**Program - 4**

**Problem Statement** –We are basically supposed to implement Matrix Multiplication and test Canon’s algorithm which is a parallel algorithm for multiplying matrices that are square in dimension (or almost square) and the dimension is divisible by the square root of the number of processors.

The matrix multiplication has been compared for:

1. OpenMp
2. OpenACC
3. MPI
4. MPI and OpenMp
5. MPI and OpenACC

* Input: Two matrices. (MatrixA and MatrixB)
* Output: Product of the two input matrices. (MatrixC)

**Approach to the Solution**

**Design Approach**:

**INIT-METHOD:** The initialization method has been written for the program for the simple use case where in the matrix A and matrix B have the entries such as:

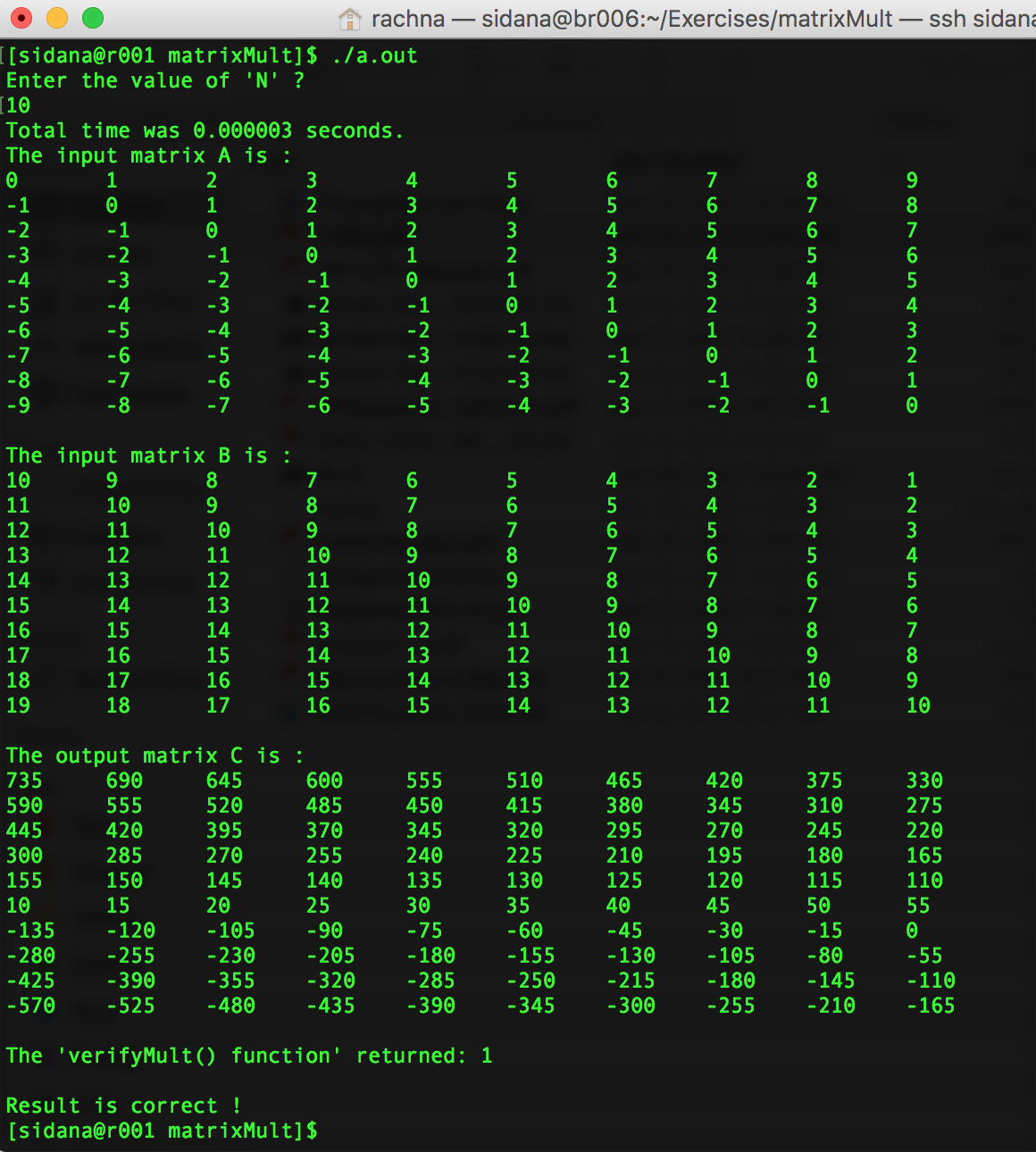
MatrixA[i][j] = j-i;

MatrixB[i][j] = n-j+i;

The matrices are multiplied according to the algorithm and the result is saved in MatrixC, the result of matrix multiplication is verified using the function **verifyMult**() in the code.  
The function **verifyMult**() generates a new matrixD such that the elements of matrixD are calculated as follows:

MatrixD[i][j] = ∑ (k-i)(n-j+k) where 0<=k<n and n is the size of the matrix.

If MatrixD is same as MatrixC, the function **verifyMult()** returns 1 else it returns 0.

****

**Sequential Algorithm:**

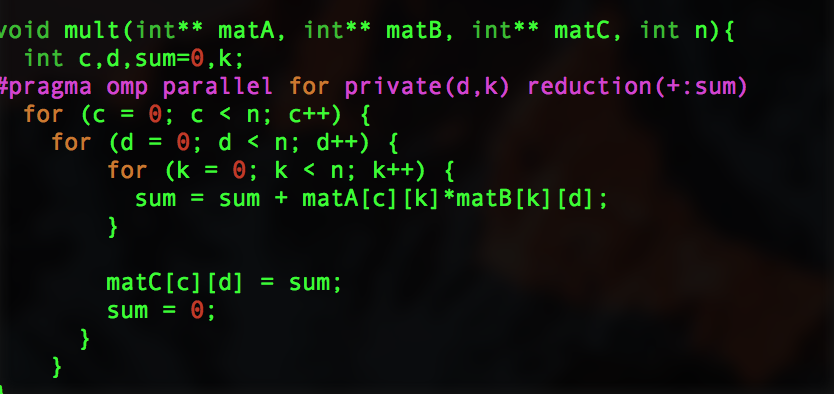
|  |  |
| --- | --- |
| Size | Serial Execution Time |
| 500 | **0.2254** |
| 1000 | **3.572** |
| 1200 | **14.159** |
| 1500 | **31.455** |

