Project 2: Multithread LCS Final Report

December 9, 2017

Sabbir Ahmed

1 Description

The longest common subsequence (LCS) problem is to be implemented for the assignment. The objective of this project is to design, analyze, and implement a multithreaded version of the LCS length algorithm. A memoized or bottom-up approach. In either case, the goal is to design an algorithm that makes efficient use of multiple processors.

Analysis is required once the LCS length algorithm is designed. The work $T_1(m, n)$ and the span $T_\infty(m, n)$ are to be computed, where m and n are the lengths of the input sequences X and Y, respectively. Finally, the parallelism is to be computed along with an estimate of a range of parameters (m, n, and P) for which linear or near-linear speed-up may be expected.

The last part of the project is to implement the LCS length algorithm in C or C++ using OpenMP on UMBC's maya cluster and measure its performance empirically using 1, 2, 4, and 8 processors (optionally, it may be tested on 16 processors; most maya nodes have eight cores, but some have 16). Test data and LCS lengths of various sizes are to be generated to demonstrate the performance characteristics of the algorithm. The algorithm to recover the LCS must also be implemented, but this need not be multithreaded.

2 Initialization

The matrix is initialized to store the LCS before computation. Since the implementation utilizes a two-dimensional array allocated on the heap to represent the matrix, initialization was possible in $\Theta(m)$ time. Since Milestone 2, the initialization loop was parallelized to bring the runtime down to $\Theta(\lg(m))$. This method is faster than iterating through all the cells of the matrix to assign each to a placeholder. Algorithm 2.1 provides the implementation used to initialize the matrix.

Reading the sequences from the one-line input files, truncating them if necessary and storing them into buffers all take constant time operations.

Algorithm 2.1: Initialization of the Longest Common Subsequence Matrix

```
1  lcs_matrix = pointer array(m + 1);
2  parallel for i = 0 to m + 1
3       lcs_i = pointer array(n + 1);
```

3 Serial Algorithm

Milestone 1 required implementation of the serial LCS length algorithm and its analysis. Table 1 visualizes the serial implementation of computing the LCS length of sample strings using the algorithm described in equation (15.9) of *Introduction to Algorithms* (1).

Table 1: LCS matrix constructed when comparing GCGTCA and ACGAA

	∅	G	C	G	T	C	A
∅	0	0	0	0	0	0	0
A	0	0	0	0	0	0	1
C	0	0	1	1	1	1	1
G	0	1	1	2	2	2	2
A	0	1	1	2	2	2	3
A	0	1	1	2	2	2	3

In Milestone 2, it was observed that the value of the cells depended on the cells above, left or top-left of itself. The cells in the other directions have no effect. This property allowing for subproblems to be independently solvable builds the foundation for the algorithm for the parallel implementation to find the LCS length.

Algorithm 3.2 provides the pseudocode for computing the length of the LCS. The serial algorithm utilizes the bottom-up approach to compute the LCS.

Algorithm 3.1: Serial Longest Common Subsequence Length

```
1  function LCS-LENGTH(X, Y, m, n)
2      // allocate (m + 1) × (n + 1) LCS matrix
3      lcs = new matrix[1, 2, .., m + 1][1, 2, .., n + 1]
4      for i = 0 to m
5          for j = 0 to n
6              // if upper-leftmost cell, content is 0
7              if (i == 0 or j == 0)
8                  lcs_{i,j} = 0
9              // if equal, content is the left diagonal value incremented by 1
10             else if (X_{i-1} == Y_{j-1})
11                 lcs_{i,j} = lcs_{i-1,j-1} + 1
12             // maximum between previous row and previous column values
13             else
14                 lcs_{i,j} = max(lcs_{i-1,j}, lcs_{i,j-1})
15     return lcs_{m,n} // the last value of the matrix is the length
```

3.1 Time Complexity

The running time of the serial implementation provides the work, T_1, of the algorithm. Simple analysis of the algorithm provided in the snippet Algorithm 3.2 suggests a non-linear running time from the nested loop. The inner loop (line 5) iterates n times with several constant conditional checks, while the outer loop (line 4) iterates m times. The runtime is therefore:

$$T_1(m, n) = \Theta(mn)$$

The work for the algorithm, $\Theta(mn)$, is quadratic if $m = n$.

