CS 6376.001 – Parallel Processing
Instructor: Dr. Richard Goodrum
TA: Kenneth Platz

Program 4 – Matrix Multiplication with MPI, OpenMP, and OpenACC

Tarunesh Verma (txv140330)
tarunesh.verma@utd.edu

Problem Statement

The objective of this program is to design and implement Cannon's Algorithm, a distributed algorithm for matrix-matrix multiplication and parallelize it with MPI, OpenMP, and OpenACC. In addition to the algorithm, the test data generation code has to also be designed and implemented.

Approach to Solution

Cannon's Algorithm multiplies two matrices square matrices A and B (of dimensions N x N) to give a result matrix C. As this is a distributed algorithm, the multiplication is done across Processes using partial blocks of matrices A and B. The sizes of the blocks of A and B are $^{N}/_{\sqrt{P}}$, where N is the dimension of the matrix and P is the number of Processes. A partial result is computed in each process, which is then added on to by a new pair obtained by the Process using rotation. Due to the nature of the division, a process mesh of dimensions $\sqrt{P}\ X\ \sqrt{P}$ is generated. Blocks of A move 1 place left and blocks of B move 1 place above until they reach their starting position. At the end, the blocks of C across nodes contain the partial results, which can be combined and used as required.

The first step was to dynamically allocate and initialize the three matrices, A, B, and C. The next step was to skew A and B to prepare them for multiplication. Following that, multiplication (recursive, block-oriented) and shifting was performed to generate the blocks of results. Finally, these blocks were combined to create the result matrix.

The exact implementation is described in the next section.

A batch script was written to request resources (nodes, partition, number and type of GPUs, running time), set-up email notifications, and run the program.

Description of Solution

In this program, the size of the block matrices is chosen to be 16K x 16K; the dimension N of the three matrices turns out as N = 16K * sqrt(P). Based on the above values, each process (node) generates its own blocks of A and B. These blocks are multiplied to create C, which stores the partial result. Blocks A and B are then rotated using MPI commands, i.e. each node sends its blocks to another node and receives a new pair to multiply. This result is added to the node's running total ofthe block C. Thus, each node is responsible for computing a part of C.

After initializing MPI variables, a process mesh was setup (using process size/width of 16K x 16K and the number of Processes) with a Cartesian topology with wrapping of the edges. A mesh communicator was generated which was used to determine the ranks of the individual blocks and aid in block operations.

The time was measured at this point to evaluate the performance.

Then memory was allocated for the three matrices and they were initialized as follows: elements of block A were given the value $j - i$, and the elements of block B were given the value $N - j + i$.

i and j act as global variables (value increasing across nodes to generate uniform matrices). This was accomplished by multiplying the mesh coordinates (locations of the blocks in the process mesh) with the Process width/size and adding it to j and i;

At this point, the matrices can be printed if required.

To prepare for multiplication, the matrices need to be skewed such that elements $A[i][j]$ and $B[i][j]$ of the Process Mesh (i.e. block matrices) are in the correct position of multiplication - $A[i][j]$ is moved i places to its left and $B[i][j]$ is moved j places above. To do this, the following to functions were used:

$$MPI\_Cart\_shift(MPI\_Comm\ comm, int\ direction, int\ disp, int * rank\_source, int * rank\_dest)$$

$$MPI\_Sendrecv\_replace(void * buf, int\ count, MPI\_Datatype\ datatype, int\ dest,$$

$$int\ sendtag, int\ recvtag, MPI\_Comm\ comm, MPI\_Status\ \&status)$$

The first function uses the mesh communicator and specified direction and displacement values to determine the source and destination ranks of the block. Directions 1 and 0 was used for shifting left and above respectively and the mesh coordinates were used as the displacement values.

The second function takes a pointer to the value to send along with the total number of elements, the datatype, the destination and any associated tag, the source and any associated tag, and the communicator for the mesh to send the value specified by buf and replace it with the value received; it returns a status indicating whether the send/receive was successful. The addresses of the first elements of the matrices along with their sizes, type, source and destination (determined from the previous function), no tags, and the communicator are used.

Then, for $\sqrt{P}$ times, multiplication and rotation (displacement of 1) are performed on A and B. The result can be printed if so desired. The execution time and performance are measured, and the resources are deallocated.