**Language used**: C

**Tools used**: ssh client ( Mac Terminal)

1. **OpenMP:**

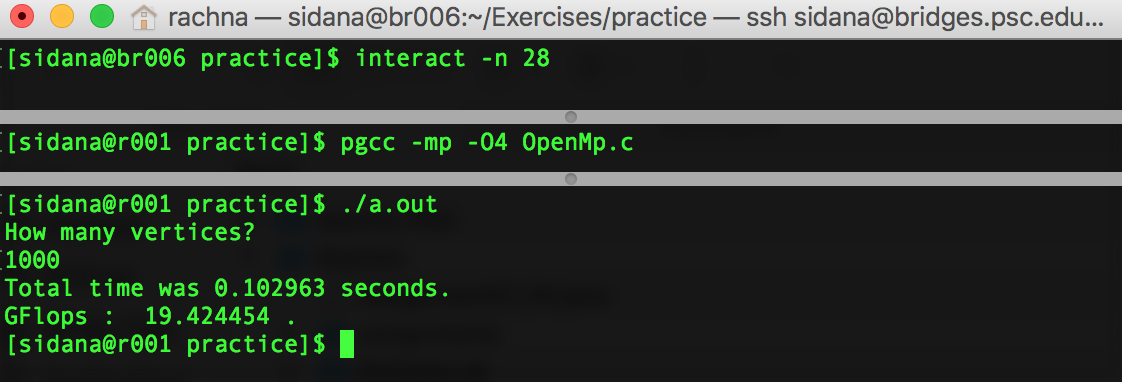
The code was analyzed thoroughly. The **for** loops were scrutinized for data dependency and the for loop was thus parallelized to increase the performance of the code.

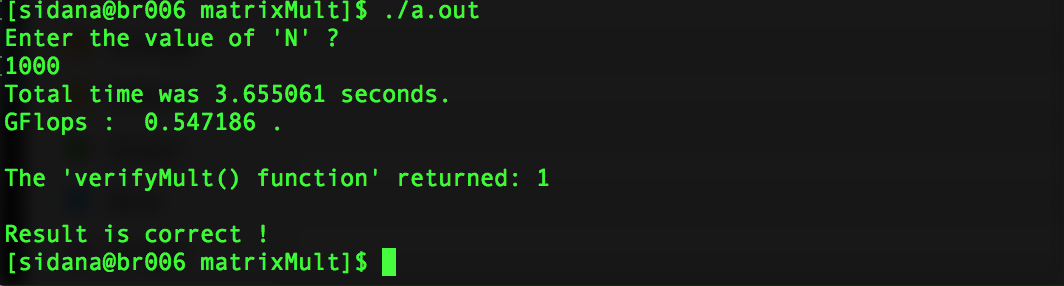


**Solution Description**

**BUILDING and EXECUTION-**

1. Log on to the bridges.
2. Shift to compute node requesting all the cores.
3. Compile the code.
4. Run the compiled code on various number of cores.
5. Note the execution time against various number of core



****

**ANALYSIS**

**Execution Time -** The execution time corresponding to various number of cores was analyzed. The following table records the time of execution the parallel program took. Clearly, as the number of threads allocation to the code increased, the time for execution dropped.

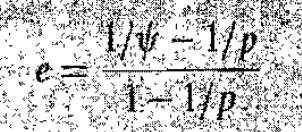
|  |  |
| --- | --- |
| Number of threads | Execution Time (in seconds) |
| 1 | 3.655061 |
| 1 | 3.568483 |
| 1 | 3.629898 |
| 2 | 1.754671 |
| 2 | 1.895487 |
| 3 | 1.255273 |
| 3 | 1.260259 |
| 4 | 0.942000 |
| 4 | 0.941677 |
| 5 | 0.760723 |
| 5 | 0.770502 |
| 6 | 0.632302 |
| 6 | 0.630087 |
| 7 | 0.608255 |
| 7 | 0.599743 |
| 8 | 0.501367 |
| 8 | 0.528323 |
| 9 | 0.473593 |
| 9 | 0.461316 |
| 10 | 0.428882 |
| 10 | 0.427178 |
| 12 | 0.370475 |
| 12 | 0.373956 |
| 14 | 0.345170 |
| 14 | 0.314278 |
| 16 | 0.278716 |
| 16 | 0.278280 |
| 18 | 0.252650 |
| 18 | 0.255996 |
| 20 | 0.223301 |
| 20 | 0.221540 |
| 22 | 0.203776 |
| 22 | 0.205341 |
| 24 | 0.200661 |
| 24 | 0.202339 |
| 26 | 0.182028 |
| 26 | 0.176882 |
| 28 | 0.158482 |
| 28 | 0.162761 |

**SPEEDUP AND KARP FLATT METRIC**

Speedup is the ration between sequential execution time and parallel execution time.

The following table maps the speedup as the function of various cores assigned to the parallel program.

The parameter “e” in Karp flatt metric can be calculated as:

****

The following tables show the values

|  |  |  |  |
| --- | --- | --- | --- |
| No of Processors | Execution Time | SpeedUp | “e" |
| 2 | 1.895487 | 2.083046337 | 0.03718487 |
| 3 | 1.255273 | 2.911765807 | 0.01515132 |
| 4 | 0.941677 | 3.881438115 | 0.010181955 |
| 5 | 0.760723 | 4.804719983 | 0.010160843 |
| 6 | 0.632302 | 5.780562136 | 0.007592267 |
| 7 | 0.599743 | 6.094378759 | 0.024766573 |
| 8 | 0.501367 | 7.290190619 | 0.013909285 |
| 9 | 0.461316 | 7.923117776 | 0.016989559 |
| 10 | 0.427178 | 8.556295034 | 0.018747795 |
| 12 | 0.370475 | 9.86587759 | 0.019664863 |
| 14 | 0.314278 | 11.63002501 | 0.015675441 |
| 16 | 0.278716 | 13.113926 | 0.014671803 |
| 18 | 0.25265 | 14.46689491 | 0.014365882 |
| 20 | 0.22154 | 16.49842466 | 0.011170366 |
| 22 | 0.203776 | 17.93666084 | 0.010787534 |
| 24 | 0.200661 | 18.21510408 | 0.013808168 |

**GFLOPS**

**1000 X 1000**

FLOPS is a measure of computer performance.

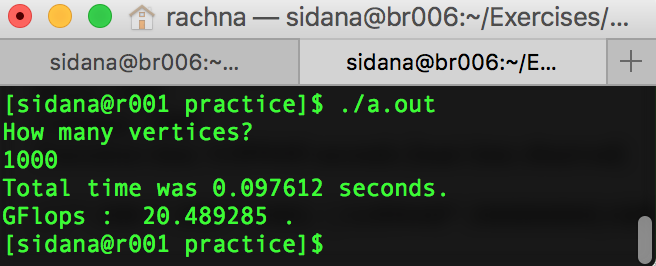
It can be calculated as:

(2 \* n \* n \* n – n \* n) / time where matrix size is n X n.

