SAURAV RAI
17558

MATRIX MATRIX MULTIPLICATION

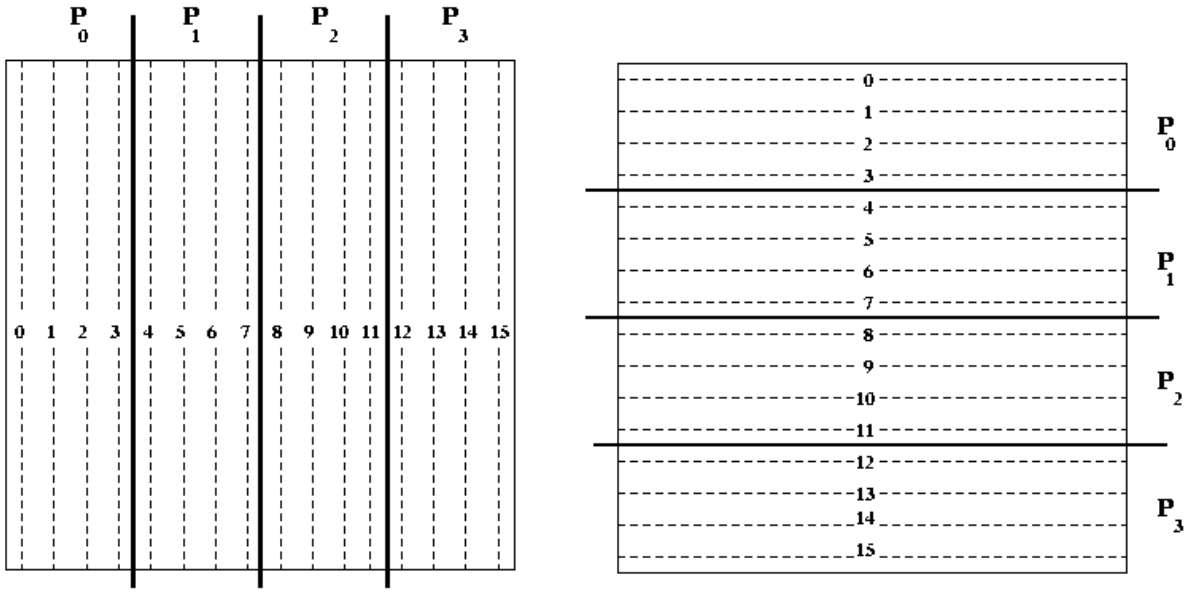
Objective:

• To write a MPI CUDA program, for computing the matrix -matrix multiplication on p processors of a message passing cluster using block striped partitioning of a matrix.

Description:

• In the striped partitioning of a matrix, the matrix is divided into groups of contiguous complete rows or columns, and each processor is assigned one such group.

The partitioning is called block-striped if each processor is assigned contiguous rows or columns. Striped partitioning can be block or cyclic. In a row-wise block striping of an nxn matrix on p processors (labeled *P0 , P1, P2,..., P -1*) , processor Pi contains rows with indices *(n/p)i, (n/p)i+1, (n/p)i+2, (n/p)i+3,....., (n/p)(i+1)-1.* A typical column-wise or row-wise partitioning of 16 x 16 matrix on 4 processors is shown in the following Figure 2.



(a) Columnwise block stripping (b) Rowwise block stripping

• The matrix A of size n x n is striped row-wise among p processes so that each process stores n/p rows of the matrix. We assume that the vector x of size n x 1 is available on each process . Now, process Pi computes the dot product of the corresponding rows of the matrix A[*] with the vector B[*] by calling the CUDA kernel Matrix Vector Multiplication and accumulate the partial result in the vector My Result vector[*]. Finally, process P0 collects the dot product of different rows of the matrix with the vector from all the processes.

