Name: Ashwin Ashokan

UIN:230002171

HW2 CSCE 689-Parallel Computing

1. Description on How to Compile and run the code on a parallel computer:

- The Entire source code is present in the folder "**code**", paste and execute the following commands
- Enter **"make**" command within the given folder.
- It will create an executable "serMatInv.exe"
- To execute the .exe file in dedicated mode you can use the .job file provided by running the **"bsub < computeinverse.job**.
- To run it individually you must first execute "export OMP_NUM_THREADS=20" first prior to regular execution.
- But however, to simply run the file on a single node you can just do regular execution such as "./serMatInv.exe 1024 20"
- The first number = rank of matrix, second argument is number of threads, based on the setting of environment variable which should be done prior hand.

2. Strategy to Parallelize the Matrix Inverse Algorithm:

- I have used "**Gauss-Jordan Elimination Method"** to calculate the inverse of a matrix once the size dimension of matrix goes down to 16.
- Wherein I have used **"#pragma omp parallel for"** and used clause **schedule (dynamic, num_rows/omp_get_num_threads())** to split the for loop as work batches across different threads.
- Since recursion **R11** and **R22** in the compute inverse function can be done independently without any race condition I have use openmp's **task construct** to run each of the recursion to as an independent task.
- Since each recursive task runs on a separate thread, the visualization of the recursion across different threads will look like a binary tree.
- And moreover since Matrix Multiplication runs in $O(n^3)$ time, I have used **#pragma omp for** to split the workload of for loop across different threads which will further parallelize the outer loop running across different threads
- However, since **R12** is dependent on the computation of **R11 and R12**, synchronization must be done to ensure that there is no Race condition between the two tasks.
- Hence, I have use **#pragma taskwait,** i.e. it waits until the synchronization of both the R11 and R22 task to finish and then compute the R12 part of the matrix.
- Overall, there is significant improvement, but however there is too much copying and modularity involved that causes stagnation in speedup certain Matrix Size,and thread size.

- By setting the ptile = 20, I assure that 20 cores of the node are assigned for computation.
- Since there is too much of overhead in thread allocation and task creation for recursive tasks, threads don't parallelize the code as much as expected.
- The inverse of the matrix is computed properly, but somehow its inverse error rate increase by parallelizing the code, which I am not sure why it happens.
- The tasks consume a considerable time for allocation and private variable allocation, hence even though speedup increases linearly, it tends to increase the error rate proportionally.
- However, the serial code computes the inverse perfectly, and I have used complete object-oriented approach for modularity and ease of understanding on how the code works. The code is almost self-explanatory.

3. SPEEDUP AND EFFICIENCY TABLE FOR VARIOUS MATRIX SIZES AND THREADS ALLOCATED.

Processor Count = 1

MATRIX SIZE	SPEEDUP	EFFICIENCY
128	0.858	0.858
256	0.888	0.888
1024	0.756	0.756
2048	0.752	0.752

Processor Count=8

MATRIX SIZE	SPEEDUP	EFFICIENCY
128	1.00645	0.125806
256	2.0645	0.258063
1024	3.8654	0.483175
2048	4.1223	0.515288

Processor Count=20

MATRIX SIZE	SPEEDUP	EFFICIENCY
128	1.5646	0.07823
256	3.002	0.1501
1024	4.562	0.2281
2048	5.231	0.26155