PAP - Assignment - 02

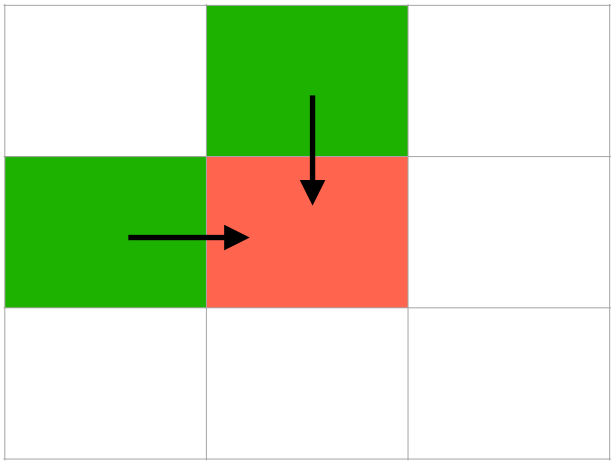
Roham Jabbari - a12211638

In this assignment, we were tasked to implement Length of the longest common subsequence (LLCS) using OpenMP.

In the serial implementation of LLCS, we fill up each block of the matrix M one by one. Since X and Y strings are very large, this approach needs a long time to finish. However, if we come up with a way to fill the elements simultaneously, we might get a good speedup.
(Serial Speed: 14.58)

First, we need to come up with an algorithm for traversing the matrix elements to get rid of data dependency in the matrix. For filling an element of the matrix, we need the last two elements to be available.

In the picture, the red elements has dependency on green elements. It has to wait for the green elements to be present.



To handle this issue, instead of traversing the matrix in a row/column order, we do it in an anti-diagonal manner.
In this algorithm, we can make sure that for each element that we reach, the dependencies are already accomplished.

		A	B	B	D	C
	0	0	0	0	0	0
B	0	0	1	1	1	1
A	0	1	1	1	1	1
B	0	1	2	2	2	2
D	0	1	2	2	3	3
C	0	1	2	2	3	4

In this piece of code, we have a loop which is incrementing on each anti-diagonal.

```
35 //ANTI-DIAGONAL VERSION NO PARALLEL
36 unsigned long long llcs_antidiagonal(const char *X, const char *Y, unsigned int **M)
37 {
38     unsigned long long entries_visited = 0;
39     int row, col, diagonal;
40     int t = 1024;
41     for(diagonal = 2; diagonal <= LEN+LEN; diagonal+=t){
42         for(row =  MIN(LEN-t+1, diagonal-1);row >= MAX(1, diagonal-LEN);row-=t)
43         {
44             col = diagonal - row;
45             for (int i = row;i < row + t; i++)
46             {
47                 for (int j = col; j < col + t; j++)
48                 {
49                     if (X[i-1] == Y[j-1]) {
50                         M[i][j] = M[i-1][j-1] + 1;
51                     } else if (M[i][j-1] < M[i-1][j]) {
52                         M[i][j] = M[i-1][j];
53                     } else {
54                         M[i][j] = M[i][j-1];
55                     }
56                     entries_visited++;
57                 }
58             }
59         }
60     }
61     return entries_visited;
62 }
```

This loop is giving us each anti-diagonal number and we can get the row and column (similar to Z-Curve from the last assignment) using the following code:

```
42         for(row =  MIN(LEN-t+1, diagonal-1);row >= MAX(1, diagonal-LEN);row-=t)
43         {
44             col = diagonal - row;
```

