Proceedings of IOE Graduate Conference, 2016
pp. 329–336

A Map-Reduce Model to Find Longest Common Subsequence using Non-alignment Based Approach

Narayan Prasad Kandel [1], Shashidhar Ram Joshi [2]

Department of Electronic & Computer Engineering, Pulchowk Campus, Institute of Engineering, Tribhuvan University, Nepal
Corresponding Email: [1] npk.And@gmail.com

Abstract

Biological sequences Longest Common Subsequence (LCS) identification has significant applications in bioinformatics. Due to the emerging growth of bioinformatics applications, new biological sequences with longer length have been used for processing, making it a great challenge for sequential LCS algorithms. Few parallel LCS algorithms have been proposed but their efficiency and effectiveness are not satisfactory with increasing complexity and size of the biological data. An non-alignment based method of sequence comparison using single layer map reduce based scalable parallel algorithm is presented with some optimization for computing LCS between genetic sequences.

Keywords

Longest Common Subsequence – MapReduce – Hadoop

1. Introduction

Biological sequence comparison programs have revolutionized the practice of biochemistry, molecular and evolutionary biology. Pairwise comparison is the method of choice for many computational tools developed to analyze the deluge of genetic sequence data [1].

A fundamental operation in bioinformatics involves the comparison of genetic (DNA) sequences. The similarity between genetic sequences is a strong indicator of evolutionarily preserved characteristics. This property has been successfully used in determining pathologically important bacteria, viruses and fungi.

Among many sequences comparison tools, the common techniques include alignment based sequences comparison technique. It can be further divided into global alignment based technique and local alignment based technique. Global alignment (eg: Needleman-Wunsch [2]) based technique align entire section whereas local alignment based technique (eg: Smith - Waterman [3]) align the smaller sections. The choice of alignment method is depend on the type of analysis desired. However, both these methods are heavily dependent on the quality of sequence data and slight variance resulting from experimental or technical limitations can significantly affect the comparison results.

An alternative approach like alignment-free technique is becoming increasingly important in dealing with the exponential growth of genetic sequence data, classification and the grouping of organisms based on these sequences as such approaches matches the relative (as opposed to the exact) order of the base pairs in the sequence. Advancements in sequencing technology have provided a deluge of genetic data. The Genbank, a public repository of genetic sequence data, reported 194463572 sequence records in its 214th release on June 15, 2016. Analyzing such large datasets on uniprocessor machines is an extremely time-consuming process. It is imperative, therefore, to harness the power of high-performance computing to facilitate our understanding of this high throughput data.

2. LITERATURE REVIEW

A large number of research has been conducted in finding similarities between two gene species. The Needleman–Wunsch [2] algorithm was the first application of dynamic programming to find a global alignment between two sequences. This algorithm leads to the evolution of various efficient LCS algorithms. The major drawback of the Needleman-Wunsch algorithm is that it is only suitable for comparing two sequences with similar length. Later, Hirschberg algorithm [4], which is also know as divide and conquer version of Needleman-Wunsch Algorithm, was evolved from Needleman- Wunsch algorithm with some optimization. Hunt-Szymanski [5] propose an optimization to Hirschberg algorithm.

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 [6]. Fast LCS's efficiency has been further improved by Efficient Fast Pruned LCS EFP_LCS [7]. A parallel LCS algorithm [8] based on dynamic programming has also been proposed. Li, Wang & Bao [9] tried to solve LCS problem using automaton based technique in multi-level hadoop mapreduce. Beside that, Bohara & Joshi [10] implement multi-level alignment based hadoop mapreduce technique to solve the lcs problem.

2.1 Needleman-Wunsch algorithm

The Needleman–Wunsch algorithm performs a global alignment of two sequences. It is commonly used in bioinformatics to align protein or nucleotide sequences. The algorithm was published in 1970 by Saul B. Needleman and Christian D. Wunsch. The Needleman–Wunsch algorithm is an example of dynamic programming and was the first application of dynamic programming to biological sequence comparison. It is sometimes referred to as the Optimal matching algorithm. This global sequence alignment method explores all possible alignments and chooses the best one (the optimal global alignment). It does this by reading in a scoring matrix and a gap penalty (penalties) that contains values for every possible residue or nucleotide match and summing the matches taken from the scoring matrix.

