The University of Texas at Dallas

CS 6376

PARALLEL PROCESSING

Programming Assignment 4

Anay Sonawane (ass151130)

1. Problem Statement:

To implement the Cannon'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 must 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 Cannon's algorithm using OpenMP:

a. Approach to the 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.

I have implemented the Cannon's algorithm using OpemMP threads. Suppose I have used p threads, then every OpenMP thread will find the matrix multiplication of $N/\sqrt{p}$ size of input matrices. Where N is the size of input matrix. So to implement Cannon's algorithm I have to use 1, 4 or 16 number of OpenMP threads, because the number should be perfect square.

b. Solution Description:

First I aligned both the input matrices as per the requirement by the Cannons Algorithm. Then I passed the matrix to the function matrix_mult() for the further processing. There I divided the matrix into size size of $N/\sqrt{p}$ blocks, and allot each block to the OpenMP thread, according to its thread number. Then I carried out submatrices multiplication and fill the partial product into the Output Matrix. After that I left shifted the input matrix A and Up shifted the input matrix B. Then again call the matrix_mult() for multiplication of new shifted matrices. Repeat this process for $\sqrt{p}$ times and store the result in output matrix C.

I have used OpenMP pragmas in matrix_mult() function to parallelize the process using OpenMP thread.

```
#pragma omp parallel default(none) private(..) shared(..) num_threads(threads)
{
    /*
        Cannon algorithm
    */
}
```

num_threads(threads) will use the 'threads' number of OpenMP threads. Using 'job_openmp.job' script I have executed the code for 1,4 and 16 thread for various matrix dimensions.

You can execute the batch script using 'sbatch job_openmp.job' on the bridges. I have saved the output in the 'openmp.csv' file after execution of the script automatically.

c. Results:

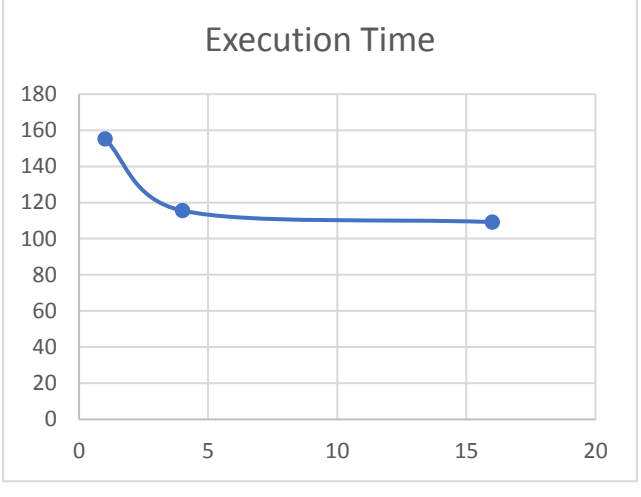
i. Execution time and performance:

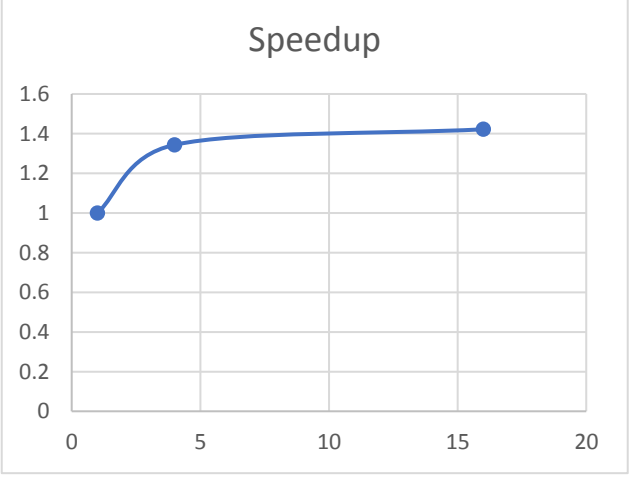
I have measured the performance by calculating speedup and GFLOPS required for the execution of the code for the given number threads and input size matrix.

