A COMPARATIVE STUDY OF VARIOUS PARALLEL LONGEST COMMON SUBSEQUENCE (LCS) ALGORITHMS

M.V.Ramakrishnan[#1], Prof.Mrs.Sumathy Eswaran[#2]

(1)M.Tech student in Computer Science and Engineering

(2) Professor in Computer Science and Engineering

Dr.MGR Educational and Research Institute, Chennai, TamilNadu, INDIA

***Abstract*— Sequence alignment is one of the most important tasks in bio-informatics or computational biology field. It helps identifying the similarity between the biological sequences. Longest Common Subsequence is the fundamental problem for sequence alignment techniques. Due to the emerging growth in bio-informatics applications, new biological sequences with longer length have been used for processing. Sequential algorithmic implementations take more time to find Longest Common Subsequence. Sequential implementations sometimes become intractable for longer biological sequences. To compute Longest Common Subsequence of longer biological sequences more efficiently and quickly, parallel algorithms are used. This paper presents a comparative study of three parallel LCS algorithms.**

***Keywords*— Dynamic Programming, EFP_LCS, FAST LCS, Parallel Algorithms, Parallel LCS.**

I. INTRODUCTION

Sequence alignment is the method of arranging the biological sequences like DNA, protein, etc. Sequence alignment can be carried out by variety of methods like pair wise alignment, multiple sequence alignment. The main purpose of sequence alignment is to identify functional and structural relationship among biological sequences.

The longest common subsequence (LCS) identification of biological sequences is an essential step in sequence alignment. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. This makes the subsequence different from the substring as a substring should appear in a contiguous relative order. A common subsequence is maximal or is a longer subsequence if it is of length maximal. For example, sequences {B,C,B,A} and {B,D,A,B} are the longest common subsequences of {A,B,C,B,D,A,B}and of {B,D,C,A,B,A}. The performance of LCS identification depends on three factors: Efficiency of algorithm, Dissimilarity between sequences and Length of sequences. Since sequence properties and databases increase in size drastically, the resource needed to compute the result also increases. It is very difficult to calculate LCS of longer sequences using sequential algorithmic implementations. For efficient computation of LCS, Parallel algorithms and heuristic algorithms are used.

II. RELATED WORK

The Needleman–Wunsch algorithm [1] was the first application of dynamic programming which provides a global alignment between two sequences. This algorithm leads to the evolution of various efficient LCS algorithms. It is only suitable if the two sequences are of similar length. The Smith-Waterman algorithm [2] evolved from Needleman- Wunsch algorithm provides a local alignment of biological sequences.

Various parallel algorithms like CREW PRAm model, Systolic arrays have been proposed in the earlier days to reduce the computation time. In the recent time Wan, Liu, Chen proposed Fast LCS algorithm [3] with time complexity |LCS(*X, Y*). Fast LCS's efficiency has been further improved by EFP_LCS [4] by 40% to 60%. A parallel LCS algorithm [5] based on dynamic programming has also been proposed.

III. ALGORITHMS UNDER STUDY

In this paper the performances of Parallel algorithms like FAST LCS, EFP_LCS and Parallel LCS have been tested on various datasets ranges from 50 to 900.

Fast LCS

Let X={x1,x2,…,xn} and Y={y1,y2,…,yn} be two biological sequences where xi,yi ∈{A,C,G,T}. An array CH of 4 characters is defined such that CH (1) =’A’, CH (2) =’C’, CH (3) =’G’, CH (4) =’T’. To compute LCS, successor table need to be built for two strings. The successor tables are denoted as TX and TY. The successor table entries are defined as follows:

$$T(i,j) = \begin{cases} \min\{k \mid k \in S(i,j)\} & \text{when } S(i,j) \neq \emptyset \\ - & \text{Otherwise} \end{cases}$$

S(i, j) = {k | x_k = CH(i), k>j} where i= 1,2,3,4 and j = 0,1,2,….,n. Let X = "T C A G A T" and Y = "A G T C G T A". Their successor tables TX and TY are shown in Table 1.

