CMSC 441 LCS Project 2 Report

By,

Joshua Standiford

Chris Sidell

Project Overview:

For this project our task was to design and implement a serial and parallel algorithm of the longest common subsequence (LCS). The basis of our serial design was out of the *Introduction to Algorithms 3rd. edition* textbook. After estimating the runtime of the serial version, we implemented a parallel version of the LCS algorithm. We tested the parallel version against multiple CPUs with various inputs. Afterwards we implemented a memory efficient parallel version of the parallel LCS algorithm. Reducing the memory usage from (m * n) to (m + n). Below is an analysis and inspection into the implementation of our groups LCS algorithms.

Design:

The design of the serial algorithm is straight from the book and straight forward design. Pseudocode is found at the bottom of the report. The parallel algorithm due to the nature cannot be translates 1-for-1 from the serial implementation. So, to parallelize the computation we must serially iterate diagonally from top left to bottom right and compute the diagonal between n and m. It must be done serially as you need the boxes to the left and top to be able to reliably compute the current position.

The memory efficient implementation does the same except of keeping the whole array it keeps the previous diagonal and computes the current diagonal using at max the length of a diagonal n to m.

Calculations:

Work Calculation:

We define work law to be the summation of all runtimes of a serial algorithm.

$$T_1(n) = \theta(1) + \theta(m * n) + \cdots + \theta(1)$$

Since $\theta(m * n)$ dominates the runtime in the summation above we can ignore all the constant work that's done with an m and n sufficiently large.

$$T_1(n) = \theta(m * n)$$

If we bound the parameters m and n such that $m = n$ then

$$m = n \; is \; worst \; case \; therefore \; if \; m = n, then \; m * n = n^2$$

$$T_1(n) = \theta(n^2)$$

Span Calculation:

The span calculation is based off

$$T_\infty(n) = \theta(lg(n)) + max_{1 \le i \le n}\ iter_\infty(i)$$

When analyzing the parallel LCS code, we can see that the outer most for loop will take

$$\theta(n + m - 1)$$

Within the algorithm there is constant time work we will denote as:

$$\theta(1)$$

Replacing the span calculations within the span law we get the equation

$$T_\infty(n) = \theta(lg(n)) + \theta(lg(m * n)) + \theta(n + m - 1) + \theta(1) + \cdots + \theta(1)$$

For the equation above. Given an n and m sufficiently large, we can reduce ignore the constant time work and can remove any constant time addition / subtraction :

$$\theta(n + m)$$

The theta above clearly dominates the runtime of the algorithm. Since any runtime greater than logarithmic runtimes will always take longer to run, then we can remove all logarithmic runtimes from the equation.

There's only one non-logarithmic and non-constant runtime left. However defined above we bound the parameters m and n such that $m = n$ therefore

$$\theta(n + m) = \theta(n + n) = \theta(2n)$$

We can reduce the runtime above by the definition of the theta runtime, where

$$\theta(2n) = \theta(n)$$

This is because the asymptotic complexity of a linear runtime is such that

$$\theta(n) = \theta(c * n)\ where\ c > 0$$

Therefore the span of the parallel LCS algorithm is

$$T_\infty(n) = \theta(n)$$

Parallelism Calculation

The parallelism calculation is based off

$$Parallelism = \frac{T_1}{T_\infty}$$

Computing the *parallelism* is quite easy, we simply plug in the runtimes for work over span.

$$parallelism = \frac{\theta(n^2)}{\theta(n)} = \theta(n)$$

Linear-speed-up Estimation

First we'll define the term's we're going to be using.

$$T_p = \text{running time on P processors.}$$

We are bounding the inputs such that

$$m = n$$

And if we define linear speed-up to be the ratio

$$\frac{T_1}{T_p} = \theta(P)$$

Thus the amount of work that can be maximized is at most **P**. This is defined as perfect linear speedup.

For our LCS program, we can expect to see linear speedups in the range of the following parameters.

$$m = n \text{ where } P \leq n \text{ and available } P \text{ is } \infty$$

We should expect to see that as long as the # of available processors are less than or equal to the input n, where **n** = **m** then we should see a linear-speedup. This is bounded by the # of available processors the computer has.