2.2 Hirschberg algorithm

Hirschberg's algorithm [4] is an optimize version of Needleman-Wunsch algorithm that used dynamic programming algorithm to find the optimal sequence alignment between two strings. Optimality is measured with the Levenshtein distance, defined to be the sum of the costs of insertions, replacements, deletions, and null actions needed to change one string into the other. Hirschberg's algorithm is commonly used in computational biology to find maximal global alignments of DNA and protein sequences.

If x and y are strings, where $length(x) = n$ and $length(y) = m$, the Needleman-Wunsch algorithm finds an optimal alignment in $O(nm)$ time using $O(nm)$ space. Hirschberg's algorithm is a clever modification of the Needleman-Wunsch Algorithm which still takes $O(nm)$ time, but needs only $O(min\{n,m\})$ space.

2.3 Hunt-Szymanski algorithm

Hunt-Szymanski algorithm [5] present an improved version of the Hirschberg algorithm. It used row-wise processing technique where right to left traversal is done to find the lcs. It solves the problem of recovering an LCS in $O(|M|log(n))$ where $|M|$ denotes the number of all matches.

2.4 Fast LCS algorithm using Map Reduce

Li, Wang & Bao [9] purpose finite automaton based technique that implements fast lcs [6] algorithm to find the multiple longest common subsequences. The authors suggested that time required for calculating MLCS is reduced significantly using FACC Technique in compared to the time required using fast LCS [6], it uses multilevel map reduce technique and map reduce is used particularly to construct successor table of different string. Multilevel map reduce program based on fast lcs algorithm is presented by Bohara & Joshi in [10]. Though it is implemented via map reduce and time might be reduced if number of the cluster devices increase but much of the time is invested in creating, managing the map reduce jobs and waiting for the results of the previous job so it fails in utilizing the power of the distributed computing due to the multilevel map reduce strategy.

2.5 MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key [11].

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The runtime system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system. A typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day. MapReduce provides an abstraction that involves the programmer defining a "mapper" and a "reducer," with the following signatures:

Map: (value 1, key1) $\longrightarrow$ list (key2, value2)

Reduce: (key2, list (value2) $\longrightarrow$ list (value2).

3. METHODOLOGY

The longest common subsequence algorithm finds the longest subsequence between two strings. In contrast to the substring, the subsequence denotes a series of letters from the string which while being in order, need not be consecutive. For example, between ATCG and CTCAG, the longest common substring is TC, while the longest common subsequence is TCG.

LCS can help identify the key nucleotides across genetic sequences and is considerably less affected by the occasional sequencing error. This method is also useful for identifying potential regions of small mutations by analyzing the portions of the string not present in the LCS.

The basic block diagram of the program execution is shown in figure 1 and explained below.

1. Input string. It is converted to hadoop input by splitting it in multiple parts. The process of splitting given string to the multiple parts is known as preprocessing.
2. Splitted String. It is now feed to mapper to calculated the lcs of each small chunk.
3. Preliminary lcs. This is output of the mapper. Now it is feed to combiner which used parallel algorithm to find intermediate lcs.
4. Intermediate lcs. This is output of the combiner. In reducer, Multiple intermediate lcs are merged together using parallel algorithm recursively until final lcs is generated.
5. Final lcs. output of the reducer.

