Develop a matrix processing program that runs in parallel using MPI. The program should do an element-by-element (C[i] = A[i]*B[i]) multiplication in parallel. The element-by-element multiplication function should take three matrices (A, B, and C; all of size n x n) as arguments (passed by reference): two matrices (A and B) should be used as input, and the third for output (C). You can pass additional arguments to the function as well. In addition, each function should have two versions: one that operates on a matrix of single-precision floating point numbers, and another that operates on a matrix of double-precision floating point numbers. I would suggest creating two .C programs (you can compile and run these separately), one for single-precision floating point matrices, and another for double-precision floating point matrices.

Also include a function that performs a sum across all matrix elements and run it on your product matrix (C) after the element-by-element multiplication. This should be accomplished using a single MPI_Reduce() call followed by a for loop to sum across the resulting array.

Command line arguments: From the command line you should pass the matrix size, the name of a file containing a matrix, and a flag indicating whether you want to print results to stdout. For example:

```plaintext
mpirun -n 4 ./process_matrix -f my_matrix.txt -n 3 -o
```

could launch a four-process MPI program for processing a 3 x 3 matrix written in my_matrix.txt, and write the results (the product matrix and the sum across all matrix elements) to stdout. Like in the previous assignment, if no file name is passed, your code should fill out the matrix arrays with random floating point numbers between 0 and 20. Matrix A and matrix B should be filled with different numbers unless you are reading a matrix from a file, in which case you can just let A = B.

Use MPI commands (mpicc, mpirun) available through Intel’s library to build and run the program (setpksgs –a intel_cluster_studio_compiler). Run extensive tests on your code to ensure that it is producing consistent results with the serial version.

Run benchmarks of your code, and generate plots for speedup and efficiency as a function of matrix size and number of MPI processes (4 plots total: speedup vs. nprocesses; speedup vs. n; efficiency vs. nprocesses; efficiency vs. n). In each plot, include results for single- and double-precision floating point matrix processing.
For the benchmarks, you can use the Linux “time” command to measure the execution time (use “real” time). Generate random input matrices and do not output the product matrix to stdout. Test the following conditions for both your single-precision and double-precision programs:

processes = 1; n=100,500,1000,2000,5000
processes = 4; n=100,500,1000,2000,5000
processes = 8; n=100,500,1000,2000,5000
processes = 16; n=100,500,1000,2000,5000
processes = 32; n=100,500,1000,2000,5000

Submit a SLURM script for running your benchmarks. To ensure your benchmarks get enough wall time and memory, I would recommend a script that looks something like this:

```
#!/bin/bash
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=8
#SBATCH --constrain=intel
#SBATCH --time=1-00:00:00
#SBATCH --mem=50G
setpkgs -a intel_cluster_studio_compiler
export I_MPI_PMI_LIBRARY=/usr/scheduler/slurm/lib/libpmi.so
echo "Running benchmark with p=1,n=100..."
time srun --nodes=1 --ntasks=1 ./mat_multiply -n 100
echo "Running benchmark with p=4,n=100..."
time srun --nodes=1 --ntasks=4 ./mat_multiply -n 100
echo "Running benchmark with p=8,n=100..."
time srun --nodes=1 --ntasks=8 ./mat_multiply -n 100
echo "Running benchmark with p=8,n=100..."
time srun --nodes=2 --ntasks=16 ./mat_multiply -n 100
echo "Running benchmark with p=16,n=100..."
time srun --nodes=4 --ntasks=32 ./mat_multiply -n 100
...
...
```

To save yourself time, you could put the single-precision and double-precision versions of the code in two separate sub-directories, but call the executable by the same name. That way, you can use the exact same SLURM script for running the different sets of benchmarks.
Comment on the performance of your code and point me to the location of your programs.