In our case it will be:

[ 2 \* 1000 \* 1000 \* 1000 – 1000 \* 10000] / [ 0.097612 \* 1000000000] **GFLOPS**

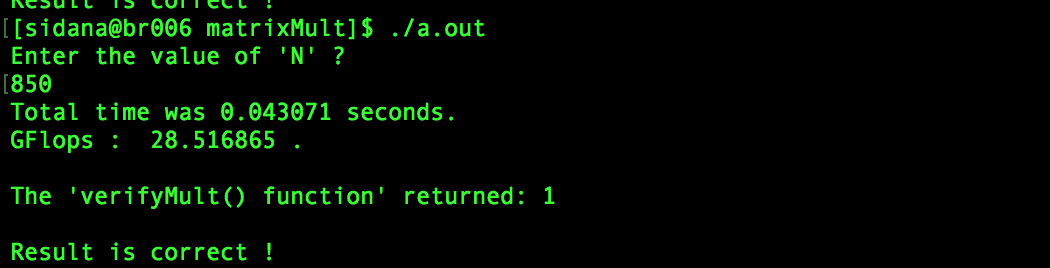
= 20.489285 GFLOPS



**850 X 850**

[2 \* 850 \* 850 \* 850 – 850 \* 850] / [ 0.043071 \* 1000000000] **GFLOPS**

= **28.516865 GFLOPS**

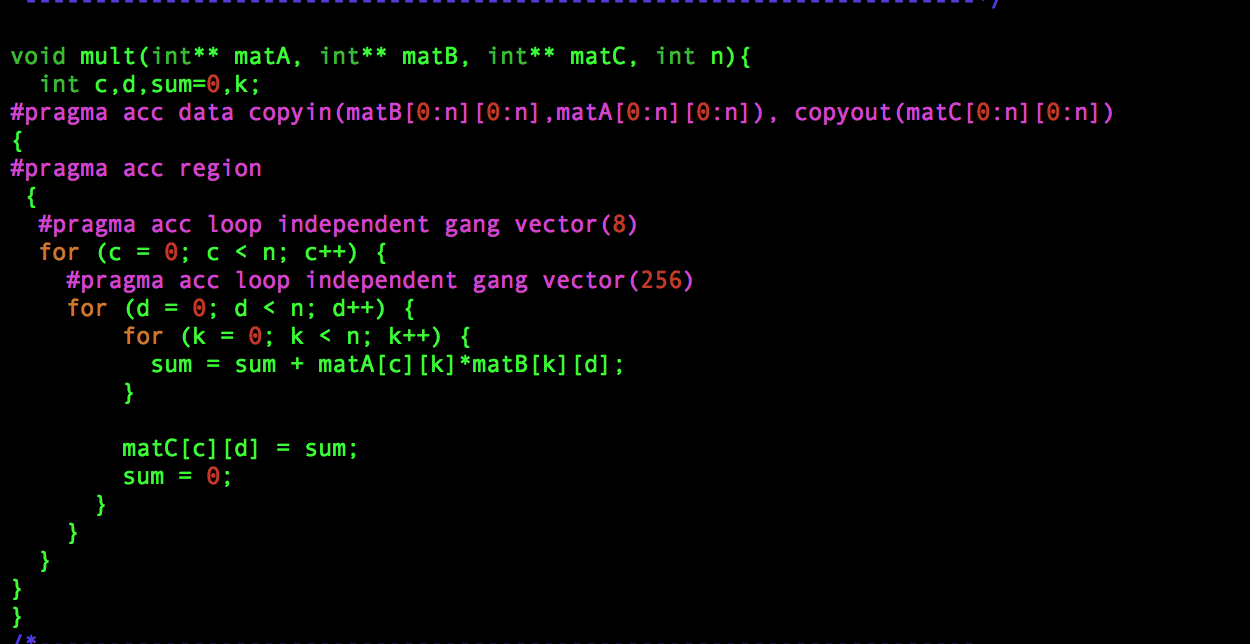
****

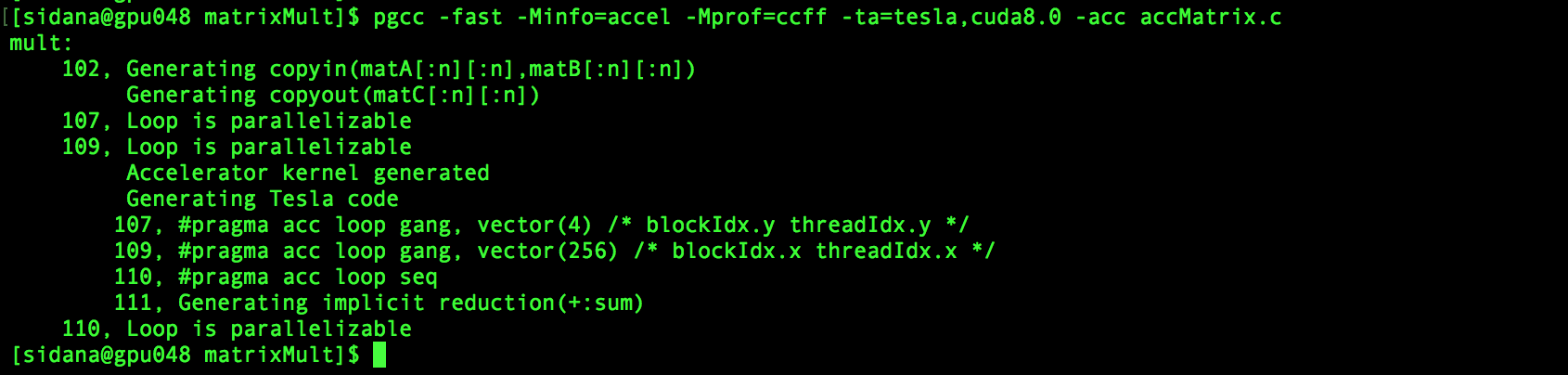
1. **OpenACC:**

**Approach to the Solution**

**Design Approach**

**Identify Parallelism**- The two outer for loops are, independent and can be executed in parallel while the innermost for loop has data dependency and hence cannot be executed in parallel. Also, we can use the directive “copy” so that not much time is spent on data copy in and out of the host.