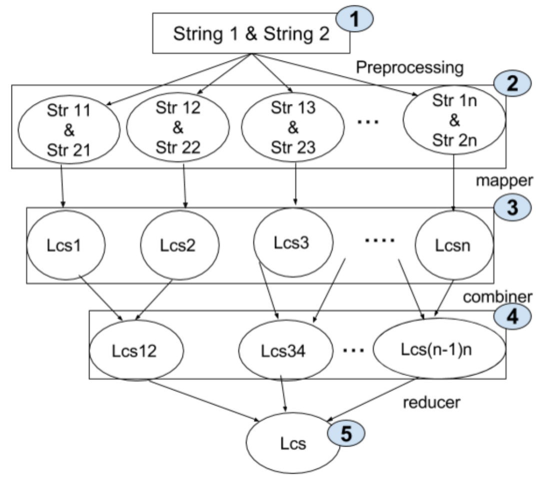

Figure 1: Program Execution Block Diagram

3.1 Computing LCS using Row-wise processing Technique

The row-wise processing technique [5] is inherited from the traditional approach for filling the dynamic programming table. However, this time, we concentrate only on those table entries which correspond to a match. Each dominant match defines a new corner to a contour line. To maintain the columns where all contour lines cross the current row, we use the array MinYPrefix[1..p], where MinYPrefix[l] gives the Y-index where the l'th contour line is located. As the name of the array suggests, the value of MinYPrefix[l] may be regarded as a cursor, which indicates the minimum length prefix of Y that is needed to produce a common subsequence of length l with the first i elements of X. Value p denotes r(X[1..i], Y[1..n]), that is, the number of contour lines crossing row i. Initially, the values of MinYPrefix are initialized to 'undefined'.

Table 1: Table for computing lcs

Row	MinYPrefix						
	0	1	2	3	4	5	6
0	0						
1	0	3					
2	0	2	5				
3	0	1	4				
4	0	1	4				
5	0	1	2	5			
6	0	1	2	5	8		

Given the example strings X=abcdbb and Y=cbacbaaba, the values of the array change as follows (undefined values are represented by n+1; the leftmost entry acts as sentinel and is set to zero):

To maintain the MinYPrefix values when moving from row to row, we need the following result.

Update rule: Let us assume that we are processing row i. For each open interval $MinYPrefix[l]..MinYPrefix[l+1]$, (l=0..r), find the matches (i,j) which fall into it (i.e. matches for which the j value is in the interval). The right boundary of the interval is kept unchanged, of no such match exists. Otherwise, it is updated to the smallest such j value (leftmost match in the interval). Note that the updates are simultaneous.

For example, when moving from row 2 to row 3 in the above example, we notice that $X[3] = Y[4]$ and $MinYPrefix[1] < 4 < MinYPrefix[2]$, so we update $MinYPrefix[2]$ to 4. The general scheme for advancing in the dynamic programming table is the following

```
begin
  for i:=1 to m
      do MinYPrefix[i] := n+1;
  MinYPrefix[0] := 0; r := 0;
  for i := 1 to m do
    /*Update array values for row i*/
    for j := 0 to r
      if range[MinYPrefix[j]+1..
         MinYPrefix[j+1]-1] contains
         matches then
        begin
          MinYPrefix[j+1] :=
            min{l|(i, l) is a match in
             this range};
          if j=r then r:=r+1;
        end
    return r;
end;
```

The Algorithm used to backtrack the LCS is as follow

```
begin
  last_char = True
  lcs_str1_index = []
  for i := lcs_length to 0
    for j := x_str_length to 0
      if i+1 > len(MinYPrefix[j-1]) or
        MinYPrefix[j][i] <
              MinYPrefix[j-1][i]
          if last_char
            last_char = False
          lcs_str1_index.append(j-1)
      elif not last_char &
          MinYPrefix[j][i]
              == MinYPrefix[j-1][i] &
          str1[j-1] = str2[MinYPrefix[j][i]
        lcs_str1_index.append(j-1)
  lcs_str = ''
  for s_index in lcs_str1_index
    lcs_str = str1[s_index] + lcs_str
  return lcs_str
end;
```

In this backtracking algorithm, different logic is implemented between last char of the LCS index and other indexes. This is done in order to extract the LCS that has minimum ending index among all possible ending indexes.

3.2 Parallel Implementation

A scalable parallel version of the LCS algorithm, proposed in [1], is outlined in figure 2. First, each string is divided across the processors and the LCS of the substrings in each processor (LCS1 and LCS2) are computed. Then the portions of strings (grey areas) that were beyond the first and last positions in the LCS are interchange and LCS for these previously unused strings is computed. Finally, the respective portions are combined to obtain the complete LCS.

