxx.out.

Output of OpenACC for matrix size 10:



```
sonawane@br005:~/program3/openacc
5
Total_Time=0.516982 Performance=0.000000
output:
0 1 2 3 4
1 0 1 2 3
2 1 0 1 2
3 2 1 0 1
4 3 2 1 0
[sonawane@gpu047 openacc]$ ./a.out
Enter the dimensions of matrix:
10
Total_Time=0.522750 Performance=0.000004
output:
0 1 2 3 4 5 6 7 8 9
1 0 1 2 3 4 5 6 7 8
2 1 0 1 2 3 4 5 6 7
3 2 1 0 1 2 3 4 5 6
4 3 2 1 0 1 2 3 4 5
5 4 3 2 1 0 1 2 3 4
6 5 4 3 2 1 0 1 2 3
7 6 5 4 3 2 1 0 1 2
8 7 6 5 4 3 2 1 0 1
9 8 7 6 5 4 3 2 1 0
[sonawane@gpu047 openacc]$
```

C. Results:

i. Execution time and performance:

We have measured the performance by calculating speedup and GFLOPS required for the execution of the code for the given number threads and input size matrix.

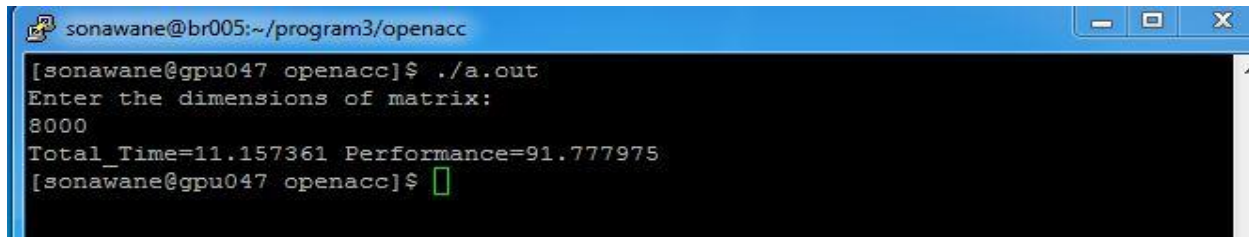
$$\text{Speedup} = \text{Serial execution time} / \text{Parallel Execution time}$$

$$\text{GFLOPS} = 2 * N^3 / T$$

Where N = Matrix size and T = execution time

When I ran the code for 1 GPU using batch script of GPU partition, the execution time I required for 8000 matrix size is: 11.157361

Maximum Performance I got: 91.777975 GFLOPS



```
sonawane@br005:~/program3/openacc
[sonawane@gpu047 openacc]$ ./a.out
Enter the dimensions of matrix:
8000
Total_Time=11.157361 Performance=91.777975
[sonawane@gpu047 openacc]$
```

5. Implementation of Floyd's algorithm using MPI:

a. Approach to the solution:

In order to optimize the algorithm further I have used the MPI library on the XSEDE's bridge environment. It has 752 RSM nodes: HPE Apollo 2000s, with 2 Intel Xeon E5-2695 v3 CPUs (14 cores per CPU), 128GB RAM and 8TB on-node storage.

In order to use the MPI with the Floyd's program we have to divide the given input matrix row wise and send them to each node for further processing.

Suppose we have matrix of size 'n' and we have 'p' nodes. Then each node will process n/p rows of matrix.

After processing collect the result from all the nodes and then display the output.

b. Solution Description:

First we have to broadcast the dimension of matrix 'n' to all other nodes using:

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Then scatter all the n/p rows to the p nodes using MPI_Scatter function:

```
MPI_Scatter(arr, n * (n/p), MPI_INT, local_arr, n * (n/p), MPI_INT, 0, MPI_COMM_WORLD);
```

Then in the compute_shortest_path() broadcast every 'k' row to all nodes for the processing

```
MPI_Bcast (tmp, n, MPI_INT, root, MPI_COMM_WORLD);
```

After processing of Floyd algorithm, on every node collect the output from every node before displaying the output.

```
MPI_Gather(local_arr, n * (n/p), MPI_INT, temp_arr, n * (n/p), MPI_INT, 0, MPI_COMM_WORLD);
```

Then using the prgm3_mpi.sh script run the algorithm for nodes 1,2,4,5,8,10,16,20 to calculate the execution time and performance for the implemented algorithm.