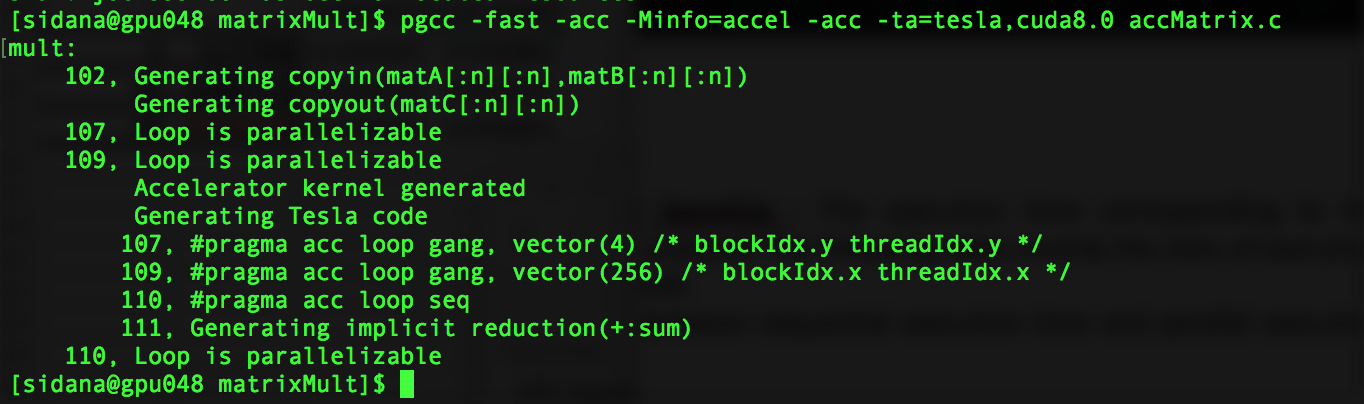
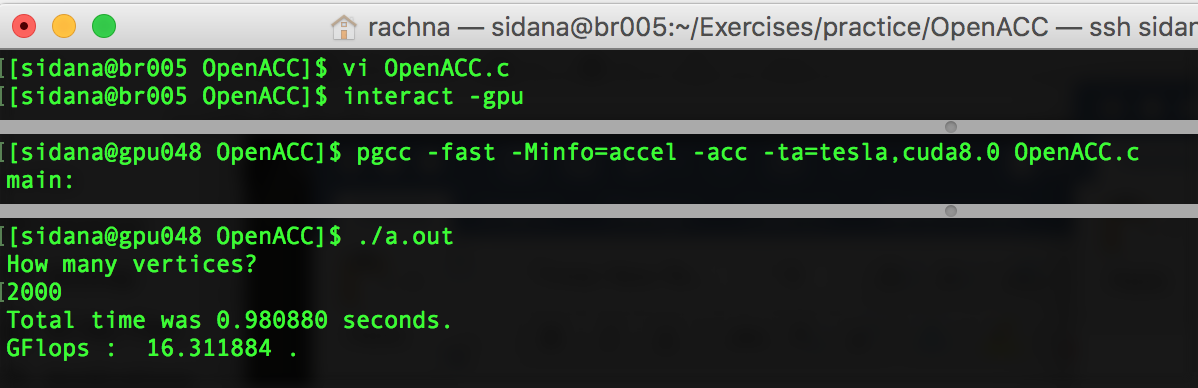


Using these directives, the code was compiled and the performance was analyzed. The profiling was activated to, analyze where the major time is being spent.

**Solution Description**

**BUILDING and EXECUTION-**

1. Log on to the bridges.
2. Shift to compute node requesting gpu mode.
3. Compile the code.
4. Run the compiled code.
5. Wait for the program to be executed.

The following screen captures the steps:

**ANALYSIS**

**Execution Time and SpeedUp -** The execution time corresponding to the parallelized code was analyzed. The following table records the time of execution the parallel program took.

Speedup is the ratio between sequential execution time and parallel execution time.

The execution time and speedup of openMP and openACC are noted in the table below:

* Problem Size – 1000 X 1000

|  |  |  |
| --- | --- | --- |
| Execution | Time | Speedup |
| CPU Serial | 3.655061 | 1 |
| CPU OpenMp 2 Threads | 1.895487 | 1.928296527 |
| CPU OpenMp 4 Threads | 0.941677 | 3.881438115 |
| CPU OpenMp 8 Threads | 0.501367 | 7.290190619 |
| CPU OpenMp 10 Threads | 0.427178 | 8.556295034 |
| CPU OpenMp 20 Threads | 0.22154 | 16.49842466 |
| CPU OpenMp 28 Threads | 0.159152 | 22.96585026 |
| OpenACC GPU | **0.581150** | **6.28935903** |

The following table maps the execution time for different matrix size:

|  |  |
| --- | --- |
| Size (NXN) | Execution Time |
| 2000 X 2000 | 0.759803 |
| 3000 X 3000 | 0.958283 |
| 4000 X 4000 | 1.305490 |
| 5000 X 5000 | 2.033351 |
| 8000 X 8000 | 5.731685 |

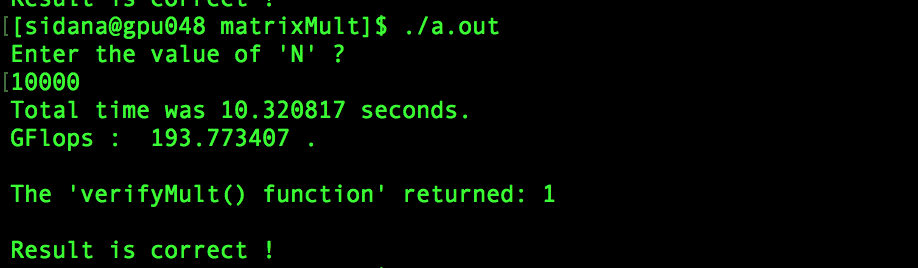
**GFLOPS**

FLOPS is a measure of computer performance.

**Problem Size 10000 X 10000**

[ 2 \* 10000 \* 10000 \* 10000 – 10000 \* 10000] / [10.320817\* 1000000000] **GFLOPS**

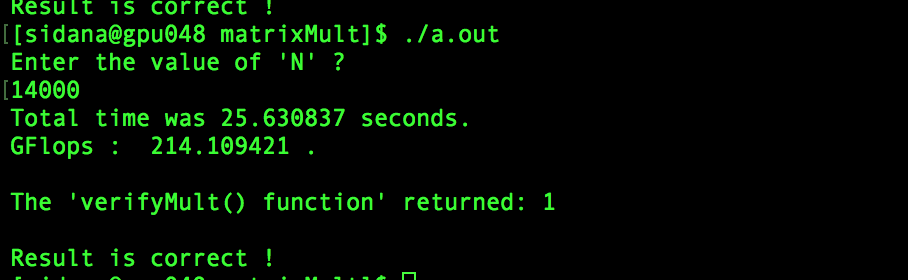
=193.773407 GFLOPS

****

**Problem Size 13000 X 13000**

[ 2 \* 13000 \* 13000\* 13000 – 13000 \* 13000] / [ 25.630837 \* 1000000000] **GFLOPS**

=214.109421 GFLOPS

****

1. **MPI**

**Approach to the Solution**

**Design Approach**

**Identify Parallelism and Its Implementation**- The Message-Passing Interface Standard(MPI) is a library that allows you to do problems in parallel using message-passing to communicate between processes. Here in this problem we need to do parallelize the matrix multiplication using MPI only. Initially we have seen results from OpenMp and OpenAcc.

