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 have the above four functionalities along with the serial mode
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 (0/1/2/3/4)
~~Your code should be executed in the following way: `./a.out 6 input.txt 8 3`~~
Your code will be executed in the following way: `bash run.sh 6 input.txt 8 3`
You also need to provide a compile.sh which would be run in the following way : `bash compile.sh` before `run.sh`. You are **NOT** allowed to use any optimisation flags (like -O3) during compilation.

Here your program should be expecting a 6*6 matrix in the file input.txt and run 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(0/1/2/3/4)>_<number of threads/processes(2/4/8/16)>.txt`

Everyone having different printing functions will affect the running time comparison and that is why we are standardising the precision(12) and providing the code for printing. For all strategies, the printing function/code should be the same as given below. **Non-compliance will lead to zero marks. Write both matrices to files serially. Do not try to parallelise the output writing part. That is not the aim of the assignment.**

```
void write_output(char fname[], double** arr, int n ){
        FILE *f = fopen(fname, "w");
        for( int i = 0; i < n; i++){
                for(int j = 0; j < n; j++){
                        fprintf(f, "%0.12f ", arr[i][j]);
                }
                fprintf(f, "\n");
        }
        fclose(f);
}
```

Regarding correctness evaluation,

1. U and L are supposed to be triangular (upper and lower respectively)
2. | det(U) - 1 | < 1e-3
3. We compute A_dash = L*U, and check that |A_dash[i][j] - A[i][j]| < 1e-3 for all i,j

Include 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?

Bash scripts you need to provide:

1. compile.sh which compiles your code. Specifically running “bash compile.sh” in your submission folder should create all the binaries that you require.
2. run.sh as specified above.

Submission Instruction :

1. There should be only one submission per group
2. Upload EntryNumber1_EntryNumber2.zip on moodle. Ex. 2016CS50625_2016CS50619.zip (If you are not in a team, upload EntryNumber.zip on moodle). On unzipping(running unzip file.zip) it should produce one folder. The folder should have the same name as the zip file. This folder should contain all the source files and all the bash scripts. In addition, it should also contain the report.
3. Incorrect submissions or missing entry numbers from the zip would lead to **zero** marks.

Compiler Specifications

1. gcc version 7.5.0
2. mpiexec version ~~1.10.2~~ **2.1.1**

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