Quang Anh Hoang (qh62@drexel.edu)

Gaussian Elimination

ECEC 622: Parallel Computer Architecture
Professor Naga Kandasamy

1. Introduction
   Gaussian elimination is a method which is used for solving a matrix equation: Ax = B, where A is an n x n matrix, and x and B are n x 1 matrices. This method's goal is to achieve a specific form of A, which is called "row echelon form". In this form, the coefficients lying on the diagonal line of the matrix is reduced to 1. Those coefficients on the lower half of the diagonal is reduced to 0, and those on the other half will have different values so that the matrix does not change its value.
   Gaussian elimination involves two steps to reduce each row consecutively: division step and elimination step. In division step, every coefficient is divided by the leading coefficient (the one lying on the diagonal line of the matrix). This will make the leading coefficient's value to be 1, while the remainings have a specific value so that the ratio between each coefficient and the leading one of that row does not change. In elimination step, for each row below the divided one, each coefficient is reduced by a multiple of that from the same column but on the divided row, and the multiplier will be the one from that row sharing the column with the leading coefficient in that iteration. This will make that coefficient's value to be 0, and eventually all coefficients below the diagonal line will be 0.
   The main task of this assignment is parallelizing the division and elimination steps with a barrier synchonization in between to speedup the program. For each iteration, from the top row to bottom one, there will be a number of threads generated to execute dividing function in parallel (using chunking method on coefficients on the right of leading one). Then barrier sync is used to wait for all threads to finish their task before moving on to the eliminating function, where each thread takes a number of rows (approximately the same in chunking method) to reduce the coefficients.
2. Implementation Design
   2.1. Threads' function:
   The first function assigned to each [worker] thread is a general function type-casting the argument from the main thread, and in this thread function, two other functions are called: divide and eliminate functions. Between those two functions, there is a barrier sync function to synchronize all threads before moving to the next function. In this assignment, the barrier sync is from the pthread library:
   *pthread_barrier_wait(pthread_barrier_t *barrier)*

   a) Divide function:
      This function starts with one row, from left to right, takes the leading coefficient's value and divide those on the right with it. Each thread is assigned to take in a number, called "chunksize", of coefficients. Chunksize is calculated as the number of coefficients on the right divided by the number of threads and rounded down to the nearest integer. To accommodate the last few rows where the number of remaining elements is smaller than number of threads, the first threads will be assigned with 1 coefficient when others do not have any coefficient to execute. Each thread also starts at different point, called "offset". Offset is computes as the row number it is executing on plus the thread id plus one (this value of one serves to move the starting point of each thread to the right by one, so that the value of leading coefficient remains the same throughout executions of all threads.

b) Eliminate function:
This function’s job is eliminating all rows below the divided one to the echelon form, in which on the same column of the leading coefficient, ones on lower rows are reduced to 0. Since these rows can be “eliminated” in parallel, each thread is assigned to work on a range of rows. The range is also called “chunksize”, computed by dividing the number of remaining rows by number of threads and rounded down to the nearest integer. Similar to divide function, in the last few iterations, when number of rows is smalled than number of threads, the first threads (with small thread id) will have the chunksize value to be 1, while others’ are 0. On each row within the assigned range, each thread takes the coefficients on that row and reduced by a multiple of coefficients on the divided row but at the same column respectively.

3. Result
There are twelve different configurations used in this assignment to examine the performance of parallelizing Gaussian Elimination, with three different values of number of threads: 4, 8 and 16; four different values of matrix size: 512 x 512, 1024 x 1024, 2048 x 2048, and 4096 x 4096. Each configuration is run for 20 times and speedups are recorded and shown in the tables below for each method:

	4 threads	8 threads	16 threads
512 x 512	0.3175764	0.24986215	0.2123914
1024 x 1024	0.6565998	0.5875595	0.49665835
2048 x 2048	0.9366296	0.84488045	0.7825092
4096 x 4096	0.9590732	0.92751255	0.9092636

Table 1: Average speedup of multi-thread Gaussian Elimination function, chunking method for both divide and eliminate functions

	4 threads	8 threads	16 threads
512 x 512	1.285614	0.381968	0.302054
1024 x 1024	0.855325	0.704643	0.566965
2048 x 2048	1.215615	0.875288	0.896119
4096 x 4096	1.027174	1.11119	0.941366

Table 2: Maximum speedup of multi-thread Gaussian Elimination function, chunking method for both divide and eliminate functions

According to the collected result, the average speedup on every configuration cannot surpass 1, which means no performance accelaration was gained. This can be explained as the overhead of creating multiple threads using this method is too large that at the given matrix size, no speedup could be achieved. This issue could be relieved by using better algorithm and cutting down on excessive latency.
However, there are some positive results at maximum speedup of this program: with 4 threads, the program can get up to 1.28 times speedup executing 512 x 512 matrix, and 1.21 times for 2048 x 2048 matrix. According to the tables of results, using 4 threads is proved to perform better than 8 and 16 threads, as the overhead would be too large and cannot outweigh the single thread program.