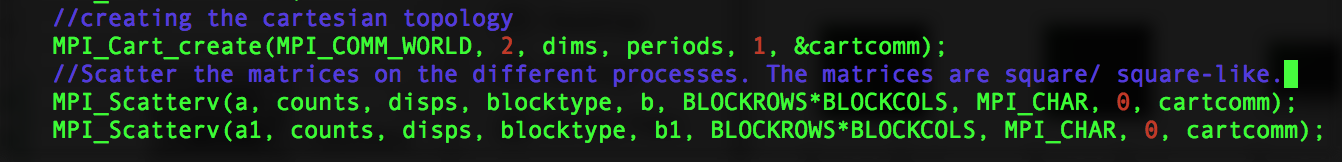
**Implementation Summary**- Initially the main() function is responsible for initialization of the input matrices corresponding to the size specified in the code. It is also responsible to verify that the result is correct. In MPI, we use multiple processes and the data of the input matrices is scattered to the different processes. Cannon’s algorithm for parallel matrix multiplication has been implemented according to the following steps:

INPUT- size of matrix (n) , matrixA (n\*n) and matrixB(n\*n)

GOAL- Product of matrixA and matrixB using p processes

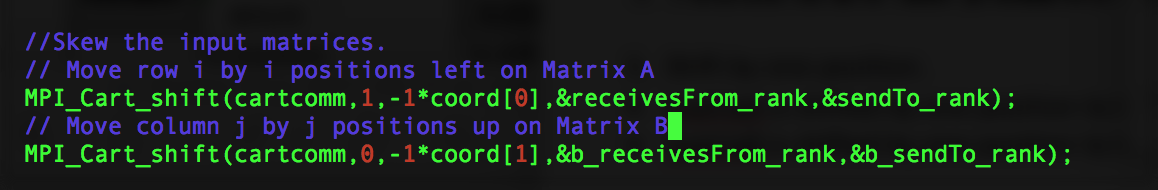
1. First Matrix A and Matrix B are initialized.
2. The data of matrixA and matrixB is scattered to different processes.

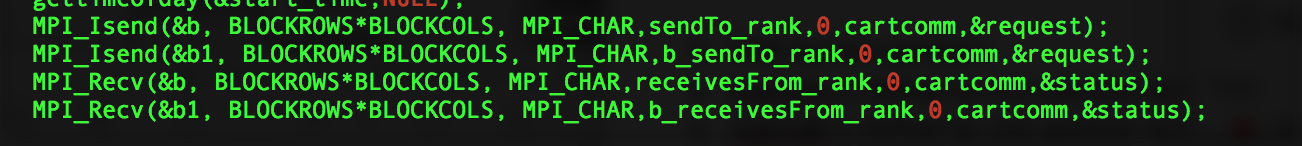
* The block on each process is of size n/sqrt(p)\* n/sqrt(p).



1. The matrices are skewed.

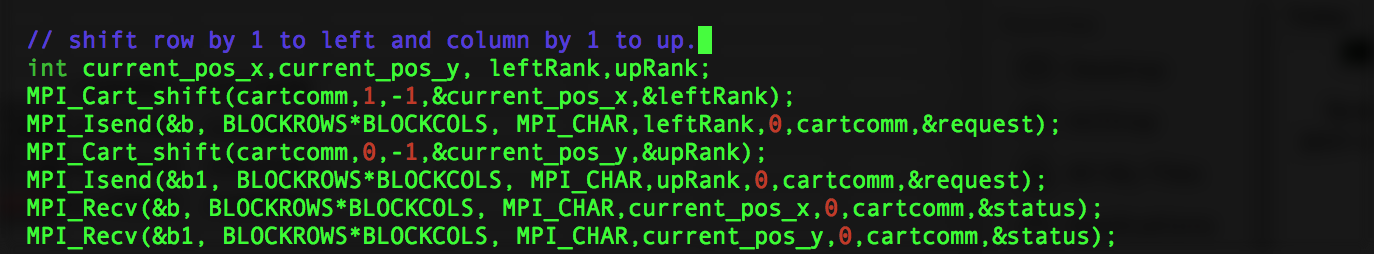
* matrixA is skewed by circular shifting ith row by i places to the left.
* matrixB is skewed by circular shifting jth column by j places to the up.





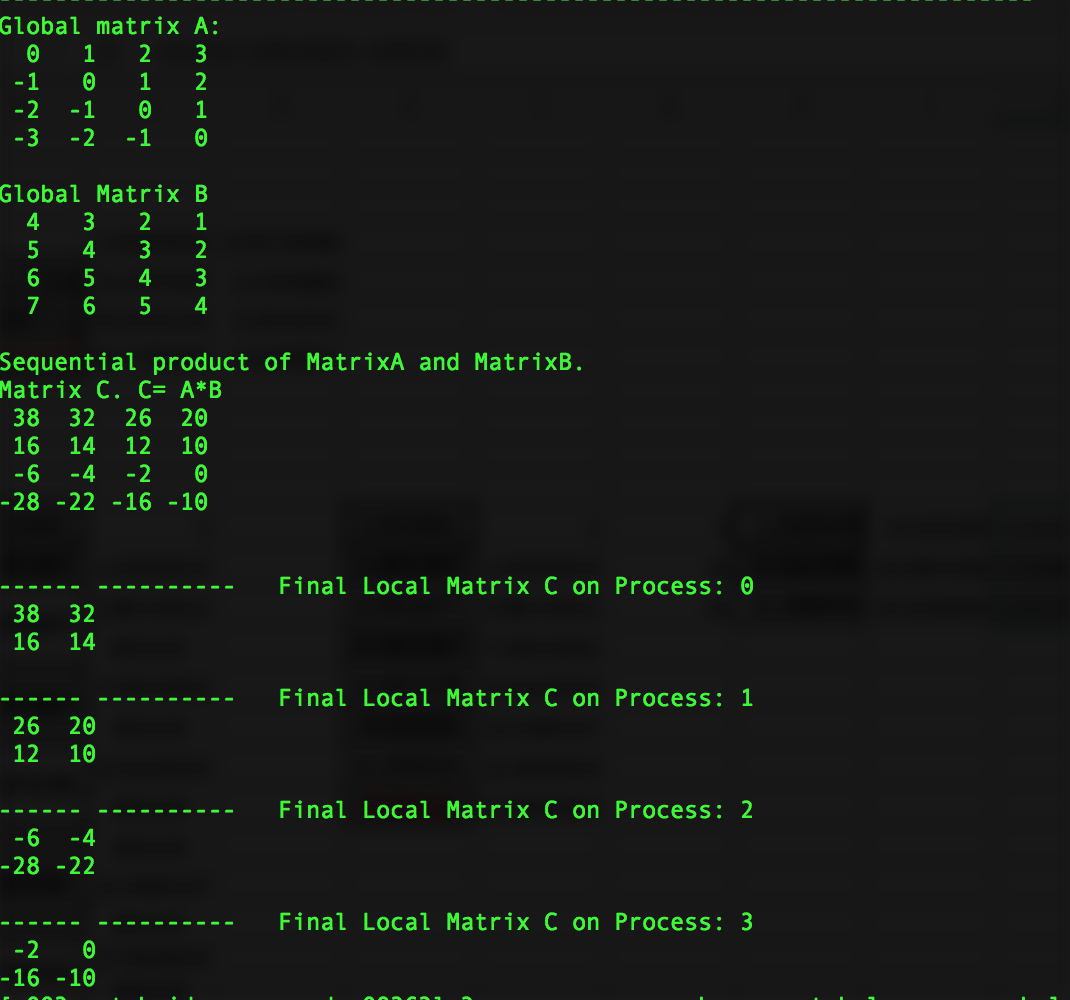
1. The 2 elements on each process are multiplied, it forms the partial element of matrixC.
2. Shift by one position.