Predictions:

The expectation of the serial algorithm based off the work calculation was that it was going to take a fair amount of time for algorithm to run. Algorithms that have an (n^2) runtime get exponentially slower with more input. Therefore, we should notice a steep increase in runtime the larger m and n get.

For the parallel algorithm, we are going to test the algorithm on 1, 2, 4, 8, and 16 CPUs. My expectation for a single processor is that it's going to about the same as the serial implementation above. I expect that it might take a bit longer, since the parallel LCS algorithm is a different implementation than the serial version. Once the algorithm goes through 2 to 16 processors respectively, the expectation is to see increasingly faster runtimes the more processors we throw at it. The limitation of this however is that we can only have as many threads running on the algorithm as there are diagonal spaces allowed per iteration. So this algorithm will operate very quickly with very large inputs and with a high number of processors.

Empirical data on following pages.

Empirical Performance:

Serial Runtime Data

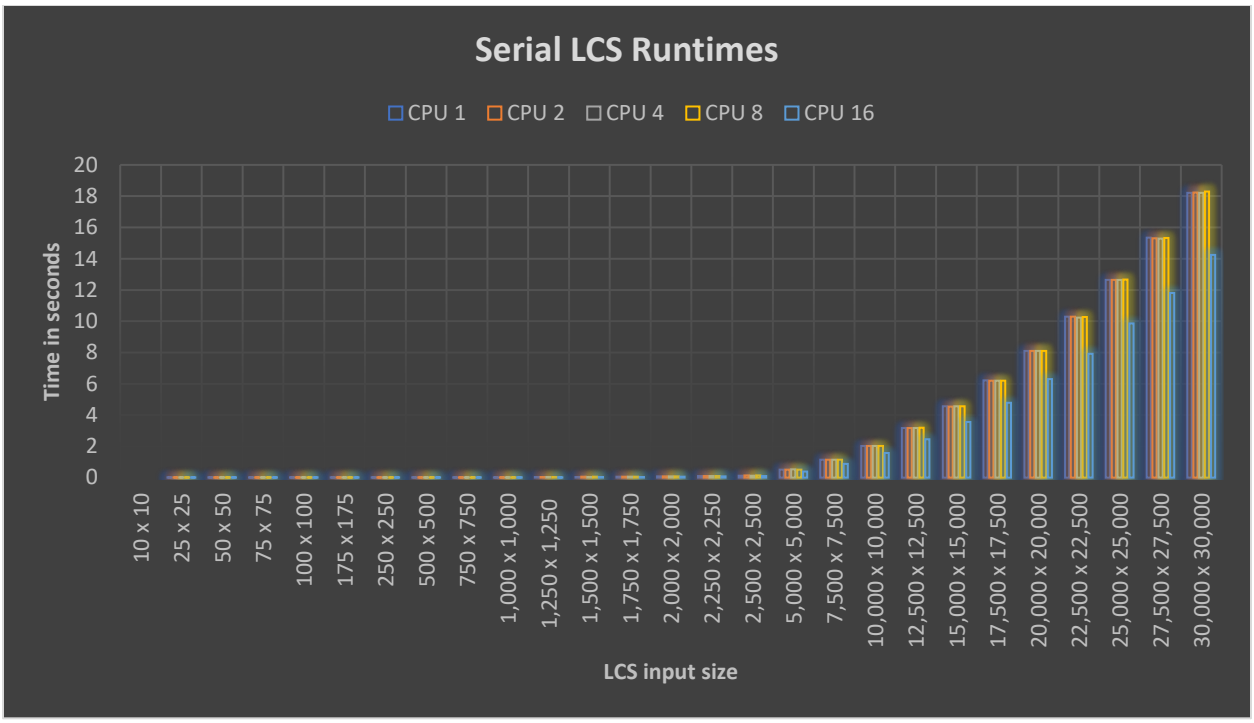
Serial	CPUs and Runtime in Seconds				
Input size	CPU 1	CPU 2	CPU 4	CPU 8	CPU 16
10 x 10	0.000005	0.000005	0.000005	0.000005	0.000002
25 x 25	0.000026	0.000025	0.000025	0.000026	0.000021
50 x 50	0.000102	0.000112	0.000102	0.000102	0.000055
75 x 75	0.00023	0.000228	0.000228	0.000123	0.000102
100 x 100	0.00053	0.000522	0.000411	0.000404	0.000182
175 x 175	0.001549	0.001407	0.001408	0.001393	0.000526
250 x 250	0.002951	0.003013	0.002959	0.002956	0.001081
500 x 500	0.010223	0.011362	0.010975	0.011535	0.004003
750 x 750	0.01773	0.017661	0.016355	0.019449	0.008854
1,000 x 1,000	0.02678	0.024278	0.025022	0.026871	0.015937
1,250 x 1,250	0.036291	0.038744	0.03653	0.040286	0.024758
1,500 x 1,500	0.050683	0.053696	0.050792	0.052726	0.035802
1,750 x 1,750	0.067075	0.069961	0.068546	0.069465	0.049345
2,000 x 2,000	0.088486	0.089588	0.085371	0.087549	0.064176
2,250 x 2,250	0.109203	0.108848	0.108779	0.107905	0.080113
2,500 x 2,500	0.131658	0.134514	0.131184	0.134661	0.09876
5,000 x 5,000	0.511447	0.514625	0.531007	0.515013	0.396672
7,500 x 7,500	1.14391	1.153935	1.147661	1.146833	0.883282
10,000 x 10,000	2.032315	2.026254	2.026127	2.030012	1.579718
12,500 x 12,500	3.170787	3.164234	3.170658	3.197318	2.447563
15,000 x 15,000	4.567589	4.549469	4.56392	4.57165	3.559267
17,500 x 17,500	6.217164	6.195609	6.196033	6.209626	4.788234
20,000 x 20,000	8.105185	8.100596	8.107197	8.106887	6.302145
22,500 x 22,500	10.284483	10.285022	10.238199	10.282908	7.929381
25,000 x 25,000	12.656203	12.652316	12.653995	12.672089	9.85427
27,500 x 27,500	15.351078	15.3079	15.293586	15.320314	11.803007
30,000 x 30,000	18.205795	18.242643	18.205094	18.293451	14.24954

