4005-735 Parallel Computing I
Project Report:
Longest Common Subsequence

Adam Risi
Rochester Institute of Technology
ajr7708@rit.edu

Richard Sarkis
Rochester Institute of Technology
res7008@rit.edu

February 14th, 2011

1 Abstract

The Longest Common Subsequence (LCS) problem is common to bioinformatics and a classic in computer science, generally considered NP-hard. The problem is commonly encountered in bioinformatics in the comparison of DNA which results in strings that are a combination of DNA base pairs representing an organism's DNA sequence. These sequences are typically very long in length resulting in a computationally intensive problem. By determining the longest common subsequence of DNA, biologists can determine the relative closeness of two organisms. The goal of this project was to research methods by which one could parallelize the LCS problem to speed up string comparison. Currently, the most common methods used by used in bioinformatics is BLAST and FASTA. Both algorithms emphasize speed of comparison at the expense of sensitivity of determining optimal alignments of sequences. Algorithms like Smith-Waterman provide exactness at the expense of speed, and in this problem area this project aims to research the current state of exact, parallelizable algorithms usable in DNA analysis.

2 Computational Problem

The Longest Common Subsequence (LCS) problem is common to bioinformatics and a classic in computer science. The problem is one in which given two sequences $X =< x_1, x_2, \cdots, x_m >$ and $Y =< y_1, y_2, \cdots, y_n >$ we attempt to find the maximum-length common subsequences of X and Y. A subsequence is not the same as a substring (subsequences do not need to be consecutive in the string). Given the string $X =< x_1, x_2, \cdots, x_m >$, string $Z =< z_1, z_2, \cdots, z_k >$ is then a subsequence of X if there is a strictly increasing sequence $< i_1, i_2, \cdots, i_k >$ of indices of X such that for all $j = 1, 2, \cdots, k$ we have $x_{i_j} = z_j$ [2].

The problem is commonly encountered in bioinformatics in the comparison of DNA which is represented by the four DNA bases adenine, guanine, cytosine, and thymine represented relatively by the letters A, C, G, T. This results in strings that are a combination of these letters representing an organism's DNA sequence. These sequences are typically very long in length resulting in a computationally intensive problem.

The problem is generally considered NP-hard problem with an arbitrary number of input sequences but with a constant number of inputs the solution time is polynomial through the use of dynamic programming [1].

Parallelizing the LCS problem results in great increases in speed in the decoding of the genetic-derived character strings being compared. By determining the longest common subsequence of DNA, biologists can determine the relative closeness of two organisms.

The goal of this project is to determine the speed-up of this problem when using parallelization techniques and compare these performance metrics with a sequential single-CPU experiment.

3 Paper Analysis: Liu et al

3.1 Problem Description

With it being one of the primary tasks in bioinformatics, finding the LCS of a biosequence efficiently is imperative. Currently, there are various paralleled LCS implementations that vary in space and time complexity. They all seek to produce exact, correct results that sequential algorithms like Smith-Waterman and Needleman-Wunsch have been producing since the early 1980s. This paper presents an algorithm called $FAST\_LCS$ that promises better performance over other parallel LCS algorithms.

3.2 Contributions

Their contribution is their $FAST\_LCS$ algorithm which works in
$max8 * (n + 1) + 8 * (m + 1), L$ space, and $O(|LCS(X, Y)|)$ time, where m and n are the lengths of strings X and Y respectively, L is the number of identical character pairs and $|LCS(X, Y)|$ is the length of the LCS of X, Y [6]. A primary data structure for the algorithm builds a successor table TX and TY for the strings X and Y respectively, where each entry in these tables lists the successor of character $CH(i)$ in the string.

$$TX(i,j) = \begin{cases} min\{k|k \in SX(i,j)\} & SX(i,j) \neq \phi \\ - & otherwise \end{cases}$$

[6]

Using the successor tables, a second table categorizing identical character pairs of X and Y is called $S(X)$ and $S(Y)$ respectively. A parallel computation, and the basic operation of the algorithm is producing the direct successors of each identical pair at each time step.

3.3 Investigative Use

The research into LCS algorithms, especially parallelizable ones is an ongoing area of active research; unfortunately it was clear that a unique solution, even a naive one was not approachable withing the context of this project. Instead, it was decided to choose the algorithm as described in [6] and implement it in Parallel Java.

4 Paper Analysis: Liu et al

4.1 Problem Description

4.2 Contributions

4.3 Investigative Use

5 Problem Solution

This project involves two sets of deliverables, a sequential version utilizing the Dynamic algorithm for LCS, and a parallel LCS algorithm developed by [6].

5.1 Sequential

We investigated the single-processor, single-threaded version of an LCS algorithm using the common Dynamic table generation algorithm. Refer to section 6 for benchmarks recorded for this sequential algorithm, compared against our parallelized algorithm. This code was written in Java using the Parallel Java libraries [3].