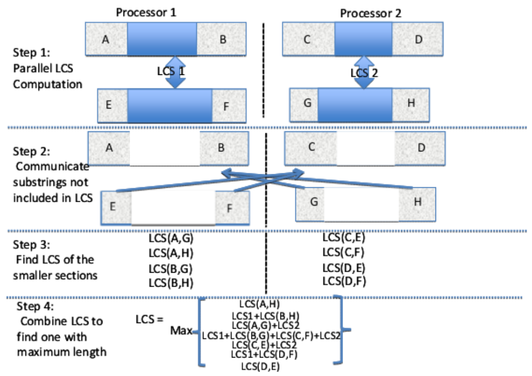

Figure 2: A Schematic Diagram of the Parallel LCS Algorithm

4. Result and Discussion

In implementing MapReduce strategy, the program is divided into 4 steps, which are: (1) Preliminary Step, (2) Mapper Step, (3) Combiner Step and (4) Reducer Step. The output of the one step is fed to the consequent next step as input and final step output is received as program output. We have used JSON format as an intermediate data format. The example of input data and output data format for each of the processes is as follows:

1. Preliminary Step

   Here, we have to calculate the LCS of strings str1 and str2. First, we take partition size as 15 and divide each string as a 15 char substring and keep the string sequence order. This work is done in Preliminary Step.

   Input
   str1=ABCBDABATCGACGATCGGGGTTCT
   TCACCACGGGGTTCTTCACCAGAGTTATCT
   str2=BDCABACTCAGGCACCGCAGTGACA
   AAAGTCGCAGTGACAAAAGTCAGGACGGC
   Partition size: 15

   Output
   1 ”a”: ”ABCBDABATCGACGA”, ”index”: 0, ”b”: ”BDCABACTCAGGCAC”
   2 ”a”: ”TCGGGGTTCTTCACC”, ”index”: 1, ”b”: ”CGCAGTGACAAAAGT”
   3 ”a”: ”ACGGGGTTCTTCACC”, ”index”: 2, ”b”: ”CGCAGTGACAAAAGT”
   4 ”a”: ”AGAGTTATCT”, ”b”: ”CAGGACGGC”, ”index”: 3

2. Mapper Step
   The preprocess string is fed to the mapper. Mapper Process is responsible for finding LCS of the small substring of str1 and str2. Along with LCS, it also calculates A, B, E, F and its index which is helpful to calculate combine LCS in an upcoming step.

   Input
   Mapper1: ’a’: ’ABCBDABATCGACGA’, ’index’: 0, ’b’: ’BDCABACTCAGGCAC’
   Mapper2: ’a’: ’TCGGGGTTCTTCACC’, ’index’: 1, ’b’: ’CGCAGTGACAAAAGT’
   Mapper3: ’a’: ’ACGGGGTTCTTCACC’, ’index’: 2, ’b’: ’CGCAGTGACAAAAGT’
   Mapper4: ’a’: ’AGAGTTATCT’, ’b’: ’CAGGACGGC’, ’index’: 3

3. Combiner Step
   Combiner step combines the output of the 2 mappers within the system to give intermediate output.

   Input
   Combiner1:
   Key: 0
   Value: [[0, ’A’: ’ABC’, ’a’: 0, ’B’: u”, ’E’: u”, ’lcs’: ’BDABATCAGA’, ’F’: ’C’, ’f’: 15, ’b’: 15, ’e’: 0, ’s2’: ’BDCABACTCAGGCAC’, ’s1’: ’ABCBDABATCGACGA’], [1, ’A’: ’T’, ’a’: 0, ’B’: u”, ’E’: u”, ’lcs’: ’CGGTAC’, ’F’: ’AAAAGT’, ’s2’: ’CGCAGTGACAAAAGT’, ’f’: 15, ’b’: 15, ’e’: 0, ’s1’: ’TCGGGGTTCTTCACC’]]

   Combiner2:
   Key: 1
   value: [[2, ’A’: ’A’, ’a’: 0, ’B’: u”, ’E’: u”, ’lcs’: ’CGGTAC’, ’F’: ’AAAAGT’, ’s2’: ’CGCAGTGACAAAAGT’, ’s1’: ’ACGGGGTTCTTCACC’, ’f’: 15, ’b’: 15, ’e’: 0], [3, ’A’: u”, ’a’: 0, ’B’: u”, ’E’: ’C’, ’lcs’: ’AGGAC’, ’F’: ’GGC’, ’s2’: ’CAGGACGGC’, ’s1’: ’AGAGTTATCT’, ’f’: 9, ’b’: 10, ’e’: 0]]