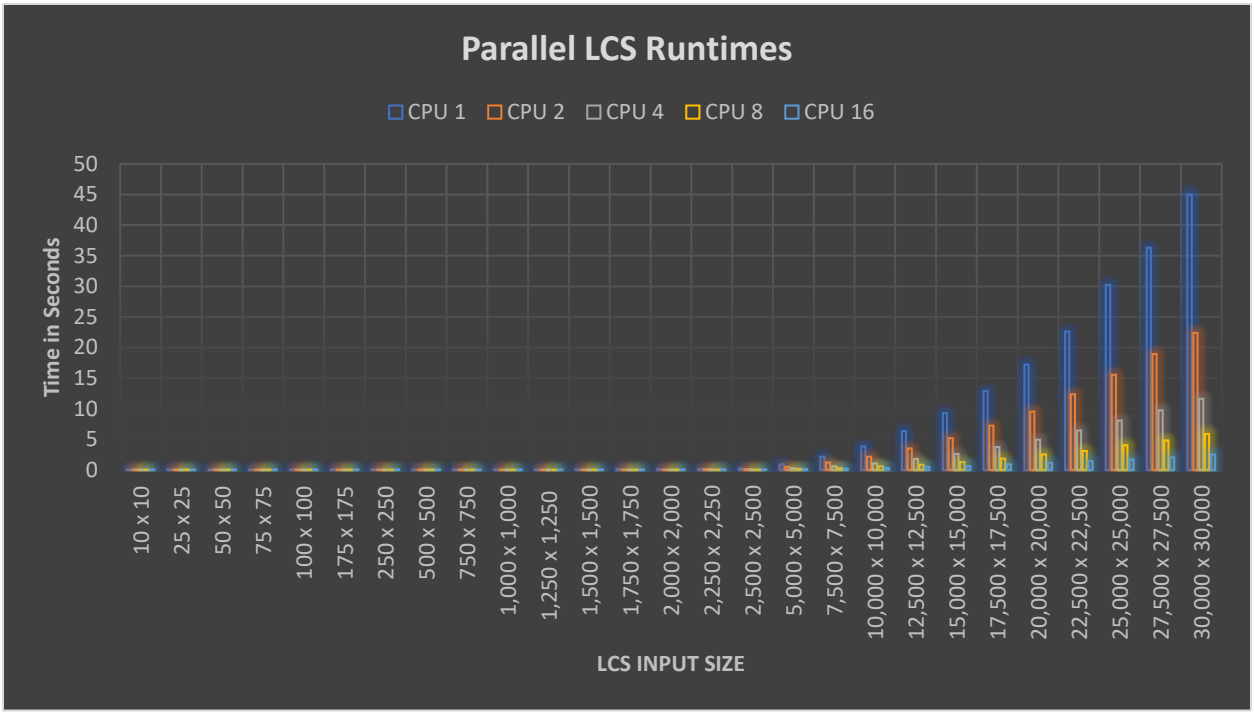
Parallel Runtime Data

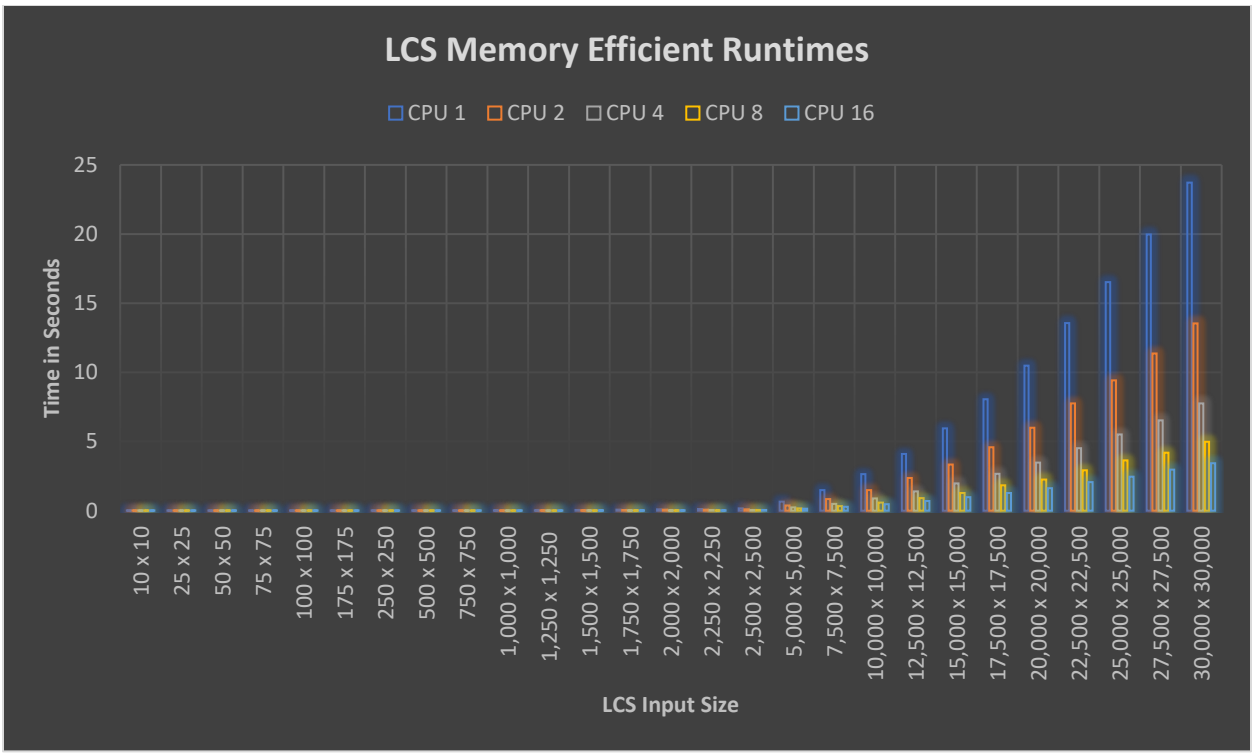
Parallel	CPUs and Runtime in Seconds				
Input size	CPU 1	CPU 2	CPU 4	CPU 8	CPU 16
10 x 10	0.000028	0.000179	0.000258	0.000371	0.000422
25 x 25	0.000072	0.000381	0.000403	0.000521	0.000601
50 x 50	0.000198	0.000597	0.000518	0.00066	0.000813
75 x 75	0.00049	0.000671	0.000721	0.000785	0.001159
100 x 100	0.000842	0.001018	0.001129	0.001184	0.001668
175 x 175	0.002209	0.002181	0.001812	0.001823	0.002652
250 x 250	0.002874	0.003387	0.002618	0.002876	0.00368
500 x 500	0.008859	0.009768	0.006685	0.005299	0.006911
750 x 750	0.016808	0.015785	0.011328	0.007322	0.010195
1,000 x 1,000	0.030712	0.022057	0.015641	0.010488	0.014563
1,250 x 1,250	0.048662	0.035343	0.021419	0.017548	0.018298
1,500 x 1,500	0.071463	0.047607	0.027002	0.019819	0.026714
1,750 x 1,750	0.098836	0.065561	0.037614	0.026792	0.030848
2,000 x 2,000	0.130177	0.085987	0.046703	0.032783	0.029263
2,250 x 2,250	0.16484	0.103802	0.057866	0.036123	0.033444
2,500 x 2,500	0.203756	0.125321	0.067985	0.043438	0.042173
5,000 x 5,000	0.917295	0.485617	0.280697	0.149459	0.110557
7,500 x 7,500	2.138821	1.193619	0.581892	0.321084	0.198746
10,000 x 10,000	3.841368	2.140341	1.073969	0.567078	0.328026
12,500 x 12,500	6.325828	3.522792	1.762813	0.867938	0.47751
15,000 x 15,000	9.264918	5.184949	2.64153	1.275007	0.662077
17,500 x 17,500	12.879671	7.209504	3.69968	1.835184	0.892528
20,000 x 20,000	17.189502	9.504888	4.931689	2.501318	1.183977
22,500 x 22,500	22.577094	12.350231	6.430517	3.115865	1.413076
25,000 x 25,000	30.258455	15.527944	8.070663	4.017151	1.694669
27,500 x 27,500	36.305059	18.957842	9.733501	4.811172	2.061413
30,000 x 30,000	44.994047	22.409602	11.607725	5.845212	2.496002