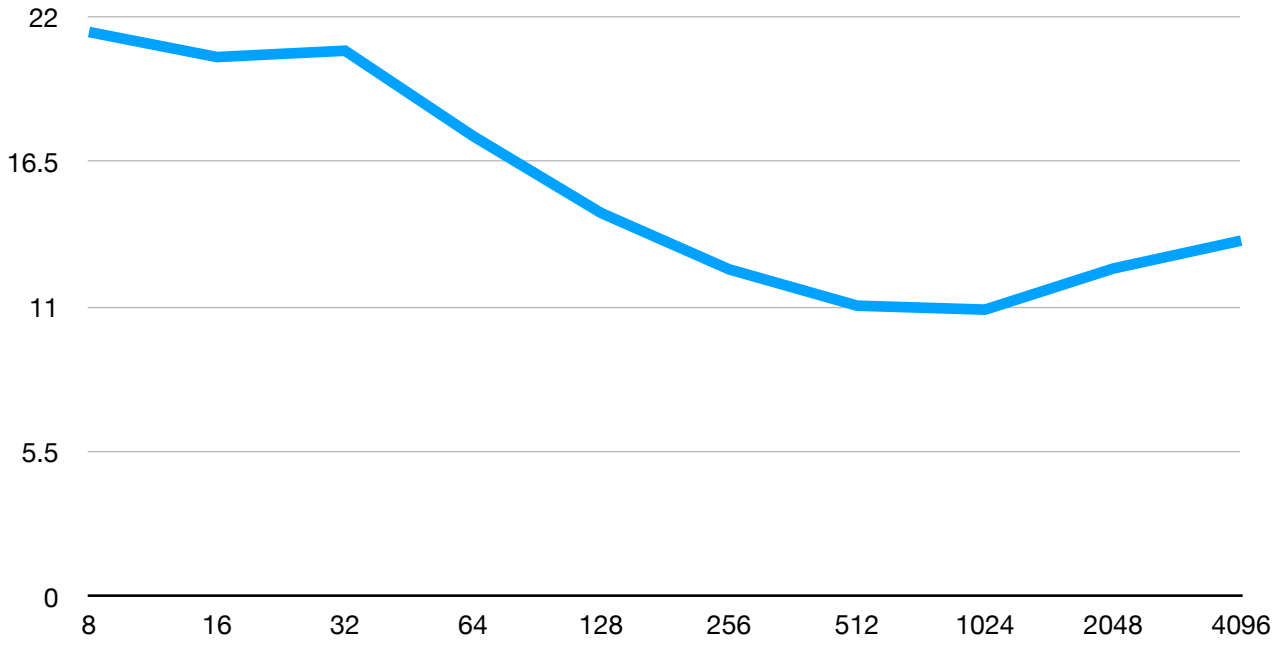
(Note: the part with MIN and MAX is used since we have to iterate until we reach the biggest anti-diagonal and then again to the last, smallest anti-diagonal)
(Source: https://github.com/drberg1000/LCS)

Also, the following block of code was added to make our code NUMA-Aware. This way, we are using cache to get a better speed. We are basically turning our matrix into smaller tiles to fit better in local cache.

```
45             for (int i = row;i < row + t; i++)
46             {
47                 for (int j = col; j < col + t; j++)
48                 {
49                     if (X[i-1] == Y[j-1]) {
50                         M[i][j] = M[i-1][j-1] + 1;
51                     } else if (M[i][j-1] < M[i-1][j]) {
52                         M[i][j] = M[i-1][j];
53                     } else {
54                         M[i][j] = M[i][j-1];
55                     }
56                     entries_visited++;
57                 }
58             }
```

In the following table and chart we can see different speeds with different tile sizes (t).

Tile size (t)	8	16	32	64	128	256	512	1024	2048	4096
Time	21.44	20.49	20.73	17.49	14.57	12.43	11.05	10.90	12.46	13.52



(Note: by using a better algorithm without any parallel code we get 0.13x speedup)

Now it is time to make the code find the elements simultaneously using OpenMP and parallel programming.
There are two approaches used in this assignment, explicit tasks and taskloop.

1. Explicit tasks:

```
67  unsigned long long llcs_parallel_tasks(const char *X, const char *Y, unsigned int **M)
68  {
69      unsigned long long entries_visited = 0;
70      int row, col, diagonal;
71      int t =  512;
72
73  #pragma omp parallel default(shared)
74      {
75  #pragma omp single
76          {
77              for(diagonal = 2; diagonal <= LEN+LEN; diagonal+=t){
78                  for(row =  MIN(LEN-t+1, diagonal-1);row >= MAX(1, diagonal-LEN);row-=t)
79                  {
80                      col = diagonal - row;
81                      int temp = 0;
82  #pragma omp task default(none) \
83  shared(M, entries_visited ) \
84  firstprivate(X, Y, temp, t, diagonal, row, col) \
85  depend(in:M[row+t-1][col-1], M[row-1][col+t-1]) \
86  depend(out:M[row+t-1][col+t-1])
87                      {
88                          for (int i = row;i < row + t; i++)
89                          {
90                              for (int j = col; j < col + t; j++)
91                              {
92                                  if (X[i-1] == Y[j-1]) {
93                                      M[i][j] = M[i-1][j-1] + 1;
94                                  } else if (M[i][j-1] < M[i-1][j]) {
95                                      M[i][j] = M[i-1][j];
96                                  } else {
97                                      M[i][j] = M[i][j-1];
98                                  }
99                                  temp++;
100                             }
101                         }
102 #pragma omp critical
103                         {
104                             entries_visited += temp;
105                         }
106                     }
107                 }
108             }
109         }
110     }
111     return entries_visited;
```