Implementation of Matrix Vector Multiplication :

Step 1 : Five arrays are required for computation. One each, for storing Matrix, vector, result vector, some part of the Matrix and the some part of the result vector.
Step 2 : Root processor initializes the Matrix, vector and the resultant vector. The matrix and the vector are constructed by assigning one to each element and the result vector is initialized to zero.
Step 3: Vector B is broadcasted to all the processors from the root node.
Step 4 : Scatter size is computed.
Step 5 : Memory is allocated for MyMatrixA and the MyResultVector by all the nodes.
Step 6 : Process with rank 0 distributes the Matrix using MPI_Scatter on to p processes.
Step 7 : Similar vectors are allocated on the device. The values of the vectors in the host machine are copied to the vectors on the device machine.
Step 8 : Each node computes the Matrix vector multiplication by calling the CUDA kernel MatrixMatrixMultiplication and copies back the result to MyResultVector on the host.
Step 9 : Processor 0 gathers from all the nodes and form the final result vector.
Step10 : Process with rank 0 prints the resultant vector.

CUDA API used :

To Allocate memory on device-GPU :
cudaMalloc(void** array, int size)

To Free memory allocated on device-GPU:
cudaFree(void* array)

To transfer from host-CPU to device-GPU:
cudaMemcpy((void*)device_array, (void*)host_array, size, cudaMemcpyHostToDevice)

To transfer from device-GPU to host-GPU:
cudaMemcpy((void*)host_array, (void*)device_array, size, cudaMemcpyDeviceToHost)

INPUT

The input to the problem is given as arguments in the command line. It should be given in the following format . Suppose the dimension of the matrix is mxn, size of the vector is n and the number of processors is *p*.

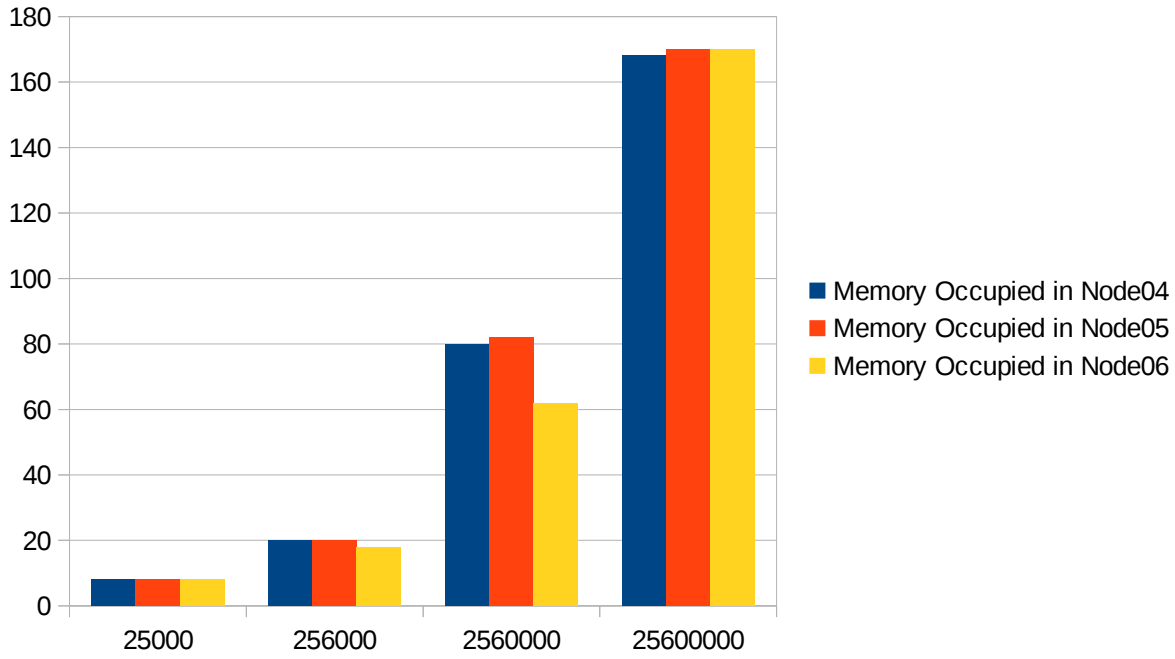
mpirun -n p ./Matrix_Vector_Multiply m n n

Process 0 generates the Matrix and the vector.

OUTPUT

Process 0 prints the final vector.

Input Size	Memory Occupied in Node04	Memory Occupied in Node05	Memory Occupied in Node06
25000	8	8	8
256000	20	20	18
2560000	80	82	62
25600000	168	170	170



RESULT : The memory is almost equally distributed to all the three nodes.Hence the work is also equally distributed among them.