Memory Efficient Data

Mem Parallel	CPUs and Runtime in Seconds				
Input size	CPU 1	CPU 2	CPU 4	CPU 8	CPU 16
10 x 10	0.000018	0.000077	0.000069	0.000091	0.000154
25 x 25	0.000059	0.000162	0.00018	0.000217	0.000341
50 x 50	0.000184	0.000366	0.000343	0.000389	0.000635
75 x 75	0.000364	0.000603	0.000551	0.000631	0.000968
100 x 100	0.000612	0.000944	0.000836	0.001003	0.001556
175 x 175	0.001837	0.002078	0.001752	0.001799	0.002613
250 x 250	0.002046	0.003449	0.002992	0.0026	0.003827
500 x 500	0.007218	0.006301	0.005668	0.005551	0.007735
750 x 750	0.014777	0.0119	0.008419	0.008022	0.011291
1,000 x 1,000	0.027438	0.019638	0.013187	0.012232	0.016629
1,250 x 1,250	0.040597	0.028238	0.019153	0.016814	0.021389
1,500 x 1,500	0.058473	0.039851	0.026333	0.021698	0.031629
1,750 x 1,750	0.079248	0.053103	0.03425	0.026878	0.036945
2,000 x 2,000	0.103184	0.067226	0.041872	0.032567	0.036137
2,250 x 2,250	0.130983	0.083248	0.051671	0.039909	0.041855
2,500 x 2,500	0.162043	0.101497	0.062543	0.048527	0.055398
5,000 x 5,000	0.655244	0.389704	0.234007	0.162534	0.151584
7,500 x 7,500	1.480399	0.851017	0.500805	0.340316	0.290534
10,000 x 10,000	2.651649	1.498443	0.891415	0.585914	0.473663
12,500 x 12,500	4.103374	2.372456	1.391356	0.900822	0.696029
15,000 x 15,000	5.960758	3.348719	1.972799	1.276948	0.980189
17,500 x 17,500	8.056751	4.597962	2.670373	1.826223	1.287721
20,000 x 20,000	10.496159	6.000715	3.472724	2.242714	1.635238
22,500 x 22,500	13.567096	7.755403	4.516649	2.928004	2.066346
25,000 x 25,000	16.517959	9.413097	5.512188	3.636042	2.45959
27,500 x 27,500	19.97082	11.376806	6.535225	4.18839	2.964348
30,000 x 30,000	23.723062	13.539283	7.769509	4.975036	3.420953