Run the script using 'sbatch job_mpi.sh'

Output for matrix size of 10 with MPI:

```

sonawane@br005:~/program3/openmp
[sonawane@br005 mpi]$ mpicc floyd_mpi.c -lm
[sonawane@br005 mpi]$ mpirun -n 5 ./a.out 10
Processes=5 Total_time= 0.000074 Performance=0.027060
output:
0 1 2 3 4 5 6 7 8 9
1 0 1 2 3 4 5 6 7 8
2 1 0 1 2 3 4 5 6 7
3 2 1 0 1 2 3 4 5 6
4 3 2 1 0 1 2 3 4 5
5 4 3 2 1 0 1 2 3 4
6 5 4 3 2 1 0 1 2 3
7 6 5 4 3 2 1 0 1 2
8 7 6 5 4 3 2 1 0 1
9 8 7 6 5 4 3 2 1 0
[sonawane@br005 mpi]$

```

c. Results

i. Execution time and performance:

We have measured the performance by calculating speedup and GFLOPS required for the execution of the code for the given number threads and input size matrix.

$$\text{Speedup} = \text{Serial execution time} / \text{Parallel Execution time}$$

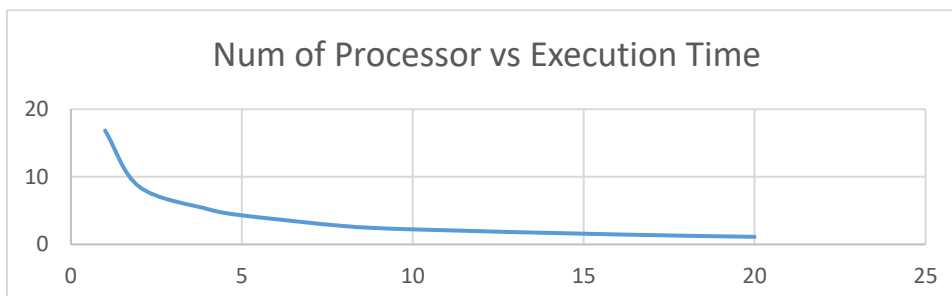
$$\text{GFLOPS} = 2 * N^3 / T$$

Where N = Matrix size and T = execution time

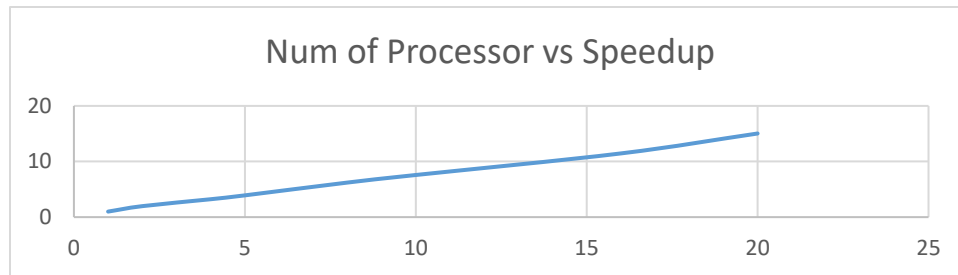
I ran the code for processes 1,2,4,5,8,10,16 and 20 for 2000 matrix size:

Num of processor	Execution Time	Speedup	Performance
1	16.833182	1	0.950504
2	8.526014	1.974331968	1.87661
4	5.228562	3.219466844	3.060115
5	4.302179	3.912710745	3.719046
8	2.707731	6.216711335	5.909006
10	2.22151	7.577360444	7.202309
16	1.469348	11.45622548	10.889184
20	1.119258	15.03959051	14.295183

