Assignment 2 - COL380

Data races

For loop j:

The value of L[i][k] is read when computing L[i][j] where k is less than j. It means that to compute L[i][j], we need correct values of L[i][k] for 0<=k<j, which are computed in the previous iterations. So, we are accessing L[i][j] in multiple iterations where one of these is a read and other is write.

For loop i:

In this loop, we are accessing the values of L[i][j] where j is constant (for this loop) and i is different for every iteration. In the sub-loop with variable k, we use the values varying the y coordinates i.e. we access L[i][k] where k is varied from 0 to j-1. As these values are already set in the previous iterations of loop j, they are now fixed and will not cause any data races.

For loop k:

We are accessing the value of sum calculated in the previous iteration to calculate the new value of sum in the current iteration. Hence sum is both being written and read in different iteration, so data races present.

Strategy 1: Parallel for

To parallelize the code using parallel for loop, we had to select a loop in the function for parallelization. Parallelizing the for loop with variable j would be most efficient, but as mentioned above it contains data race. Therefore, the next loop that is the for loop with variable i is being parallelized.
The only possibility of data races in this parallelization would be if variable sum is common in all loops as it is being read and written in each iteration. To avoid this issue, all private and shared variables are mentioned when the command for parallelization is written.

Strategy 2: Sections

To parallelize the code using sections, we divided each iteration of loop j into two sections, one for lower triangular matrix and other for upper triangular matrix. But doing this alone would have resulted in a data race as L[j][j] was updated in the lower triangular matrix region and was being used in the upper triangular matrix region. To avoid this data race, the value of L[j][j] was calculated before parallelizing the code, so that whenever it is accessed it is already updated.

Strategy 3: Parallel for and sections

The two strategies used above, parallelized the code by sections and loops individually. To use them together, we simply merged both the approaches i.e., made sections for lower and upper triangular matrix calculations and parallelized the loops with variable i in each of these sections. The data dependency handling is done in the same way as done in the previous two strategies.

Strategy 4: MPI

To parallelize the code using MPI, we decided to implement the parallelization in the loop for k. We are assigning each a process a chunk of iterations so that every process can calculate its partial sum and then sends it back to the root process which can calculate the total sum and then use it to change the values of L[i][j] and U[j][i] in the (i,j) iteration.
Since, to calculate the correct sum in each iteration (i,j) every process needs the updated values of L and U from the previous (i-1,j) iteration. To accomplish this, at the end of each (i,j) iteration, the root process updates L and U and then sends the updated matrices to every process. Except for the first time, when each process has the original L and U, other processes before starting to calculate their partial sums, they first receive the updated values of L and U from the root process and then use these to calculate the partial sums.

Results:

For n=100, the results are tabulated below:

Number of threads	**Strategy 0**	**Strategy 1**	**Strategy 2**	**Strategy 3**	**Strategy 4**
2	0.017501	0.016946	0.019111	0.021978	0.204523
4	0.021413	0.021097	0.019348	0.041671	0.407620
8	0.018696	0.022183	0.028082	0.072788	1.353400
16	0.018519	0.026545	0.022720	0.126374	2.065548

MPI is unexpectedly more time because of the low value of n. Also, the current implementation of MPI is only parallelising the loop of k. If additional parallelizations are done for the i and j loop, the performance can improve significantly.