The Crout matrix decomposition is an LU decomposition that decomposes a matrix into a lower triangular matrix (L`), an upper triangular matrix (U) and, although not always needed, a permutation matrix (P). It was developed by Prescott Durand Crout. Crout method returns a lower triangular matrix and a unit upper triangular matrix.
So, if a matrix decomposition of a matrix A is such that:

```
A = L`DU
```

being L` a unit lower triangular matrix, D a diagonal matrix and U a unit upper triangular matrix, then Crout's method produces

```
A = (L`D)U = LU
```

Sequential Program.

```
void crout(double const **A, double **L, double **U, int n) {
        int i, j, k;
        double sum = 0;
        for (i = 0; i < n; i++) {
                U[i][i] = 1;
        }
        for (j = 0; j < n; j++) {
                for (i = j; i < n; i++) {
                        sum = 0;
                        for (k = 0; k < j; k++) {
                                sum = sum + L[i][k] * U[k][j];
                        }
                        L[i][j] = A[i][j] - sum;
                }
                for (i = j; i < n; i++) {
                        sum = 0;
                        for(k = 0; k < j; k++) {
                                sum = sum + L[j][k] * U[k][i];
                        }
                        if (L[j][j] == 0) {
                                exit(0);
                        }
                        U[j][i] = (A[j][i] - sum) / L[j][j];
                }
        }
}
```

NOTE: the input to the program is a square matrix A, and the outputs are a lower triangular matrix L and a **unit** upper triangular matrix U such that A = LU.

0. Strategy 0 is the sequential program (that's already been implemented, you must include this in your code)

Implement the following versions using OMP(strategy 1,2,3) and MPI(strategy 4):

1. Develop the first version using `parallel for' construct with other appropriate clauses. (Marks: 4)

2. Develop the second version using `sections' construct with other appropriate clauses. (Marks: 4)
3. Use both `parallel for' and `sections' constructs with other appropriate clauses to develop the parallel program. (Marks: 6)

Do not use `reduction' or `atomic' clauses in any implementation.

4. Write an MPI version that solves the problem in a distributed manner. (Marks: 6)

The program should contain four functions: One for the serial program and other four for other versions as specified above.

Compute the results for 2, 4, 8, 16 threads (for OpenMP) or processes(MPI).

Input format:
(i) n : number of rows and columns of the square matrix
(ii) filename that contains an n*n matrix (A)
(iii) number of threads
(iv) strategy (1/2/3/4)
Your code should be executed in the following way: `./a.out 6 input.txt 8 3`
Here we your code should be expecting a 6*6 matrix in the file input.txt and run the strategy 3 using 8 threads

Output format: for each strategy print the two matrices (L and U) to individual output files.
Output file name: `output_(L/U)_<strategy(1/2/3/4)>_<number of threads/processes(2/4/8/16)>.txt`
The double values should have precision upto three places.

Submit a report:
(1) Explain your approaches.
(2) In each of these cases which locations have potential data races? How do you guarantee correctness by avoiding data races without using atomic construct?

==========

Reference: https://en.wikipedia.org/wiki/Crout_matrix_decomposition