Assignment 2

Loop labels :-

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>

// Sequential Programm for computing LU decomposition
void crout_sequential(double **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];
        }

    }
}
```

LOOP 1

LOOP 2

LOOP 3

LOOP 4

LOOP 5

Analysing the sequential program :-

1. The sequential program has 3 nested loops, namely, Loop 1,2,3 and Loop 1,4,5. So, we can conclude that whatever holds for Loop 1,2,3 will be almost true for Loop 1,4,5 as well.

2. The outermost loop in the sequential program, i.e. Loop 1 cannot be prallelised because of the data race prevailing across the loops, eg.,
when j = 0, L[0][0] is written and when j = 1, L[0][0] is read; which shows the True Dependency prevailing in it.

3. The innermost loops, i..e, Loop 3, 5 can be easily parallelised with the loop just above them, that is Loop 2 and 3 can be parallelised together; Loop 4 and 5 can be parallelised together.

4. The two loops inside Loop 1 namely , Loop 2 and 4 cannot be parallelised because of the dependency of the value L[j][j] which is basically written in Loop 2 (upper loop) and is read in every iteration of the Loop 4 (lower loop). But because of the fact that this limitation only exist for a single value of i = j, hence this dependency can easily be removed if we first calculate the value of L[j][j] and then we can parallelise the remaining part of the Loop 2

(upper loop) together with the Loop 4 (lower loop).

STRATEGIES :-

1. STRATEGY 0 :-

-> The label 0 is given to the **sequential code** provided for LU decomposition in this assignment. No change has been made to that code.

2. STRATEGY 1 :-

-> In this strategy the **"parallel for"** construct of the openmp library is used.

- **Approaches Tried :-**
  To parallelise the loop the first approach we tried was as follows :-
  → The data dependency of the outermost loop was not removable, hence it wasn't parallelised.
  → The inner loops were parallelised, i.e., Loop 2 and 3 were parallelised together by collapsing them together and same was the case with Loop 4 and 5. For collapsing "collapse(2)" construct of omp library has been used.
  → All the global pointers wer made local, so that different copies can be passed to different threads.
  → When we collapse Loop 2,3 then the values of the Lower matrix could not be changed in the loop and same is the case with the lower loop and the Upper matrix. So, for updating them we made another loop just at the end of the upper loops 2,3 for updating L and another loop just at the end of the upper loops 4,5 for updating U.
  → So, in the collapsed parallelised loops as well instead of the local variable sum, we used a sum_array[n-j], so that we can store all the sums which we get in different iterations and then finally update L and U in one complete loop.
  → To update the sum_array in the parallelised loop we have to use "critical" construct so as to avoid the bugs.
  → Loop 2,3 are parallelised first and after completion of those Loop 4,5 are parallelised.

- **Problems with this approach :-**
  → The result of this approach is absolutely correct but the main problem which exist is that this approach takes a lot of time to complete.
  → The much more time taken by this approach can be attributed to the existence of the critical section in each iteration of the loop. So, as we know that the more the number of critical sections the more is the time taken by the algorithm, so we have tried another approach which takes less time.

- **New startegy :-**
  → In the new strategy the initialisation of the elements of the matrix L and U, and

then the initialisation of the diagonal entries of the matrix U is done in the same way using pragma_omp_parallel_for and pragma_omp_barrier at the end of the loop.
→ Loop 1 is not parallelised in this strategy as well because of the fact that the loop carried dependency in loop 1 can't be removed.
→ We did not collapsed Loop 2,3 and neither Loop 4,5 and we decided to prallelise only the outer loops that is loop 2 and loop 4 because as taught in the class, we muct always try to parallelise the outer loop if it is possible because the benefit of time which it gives is greater than the one given by the inner loop.
→ In this way the critical sections are reduced to 0. And all the loop variales, variable sum etc. are made local so that copies of those can be used by different threads.
→ In the analysation of the sequential code we came to the 4th conclusion(written above). So in our new strategy we handled the data dependency in Loop 2 and 4 by running an iteration of i=j, before parallelising so that both the loop Loop 2 ans 4 can be parallelised together and hence reducing half of the work.

- **How possible data race is avoided :-**
  → The first possible data race which can occur in this strategy is because of the global loop variables and the sum variable given in the sequential code. So, all the global variables are made local.
  → A significant data race can occur while parallelising loop 2 and 4 together because of L[j][j], so to avoid this the iteration i=j is carried out sequentially first before parallelising the loops combinedly.
  →And in this the check L[j][j] == 0 is also done after that sequential run of i=j itself.

3. STRATEGY 2 :-

-> In this strategy the **"sections"** construct of the openmp library is used.

- **Approaches Tried :-**
  The first approach which was tried was as follows :-
  → In the first approach the initialisation of the matrix L,U and the initialisation of the diagonal elements of the matrix U is done by dividing the loop into 8 sections.
  → Then in the innermost loop which is Loop 3 and Loop 5 the calculation of the variable sum is done parallely by dividing the sum into 8 parts and then giving each part to each section.

  The first approach which was tried was as follows :-
  → The initialisation part is the same as the above approach.
  → In the Loop 2 a recursive approach is used to parallelise it in which the Loop 2 is recursive divided in at most 4 parts and then each part is given total_threads/4 number of threads and then each part divides the Loop 3 into atmost 4 parts and those parts are carried out by the total threads given to that part.