Serial LCS Runtimes
CPU 1
CPU 2
CPU 4
CPU 8
CPU 16
Time in seconds
0
2
4
6
8
10
12
14
16
18
20
10 x 10
25 x 25
50 x 50
75 x 75
100 x 100
175 x 175
250 x 250
500 x 500
750 x 750
1,000 x 1,000
1,250 x 1,250
1,500 x 1,500
1,750 x 1,750
2,000 x 2,000
2,250 x 2,250
2,500 x 2,500
5,000 x 5,000
7,500 x 7,500
10,000 x 10,000
12,500 x 12,500
15,000 x 15,000
17,500 x 17,500
20,000 x 20,000
22,500 x 22,500
25,000 x 25,000
27,500 x 27,500
30,000 x 30,000
LCS input size


Parallel LCS Runtimes
CPU 1
CPU 2
CPU 4
CPU 8
CPU 16
Time in Seconds
0
5
10
15
20
25
30
35
40
45
50
10 x 10
25 x 25
50 x 50
75 x 75
100 x 100
175 x 175
250 x 250
500 x 500
750 x 750
1,000 x 1,000
1,250 x 1,250
1,500 x 1,500
1,750 x 1,750
2,000 x 2,000
2,250 x 2,250
2,500 x 2,500
5,000 x 5,000
7,500 x 7,500
10,000 x 10,000
12,500 x 12,500
15,000 x 15,000
17,500 x 17,500
20,000 x 20,000
22,500 x 22,500
25,000 x 25,000
27,500 x 27,500
30,000 x 30,000
LCS INPUT SIZE



Runtime Analysis:

From the data gathered running the serial and parallel algorithm up until 5000 are similar in runtimes. For larger sizes where big O takes precedence the graph clearly shows the serial implementation increasing much faster than the parallel implementation. The serial following a path much closer to n^2 where the parallel path is closer to n when inputs get larger.

This is further reinforced with LCS runtimes shown with a bar graph as you can see the single CPU implementation increases much more rapidly. These results fit closely to our predictions we made before we implemented the algorithm. We expected for the serial data to follow a steep trend as the input sizes increased. As you can see from the graphs above, the data did just that. The same thing can be said for the parallel version, we expected a more linear, leveled out trend. Again from the graphs above, we can visually see that the more processors added and for input sizes greater than 5,000 the trend follows close to a linear runtime.

Pseudocode:

Serial

```
LCS(X, Y, n, m)
    int L[n][m];
    for I to n:
        for j to m:
            if I or j = 0:
                L[i][j] = 0;
            Else if X[I – 1] = Y[j – 1]:
                L[i][j] = L[I - 1][j – 1] + 1
            Else:
                L[i][j] = max(L[I – 1][j], L[i][j – 1]
    Return L[n][m]
```

Parallel

```
PLCS(X, Y, n, m)
    int L[n][m]

    for(i = 0, i to n + m - 1)
        let col = MAX(0, i-n)
        let size = MIN(i, MIN(m-col, n))

        parallel for(j = 0, j to size)
            let l = MIN(n, i)
            let r = col + j
            let x = l - j - 1

            if X[x] == Y[r]
                if x == 0 or r == 0
                    L[x][r] = 1
                else
                    L[x][r] = L[x-1][r-1] + 1
            else
                if x == 0 and r == 0
                    L[x][r] = 0
                else if x == 0
                    L[x][r] = L[x][r-1]
```

```
                else if r == 0
                        L[x][r] = L[x-1][r]
                else
                        L[x][r] = MAX(L[x-1][r], L[x][r-1])
```