4. Reducer Process
   Finally, reducer step reduces the intermediate output from all combiner to one final lcs.

   Input:
   Key: lcs
   value: [[0, 'a': 0, 'A': u", 'b': 30, 'e': 0, 'lcs': 'BDABATCAGACCGGTAC', 'f': 30, 's2': 'BDCABACTCAGGCACCGCAGTGACAAAAGT', 's1': 'ABCBDABATCGACGATCGGGGTTCTTCACC', 'F': u", 'B': u", 'E': u"], [1, 'a': 0, 'A': u", 'b': 25, 'e': 0, 'lcs': 'CGGTACAGGAC', 'f': 24, 's2': 'CGCAGTGACAAAAGTCAGGACGGC', 's1': 'ACGGGGTTCTTCACCAGAGTTATCT', 'F': u", 'B': u", 'E': u"]]

   Output:
   "length": 28, "lcs": "BDABATCAGACCGGTACCGGTACAGGAC"

The parallel algorithm suggested by [1] highly depend on both starting index and ending index of LCS in both string. If we are able to extract LCS that lie in middle of both given string then the parallel algorithm would give us the sequential equivalent result in much more less time.

With modified LCS backtracking algorithm, for the repetition character, the index of the character that lies toward center is selected. The example cases are:

1. LCS('TCG', 'TCAGGGGGGGGGGGGGGGG') = 'TCG'
   index: str1 = [0,1,2], str2 = [0,1,3]
2. LCS('TTTTTTTTTTTTTTTTTCG', 'TCAG') = 'TCG'
   index: str1 = [15, 16, 17], str2 = [0,1,3]
3. LCS('TCGGGGGGGGGGGGGGGGG', 'TCAG') = 'TCG'
   index: str1 = [0,1, 2], str2 = [0, 1, 3]
4. LCS('TCG', 'TTTTTTTTTTCAG') = 'TCG'
   index: str1 = [0,1, 2], str2= [0, 12, 10]

Here, cases 1, 2, 3 work as expected but case 4 doesn't work as expected. This is because, when constructing matrix we only take least possible index of string_2 required to get LCS of length x when selecting n first character in string_1.

4.1 Complexity Analysis

The time complexity of core LCS computing algorithm is $O(|M|log(n))$ where $|M|$ denotes the number of all matches. The space complexity of the algorithm is $O(n^2)$.

4.2 Run Time of Algorithm

Performance is measured by running this algorithm for two input sequences. Table 2 shows the time taken to get result with a single node. Two input sequences are run for 10 times and the average time is computed. The time taken by this algorithm is listed below.

Time to compute LCS between 2 strings with length 17990 and 17990 with per process length of 1500, 500, 100 and 50 is shown in Table 2.

Table 2: Output time (in sec) comparison with different per process length of the string

S.N.	Per Process Length			
	1500	500	100	50
1	188	23	2.010	.940
2	190	22	1.951	.961
3	204	22	2.030	.919
4	197	22	2.024	.958
5	200	22	2.093	.961
6	191	23	2.012	.943
7	197	22	1.960	.961
8	195	22	1.981	.927
9	206	38	2.057	.909
10	204	24	2.021	.949
average	197	24	2.014	.943
lcs length	17984			
actual length	17984			

As shown in Table 2, time taken by algorithm is decrease with decreasing number of per process string length.

4.3 Hadoop Runtime Analysis

The scalability of the algorithm is studied by measuring the time to compute LCS between two string by running it on Amazon ElasticMapReduce platform with varying number of node which is shown on Table 3 and Table 4 respectively.

Table 3: LCS time (in sec.) comparison on different per process string length and processor number of 2 string with length 10,000

No. of Processor	Per Process String Length				
	200	500	1000	1500	2000
1	132.9	184	262.4	499.7	609
3	119.8	133.2	160.6	332.3	435
5	116.6	128.1	182.4	286.5	343.5
10	-	109.5	-	-	-
20	-	106.2	-	-	-
lcs length	6317	6416	6464	6466	6485
actual length	6514				

Table 4: LCS time (in sec.) comparison on different processor number of 2 string with length 200,000

No. of Processor	Per Process String Length
	500
10	609.8
20	587.3
lcs length	128,667
actual length	130,814

With reference to the time taken to computer LCS as presented in Table 3, it is fair to say that the algorithm is scalable. The accuracy of the algorithm goes on increasing with increase in per process length as with increase in per-process length there are less number of parallel merge.