TABLE I. SUCCESSOR TABLES TX AND TY

TX:

I	Ch(i)	0 1 2 3 4 5 6
1	A	3 3 3 5 5 - -
2	C	2 2 - - - - -
3	G	4 4 4 4 - - -
4	T	1 6 6 6 6 -

TY:

I	Ch(i)	0 1 2 3 4 5 6 7
1	A	1 7 7 7 7 7 7 -
2	C	4 4 4 4 - - - -
3	G	5 5 5 5 5 - - -
4	T	3 3 3 6 6 - -

For sequences X and Y, if $x_i = y_j$ = CH (k), then (i, j) is an identical pair of CH (k). The set of all identical character pairs of X and Y is denoted as S(X, Y).The level of an identical pair is defined as follows:

$$\text{Level}(i,j) = \begin{cases} 1 & \text{if } (i,j) \text{ is an identical character Pair} \\ \max\{\text{level}(k,l) + 1 \mid (k,l) < (i,j)\} & \text{Otherwise} \end{cases}$$

Where (i, j) and (k, l) be two identical pairs of X and Y. In this case these are (3, 1), (2, 4), (4, 5), (1, 3). The Initial Identical Pairs (IIDPs) are eligible to get entry into level 1 of Pair Table. Remaining IDPs get their level entry according to their case. After finding all the initial identical pairs, the next step would be producing all the direct successors of them. For each identical pair, the operation of producing all its direct successors is as follows:

(i, j) -> {(TX (k, i), TY (k, j)) | k = 1,2,3,4, TX (k, i) ≠ ‘-‘and TY (k, j) ≠ ‘-‘}.

For example, the successors of the identical character pair (3, 1) in example can be obtained by combining the elements of the 3rd column of TX and 1st column of TY. They are (5, 7), (-, 4), (4, 5), (6, 3).Here (-. 4) don’t represent identical character pairs. Hence after discarding (-, 4), the successors are (5, 7), (4, 5), (6, 3). In successor generating process, pruning techniques can be introduced to eliminate identical pairs which cannot produce LCS. These pruning techniques reduce the time complexity and improve the efficiency of the algorithm.

In Pruning Operation 1, if two identical pairs (i, j) and (k, l) are on the same level satisfying (k, l) > (i, j), then (k, l) can be pruned. For example (4, 5) and (6, 3) in example 1 are successors of identical pair (3, 1). Since both of them are not on the same level and (4, 5) > (6, 3), (6, 3) can be pruned. In Pruning Operation 2, If two identical character pairs (i1, j) and (i2, j) are on the same level satisfying i1 < i2, then (i2, j) can be pruned. In Pruning Operation 3, if there are identical pairs (i1, j), (i2, j), (ir, j) and i1< i2< … <ir, then (i2, j), (ir, j) can be pruned. All these pruning operations are carried out without affecting the correctness of the algorithm. After performing the pruning operations, the tracing back of LCS starts from the identical pairs with the maximum level and ends when it reaches an initial identical pair, and the trail indicates the LCS. If more than one identical pairs with the maximum level are present in the table, the tracing process can be carried out in parallel to obtain LCS concurrently.

Efficient Fast Pruned LCS (EFP_LCS)

EFP_LCS algorithm takes Fast LCS as its base algorithm and increases its efficiency by extending it. EFP_LCS algorithm also starts with successor table generation, finding all the successors of identical pairs and performing three pruning operations to increase the efficiency of the algorithm. EFP_LCS extends the first pruning operation of Fast LCS by identifying non-redundant IIDPs which results in the reduction of computation time. In addition to the three pruning operations of Fast LCS, EFP_LCS adds two more pruning operations to make the algorithm more efficient. In Pruning Operation 4, while finding successor for a pair (i, j), if either the members of column i of the TX table or members of column j of TY table are all having no entry then the pair (i, j) is redundant in the next level. Hence the pair (i, j) can be pruned in this level. In Pruning Operation 5, LCS is identified by starting with first eligible IIDP character. This Process is repeated with next IIDP. The length of 2nd LCS is compared with the previous one. Discard the lesser length one and retain only the longer one. This process is continued until all the IIDPs are over. This pruning is useful in sequential implementation of the algorithm.