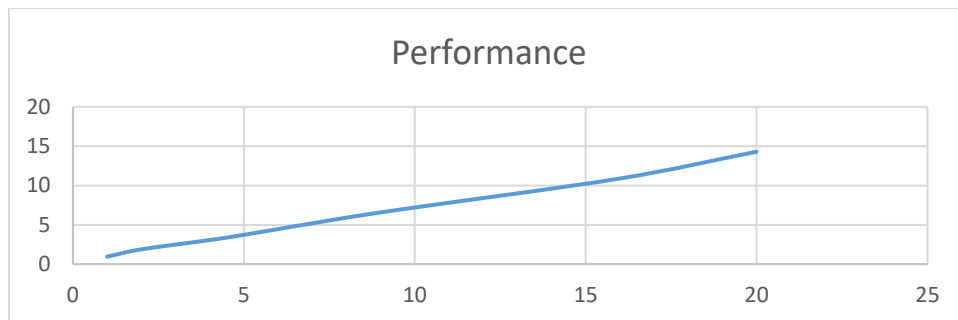
Graph for number of nodes vs Execution time:



Graph for number of nodes vs Speedup:



Graph for number of nodes vs Performance:



You can see from table and graph that performance and speedup is increased as we increased number of nodes.

Maximum Performance I got for matrix size 2000 which is 14.29 GFLOPS for 20 processes with execution time 1.11 seconds.

```
sonawane@br005:~/program3/openmp
[sonawane@br005 mpi]$ mpirun -n 28 ./a.out 4000
Processes=28 Total_time= 12.233580 Performance=10.463004
[sonawane@br005 mpi]$ mpirun -n 28 ./a.out 2000
Processes=28 Total_time= 1.645854 Performance=9.721397
[sonawane@br005 mpi]$ mpirun -n 40 ./a.out 2000
Processes=40 Total_time= 2.187592 Performance=7.313978
[sonawane@br005 mpi]$ mpirun -n 40 ./a.out 4000
Processes=40 Total_time= 16.775896 Performance=7.629995
[sonawane@br005 mpi]$ mpirun -n 80 ./a.out 8000
Processes=80 Total_time=101.619277 Performance=10.076828
[sonawane@br005 mpi]$ mpirun -n 80 ./a.out 2000
Processes=80 Total_time= 1.672269 Performance=9.567839
[sonawane@br005 mpi]$ mpirun -n 40 ./a.out 2000
Processes=40 Total_time= 2.125300 Performance=7.528349
[sonawane@br005 mpi]$ mpirun -n 20 ./a.out 4000
^C[mpiexec@br005.pvt.bridges.psc.edu] Sending Ctrl-C to processes
[mpiexec@br005.pvt.bridges.psc.edu] Press Ctrl-C again to force ab
[sonawane@br005 mpi]$ ^C
[sonawane@br005 mpi]$ ^C
[sonawane@br005 mpi]$ mpirun -n 40 ./a.out 4000
Processes=40 Total_time= 16.741706 Performance=7.645577
[sonawane@br005 mpi]$ mpirun -n 80 ./a.out 4000
```

ii. **Performance measure using Gustafson – Barsis’s law and Karp – Flatt metric:**

Gustafson Barsis’s law:

$$\psi \leq p + (1 - p)s$$

Karp_Flatt metric:

$$e = \frac{1/\psi - 1/p}{1 - 1/p}$$

Here we have considered that s= 10% i.e 10% of the code executes serially. ‘P’ is number of total processes used for the execution. In our case it is the number of nodes that we have used for the processing.

Table for Gustafson-Barsis’s law and Karp-Flatt metric:

p	Ψ	e
2	1.9	0.052632
4	3.7	0.027027
5	4.6	0.021739
8	7.3	0.013699
10	9.1	0.010989
16	14.5	0.006897
20	18.1	0.005525

6. Implementation using OpenMP and MPI:

a. Approach to the solution:

We can use the bridges environment provided by XSEDE. It has 752 RSM nodes: HPE Apollo 2000s, with 2 Intel Xeon E5-2695 v3 CPUs (14 cores per CPU), 128GB RAM and 8TB on-node storage. Every node can be used using the MPI pragmas and 28 cores at every node can be used using OpenMP pragmas.