Batch File – script_batch.job

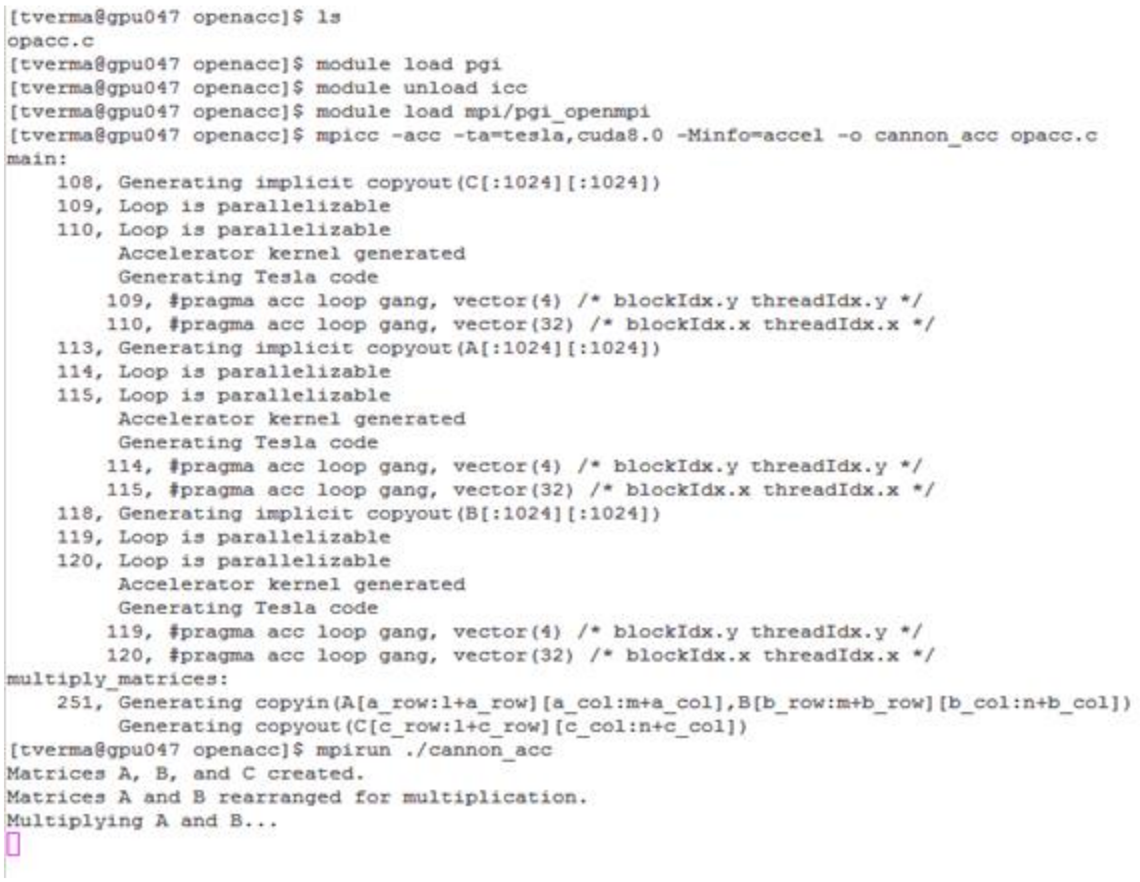
The batch file called *batch_script.job* is used to request resources and execute the program.

The script requests the resources (nodes, number of tasks per node, number of CPUs per task, GPUs if required, running time); unloads the Intel compiler and loads PGI compilers; and compiles and runs the program.

Screenshots

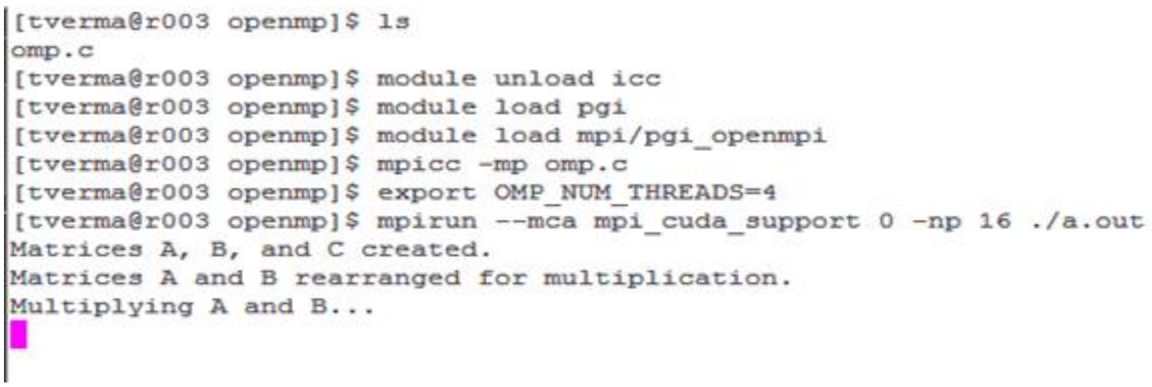
The following screenshots show the execution of the program on Bridges:

1. GPU



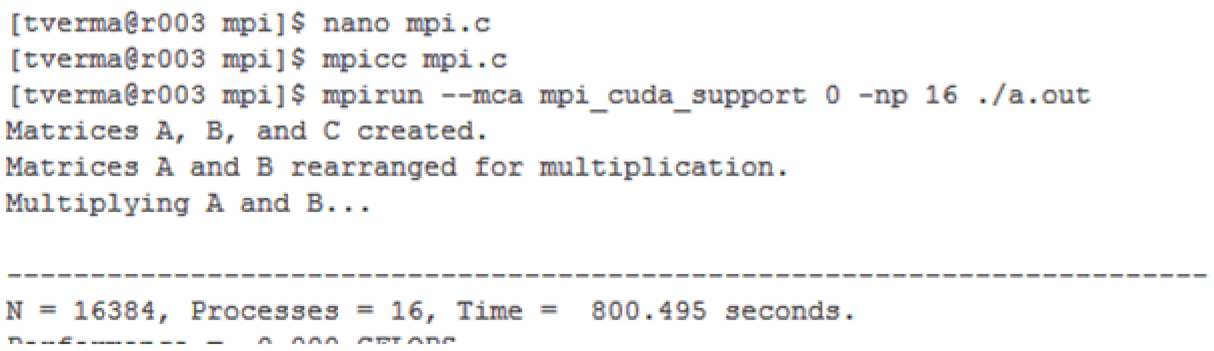
```
[tverma@gpu047 openacc]$ ls
opacc.c
[tverma@gpu047 openacc]$ module load pgi
[tverma@gpu047 openacc]$ module unload icc
[tverma@gpu047 openacc]$ module load mpi/pgi_openmpi
[tverma@gpu047 openacc]$ mpicc -acc -ta=tesla,cuda8.0 -Minfo=accel -o cannon_acc opacc.c
main:
    108, Generating implicit copyout(C[:1024][:1024])
    109, Loop is parallelizable
    110, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        109, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
        110, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
    113, Generating implicit copyout(A[:1024][:1024])
    114, Loop is parallelizable
    115, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        114, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
        115, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
    118, Generating implicit copyout(B[:1024][:1024])
    119, Loop is parallelizable
    120, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        119, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
        120, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
multiply_matrices:
    251, Generating copyin(A[a_row:1+a_row][a_col:m+a_col],B[b_row:m+b_row][b_col:n+b_col])
         Generating copyout(C[c_row:1+c_row][c_col:n+c_col])
[tverma@gpu047 openacc]$ mpirun ./cannon_acc
Matrices A, B, and C created.
Matrices A and B rearranged for multiplication.
Multiplying A and B...
```

2. OpenMP



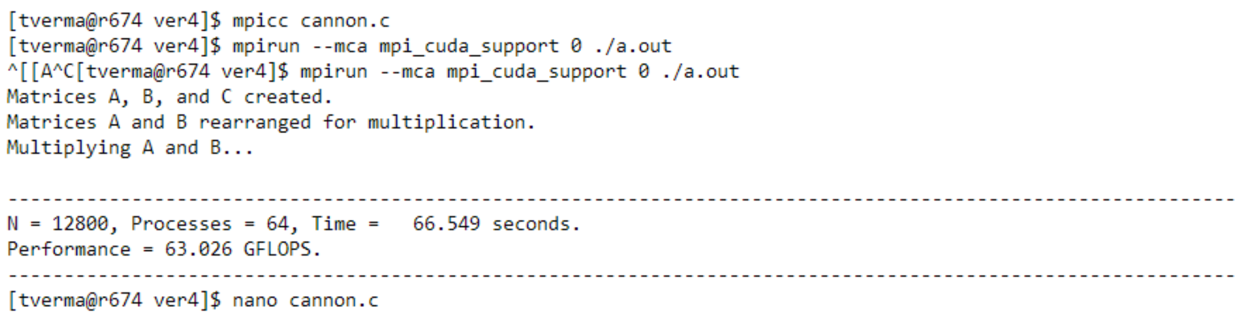
```
[tverma@r003 openmp]$ ls
omp.c
[tverma@r003 openmp]$ module unload icc
[tverma@r003 openmp]$ module load pgi
[tverma@r003 openmp]$ module load mpi/pgi_openmpi
[tverma@r003 openmp]$ mpicc -mp omp.c
[tverma@r003 openmp]$ export OMP_NUM_THREADS=4
[tverma@r003 openmp]$ mpirun --mca mpi_cuda_support 0 -np 16 ./a.out
Matrices A, B, and C created.
Matrices A and B rearranged for multiplication.
Multiplying A and B...
```

