Realization Of Parallel Non-Alignment Based Approach To Find Longest Common Subsequence Using Hadoop MapReduce

Narayan Prasad Kandel
npk.and@gmail.com

***Abstract*—The Longest Common Subsequence (LCS) identification of biological sequences has significant applications in bioinformatics. Due to the emerging growth in 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. A parallel non-alignment based map reduce approach which help to solve the problem using distributed platform, is realized using Hadoop MapReduce.**

***Keywords*—Bioinformatics, Longest Common Subsequence, MapReduce, Hadoop**

I. 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 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 the many sequence comparison tools for mining genetic information, an extremely common technique includes the alignment-based methods. These involve aligning the entire (global alignment, Needleman-Wunsch [2]) or smaller sections (local alignment, Smith - Waterman [3]) of the genetic sequences. The choice of global or local alignment is based on the type of analysis desired. However, both these methods are heavily dependent on the quality of sequence data. Slight discrepancies resulting from experimental or technical limitations can significantly affect the comparison results. An alternative approach of sequence analysis is becoming increasingly important in dealing with the exponential growth of genetic sequence data, classification and the grouping of organisms based on these sequences. Such alternative approaches include the alignment-free methods, which match 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 188372017 sequence records in its 210th release in OCT 15, 2015. 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.

II. Literature Review

Large number of research has been conducted in finding similarities between two gene species. The NeedlemanWunsch [2] algorithm 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 Hirschberg [4] algorithm evolved from Needleman- Wunsch algorithm provides optimize version of Needleman-Wunsch. Hunt-Szymanski [5] propose an optimization to Hirschberg algorithm. Various parallel algorithms like Concurrent Read Exclusive Write (CREW) Parallel Random Access Machine(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 LCSs 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. A distributed algorithm using alignment based Hadoop MapReduce model [9] has been proposed to solve problem using distributed platform.

A. Needleman-Wunsch algorithm

The NeedlemanWunsch 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 NeedlemanWunsch 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.

B. Hirschberg algorithm

Hirschberg's algorithm [4] is a dynamic programming algorithm that finds 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 simply described as a divide and conquer version of the Needleman-Wunsch algorithm. Hirschberg's algorithm is commonly used in computational biology to find maximal global alignments of Deoxyribonucleic Acid(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 $O(minn, m)$ space.

C. Hunt-Szymanski algorithm

Hunt-Szymanski algorithm [5] present an improved version of the Hirschberg algorithm. It solve the problem of recovering an LCS in $O((r + n)logn)$ time and $O(r + n)$ space where r is the total number of ordered pairs of positions at which the two sequences match and n is length of the string.

D. 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.
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 programs 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 Googles 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) list (key2, value2)
Reduce: (key2, list (value2) list (value2).

III. 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.

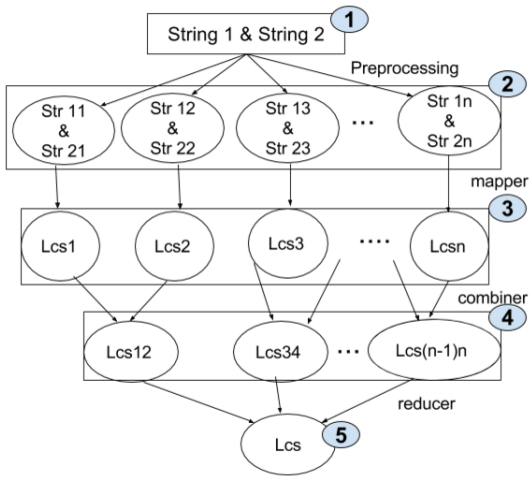

Fig. 1. Program Execution Block Diagram

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.