The dynamic table generation algorithm is a quadratic-time linear-space algorithm for the calculation of the length of the longest common subsequence. The dynamic programming LCS algorithm computes with $O(2 * min(n, m))$ space and time complexity of $O(mn)$ time, where X and Y are two input strings and $|x| = n$ and $|y| = m$.

Algorithm Description

The Dynamic programming algorithm operates by constructing a table, where each cell contains the length of the longest common subsequence up until the indices of that cell in the input sequences. As the value of a cell solely depends on the cells above, to the left, and diagonally from the cell, only two rows of the table need to be maintained at any given time. The value of the cell in the last row and column contains the length of the longest common subsequence.

```
DYNAMIC PROGRAMMING LCS-LENGTH(m, n, A, B)
 1  ▷ Initialization
 2  K[1][j] ← 0[j = 0 . . n]
 3  for i ← 1 to m
 4        do
 5           K[0][j] ← K[1][j](j = 0 . . n)
 6           for j ← 1 to n
 7                 do
 8                    if A[i] = B[j]
 9                       then
10                             K[1][j] ← K[0][j − 1] + 1
11                       else
12                             K[1][j] ← max(K[1][j − 1], K[0][j])
13  LL[j] ← K[1][j](j = 0 . . n)
```

5.2 Parallelized

The parallel algorithm uses a technique called *wavefront parallelism*. The shorter sequence is initially broken into k chunks, and each processor is made responsible for a series of columns. Each processor is tasked with computing the values for its column in blocks of h rows. As a processor finishes computing a block, it sends the data overlapping with its neighbor, the "passage band", to the next processor, who is then able to compute a block of values itself. Once the entire table is calculated, the answer remains in the final processor's computation block. Refer to 6 for the computation time, speedup, efficiency, and experimentally determined sequential fraction values for a number of test cases.

While this technique does provide a good amount of parallelism, it does not achieve load balancing when calculating the first k blocks and the last k blocks. This does not significantly effect the result; however, as the block size is so small that the unbalanced computation time is insignificant.

6 Performance Metrics

The following table provides reported metrics for various test conditions. The variables are the length of input strings A and B, resulting in increasing memory and time requirements. All test cases were run on the RIT `paranoia` cluster, which consists of 32 back end computers (named `thug01` through `thug32` each an UltraSPARC-IIe CPU, 650 MHz clock, 1 GB main memory. They are connected by a 100-Mbps switched Ethernet back end interconnection network and aggregate 21 GHz clock, 32 GB main memory [4]. Tables 1 through 9 list the trial runs of two data strings increasing in size by a power of two from 512 bytes up to 128 kilobytes. The sequential, 1 processor trial runs for the 128 kilobyte test case could not be done due to default settings on the `paranoia` CS cluster preventing jobs from running longer than 1 hour, instead these trial time values were extrapolated by applying a power function curve fitting as described in Appendix C in **??**.

A (bytes)	B (bytes)	K	T (msec)	Speedup	Eff	EDSF
512	512	seq	808	xxx	xxx	xxx
512	512	1	961	0.841	0.841	xxx
512	512	2	1275	0.634	0.317	1.653
512	512	4	1557	0.519	0.130	1.827
512	512	8	2014	0.401	0.050	2.252
512	512	16	2168	0.373	0.023	2.340

Table 1: Two 512-byte strings compared on 1,2,4,8,16 processors. Times are the average of five trial runs.

A (bytes)	B (bytes)	K	T (msec)	Speedup	Eff	EDSF
1024	1024	seq	956	xxx	xxx	xxx
1024	1024	1	854	1.119	1.119	xxx
1024	1024	2	1340	0.713	0.357	2.138
1024	1024	4	1616	0.592	0.148	2.190
1024	1024	8	1687	0.567	0.071	2.115
1024	1024	16	2660	0.359	0.022	3.256

Table 2: Two 1-kilobyte strings compared on 1,2,4,8,16 processors. Times are the average of five trial runs.

7 Conclusion

A (bytes)	B (bytes)	K	T (msec)	Speedup	Eff	EDSF
2048	2048	seq	1276	xxx	xxx	xxx
2048	2048	1	1204	1.060	1.060	xxx
2048	2048	2	1848	0.690	0.345	2.070
2048	2048	4	1834	0.696	0.174	1.698
2048	2048	8	2257	0.565	0.071	2.000
2048	2048	16	2553	0.500	0.031	2.195

Table 3: Two 2-kilobyte strings compared on 1,2,4,8,16 processors. Times are the average of five trial runs.