The implementation of MPI pragmas will be same as previous. We just need to add the OpenMP pragmas in the `compute_shortest_path()` function.

b. Solution Description:

We have to manage the matrix data same way as we manage in the previous case of Floyd using MPI.

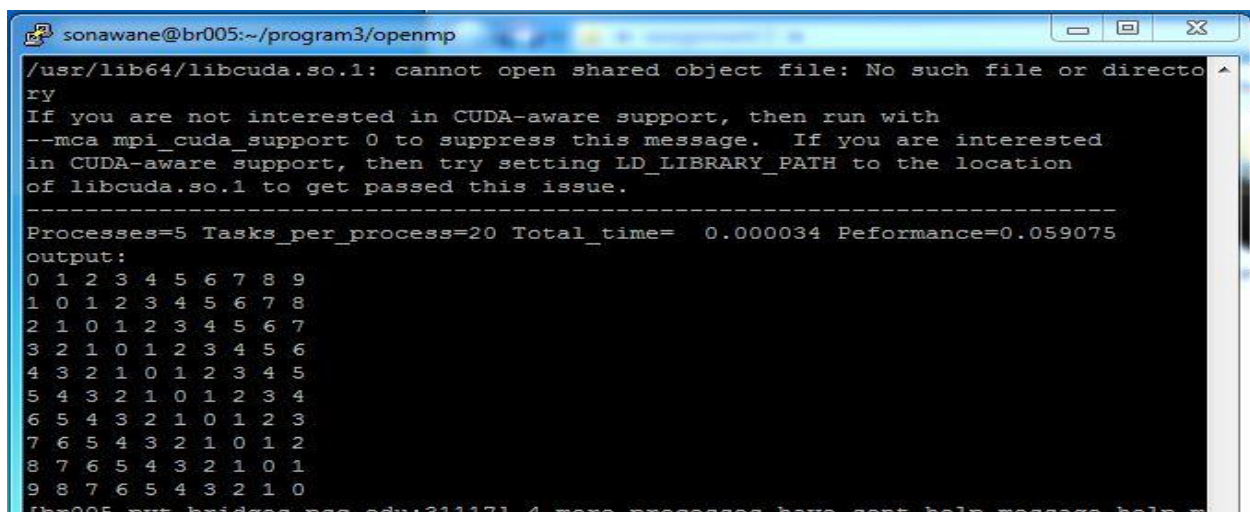
We just need to add the OpenMP pragma to the algorithm function in order to use the cores at the every node.

```
MPI_Bcast(tmp, n, MPI_INT, root, MPI_COMM_WORLD);
#pragma omp parallel for private(i,j)
for (i = 0; i < n/p; i++)
    for (j = 0; j < n; j++)
        a[i*n+j] = MIN(a[i*n+j], a[i*n+k] + tmp[j]);
}
```

Then I executed the code on bridge using the batch script. In the script I ran the code using `mpirun` for 1,2,4,5,8,10,16 and 20 nodes. Every node then ran for the 1 to 28 cores in the script.

You can run the script using '`sbatch job_openmpi.sh`'. The output will be saved in the .csv file.

Output for matrix size of 10 with MPI and OpenMP:



```
sonawane@br005:~/program3/openmp
/usr/lib64/libcuda.so.1: cannot open shared object file: No such file or directory
If you are not interested in CUDA-aware support, then run with
--mca mpi_cuda_support 0 to suppress this message.  If you are interested
in CUDA-aware support, then try setting LD_LIBRARY_PATH to the location
of libcuda.so.1 to get passed this issue.
-----
Processes=5 Tasks_per_process=20 Total_time= 0.000034 Performance=0.059075
output:
0 1 2 3 4 5 6 7 8 9
1 0 1 2 3 4 5 6 7 8
2 1 0 1 2 3 4 5 6 7
3 2 1 0 1 2 3 4 5 6
4 3 2 1 0 1 2 3 4 5
5 4 3 2 1 0 1 2 3 4
6 5 4 3 2 1 0 1 2 3
7 6 5 4 3 2 1 0 1 2
8 7 6 5 4 3 2 1 0 1
9 8 7 6 5 4 3 2 1 0
[br005.pvt.bridges.psc.edu:31117] 4 more processes have sent help message help-m
```

c. Results:

I. Execution time and performance:

We have measured the performance by calculating speedup and GFLOPS required for the execution of the code for the given number threads and input size matrix.

$$\text{Speedup} = \text{Serial execution time} / \text{Parallel Execution time}$$

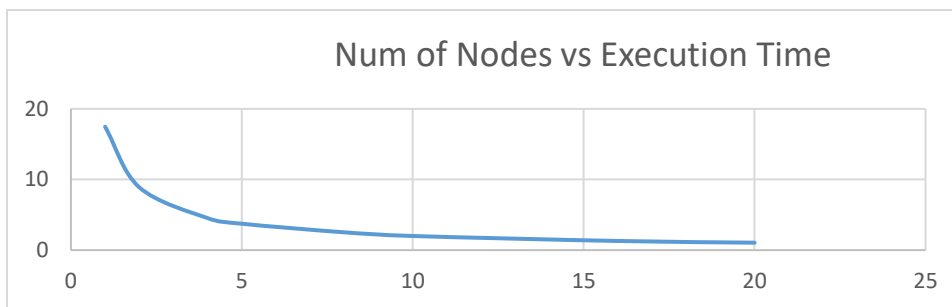
$$\text{GFLOPS} = 2 * N^3 / T$$

Where N = Matrix size and T = execution time

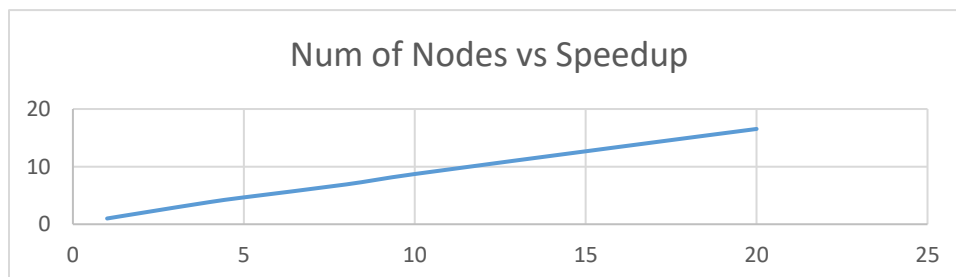
I ran the code with nodes 1,2,4,5,8,10,16 and 20 for 28 OpenMP Threads. The matrix size used for this is 2000:

Num of Nodes	OpenMP Threads	Execution Time	Speedup	Performance
1	28	17.47204	0.997424456	1.915749
2	28	8.893015	1.959632363	2.799165
4	28	4.53175	3.845543112	4.530645
5	28	3.739279	4.660534825	5.2789
8	28	2.519519	6.916812296	7.350419
10	28	2.000056	8.713276028	8.999776
16	28	1.296756	13.43895074	13.338481
20	28	1.053684	16.53915216	16.184818

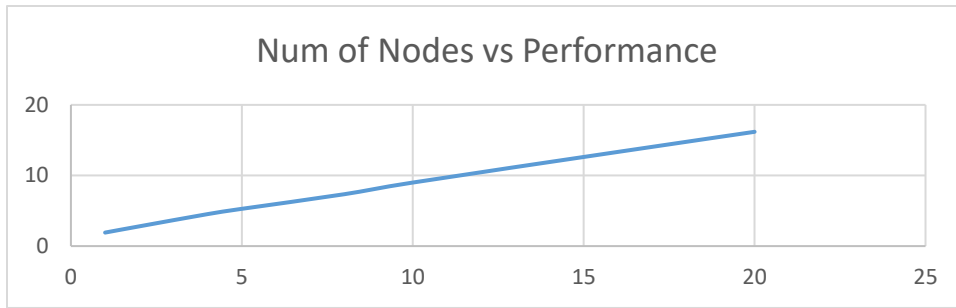
Graph for Number of nodes vs Execution time at OpenMP threads of 28:



Graph for Number of nodes vs Speedup at OpenMP threads of 28



Graph for Number of nodes vs Performance at OpenMP threads of 28:



Maximum performance I got is 16.18 GFLOPS with execution time 1.0536 seconds for matrix input size of 2000 and OpenMP threads 28.

```

sonawane@br005:~/program3/openmp
-----
[br005.pvt.bridges.psc.edu:10289] 39 more processes have sent help message help-
mpi-common-cuda.txt / dlopen failed
[br005.pvt.bridges.psc.edu:10289] Set MCA parameter "orte_base_help_aggregate" t
o 0 to see all help / error messages
Processes=40 Tasks_per_process=28 Total_time= 10.449025 Performance=12.249947
[sonawane@br005 mpi_openmp]$ mpirun -n 20 ./a.out 4000
-----
The library attempted to open the following supporting CUDA libraries,
but each of them failed.  CUDA-aware support is disabled.
libcuda.so.1: cannot open shared object file: No such file or directory
/usr/lib64/libcuda.so.1: cannot open shared object file: No such file or directo
ry
If you are not interested in CUDA-aware support, then run with
--mca mpi_cuda_support 0 to suppress this message.  If you are interested
in CUDA-aware support, then try setting LD_LIBRARY_PATH to the location
of libcuda.so.1 to get passed this issue.
-----
[br005.pvt.bridges.psc.edu:10813] 19 more processes have sent help message help-
mpi-common-cuda.txt / dlopen failed
[br005.pvt.bridges.psc.edu:10813] Set MCA parameter "orte_base_help_aggregate" t
o 0 to see all help / error messages
Processes=20 Tasks_per_process=28 Total_time=  7.975849 Performance=16.048448
[sonawane@br005 mpi_openmp]$

```

II. Performance measure using Gustafson – Barsis’s law and Karp – Flatt metric:

Gustafson Barsis’s law:

$$\psi \leq p + (1 - p)s$$

Karp_Flatt metric:

$$e = \frac{1/\psi - 1/p}{1 - 1/p}$$

Here we have considered that s= 10% i.e 10% of the code executes serially.

'P' is number of total processes used for the execution. In our case it is the number of nodes multiply by the number of cores in every node:

$P = \text{number of nodes} * \text{number of cores each node}$

Here we have calculated the 'p' using number of nodes = 20

Table for Gustafson-Barsis's law and Karp-Flatt metric:

Number of cores	p	Ψ	e
2	40	36.1	0.00277
3	60	54.1	0.001848
4	80	72.1	0.001387
5	100	90.1	0.00111
6	120	108.1	0.000925
7	140	126.1	0.000793
8	160	144.1	0.000694
9	180	162.1	0.000617
10	200	180.1	0.000555
11	220	198.1	0.000505
12	240	216.1	0.000463
13	260	234.1	0.000427
14	280	252.1	0.000397
15	300	270.1	0.00037
16	320	288.1	0.000347
17	340	306.1	0.000327
18	360	324.1	0.000309
19	380	342.1	0.000292
20	400	360.1	0.000278
21	420	378.1	0.000264
22	440	396.1	0.000252
23	460	414.1	0.000241
24	480	432.1	0.000231
25	500	450.1	0.000222
26	520	468.1	0.000214
27	540	486.1	0.000206
28	560	504.1	0.000198

7. Implementation using OpenACC and MPI:

a. Approach to the solution:

We can use the bridges environment provided by XSEDE. It has 752 RSM nodes: HPE Apollo 2000s, with 2 Intel Xeon E5-2695 v3 CPUs (14 cores per CPU), 128GB RAM and 8TB on-node storage. Every node can be used using the MPI pragmas and GPU can be used using OpenACC pragmas.

The implementation of MPI pragmas will be same as previous. We just need to add the OpenACC pragmas in the compute_shortest_path() function.

b. Solution Description:

We have to manage the matrix data same way as we manage in the previous case of Floyd using MPI.