* matrixA is shifted by one position to the left.
* matrixB is shifted by one position to the up.



1. The 2 elements on each process are multiplied again and added to the previous partial sum for matrix C.
2. Now different processes contain different blocks of matrix C.

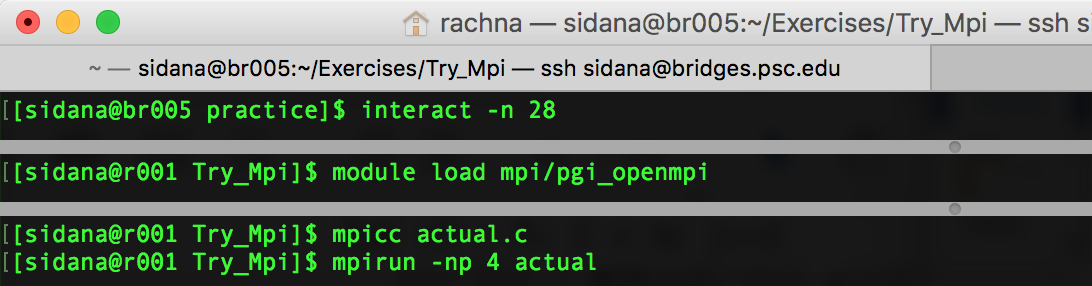
The following screen shows matrices of size 4X4 being run on 4 processes:

****

**Solution Description**

**BUILDING and EXECUTION-**

1. Log on to the bridges.
2. Shift to interactive node requesting all the cores.
3. Load the mpi module
4. Compile the code.
5. Run the compiled code on various number of processes.
6. Note the execution time.



**ANALYSIS**

**Execution Time -** The execution time corresponding to the MPI code was analyzed. The following table records the time of execution the parallel program took.

Speedup is the ratio between sequential execution time and parallel execution time.

The execution time and speedup are noted in the table below:

* Problem Size – 1000 X 1000:

|  |  |
| --- | --- |
| Processes | Time |
| 4 | 1.077718 |
| 4 | 1.118976 |
| 4 | 1.038508 |
| 4 | 1.074638 |
| 16 | 0.367080 |
| 16 | 0.362398 |
| 16 | 0.382502 |
| 16 | 0.374968 |
| 25 | 0.208922 |
| 25 | 0.191302 |
| 25 | 0.188066 |
| 25 | 0.210144 |
| 25 | 0.176549 |

The Following table and the graph below the table maps the execution time corresponding to different problem size for the MPI Program. As the problem size increases, the time to execute the multiplication also increases:

|  |  |
| --- | --- |
| Size (NXN) | Execution Time |
| 2000 X 2000 | 2.22189 |
| 3000 X 3000 | 8.79307 |
| 4000 X 4000 | 15.2379 |
| 5000 X 5000 | 32.9087 |
| 8000 X 8000 | 117.324 |

**Speedup and Karp Flatt Metric:**

The following table captures the speedup and the variable “e” of the Karp flatt metric.

|  |  |  |  |
| --- | --- | --- | --- |
| Processes | Time | Speedup | “e” |
| 4 | 1.038508 | 3.519530904 | 0.045505032 |
| 16 | 0.362398 | 10.08576482 | 0.039092954 |
| 25 | 0.188066 | 19.43499091 | 0.01193082 |

**GFLOPS**

**Problem Size 1024 X 1024**

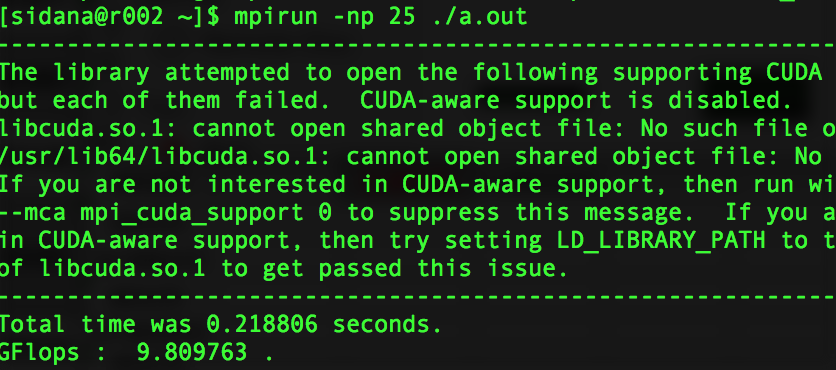
FLOPS is a measure of computer performance.

It can be calculated as:

(2 \* n \* n \* n – n \* n) / time where matrix size is n X n.

