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.

Since dominates the runtime in the summation above we can ignore all the constant work that’s done with an and sufficiently large.

If we bound the parameters and such that then

*Span Calculation:*

The span calculation is based off

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

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

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

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

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 and such that therefore

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

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

Therefore the span of the parallel LCS algorithm is

*Parallelism Calculation*

The parallelism calculation is based off

*Parallelism*

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

=

*Linear-speed-up Estimation*

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

= running time on P processors.

We are bounding the inputs such that

*=*

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

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.

*= and available P is*

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 Performance:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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 |

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 where the parallel path is closer to 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.

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])