3. MPI



```
[tverma@r003 mpi]$ nano mpi.c
[tverma@r003 mpi]$ mpicc mpi.c
[tverma@r003 mpi]$ mpirun --mca mpi_cuda_support 0 -np 16 ./a.out
Matrices A, B, and C created.
Matrices A and B rearranged for multiplication.
Multiplying A and B...

------------------------------------------------------------------------
N = 16384, Processes = 16, Time =  800.495 seconds.
```



```
[tverma@r674 ver4]$ mpicc cannon.c
[tverma@r674 ver4]$ mpirun --mca mpi_cuda_support 0 ./a.out
^[[A^C[tverma@r674 ver4]$ mpirun --mca mpi_cuda_support 0 ./a.out
Matrices A, B, and C created.
Matrices A and B rearranged for multiplication.
Multiplying A and B...

------------------------------------------------------------------------------------------------------------
N = 12800, Processes = 64, Time =   66.549 seconds.
Performance = 63.026 GFLOPS.
------------------------------------------------------------------------------------------------------------
[tverma@r674 ver4]$ nano cannon.c
```

The result shown above was taken while testing, and that's why the filename is different. The code is the same.

Performance Metrics

The Process size/width chosen by default, 16K x 16K was massive. As the number of Processes increased (1, 4, 9, 16, 25), the width increased as (16K x 16K, 32K x 32K, 48K x 48K, 64K x 64K, 80K x 80K). Given such massive sizes, the default MPI program could not complete for most of these values in 6-8 hours. Only 16K x 16K completed, generating the output matrix for 1 Process in 11610.286 seconds (a little over 3 hours), with a performance of **0.76 Gigaflops.**

Thus, I performed subsequent tests on smaller matrix sizes and varying the number of Processes shown above. The results are present below. It was found that for MPI programs, the performance increased dramatically when the number of processes was greater than 4. This is expected, as for more than 4 Processes, the computation to communication time ratio of Cannon's Algorithm is better than that of a more straightforward, row-wise approach.

The performance of the program was estimated as shown below

$$Performance = \frac{2 * \ iterations * ROWS * COLUMNS}{Execution\ Time\ (in\ nanoseconds)} = {2N^3}/{T}$$

The results from the tests are summarized below:

	N (Dimensions)	**Number of processes**	**Time taken (s)**	**Performance (GLOPS)**	**Speed-up from 16K/1 Process (shown above)**
OpenMP (4 threads)	8192	16	380.031	2.89	-----(N is not same) -----
OpenACC					
1 GPU	8192	16	93.767	11.73	-----(N is not same) -----
	16384	16	702.029	12.53	16.54
MPI	16384	16	800.495	10.99	14.50
	2048	1	17.160	1.00	
	4096	4	37.746	3.64	
	6144	9	61.733	7.51	
	8192	16	86.422	12.72	
	12800	64	66.549	63.03	
	25600	64	543.091	61.78	

Performance Metrics – Gustafson-Barsis' Law

$$\psi = p + (1 - p) * s$$

$$\psi\ is\ the\ speed - up\ (calculated\ above), p\ is\ the\ number\ of\ processors\ (shown\ above),$$
$$and\ s\ is\ the\ percentage\ of\ time\ spent\ in\ sequential\ code$$

$$s_{GPU} = 16.54 \rightarrow 16.0\ \%\ time\ spent\ in\ serial\ code.$$

$$s_{MPI} = 14.50 \rightarrow 14.5\ \%\ time\ spent\ in\ serial\ code.$$

Performance Metrics – Karp-Flatt Metric

$$e = \frac{1/_{\psi} - 1/_{p}}{1 - 1/_{p}}, where\ e\ is\ the\ experimentally - determined\ serial\ fraction,$$

$\psi\ is\ the\ speed - up, p\ is\ the\ numberof\ processors$

This is calculated only for 16K x 16K run, as we have the speed up available.

$$e_{GPU} = 0.002$$

$$e_{MPI} = 0.007$$