We just need to add the OpenACC pragma to the algorithm function in order to use the cores at the every node.

```
#pragma acc data copy(a[0:n*n/p]), create(tmp[n])
{
  for (k = 0; k < n; k++)
  { root = BLOCK_OWNER(k, p, n);
    if (root == id)
    { offset = k - BLOCK_LOW(id, p, n);
      for (j = 0; j < n; j++)
        tmp[j] = a[offset*n+j];
    }
  }

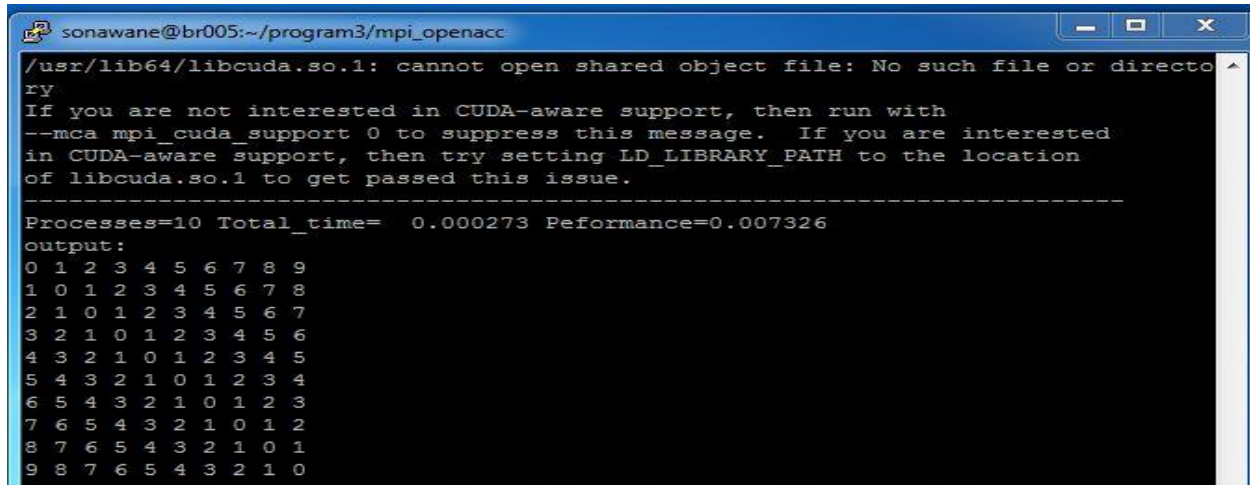
  #pragma acc update host(tmp)
  {
    MPI_Bcast (tmp, n, MPI_INT, root, MPI_COMM_WORLD);
  }

  #pragma acc update device(tmp)
  #pragma acc parallel
  {
    #pragma acc loop independent
    for (i = 0; i < n/p; i++){
      #pragma acc loop independent
      for (j = 0; j < n; j++)
        a[i*n+j] = MIN(a[i*n + j], a[i*n+k] + tmp[j]);
    }
  }
}
}
```

Then I executed the code on bridge using the batch script. In the script I ran the code using mpirun for 1,2,4,5,8,10,16 and 20 nodes with the one GPU.

You can run the script using 'sbatch job_accmpi.sh'. The output will be saved in the .csv file.

Output for matrix size 10 with MPI and OpenACC



```
sonawane@br005:~/program3/mpi_openacc
/usr/lib64/libcuda.so.1: cannot open shared object file: No such file or directory
If you are not interested in CUDA-aware support, then run with
--mca mpi_cuda_support 0 to suppress this message.  If you are interested
in CUDA-aware support, then try setting LD_LIBRARY_PATH to the location
of libcuda.so.1 to get passed this issue.
-----
Processes=10 Total_time= 0.000273 Performance=0.007326
output:
0 1 2 3 4 5 6 7 8 9
1 0 1 2 3 4 5 6 7 8
2 1 0 1 2 3 4 5 6 7
3 2 1 0 1 2 3 4 5 6
4 3 2 1 0 1 2 3 4 5
5 4 3 2 1 0 1 2 3 4
6 5 4 3 2 1 0 1 2 3
7 6 5 4 3 2 1 0 1 2
8 7 6 5 4 3 2 1 0 1
9 8 7 6 5 4 3 2 1 0
```

c. Results:

i. Execution time and performance:

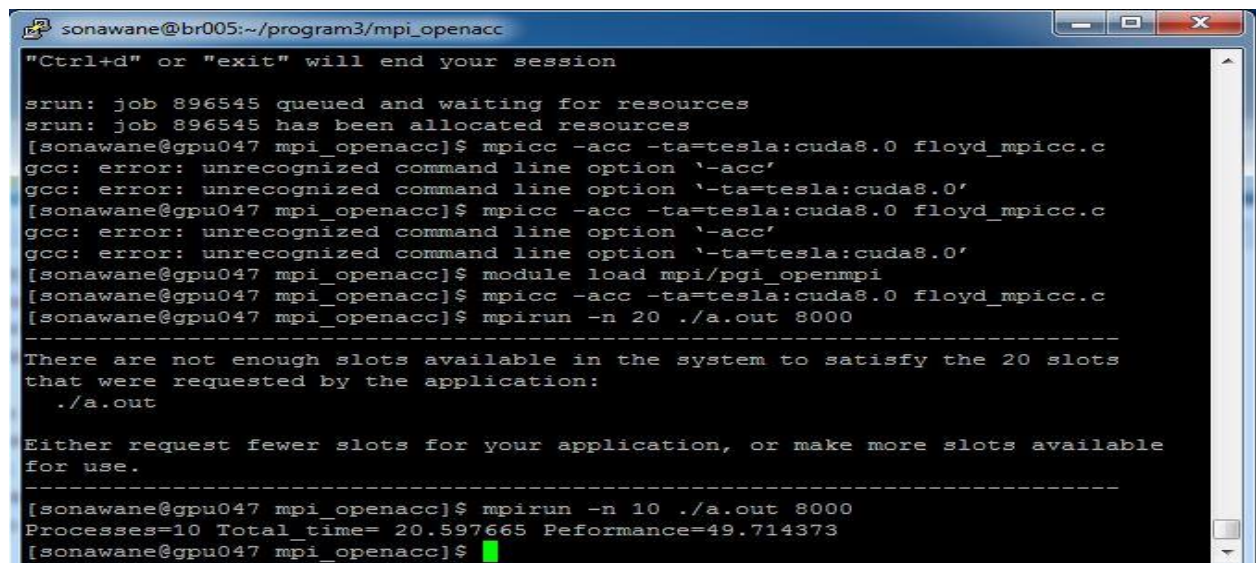
We have measured the performance by calculating speedup and GFLOPS required for the execution of the code for the given number threads and input size matrix.

Speedup = Serial execution time / Parallel Execution time

GFLOPS = $2 * N^3 / T$

Where N = Matrix size and T = execution time

Maximum performance I got is 49.71 GFLOPS with execution time of 20.59 seconds for the matrix size 8000.



```
sonawane@br005:~/program3/mpi_openacc
"Ctrl+d" or "exit" will end your session
srun: job 896545 queued and waiting for resources
srun: job 896545 has been allocated resources
[sonawane@gpu047 mpi_openacc]$ mpicc -acc -ta=tesla:cuda8.0 floyd_mpicc.c
gcc: error: unrecognized command line option '-acc'
gcc: error: unrecognized command line option '-ta=tesla:cuda8.0'
[sonawane@gpu047 mpi_openacc]$ mpicc -acc -ta=tesla:cuda8.0 floyd_mpicc.c
gcc: error: unrecognized command line option '-acc'
gcc: error: unrecognized command line option '-ta=tesla:cuda8.0'
[sonawane@gpu047 mpi_openacc]$ module load mpi/pgi_openmpi
[sonawane@gpu047 mpi_openacc]$ mpicc -acc -ta=tesla:cuda8.0 floyd_mpicc.c
[sonawane@gpu047 mpi_openacc]$ mpirun -n 20 ./a.out 8000
-----
There are not enough slots available in the system to satisfy the 20 slots
that were requested by the application:
./a.out
Either request fewer slots for your application, or make more slots available
for use.
-----
[sonawane@gpu047 mpi_openacc]$ mpirun -n 10 ./a.out 8000
Processes=10 Total_time= 20.597665 Performance=49.714373
[sonawane@gpu047 mpi_openacc]$
```