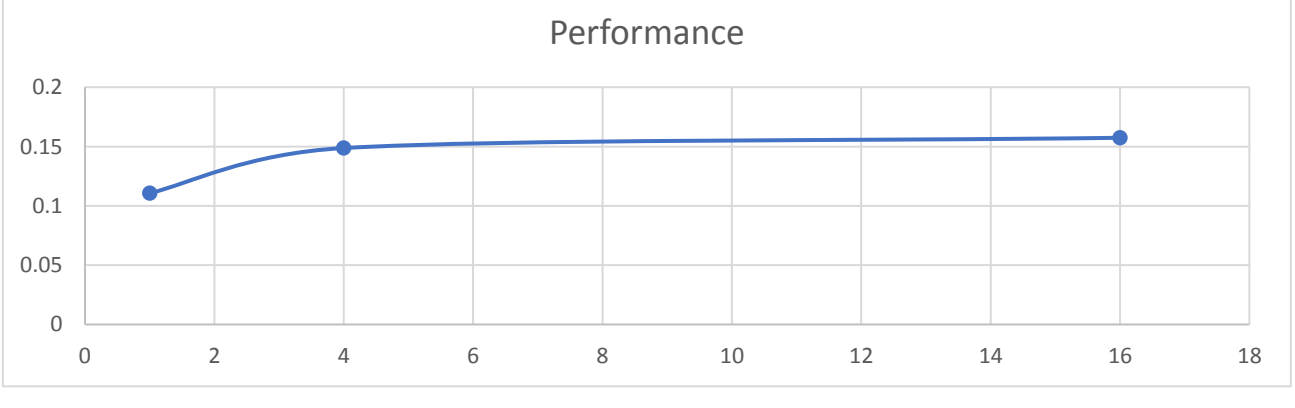
Speedup = Serial execution time / Parallel Execution time
GFLOPS = 2* N^3/T
Where N = Matrix size and T = execution time

I ran the code for processes 1,4 and 16 as only processes with square number supported for this algorithm. I ran this code on various size of matrices. Here is the results for matrix size of 2048.

OpenMP threads	Execution Time	Performance	Speedup
1	155.237914	0.110668	1
4	115.531129	0.148703	1.343689059
16	109.17408	0.157362	1.421930132







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 2048 as input matrix size.

The maximum Performance we could achieve: 0.504183 for matrix size 300 with OpenMP threads 4.

ii. Performance measure using Gustafosn – 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= 25% i.e 25% of the code executes serially. P is number of total processes used for the execution. In our case it is from 1,4 and 16.

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

Num of Processors	gustafson	Karp-flatt
4	3.25	0.076923
16	12.25	0.020408

3. Implementation of Cannon'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.
As given in Cannons algorithm, first align both the input matrices. In order to use the MPI with the Cannon's program we have to divide the given input matrices A and B in to the square sub matrices of each size n/√p. Where n is the order of matrix while p is number of nodes in MPI. Then send the submatrices of A and B both to each node. Each node then calculates the product of submatrices and returns the result in submatrices of output C. Then left shift the whole matrix A and up shift the whole matrix B. Iterate this algorithm for √p times.

b. Solution Description:

Initially I divide the matrix A and B into submatrices of size n/√p, where p is the total number of nodes used in the MPI. Then I allotted them to every node according to the rank of the node.

I have used the cartesian topology to align, left shift and right shift of matrices among the MPI nodes. To do the communication of matrices we have to create the new communicator to which the topology information is attached.

MPI_Cart_create(MPI_COMM_WORLD,2,dims,periods,1,&comm_2d);

Then using the rank get the coordinates in cartesian topology:
MPI_Comm_rank(comm_2d, &my2drank);
MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

Then compute the ranks of up and left submatrices in the topology using:
MPI_Cart_shift(comm_2d, 1, -1, &rightrank, &leftrank);
MPI_Cart_shift(comm_2d, 0, -1, &downrank, &uprank);

Then I have done the initial alignment of matrices A and B using MPI_Cart_shift and MPI_Sendrcv_replace:
MPI_Cart_shift(comm_2d, 1, -mycoords[0], &shiftsource,&shiftdest);
MPI_Sendrecv_replace(A_Block[id], nlocal*nlocal, MPI_INT, shiftdest, 1,shiftsource, 1, comm_2d, &status);
MPI_Cart_shift(comm_2d, 0, -mycoords[1], &shiftsource,&shiftdest);
MPI_Sendrecv_replace(B_Block[id], nlocal*nlocal, MPI_INT, shiftdest, 1,shiftsource, 1, comm_2d, &status);

Then I iterate the following algorithm for √p times to find the multiplication of input matrices

i. product of matrix***: matrix_mult(nlocal,A_Block[id],B_Block[id],C_Block[id]);***
ii. Left shift the matrix A by nlocal size:
MPI_Sendrecv_replace(A_Block[id], nlocal*nlocal, MPI_INT, leftrank,1, rightrank, 1, comm_2d, &status);
iii. Up shift the matrix B by nlocal size:
MPI_Sendrecv_replace(B_Block[id], nlocal*nlocal, MPI_INT, uprank, 1, downrank, 1, comm_2d, &status);

After this loop store the output from each block in the output matrix C.