[2 \* 1024 \* 1024 \* 1024 – 1024\*1024] / [0.218806\* 1000000000] **GFLOPS**

=**9.809763 GFLOPS**

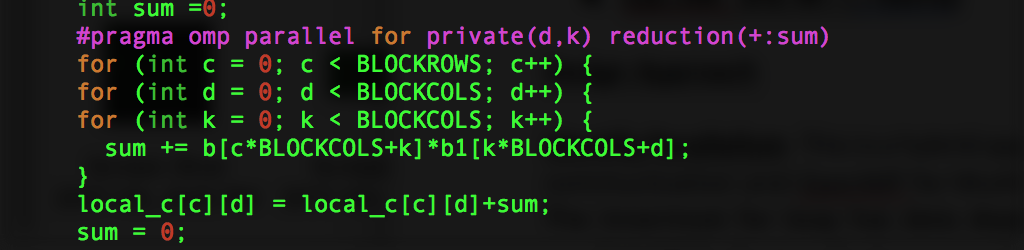


1. **OpenMP and MPI ( Hybrid)**

**Design Approach**

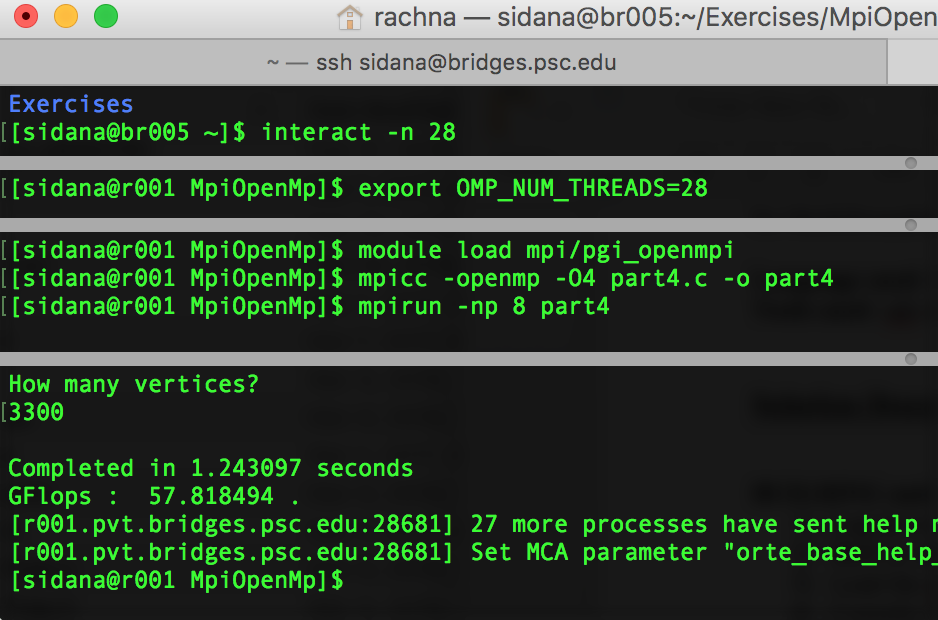
**Identify Parallelism**- This is a hybrid approach in which we use MPI for inter process communication and OpenMP for Multithreading.

The innermost for loop has data dependency and hence cannot be executed in parallel. While the two for loops are, independent and can be executed in parallel.

****

**BUILDING and EXECUTION:**

1. Log on to the bridges.
2. Shift to interactive node requesting ’n’ number of the cores.
3. Load the mpi module
4. Compile the code.
5. Run the compiled code on various number of processes and different number of threads.
6. Note the execution time.



**ANALYSIS**

**Execution Time and SpeedUp -** The execution time corresponding to various number of processes in case of MPI was analyzed. The number of threads were 28. The following table records the time of execution the parallel program took. Clearly, as the number of processes increased, the time for execution dropped.

Speedup is the ratio between sequential execution time and parallel execution time.

* Problem Size – 1000 X 1000

|  |  |
| --- | --- |
| Processes | Time |
| 4 | 0.272060 |
| 4 | 0.290468 |
| 4 | 0.257804 |
| 4 | 0.277429 |
| 16 | 0.085712 |
| 16 | 0.087083 |
| 16 | 0.094396 |
| 16 | 0.087812 |
| 16 | 0.093210 |
| 25 | 0.053123 |
| 25 | 0.056627 |
| 25 | 0.049856 |
| 25 | 0.044154 |

The flowing table and the graph below table maps the execution time for different problem sizes for MPI and OpenMp Hybrid model. With the increasing size the time taken for execution increases:

|  |  |
| --- | --- |
| Size (NXN) | Execution Time |
| 2000 X 2000 | 0.454985 |
| 3000 X 3000 | 1.874071 |
| 4000 X 4000 | 5.278161 |
| 5000 X 5000 | 9.709297 |
| 8000 X 8000 | 34.476830 |

**Speedup and Karp Flatt Metric:**

The following table captures the speedup and the variable “e” of the Karp flatt metric. The better than expected speedups yield a negative Karp Flat Metric.

|  |  |  |
| --- | --- | --- |
| Processes | Speedup | “e” |
| 4 | 13.43476071 | -0.234088296 |
| 16 | 42.64351549 | -0.041653094 |
| 25 | 82.77983875 | -0.029083087 |

**GFLOPS**

FLOPS is a measure of computer performance.

It can be calculated as:

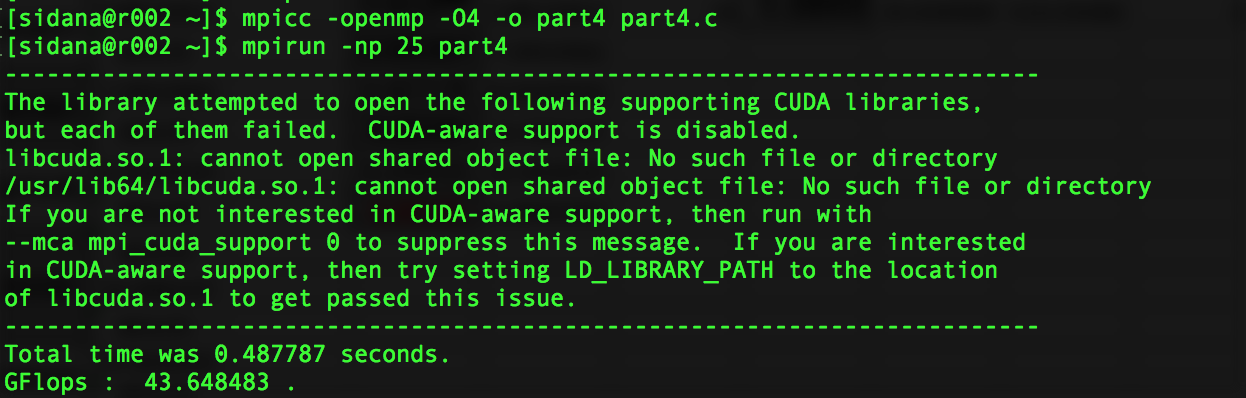
[2\*n\*n\*n-n\*n]/time

**Problem Size 2200 X 2200**

Execution Time: 0.487787 seconds

GFLOPS: [2\*2200\*2200\*2200-2200\*2200]/0.487787 \* 10^9

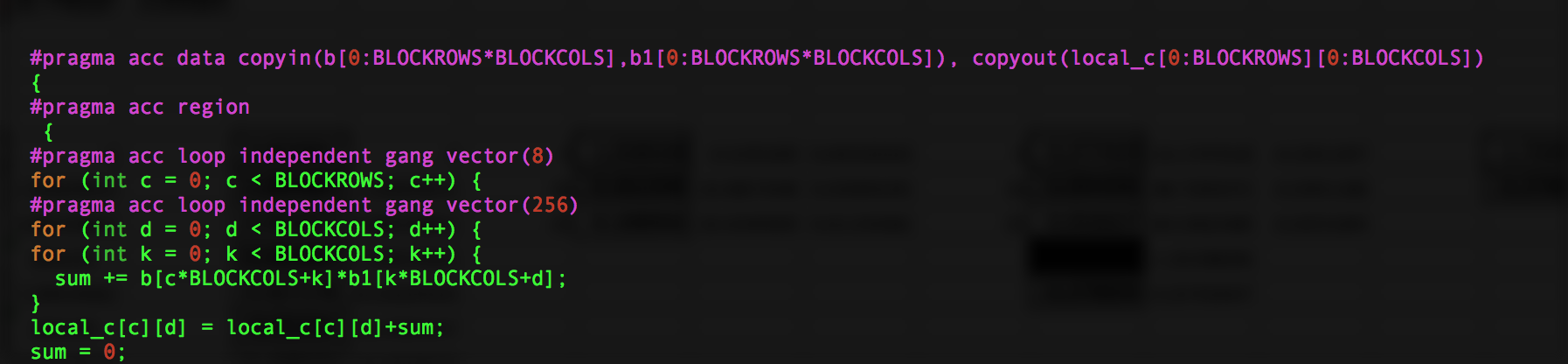
GFLOPS: 43.648483

****

1. **OpenAcc and MPI ( Hybrid)**

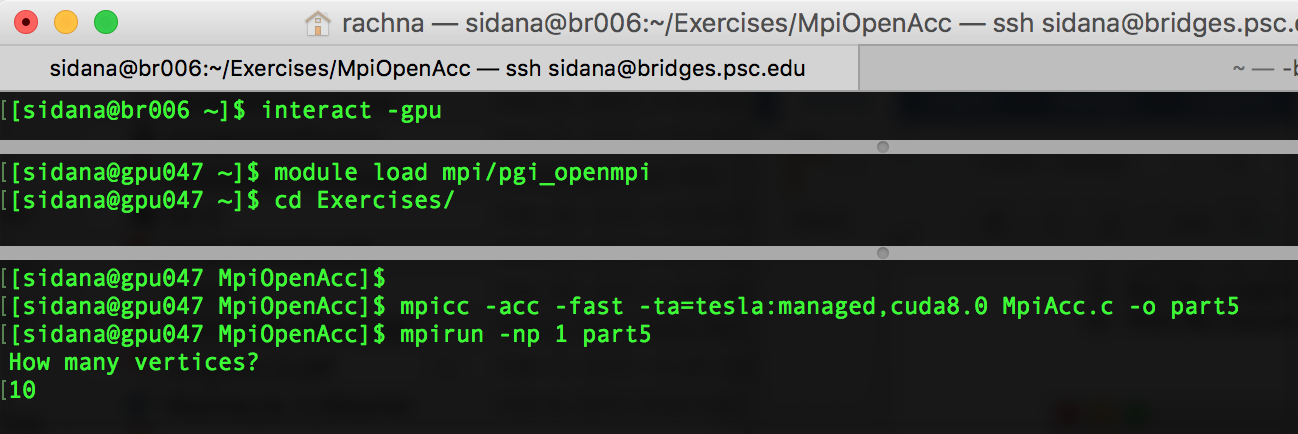
**Identify Parallelism**- This is a hybrid approach in which we use MPI for inter process communication and OpenAcc.

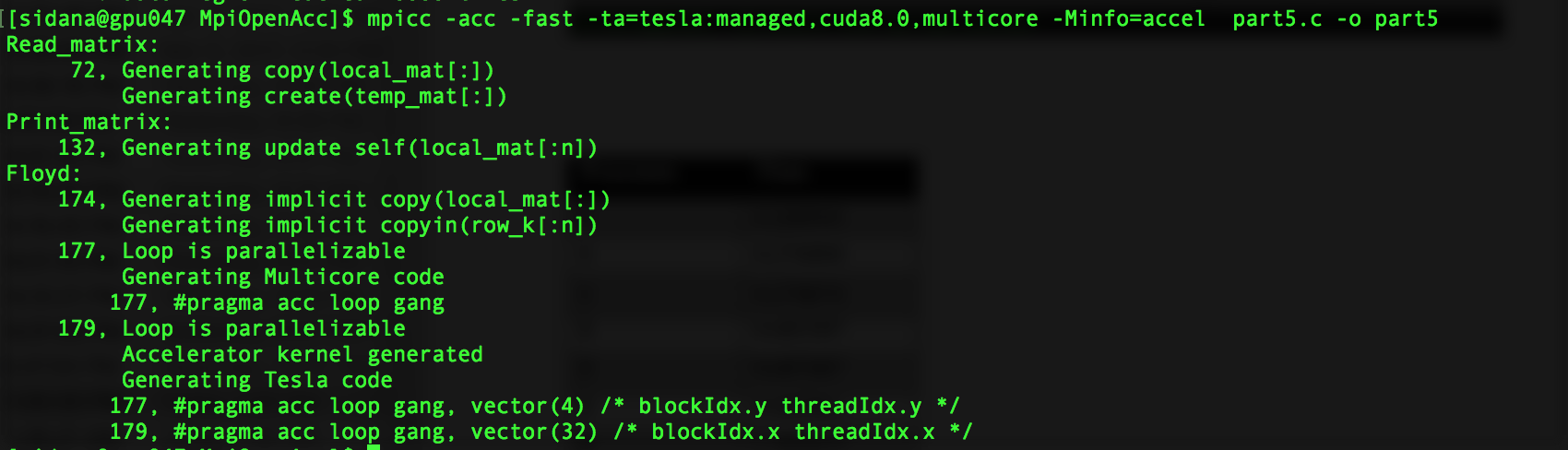
The innermost for loop has data dependency and hence cannot be executed in parallel. While the two for loops are, independent and can be executed in parallel. Also, we can use the directive “copy” so that not much time is spent on data copy in and out of the host.

****

**BUILDING and EXECUTION:**

1. Log on to the bridges.
2. Shift to interactive node requesting gpus.
3. Load the mpi module
4. Compile the code.
5. Run the compiled code on various number of processes.
6. Note the execution time.





The following table maps the execution time for OpenAcc and MPI hybrid model for different number of processes:

|  |  |
| --- | --- |
| Processes | Time |
| 4 | 1.719486 |
| 4 | 1.720063 |
| 4 | 1.744799 |
| 4 | 1.749278 |
| 16 | 6.241710 |
| 16 | 5.680860 |
| 16 | 6.162891 |

**SpeedUp and Karp Flatt Metric:**

The following table maps the speedup and the variable ”e” of the karp flatt metric.

|  |  |  |  |
| --- | --- | --- | --- |
| Processes | Time | Speedup | “e” |
| 4 | 1.720063 | 2.124957632 | 0.294130157 |
| 16 | 5.680860 | 0.643399239 | 1.591194584 |

**GFLOPS**

**Problem Size 3200 X 3200**

[ 2 \* 3200 \* 3200\* 3200 – 3200 \* 3200] / [6.225576\*1000000000] **GFLOPS**

= **10.525252 GFLOPS**