  → The same approach is used to parallelise Loop 4 and 5.

- **Problems with this approach :-**
  → The first approach takes a lot of time because of few reasons, first being that Loop 2 ans 4 are not parallelised and only 3 and 5 are parallelised. But Loop 2 and 4 being the outer one contributes to more time and parallelising 3 and 5 doesn't effect much.
  → In the first approach to update the value of sum we used critical construct in each section which contributes in increasing the time.
  → In the seond approach the nestism is a lot which contributes to the increased time because as nestism increases, the forking and joining of threads at every call increases the overhead.

- **New strategy :-**
  → The new startegy is almost same as the first approach tried in this.
  → The initialisation of U and L and that of the diagonal elements of U is done by dividing the loop into 8 sections.
  → Loop 2 and Loop 4 is parallelised together by dividing into 8 sections and using two different variables for the different sums being calculated in the Loop 2 and 4.
  → To avoid the race of L[j][j] between loop 2 and loop 4, we carry out the iteration i-j sequentially.

- **How possible data race is avoided :-**
  → A data race can occur while parallelising loop 2 and 4 together because of L[j][j], so to avoid this the iteration i=j is carried out sequentially first before parallelising the loops combinedly. The check L[j][j] == 0 is also done after that sequential run of i=j itself.
  → Possible data race which can occur in this strategy is because of the global loop variables and the sum variable given in the sequential code. So, all the global variables are made local.

4. STRATEGY 3 :-

-> In this strategy the **"parallel for"** and **"sections"** construct of the openmp library are used together.

- **Approaches Tried :-**
  To parallelise the loop the first approach tried was :-
  → The initialisation part of the strategy follows the same sections method.
  → The initialisation of the diagonal elements of U is done by using a recursive method in which in in each the threads are divided to the blocks in a recursive manner.
  → Nested parallelism was tried to parallelise Loop 2 ans 4 together in which at first both of them were divided into two sections using "pragma omp sections" and after that each section was parallelised using "pragma omp parallel for".

- **Problems with this approach :-**
  → The time taken by this approach is even more that of the strategy 1 as well as strategy 2 because of the nested parallelism which is being implemented in this.Nested parallelism increases the overhead of the forking and joining at each of the step and hence the time subsequently increase. Hence it is discouraged by default.
  → So, the option which is left is to use the strategy from 1 and 2 which takes less time in the main loop.

- **New strategy :-**
  → In the new strategy, the initialisation part of the strategy follows the same sections method.
  → The initialisation of the diagonal elements of U is done by using a recursive method in which in in each the threads are divided to the blocks in a recursive manner.
  → Since Strategy 1 takes less time than the Strategy 2, the loops inside the Loop 1 are parallelised in the same way as done in the Strategy 1, which includes a sequential run of i=j followed by the combined parallelisation of Loop 2 and Loop 4.

- **How possible data race is avoided :-**
  → Since this is a combined strategy of 1 and 2 , so all the measures taken in Strategy 1 and 2 to avoid data dependencies are used in this as well.

5. STRATEGY 4 :-

-> In this strategy the **"mpi interface"** of the mpi library has been used.

- **Approaches Tried :-**
  → In this the first approach which was tried was based on the Trapezoidal sum parallelisation using MPI interface given in the book by Peter Pacheco.
  → According to the implementation with refrence to this approach, the sum in Loop 3 and Loop 5 was being divided into different parts and each part was being calculated using different MPI processess.
  → Then all the processes send their sum to the process 0 which then adds up all the sum and writes it in the concerned block of matrix L and U.
  → Finally process 0 broadcasts this to the other processes.

- **Problems with this approach :-**
  → This approach was computationally too heavy because a lot of communication was being done among the processes.
  → And hence this was failing even if we give more than two proccesses as input.

- **New strategy :-**
  → In this approach the loops Loop 2 and Loop 4 are being parallelised by the different processes generated by MPI_Init.
  → The value variable of these loops is distributed to the processes in a cyclic partioning manner, i.e., the process having the = i1 % comm_size is given the $i1^{th}$ iteration.
  → And then as each processes calculate the block of L and U it broadcasts it to the other processes using MPI_Bcast.

- **How possible data race is avoided :-**
  → Values are broadcasted among different processes regularly so that no process uses a wrong value.
  →Possible data race can occur between Loop 2 and Loop 4 and to avoid this Loop 2 is carried out before Loop 4.
  → To avoid any data race of the loop variable, local loop variable is given to every iteration of the loop instead of having just a global loop variable.

Results :-

For a square matrix of rows = columns = 100.

Threads	Strategy 0	Strategy 1	Strategy 2	Strategy 3
2	0.001612	0.003071	0.001335	0.000998
4	0.001612	0.001018	0.001893	0.002987
8	0.001612	0.002813	0.002028	0.002846
16	0.001612	0.005406	0.002321	0.005433

For a square matrix of rows = columns = 1000.

Threads	Strategy 0	Strategy 1	Strategy 2	Strategy 3
2	3.063346	1.952806	4.124936	2.100750
4	3.063346	1.524797	3.653296	1.681314
8	3.063346	1.481175	3.565294	1.728593
16	3.063346	1.633080	3.456821	1.711813

Submitted By :-

Nikita Bhamu → 2018CS50413
Vijay Kumar Meena → 2017CS50421