Table 4 shows the time required to compute LCS of the strings with length 2,00,000. It is discovered that the time required to compute the LCS of strings having length 2,00,000 with 500 per process string length using 10 nodes is equal to the time taken to compute the LCS of strings having length 10,000 with 2000 per process string length using 1 node. This shows that, it is feasible to compute the LCS between 2 strings having very large length using group of low processing power computer. This will help to do cost effective analysis of the similarities between genetic sequences within no time.

The time required to find the LCS between 2 string is better than the MapReduce algorithm proposed by Bohara et. al [10]. Bohara et. al performed the study to find whether it is possible or not to find LCS using MapReduce approach. Bohara suggested that it is possible to calculate the LCS using MapReduce and time required will be reduced with the addition of the node. But Bohara didn't included the length of the input string so it is difficult to conclude how fast this algorithm is.

4.4 Verification

The output result is verified by comparing it with sequential program result. Total runtime is calculated by taking an average of 10 runtimes and it is later compared with corresponding sequential program runtime.

5. Conclusion

A basic model for MapReduce-based parallel algorithm for gene sequence comparison has been developed. Although there are few parallel algorithms for LCS computation, they are not reliable as the MapReduce-based solution in the context of fault tolerance and concurrency control. This MapReduce-based model handles all the different aspects of distributed computing from load balancing to synchronization automatically. The algorithm is highly scalable and cost effective. Hence, the large number of gene data can be processed at short period if we use a large number of nodes created from commodity computers. The time to calculate LCS can also reduce by decreasing the per process string length. But it might result in a decrease in LCS length, as there is trade-off between per process string length and accuracy.

5.1 Limitation

The parallel algorithm might not return longest common subsequence as multiple starting and ending point are possible for the same length of the LCS substring between two strings. This is because, with a different value of starting and ending sequence, we have different A, B, E, F. So, there is always a possibility of not having longest subsequence.

5.2 Further Enhancement

There is a special case where this algorithm don't work which is listed in limitation. So we can optimize algorithm to overcome the listed problem. This thesis can be extended to study relation between occurrences of multiple LCS across the same species to intra-species mutations. Also, we can enhance it to compare the runtime between different distributed platform like Apache Spark, Google dataflow and Hadoop cascading.

References

[1] S. Bhowmick, M. Shafiullah, H. Rai, and D. Bastola, "A Parallel Non-Alignment Based Approach to Efficient Sequence Comparison using Longest Common Subsequences," J. Phys.: Conf. Ser. Journal of Physics: Conference Series, vol. 256, p. 012012, Jan. 2010.

[2] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins,"Journal of Molecular Biology, vol. 48, no. 3, pp. 443–453, 1970.

[3] T. F. Smith and M. S. Waterman, "Comparison of biosequences," Advances in Applied Mathematics, vol. 2, no. 4, pp. 482–489, 1981.

[4] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Comm. Assoc. Comput. Mach., 18:6, 341–343, 1975.

[5] Hunt, James W. and Szymanski, Thomas G. May 1977. "A Fast Algorithm for Computing Longest Common Subsequences". Comm. ACM, vol.20 no.5; 350-353.

[6] Chen Y, Wan A and Liu W, "A fast Parallel Algorithm for finding the Longest Common Subsequence of multiple biosequences" , BMC Bioinformatics 7 (suppl 4), 2006

[7] Eswaran S and RajaGopalan SP, "An Efficient Fast Pruned Parallel Algorithm for finding LCS in Biosequences", Anale Seria Informatica. Vol. VIII fasc.1, 2010.

[8] Dhraief A, Issaoui R and Belghith A, "Parallel Computing the Longest Common Subsequence (LCS) on GPUs: Efficiency and Language Suitability", INFOCOMP 2011: The First International Conference on Advanced Communications and Computation, 2011

[9] Li, Yanni, Yuping Wang, and Liang Bao. "FACC: a novel finite automaton based on cloud computing for the multiple longest common subsequences search." Mathematical Problems in Engineering 2012 (2012).

[10] Bohara Jnaneshwar, Joshi Shashidhar Ram, "A MapReduce Based Parallel Algorithm for Finding Longest Common Subsequence in Biosequences", IOE Graduate Conference Journal, 2013

[11] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.