Parallel LCS based on Dynamic Programming

This algorithm is based on the filling of a score matrix through a scoring mechanism. The best score is the length of the LCS and the subsequence can be found by tracing back the table. Let m and n be the lengths of two sequences to be compared. The length of a maximal common subsequence in A = {a1, a2 ... an} and B = {b1, b2 ... bm} is determined as follows:

$$L(i,j) = \begin{cases} 0 \text{ i f } i = 0 \text{ or } j = 0 \\ L(i-1, j-1) + 1 \text{ if } a_i = b_j \\ \text{Max}(L(i, j-1), L(i-1, j)) \text{ else.} \end{cases}$$

The above scoring function is used to fill the scoring matrix row by row. The highest calculated score in the score matrix is the length of the LCS.

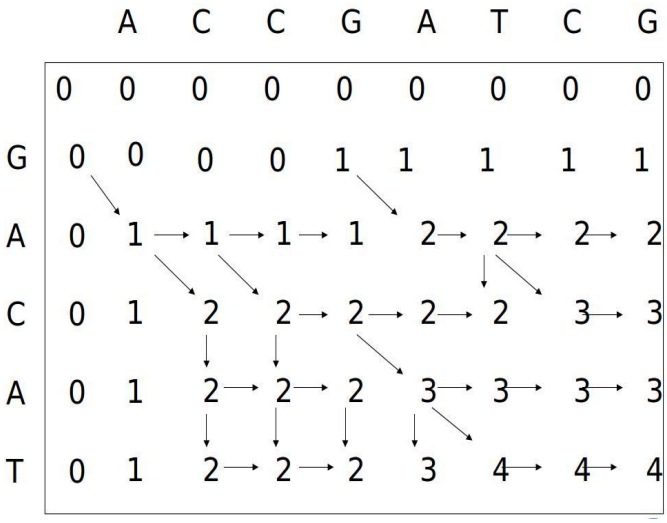

Fig 1. Example of scoring matrix

For example, in the Fig.1 the length is 4. In the scoring matrix, the LCS is traced back from the highest score point 4 to the score 1. In the scoring matrix L [i, j] totally depends on the data in the same row and the same column. Hence the value in the same column or same row cannot be computed in parallel. Hence first L1, 1 is computed then L1, 2 and L2, 1 in the same time, after that L1, 3, L2, 2 and L3, 1. This process is continued until the matrix is filled. To parallelize the dynamic programming algorithm, the score matrix is computed in the anti-diagonal direction as shown in Fig.2.

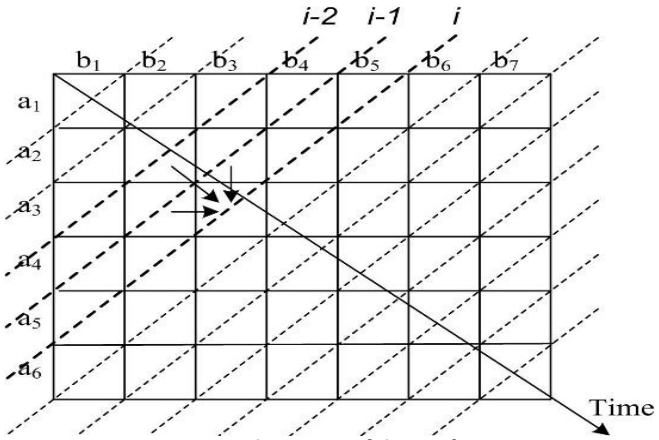

Fig 2. Anti-Diagonal Approach

IV. Experimental Results

All the three algorithms are tested on various datasets whose length ranges from 50 to 900 on two systems: 2 core and 4 core processors with 2GB memory and processor speed up to 2.20GHz. The data used in implementation has been taken from EBI [6] and EBI ftp server [7]. For LCS computation, length of sequence 1 is kept as constant (50) whereas length of sequence 2 varies from 50 to 900.

Table II and Table III summarizes the computation time for LCS of two sequences in 2-Core and 4-Core systems.