Then using the 'job_mpi.job' script run the algorithm for nodes 1,4 and 16 for various size of matrices to calculate the execution time and performance for the implemented algorithm. Algorithm, stores the output in mpi.csv file.
Run the script using 'sbatch job_mpi.job'.

c. Results

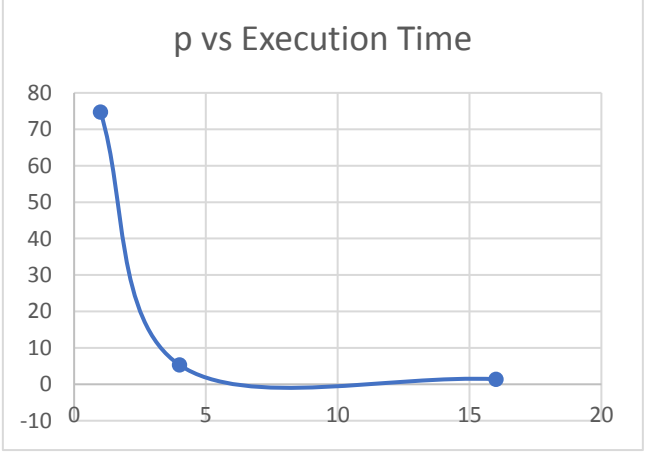
i. Execution time and performance:

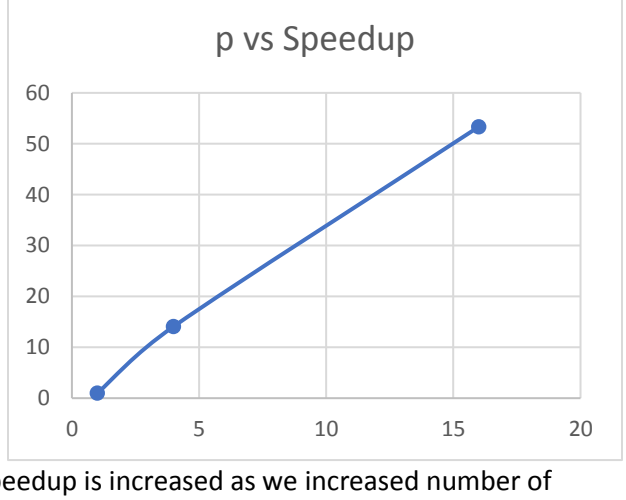
We have measured the performance by calculating speedup and GFLOPS required for the execution of the code for the given number threads and input size matrix.

Speedup = Serial execution time / Parallel Execution time
GFLOPS = 2* N^3/T
Where N = Matrix size and T = execution time

I ran the code for processes 1,4 and 16 as only processes with square number supported for this algorithm. I ran this code on various size of matrices. Here is the results for matrix size of 2048.

Number of processes	Execution Time	Performance	Speedup
1	74.75206	0.229825	1
4	5.304094	3.238983	14.09328
16	1.401306	12.2599	53.34457





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 14.040 GFLOPS for 26 processes with execution time 1.139548 seconds.

ii. **Performance measure using Gustafosn – 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= 25% i.e 25% 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:

Num of Processors	gustafson	Karp-flatt
4	3.25	0.076923
16	12.25	0.020408

4. Implementation of Cannon's algorithm using MPI and OpenMP:

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 matrix_mult() function.

b. Solution Description:

We have to manage the matrix data same way as we manage in the previous case of Cannon using MPI. We just need to add the OpenMP pragma to the matrix_mult() function in order to use the cores at the every node.

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

Then I executed the code on bridge using the batch script. In the script, I ran the code using mpirun for 1,4 and 16 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 openmpi.csv file.

c. Results:

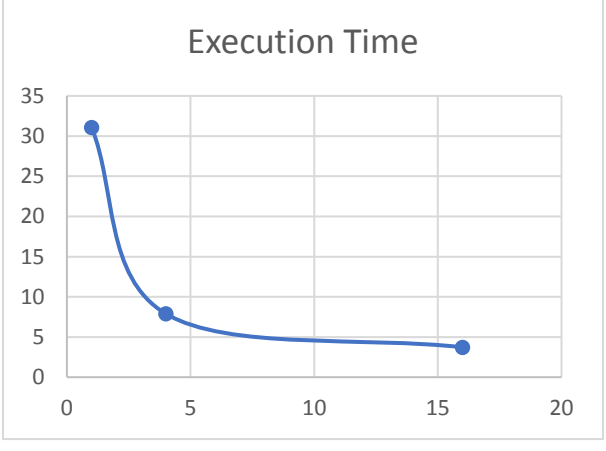
i. Execution time and performance:

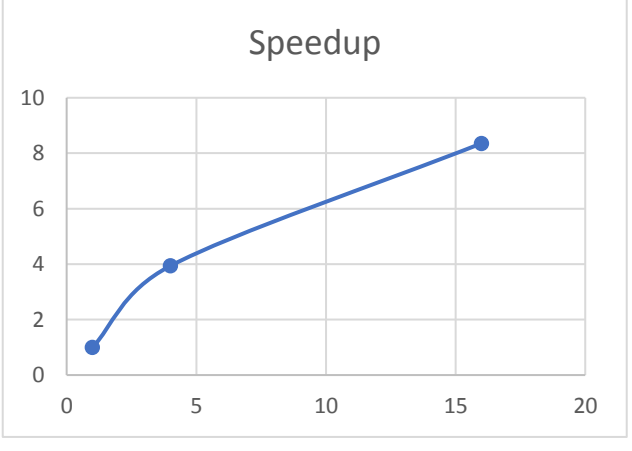
I have measured the performance by calculating speedup and GFLOPS required for the execution of the code for the given number threads and input size matrix.

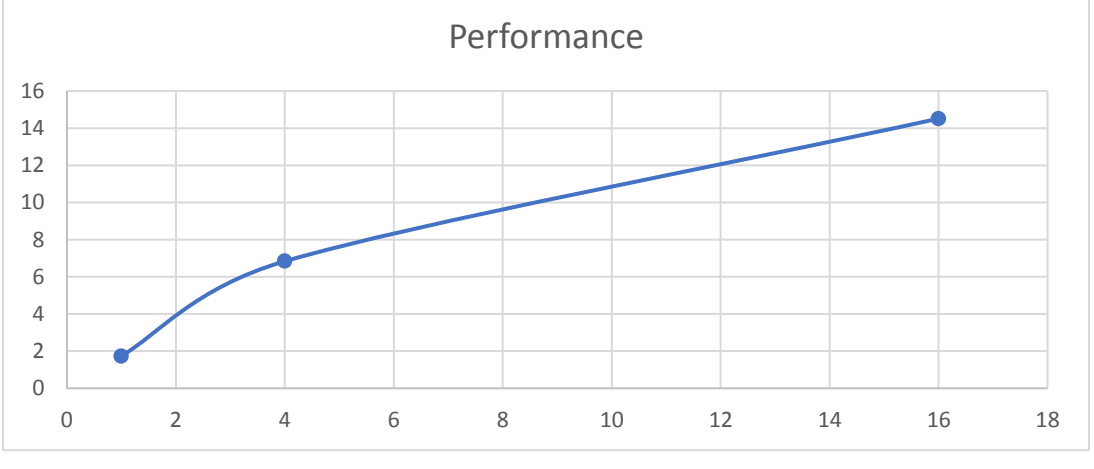
Speedup = Serial execution time / Parallel Execution time
GFLOPS = 2* N^3/T
Where N = Matrix size and T = execution time

I ran the code for MPI processes 1,4 and 16 as only processes with square number supported for this algorithm to divide the matrix. I have tested the output for all the OpenMP thareds of 1 to 28. I ran this code on various size of matrices. Here is the results for matrix size of 3000 with OpenMP threads = 2.

Num of Nodes	OpenMP threads	Execution Time	Performance	Speedup
1	2	31.09	1.736861	1
4	2	7.896811	6.838203	3.937032
16	2	3.720882	14.512899	8.355546







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

Maximum performance I got is 16.91 GFLOPS with execution time 8.581571 seconds for matrix input size of 24000 and OpenMP threads 20 and mpi nodes of 4.

ii. Performance measure using Gustafosn – 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= 25% i.e 25% 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 = number of nodes * number of cores each node

Here we have calculated the 'p' using number of cores each node = 2

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

Number of cores	Num of Processesese	gustafson	Karp-flatt
1	2	1.75	0.142857
4	8	6.25	0.04
16	32	24.25	0.010309

5. 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 matrix_mult () function.

b. Solution Description:

We have to manage the matrix data same way as we manage in the previous case of Cannon using MPI.
We just need to add the OpenACC pragma to the matrix_mult () algorithm function in order to use the cores at every node.

```
#pragma acc data copyin( a[0: n*n],b[0:n*n]), copyout(c[0:n*n] )
    {
        #pragma acc loop independent
        for (i=0; i<n; i++)
        {
            #pragma acc loop independent
            for (j=0; j<n; j++){
                #pragma acc loop seq
                for (k=0; k<n; k++){
                    c[i*n+j] += a[i*n+k]*b[k*n+j];
                }
            }
        }
}
```