In this approach, we use explicit tasks to make each tile from the matrix into a different task and put it into the task pool for our threads to work on.

```
75  #pragma omp single
```

omp single was used to tell our code that each task should be done only once. If we don't use omp single, each tile will be filled once with each thread

```
103 #pragma omp critical
104                         {
105                             entries_visited += temp;
```

omp critical was used to handle data race in our code. Since entries_visited is a shared variable, each thread is going to write something on it. Some threads may find access to this variable at the same time. This will make data race to happen. To fix this issue, we introduce a temporary variable called temp and let tell our threads to write on temp for the time being. At the end of each task, we add variable temp to entries_visited in a critical clause to handle the data race. We don't want to use critical clause in the loop because critical has a large overhead, this makes our code extremely slow.

```
82 #pragma omp task default(none) \
83 shared(M, entries_visited ) \
84 firstprivate(X, Y, temp, t, diagonal, row, col) \
85 depend(in:M[row+t-1][col-1], M[row-1][col+t-1]) \
86 depend(out:M[row+t-1][col+t-1])
```

To turn each tile into a different task we use omp task.

default was used for better code understanding.

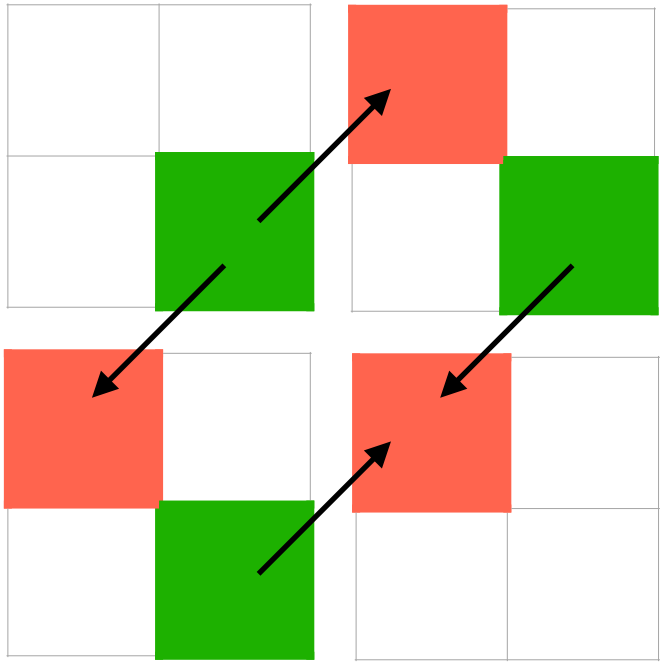
M and entries_visited are introduced to each task as shared since they will be written on using each task.

X, Y, t, diagonal are constants for each task. It really doesn't make any different if we introduce them as shared. However, the principle is to keep variables as first private if possible.

temp, row, col are unique in each task. Therefore, they are introduced as first privates.

The most important part of this assignment was to handle the dependency in this algorithm. For one core we did this by using anti-diagonal. However, when multiple cores are working on our matrix, multiple tiles will be handled simultaneously. This means a thread may be working on a tile that does not have its dependencies fulfilled. To handle this issue we have to introduce the dependencies to each task.

Each red element (start of each tile) depends on two green elements (end of tiles on left and top). This means each tile has to wait until the tiles on the left and top are completed. To make our threads understand this, we use depend-in and depend-out. Depend-in consists of two elements (the last element from the dependency tiles) and Depend-out consists of one element. (The last element of each tile to be given to the next task).