TABLE II. COMPARISON OF COMPUTATION TIME FOR LCS OF 2 SEQUENCES IN 2-CORE PROCESSOR

		Fast LCS	**EFP LCS (Sequential)**	**EFP LCS (Parallel)**	**Parallel LCS(Dynamic Programming)**
Xlen	**Y len**	**Elapsed time in secs**	**Elapsed time in secs**	**Elapsed time in secs**	**Elapsed time in secs**
XLen: 50	50	2.838	2.629	2.45	5.32
XLen: 50	100	2.894	2.791	2.56	5.68
XLen: 50	150	3.181	2.98	2.69	5.97
XLen: 50	200	3.275	3	3	6
XLen: 50	250	3.346	3.143	3.01	6
XLen: 50	300	3.453	3.288	3.12	6
XLen: 50	350	4.250	3.984	3.406	6
XLen: 50	400	4.360	4.231	4.065	7
XLen: 50	450	4.516	4.45	4.23	8
XLen: 50	500	4.985	4.828	4.484	8
XLen: 50	700	5.922	5.015	4.75	10
XLen: 50	900	6.078	5.437	4.859	13

TABLE III. COMPARISON OF COMPUTATION TIME FOR LCS OF 2 SEQUENCES IN 4-CORE PROCESSOR

On comparison between FAST LCS and EFP_LCS, the latter one proved to be more efficient in terms of memory efficiency as well as resource utilization. EFP_LCS speeds up the computation by over 40% to 60% when compared to FAST LCS. When comparing all the three algorithms, a different criterion had to be used. FAST LCS and EFP_LCS were capable of computing LCS of multiple sequences and more than one LCS between them whereas algorithm based on dynamic programming can compute LCS for only two sequences. Hence all three algorithms were tested for performance only based on two sequences.

		Fast LCS	EFP LCS (Sequential)	EFP LCS (Parallel)	Parallel LCS(Dynamic Programming)
Xlen	**Y len**	**Elapsed time in secs**	**Elapsed time in secs**	**Elapsed time in secs**	**Elapsed time in secs**
XLen: 50	50	1.875	1.703	1.628	5.062
XLen: 50	100	1.875	1.781	1.643	5.187
XLen: 50	150	1.875	1.781	1.644	5.281
XLen: 50	200	1.906	1.828	1.675	5.421
XLen: 50	250	1.937	1.875	1.698	5.437
XLen: 50	300	2.062	1.89	1.762	5.515
XLen: 50	350	2.125	1.922	1.794	5.625
XLen: 50	400	2.140	1.953	1.871	5.64
XLen: 50	450	2.141	2	1.972	5.687
XLen: 50	500	2.203	2	1.997	5.718
XLen: 50	700	2.328	2.047	2	5.8
XLen: 50	900	2.500	2.125	2.06	8.5

V. Conclusion

As a result, EFP_LCS proves to be the most efficient parallel algorithm which can compute LCS in a faster and more efficient way when compared to other two algorithms. Although LCS algorithm based on dynamic programming is parallel, it takes more time because it utilizes lot of resources. All the three algorithms can also be tested on longer sequences.

References

[1]Smith TF, Waterman MS: Identification of common molecular Subsequence, *Journal of Molecular Biology* 1990, 215:403-410.
[2]Needleman SB, Wunsch CD: A general method applicable to the search for similarities in the amino acid sequence of two Proteins, J *Mol Biol* 1970, 48(3):443-453.
[3]Yixi Chen, Andrew Wan and Wei Liu , A fast Parallel Algorithm for finding the Longest Common Subsequence of multiple biosequences , BMC Bioinformatics 2006, 7 (suppl 4): 54, ©2006 Chen et al; licensee BioMed Central Ltd.
[4]An Efficient Fast Pruned Parallel Algorithm for finding LCS in Biosequences, Anale Seria Informatica. Vol. VIII fasc. 1 – 2010.
[5]Parallel Computing the Longest Common Subsequence (LCS) on GPUs: Efficiency and Language Suitability, INFOCOMP 2011: The First International Conference on Advanced Communications and Computation.
[6]ftp://ftp.ebi.ac.uk/pub/databases/fastafiles/asd/
[7]ftp://ftp.ebi.ac.uk/pub/databases/pdb_seq/
[8] http://www.pdb.org
[9] www.ebi.ac.uk

AUTHORS PROFILE

(1)Author M.V.Ramakrishnan is an M.Tech student in Computer science & engineering, Dr.MGR University, Chennai, Tamil Nadu, India.

(2)Author, Mrs.Sumathy Eswaran, is a Professor in Computer Science & Engineering Department, Dr. MGR University, Chennai, Tamil Nadu, and India. She has published 2 papers in National Conference, 1 paper in International Conference and 7 papers in International Journals.