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 $x_{i_j} = z_j$ [3].

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 [2].

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 [8]. 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.

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

[8]

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 [8] 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 Hirschberg's LCS algorithm and a parallel LCS algorithm provided developed by [8].

5.1 Sequential

We investigated the single-processor, single-threaded version of an LCS algorithm using Hirschberg's algorithm[4]. 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 [5].

Hirschberg's algorithm, developed by Dan Hirschberg in 1975, utilizes a dynamic programming model for finding the least cost sequence alignment between two string inputs. It is a divide and conquer version of the Needleman-Wunsch algorithm, and is used in bioinformatics for finding exact, maximal global alignments of DNA base pair sequences.[1]. The algorithm is considered superior to heuristic methods like BLAST or FASTA, but results in a space complexity of $O(min(n, m))$ space and time complexity of $O(mn)$ time, where X and Y are two input strings and $|x| = n$ and $|y| = m$.

Algorithm Description

Hirschberg's algorithm is divided into three distinct parts, Algorithm A, B and C. Algorithm B is an optimization of Algorithm A, thus our sequential version of the algorithm relies only on Algorithms B and C directly.

We introduce string $C = c_1c_2 \cdots c_p$ that is a subsequence of string A, where $A = a_1a_2 \cdots a_m$ if and only if there is a mapping $F1, 2, \cdots, p \rightarrow 1, 2, \cdots, m$ such that $f(i) = k$ when $c_i = a_k$ and F is a monotonically increasing function. Therefore, C is a common subsequence of input strings $A = a_1a_2 \cdots a_m$ and $B = b_1b_2 \cdots a_n$ if C is a subsequence of A and C is a subsequence of B[4].

Algorithm B produces an output vector LL, which is a derivation of Algorithm A in [4] such that the matrix L as defined in that algorithm is reduced to requiring only rows i and $i - 1$, hence the use of a 2-row array K of j-elements wide.

The space complexity of Algorithm B as implemented in our Java version is $O(3n)$ deriving from the total allocated space of $3(n + 1)$ integers for the three local storage arrays. The input arrays are pre-allocated and passed by reference. The time complexity of Algorithm B is simply $O(mn)$ reflecting the nested for-loops executed mn times.

```
ALGORITHM-B(m, n, A, B, LL)
 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)
```

Algorithm C recursively produces a solution to the LCS sub-problem, where at its finish returns the full longest common subsequence string. In our Java implementation of Algorithm C, the space complexity is proportional to $O(m + n)$ space. The time complexity of this algorithm is $O(mn)$, factoring in linear invocations of Algorithm-B and -C inside this function.

```
ALGORITHM-C(m, n, A, B, C)
 1  if n = 0
 2     then
 3          ▷ If the problem is trivial, solve it
 4          C ← ε ▷ Empty string
 5  elseif m = 1
 6     then
 7          for j ← 0 to n
 8               do
 9                  C ← A[1]
10
11  else
12          ▷ Otherwise, divide and conquer
13          i ← ⌊m/2⌋
14          ▷ Evaluate
15          ALGORITHM-B(i, n, A_{1,i}, B_{1,n}, L1)
16          ALGORITHM-B(m − i, n, A_{n,i+1}, B_{n,1}, L2)
17          ▷ Find j
18          for j ← 0 to n
19               do
20                  M ← MAX(L1[j] + L2[n − j])
21                  k ← MIN(j) such that L1(j) + L2(n − j) = M
22
23          ▷ Solve simpler sub-problems
24          ALGORITHM-C(i, k, A_{1,i}, B_{1,k}, C1)
25          ALGORITHM-C(m − i, n − k, A_{i+1,m}, B_{k+1,n}, C2)
26          ▷ Resulting output
27          C ← STRCAT(C1, C2)
28
```

The combined use of Algorithms C and B to produce the longest common subsequence results in a runtime complexity of $O(mn)$ and space complexity of $O(m + n)$.

5.2 Parallelized

Our parallelized version has...

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 backend 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 backend interconnection

network and aggregate 21 GHz clock, 32 GB main memory [6]. 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 trial run 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.

A (bytes)	B (bytes)	K	T (msec)	Speedup	Eff	EDSF
512	512	seq	849	xxx	xxx	xxx
512	512	1				xxx
512	512	2				
512	512	4				
512	512	8				
512	512	16				

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	1130	xxx	xxx	xxx
1024	1024	1				xxx
1024	1024	2				
1024	1024	4				
1024	1024	8				
1024	1024	16				

Table 2: Two 1-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
2048	2048	seq	2586	xxx	xxx	xxx
2048	2048	1				xxx
2048	2048	2				
2048	2048	4				
2048	2048	8				
2048	2048	16				

Table 3: Two 2-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
4096	4096	seq	6036	xxx	xxx	xxx
4096	4096	1				xxx
4096	4096	2				
4096	4096	4				
4096	4096	8				
4096	4096	16				

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	12242	xxx	xxx	xxx
8192	8192	1				xxx
8192	8192	2				
8192	8192	4				
8192	8192	8				
8192	8192	16				

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	55358	xxx	xxx	xxx
16384	16384	1				xxx
16384	16384	2				
16384	16384	4				
16384	16384	8				
16384	16384	16				

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	248778	xxx	xxx	xxx
32768	32768	1				xxx
32768	32768	2				
32768	32768	4				
32768	32768	8				
32768	32768	16				

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	803872	xxx	xxx	xxx
65536	65536	1				xxx
65536	65536	2				
65536	65536	4				
65536	65536	8				
65536	65536	16				

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	3980448*	xxx	xxx	xxx
131072	131072	1				xxx
131072	131072	2				
131072	131072	4				
131072	131072	8				
131072	131072	16				

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

Bibliography

[1] Hirschberg's algorithm. *Wikipedia*. http://goo.gl/RyxLH.

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

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

[4] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18:341–343, June 1975.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.