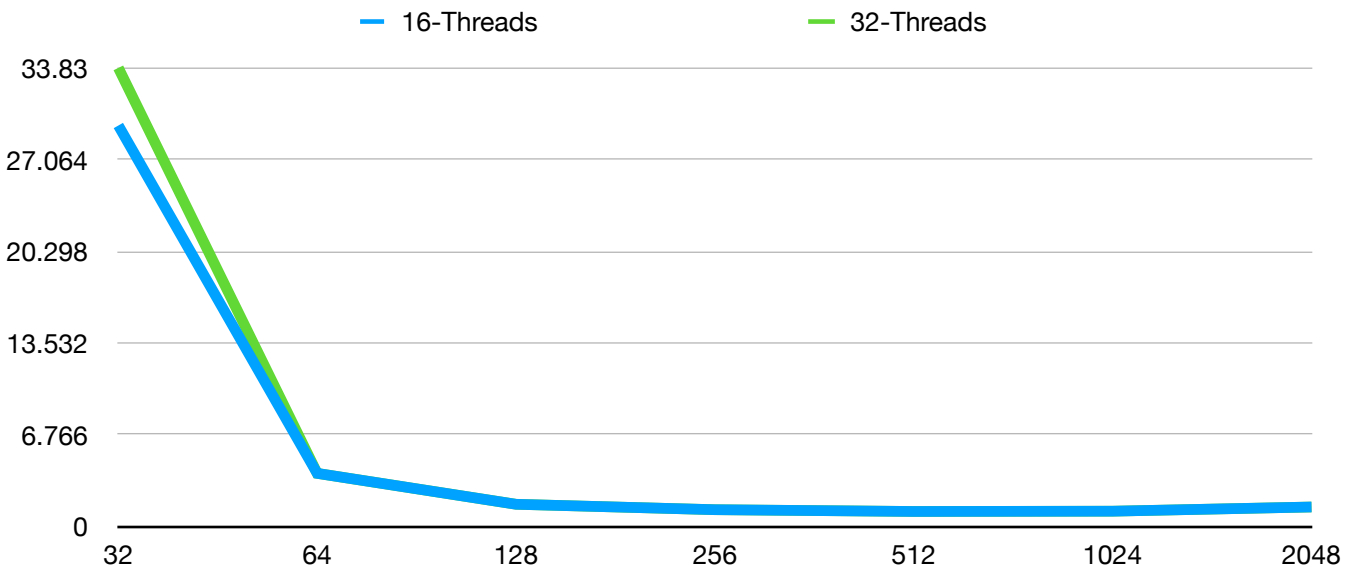
Granularities:

`OMP_NUM_THREADS=32`

Tile size (t)	32	64	128	256	512	1024	2048
Time	33.83	8.35	1.33	1.12	1.00	1.10	1.58

`OMP_NUM_THREADS=16`

Tile size (t)	32	64	128	256	512	1024	2048
Time	29.58	3.92	1.64	1.24	1.11	1.13	1.44



With changing the number of threads we can see that for very large and very small granularities, less threads show better performance. This happens because of the overhead of tasks. Also, because of the task overhead, too many tasks in the task pool cause a reduction of speed.
On the other hand, using very large tiles in our algorithm lowers the speed because of the cache.

In the anti-diagonal approach using only one thread, we had a better result using t = 1024, but it is clear that we are getting a better result using t = 512 (smaller tile size) here in explicit task. This is happening because of the balance between number of Tasks and cache optimisation.

It is worth mentioning that using t = 512 we have between 1 to around 100 tasks in the task pool (100 on the largest anti-diagonal). There may be more than 100 tasks in the pool since tasks are being generated unrelated to the ones being completed.

By looking at the plot and the data table, 512 was chosen as the optimal block size (granularity) since other block sizes have less speed.

Using explicit tasks with the best granularity, we are getting almost **14.6x speedup**.

2. Taskloop:

```
unsigned long long llcs_parallel_taskloop(const char *X, const char *Y, unsigned int **M)
{
    unsigned long long entries_visited = 0;

    int row, col, diagonal;
    int t = 512;
    #pragma omp parallel default(shared)
    {
#pragma omp single
        {
            for(diagonal = 2; diagonal <= LEN+LEN; diagonal+=t){
#pragma omp taskloop \
shared(M) \
firstprivate(X, Y, t, diagonal) \
private(row, col) \
reduction(+:entries_visited) \
grainsize(1)
                for(row =  MIN(LEN-t+1, diagonal-1);row >= MAX(1, diagonal-LEN);row-=t)
                {
                    col = diagonal - row;
                    for (int i = row;i < row + t; i++)
                    {
                        for (int j = col; j < col + t; j++)
                        {
                            if (X[i-1] == Y[j-1]) {
                                M[i][j] = M[i-1][j-1] + 1;
                            } else if (M[i][j-1] < M[i-1][j]) {
                                M[i][j] = M[i-1][j];
                            } else {
                                M[i][j] = M[i][j-1];
                            }
                            entries_visited++;
                        }
                    }
                }
            }
        }
    }
    return entries_visited;
}
```

In this approach, we use taskloop. With using taskloop, OpenMp makes the tasks by itself.

```
#pragma omp taskloop \
shared(M) \
firstprivate(X, Y, t, diagonal) \
private(row, col) \
reduction(+:entries_visited) \
grainsize(1)
```

M is introduced as shared since each task made by taskloop will have to change M.