A (bytes)	B (bytes)	K	T (msec)	Speedup	Eff	EDSF
4096	4096	seq	3066	xxx	xxx	xxx
4096	4096	1	2567	1.194	1.194	xxx
4096	4096	2	2737	1.120	0.560	1.132
4096	4096	4	2720	1.127	0.282	1.079
4096	4096	8	2641	1.161	0.145	1.033
4096	4096	16	3206	0.956	0.060	1.266

Table 4: Two 4-kilobyte strings compared on 1,2,4,8,16 processors. Times are the average of five trial runs.

A (bytes)	B (bytes)	K	T (msec)	Speedup	Eff	EDSF
8192	8192	seq	10408	xxx	xxx	xxx
8192	8192	1	10365	1.004	1.004	xxx
8192	8192	2	6883	1.512	0.756	0.328
8192	8192	4	4783	2.176	0.544	0.282
8192	8192	8	4191	2.483	0.310	0.319
8192	8192	16	4227	2.462	0.154	0.368

Table 5: Two 8-kilobyte strings compared on 1,2,4,8,16 processors. Times are the average of five trial runs.

A (bytes)	B (bytes)	K	T (msec)	Speedup	Eff	EDSF
16384	16384	seq	49159	xxx	xxx	xxx
16384	16384	1	86032	0.571	0.571	xxx
16384	16384	2	27481	1.789	0.894	-0.361
16384	16384	4	13932	3.528	0.882	-0.117
16384	16384	8	8609	5.710	0.714	-0.028
16384	16384	16	8154	6.029	0.377	0.034

Table 6: Two 16-kilobyte strings compared on 1,2,4,8,16 processors. Times are the average of five trial runs.

A (bytes)	B (bytes)	K	T (msec)	Speedup	Eff	EDSF
32768	32768	seq	321322	xxx	xxx	xxx
32768	32768	1	398561	0.806	0.806	xxx
32768	32768	2	177135	1.814	0.907	-0.111
32768	32768	4	61540	5.221	1.305	-0.127
32768	32768	8	28503	11.273	1.409	-0.061
32768	32768	16	16645	19.304	1.207	-0.022

Table 7: Two 32-kilobyte strings compared on 1,2,4,8,16 processors. Times are the average of five trial runs.

A (bytes)	B (bytes)	K	T (msec)	Speedup	Eff	EDSF
65536	65536	seq	1451538	xxx	xxx	xxx
65536	65536	1	1604761	0.905	0.905	xxx
65536	65536	2	813422	1.784	0.892	0.014
65536	65536	4	372594	3.896	0.974	-0.024
65536	65536	8	117537	12.350	1.544	-0.059
65536	65536	16	56647	25.624	1.602	-0.029

Table 8: Two 64-kilobyte strings compared on 1,2,4,8,16 processors. Times are the average of five trial runs.

A (bytes)	B (bytes)	K	T (msec)	Speedup	Eff	EDSF
131072	131072	seq	5183648*	xxx	xxx	xxx
131072	131072	1	5925532*	0.875	0.875	xxx
131072	131072	2	3208530	1.616	0.808	0.083
131072	131072	4	1635557	3.169	0.792	0.035
131072	131072	8	721019	7.189	0.899	-0.004
131072	131072	16	219387	23.628	1.477	-0.027

Table 9: Two 128-kilobyte strings compared on 1,2,4,8,16 processors. Times are the average of five trial runs.

* This value was extrapolated from the previous data points due to constraints on the `paranoia` cluster.

Bibliography

[1] Longest common subsequence problem. *Wikipedia*. http://goo.gl/5bMjT.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[3] Alan Kaminsky. Parallel java library. *Rochester Institute of Technology*. http://www.cs.rit.edu/ ark/pj.shtml.

[4] Alan Kaminsky. Parallel java on the rit cs parallel computers. *Rochester Institute of Technology*. http://www.cs.rit.edu/ ark/runningpj.shtml.

[5] Jinxian Lin and Yiqing Lv;. Computing all longest common subsequences on mpi cluster. *Computational Intelligence and Natural Computing Proceedings (CINC), 2010 Second International Conference on*, 1:386 – 389, 2010.

[6] Wei Liu, Ling Chen, and Lingjun Zou. A parallel lcs algorithm for biosequences alignment. *InfoScale '07: Proceedings of the 2nd international conference on Scalable information systems*, Jun 2007.

[7] B Ben Mabrouk, H Hasni, and Z Mahjoub. Parallelization of the dynamic programming algorithm for solving the longest common subsequence problem. *Computer Systems and Applications (AICCSA), 2010 IEEE/ACS International Conference on*, pages 1 – 8, 2010.

[8] Nuraini Rashid, Rosni Abdullah, and Abdullah Talib. Parallel homologous search with hirschberg algorithm: a hybrid mpi-pthreads solution. *ICCOMP'07: Proceedings of the 11th WSEAS International Conference on Computers*, Jul 2007.