3.2 Printing the Longest Common Subsequence

After computing the length, the matrix may be used to print the LCS itself. Printing the subsequence is done in a serial implementation since the project does not emphasize its analysis. Since its implementation is not necessary for the scope of the project,

the LCS printing functionality will be used for debugging purposes only. Algorithm 3.3 provides the pseudocode used to print the LCS.

Algorithm 3.2: Serial Longest Common Subsequence Printing

```
1  function SERIAL-LCS-PRINT(X, Y, m, n, lcs)
2      lcsstr = new string
3      cursor = lcs_{m,n} // cursor of the matrix
4      i = m, j = n // init from the bottom-rightmost cell
5      while (i > 0 and j > 0)
6          // if current character in X[] and Y[] are same
7          if (X_{i-1} == Y_{j-1})
8              lcsstr_{cursor-1} = X_{i-1} // result gets current character
9              i--, j--, cursor-- // decrement i, j and cursor
10         // find the larger of two and go to that direction
11         else if (lcs_{i-1,j} > lcs_{i,j-1})
12             i--
13         else
14             j--
15     print lcsstr
```

4 Parallel Algorithm

Milestone 2 required implementation of the parallel LCS length algorithm. This version of the implementation uses the same LCS matrix to generate the same cell values. Table 2 revisits the matrix generated by the serial implementation and highlights the independent subproblems.

Table 2: LCS matrix constructed when comparing `GCGTCA` and `ACGAA` with the diagonal subproblems highlighted

	∅	G	C	G	T	C	A
∅	0	0	0	0	0	0	0
A	0	0	0	0	0	0	1
C	0	0	1	1	1	1	1
G	0	1	1	2	2	2	2
A	0	1	1	2	2	2	3
A	0	1	1	2	2	2	3
		(1)	(2)				

The parallel implementation must generate the same values in the cells using these subproblems to avoid race conditions between threads.

To iterate through the cells in these diagonal independent subproblems, the algorithm must consider the constraint that the lengths of the diagonal subproblems do not monotonically increase. The lengths increase until the diagonal label as (1) (highlighted in) and decreases after the diagonal labeled as (2) (highlighted in).

To avoid issues with bounds of the lengths of the diagonals, the parallel algorithm splits the matrix into two halves visualized in Table 3.

Table 3: LCS matrix constructed when comparing `GCGTCA` and `ACGAA` highlighting its two halves used for the parallel algorithm.

	∅	G	C	G	T	C	A
∅	0	0	0	0	0	0	0
A	0	0	0	0	0	0	1
C	0	0	1	1	1	1	1
G	0	1	1	2	2	2	2
A	0	1	1	2	2	2	3
A	0	1	1	2	2	2	3

Algorithm 4.4 provides the pseudocode for computing the length of the LCS.

Algorithm 4.1: Parallel Longest Common Subsequence Length

```
1  function P-LCS-LENGTH(X, Y, m, n)
2      // first half of the matrix
3      for i = 1 to n
4          parallel for j = 1 to i
5              if Y_{i-j} == X_{j-1}
6                  lcs_{i-j+1,j} = lcs_{i-j,j-1} + 1
7              else if lcs_{i-j,j} ≥ lcs_{i-j+1,j-1}
8                  lcs_{i-j+1,j} = lcs_{i-j,j}
9              else
10                 lcs_{i-j+1,j} = lcs_{i-j+1,j-1}
11     // second half of the matrix
12     diagonal_len = 0
13     for i = 2 to m
14         // if the diagonal length is not at its maximum
15         if diagonal_len < (m - n)
16             diagonal_len++  // increment the maximum diagonal length
17         parallel for j = i to (n + diagonal_len)
18             if Y_{n-j+i-1} == X_{j-1}
19                 lcs_{n-j+i,j} = lcs_{n-j+i-1,j-1} + 1
20             else if lcs_{n-j+i-1,j} ≥ lcs_{n-j+i,j-1}
21                 lcs_{n-j+i,j} = lcs_{n-j+i-1,j}
```

```
22              else
23                  lcs_{n-j+i,j} = lcs_{n-j+i,j-1}
24      return lcs_{m,n} // the last value of the matrix is the length
```

4.1 Time Complexity

The running time for the parallel implementation provides the span, T_∞ of the algorithm. Computing the runtime of the parallel implementation with an undefined number of processors will yield the span of the algorithm.

