**Longest Common Subsequence :**

**Methodology:** Longest Common subsequence has been implement dynamic programing fashion,but in order attain the parallelism we try to fill the cost matrix by traversing it diagonally. If we consider the I and j as row and column index respectively, all the cells(i,j) i+j=t, are filled simultaneously in the increasing order of t. Below table shows the example of implementation.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 3 | 4 | 5 | 6 | 7 |
| 3 | 4 | 5 | 6 | 7 | 8 |
| 4 | 5 | 6 | 7 | 8 | 9 |

Here we tried to divide the larger examples into smaller chunks of different block size, so that number amount of work per thread is greater than thread creation bottleneck. But it seems that cilk\_for handles it implicitly manages the thread creation.

**Two methods were tried,**

1. Creating two cilk for loops for row and column and which is wrapped by a for loop that iterates for each diagonal. Within innermost for loop we check whether sum of row and column is equal to diagonal. Following approach involves unnecessary checks which creates lot of burden on each thread, leading to higher execution time.

2.Creating a loop structure in such a way that first for loop will handle diagonals from 1 to row size and second for loop to handle from row size+1 to total diagonal elements. Even though the execution time is very good but the parallelism and speedup for this approach is not so great.

In order to attain better cache hits, algorithm tries to consider the largest sequence along the row.

**Cilk View Report :**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Approach | Duration (ms) | Overall Parallelism | Parrallelism in Parallel region | Speed Up in Parallel Region | | I | 2640 | 5421.41  88.32  (burdened) | 8870.17  88.88  (burdened) | 2 processors: 1.90 - 2.00  4 processors: 3.78 - 4.00  8 processors: 7.06 - 8.00  16 processors: 12.43 - 16.00  32 processors: 20.09 - 32.00  64 processors: 29.03 - 64.00  128 processors: 37.33 - 128.00  256 processors: 43.56 - 256.00 | | II | 20 | 10.94  0.3  (burdened) | 23.39  0.29  (burdened) | 2 processors: 0.29 - 2.00  4 processors: 0.22 - 4.00  8 processors: 0.19 - 8.00  16 processors: 0.18 - 16.00  32 processors: 0.18 - 23.39  64 processors: 0.17 - 23.39  128 processors: 0.17 - 23.39  256 processors: 0.17 - 23.39 | |

**Cilk Screen Report :**

Cilk screen has been run on smaller instance of data, since it takes a longer time to find the longest common subsequence

|  |
| --- |
| sudhar@ubuntu:~/assignments/ConcurrentProgramming/Assignment1$ cilkscreen ./lcs  Cilkscreen Race Detector V2.0.0, Build 3566  16 asdasdahsdkasdaj  No errors found by Cilkscreen |