A. Computing LCS using Row-wise processing Technique

The row-wise processing 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 I. 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
(1)     for i:= 1 to m do MinYPrefix[i] := n+1;
(2)     MinYPrefix[0] := 0; r := 0;
(3)     for i := 1 to m do
.           /*Update the array values for row i. */
(4)         for j := 0 to r do
(5)             if range[MinYPrefix[j]+1..MinYPrefix[j+1] - 1]
.                   contains matches then
(6)              begin MinYPrefix[j+1]
.               := min { l or (i, l) is a match in this range};
(7)              if j = r then r := r+1;
.            end;
.       return r;
.    end;
```

B. 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.

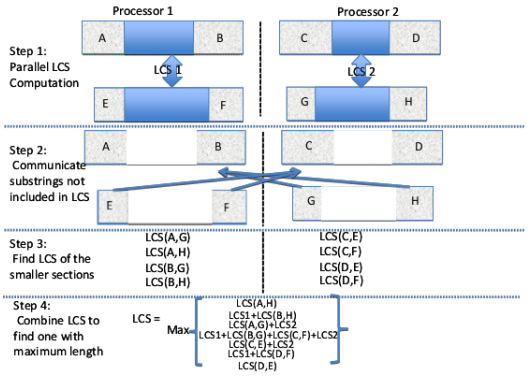

Fig. 2. A Schematic Diagram of the Parallel LCS Algorithm

IV. RESULT AND DISCUSSION

In implementing algorithm, the program is divided into 4 steps, that 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 used json format as intermediate data format. The example of input data and output data format for each of the processes is as follow.

1) Preliminary Step
Here, we have to calculated 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=ABCBDABATCGACGATCGGGGTTCTTCACCACG GGGTTCTTCACCAGAGTTATCT
str2=BDCABACTCAGGCACCGCAGTGACAAAAGTCG CAGTGACAAAAGTCAGGACGGC
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", "index": 3, "b": "CAGGACGGC"

2) Mapper Step
The preprocess string is fed to mapper. Mapper Process is responsible to find lcs of the small substring of str1 and str2. Along with lcs, it also calculate A, B, E, F and its index which is helpful to calculate combine lcs in 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', 'index': 3, 'b': 'CAGGACGGC'

3) Combiner Step
Combiner step combine 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', 's2': 'BDCABACTCAGGCAC', 's1': 'ABCBDABATCGACGA', 'f': 15, 'b': 15, 'e': 0], [1, 'A': 'T', 'a': 0, 'B': u", 'E': u", 'lcs': 'CGGTAC', 'F': 'AAAAGT', 's2': 'CGCAGTGACAAAAGT', 's1': 'TCGGGGTTCTTCACC', 'f': 15, 'b': 15, 'e': 0]]

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"

A. 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 $O(n^2)$.

B. Run Time of Algorithm

Performance is measured by running this algorithm for two input sequences. Below are the results for the time taken with single node. The configuration is Intel Core i5 CPU and 8GB of RAM. The linux version is ubuntu 14.04. Two input sequences are run for 10 times and average time is computed. The time taken by this algorithm is listed below.
String with length : 55 and 54 with lcs of length 28, per process length 15
1. 144.27113533 ms
2. 134.327173233 ms
3. 153.401851654 ms
4. 148.401021957 ms
5. 153.903961182 ms
6. 130.937099457 ms
7. 145.685195923 ms
8. 140.298128128 ms
9. 145.020008087 ms
10. 184.189081192 ms
average: 148.043465614

string with length 17990 and 17990 with lcs of length 17984 and per process length of

TABLE II. OUTPUT TIME COMPARISON WITH DIFFERENT PER PROCESS LENGTH OF THE STRING

S.N.	Per Process Length			
	1500	500	100	50
1	188051 ms	23204 ms	2010 ms	940 ms
2	190373 ms	22622 ms	1951 ms	961 ms
3	204989 ms	22590 ms	2030 ms	919 ms
4	197026 ms	22865 ms	2024 ms	958 ms
5	200978 ms	22701 ms	2093 ms	961 ms
6	191984 ms	23751 ms	2012 ms	943 ms
7	197794 ms	22773 ms	1960 ms	961 ms
8	195714 ms	22595 ms	1981 ms	927 ms
9	206259 ms	38038 ms	2057 ms	909 ms
10	204288 ms	24651 ms	2021 ms	949 ms
average	197746 ms	24579 ms	2014 ms	943 ms

It is seen that computation time reduced tremendously with smaller per process string length. Though time reduced tremendously with small number of per process string length, the output length of the LCS string also get reduced. We need to make trade-off between time and quality (length of the lcs).

C. Conclusion

A basic model for MapReduce based parallel algorithm for gene sequence comparison has been implemented. 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. Hence, the large number of gene data can be processed at short period if we use the large number of nodes created from commodity computers.

D. Limitation

The parallel algorithm might not give accurate result if multiple starting and ending point are possible for the same length of the lcs substring between two strings. This is because, with 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.

E. Further Enhancement

There is special case where this algorithm dont work which is listed in limitation. So we can optimize algorithm to overcome the listed problem. 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. 443453, 1970.

[3] T. F. Smith and M. S. Waterman, Comparison of biosequences, Advances in Applied Mathematics, vol. 2, no. 4, pp. 482489, 1981.

[4] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Comm. Assoc. Comput. Mach., 18:6, 341343, 1975.

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

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

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

[8] A. Dhraief, R. Issaoui and A. Belghith, "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] J. Bohara, S.R. Joshi, "A MapReduce Based Parallel Algorithm for Finding Longest Common Subsequence in Biosequences", IOE Graduate Conference Journal, 2013