X, Y, t, diagonal are constant in each task. It will not change if we introduce them as shared. However, based on principle we have to keep our variables as first privates if possible.

row and col are unique in each task. They will be defined in the loop.

To fix the data race issue on entries_visited, we simply put a reduction on this variable. By using reduction, a private copy for the variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

Since we are already tiling the matrix, there is no need for grainsize. Therefore, grainsize=1 gives the best performance here.

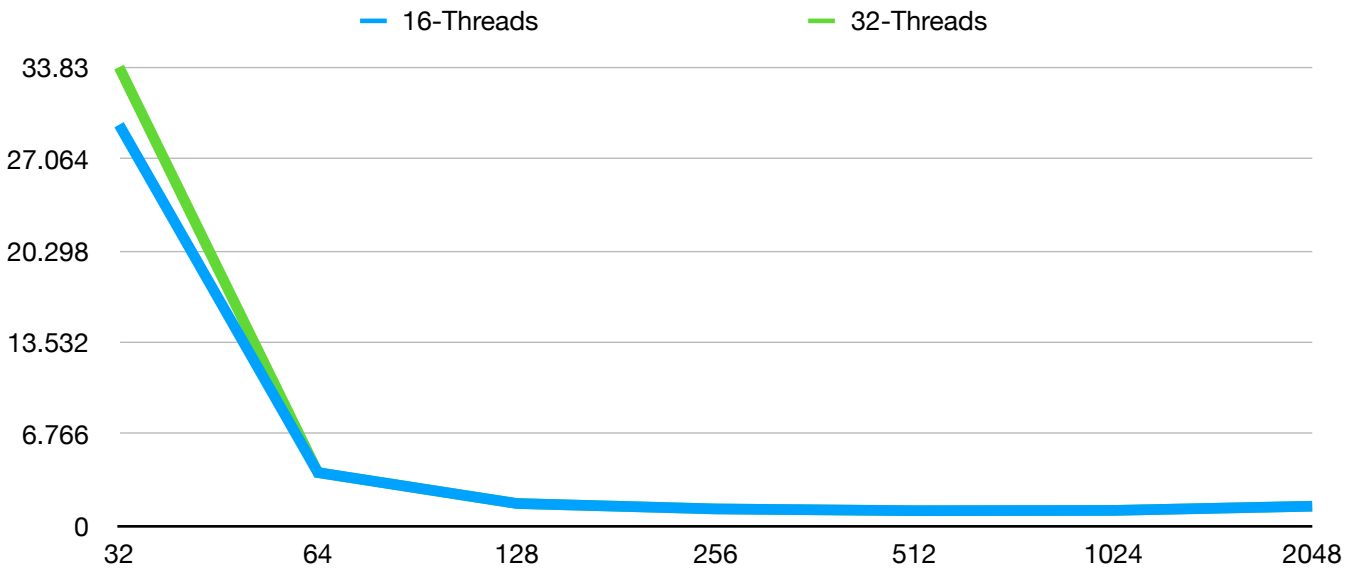
Granularities:

`OMP_NUM_THREADS=32`

Tile size (t)	32	64	128	256	512	1024	2048
Time	5.72	1.64	1.34	1.24	1.23	1.31	1.65

`OMP_NUM_THREADS=16`

Tile size (t)	32	64	128	256	512	1024	2048
Time	16.49	2.04	1.61	1.35	1.27	1.29	1.71



Taskloop approach is working slower that explicit task. The reason behind it is that Taskloop has a little bit more overhead than explicit tasking. Although the overhead is larger, it is more optimised to handle large number of tasks in the task pool as it is visible when we use smaller tile size (more tasks).

512 was chosen as the optimal block size (granularity) since other block sizes show less speed according to the plot and data tables above.

Using 32 threads we are getting **11.8x speedup** compared to the serial approach.