Then I executed the code on bridge using the batch script. In the script, I ran the code using mpirun for 1,4 and 16 nodes with the one GPU.
You can run the script using 'sbatch job_mpicc.sh'. The output will be saved in the mpi.csv file.

c. Results:

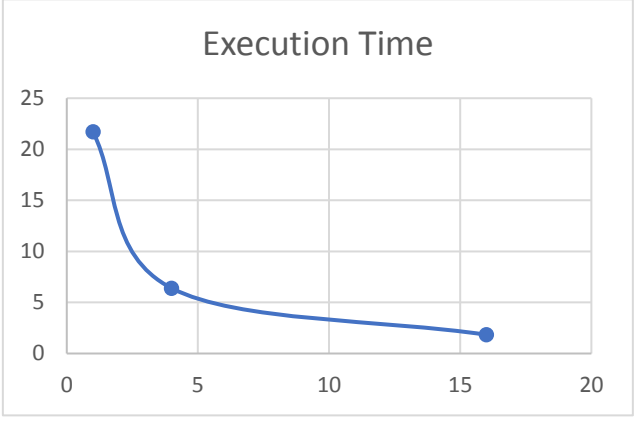
i. Execution time and performance:

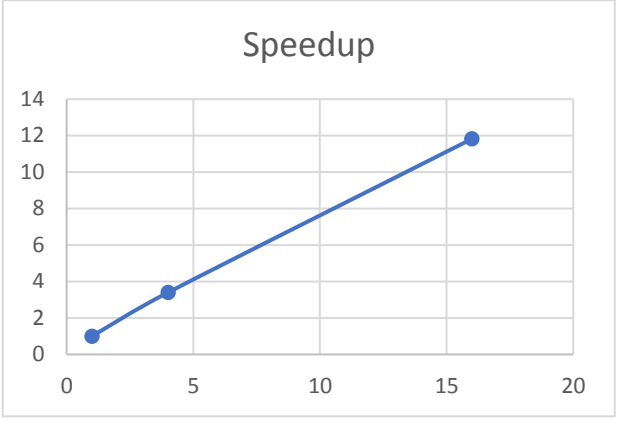
I have measured the performance by calculating speedup and GFLOPS required for the execution of the code for the given number threads and input size matrix.

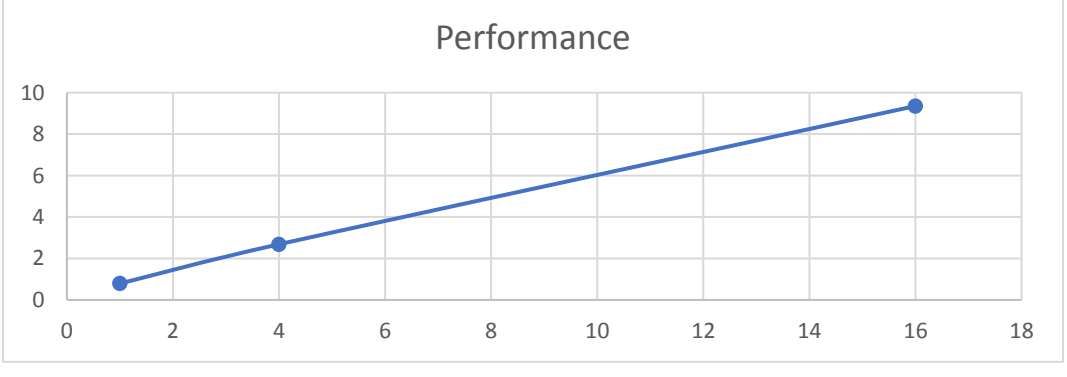
Speedup = Serial execution time / Parallel Execution time
GFLOPS = 2* N^3/T
Where N = Matrix size and T = execution time

I ran the code for 1 GPU and MPI processes 1,4 and 16 as only processes with square number supported for this algorithm. I ran this code on various size of matrices. Here is the results for matrix size of 2048.

Num of Nodes	Execution Time	Performance	Speedup
1	21.707532	0.791424	1
4	6.396048	2.686013	3.393897607
16	1.836325	9.355572	11.82118198







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

Maximum performance I got is 12.90 GFLOPS with execution time of 7.85 seconds for the matrix size 3700 with MPI threads 16 and 1 GPU