The two loops handling their corresponding half of the LCS matrix consist of symmetric operations with identical runtimes of varying sizes, m and n. Therefore, computing the runtime for one of these loops is sufficient.

The inner **parallel** loop spawns all its ITER$_\infty(j)$ concurrently with an unbounded number of processors. Since $\max_{1 \le j \le i}$ ITER(j) = $\Theta(i)$, (congruently $\max_{i \le j \le n+d}$ ITER(j) = $\Theta(i)$ for the second half of the matrix), the **parallel** loop is $\Theta(i)$.

The overhead for the loops come from the outer loops iterating for the total number of diagonals. The number of diagonals is congruent to the sum of the lengths of the sequences minus the $\varnothing$ character, since the implementation skips the first iterations to optimize space. The runtime is therefore:

$$\begin{aligned} T_\infty(m,n) &= \Theta(m+n-1) \\ &= \Theta(m+n) \end{aligned}$$

The span for the algorithm, $\Theta(m+n)$, is linear if $m = n$.

	Execution Time				
Sequence Lengths	**1 Core**	**2 Cores**	**4 Cores**	**8 Cores**	**16 Cores**
500×500	0.00132	0.00085	0.00074	0.00081	0.00619
500×1000	0.00173	0.00091	0.00093	0.00113	0.00749
1000×1000	0.00778	0.00295	0.00203	0.00191	0.00680
500×2500	0.00601	0.00351	0.00326	0.00340	0.00753
1000×2500	0.02358	0.00782	0.00536	0.00496	0.01099
2500×2500	0.06988	0.03521	0.01718	0.01211	0.01751
500×5000	0.01174	0.00798	0.00733	0.00757	0.01370
1000×5000	0.05913	0.01697	0.01156	0.01100	0.01902
2500×5000	0.16671	0.07894	0.03575	0.02002	0.03167
5000×5000	0.36594	0.17528	0.08599	0.04275	0.04227
500×10000	0.02344	0.01675	0.01582	0.01587	0.02706
1000×10000	0.14117	0.03535	0.02702	0.02553	0.03445
2500×10000	0.40571	0.18924	0.08861	0.03861	0.04907
5000×10000	0.84709	0.44313	0.19429	0.09228	0.06786
10000×10000	1.62424	0.84876	0.42658	0.21194	0.12452
500×25000	0.07587	0.04237	0.03851	0.03946	0.06207
1000×25000	0.38554	0.10670	0.06793	0.06461	0.08672
2500×25000	1.29024	0.55931	0.21134	0.09250	0.10915
5000×25000	2.56334	1.32716	0.57491	0.22862	0.14928
10000×25000	4.95778	2.61067	1.42275	0.59635	0.30502
25000×25000	12.8376	6.74628	3.61627	1.81664	0.86447
500×50000	0.33343	0.14526	0.09436	0.08132	0.11722
1000×50000	0.80969	0.39848	0.19656	0.14247	0.15441
2500×50000	2.96173	1.20282	0.53493	0.27511	0.23781
5000×50000	5.90636	3.07472	1.25649	0.57563	0.36795
10000×50000	11.7655	6.14264	3.22639	1.29430	0.66344
25000×50000	29.2438	15.3071	8.12103	4.10569	1.83671
50000×50000	59.41108	30.94504	16.52633	8.51268	4.18818
500×100000	0.75508	0.38384	0.23663	0.18413	0.22643
1000×100000	1.72948	0.85551	0.48736	0.30923	0.32154

2500×100000	6.11345	2.51991	1.16686	0.64060	0.50087
5000×100000	12.48135	6.48731	2.78458	1.23304	0.79432
10000×100000	24.24921	12.92104	6.85141	2.84215	1.41117
25000×100000	59.10952	30.98594	17.28448	8.86489	4.04930
50000×100000	118.8275	62.16034	33.55820	17.96266	9.00409
100000×100000	253.86608	135.25331	70.53762	37.20588	18.47668

5 Testing and Debugging

5.1 Milestone 1

The source code was initially developed in C++, but later translated to C. The Intel C++ Compiler, `icpc`, did not appear to properly compile the source code. The executable built without any warnings or errors, but the algorithm during its execution appeared to skip steps. The current implementation in C compiles with both `gcc` and the Intel C Compiler, `icc`.

References

(1) T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.