Longest Common Subsequence using parallelization

Harshal More
Department of Computer Science
The University of Texas at San Antonio
Texas, USA
wbg516@my.utsa.edu

Gopal Vishwakarma
Department of Computer Engineering
The University of Texas at San Antonio
Texas, USA
gopalvishwakarma19@yahoo.com

***Abstract*—Finding a longest common subsequence is one of the crucial tasks in computer science and bioinformatics. In this paper, we present the computation of LCS using parallel computing using Message Passing Interface.**

Keywords—Longest Common Subsequence (LCS), dynamic programming, parallel computing, master node, slave, Message Passing Interface (MPI), memoization.

I. INTRODUCTION

To compare two DNA sequences, we need a mechanism to be able to determine how closely related the two organisms are. Sequence similarity searching is the most widely used, and most reliable, strategy for characterizing newly determined sequences. Sequence similarity searches can identify "homologous" proteins or genes by detecting statistically significant similarity that reflects common ancestry. We can achieve this by computing LCS.

For the given two strings, LCS is a longest subsequence common to all sequences in a set of sequences. Finding LCS using traditional algorithms such as recursive method and memoization could take large amount of time as these algorithms run in polynomial time. In this paper, we propose a parallel implementation of LCS problem using MPI.

II. LCS USING DYNAMIC PROGRAMMING APPROACH

A. Dynamic Programming approach

We will use dynamic programming approach with running time of O(mn) where m is the length of the first input string and n is the length of the second input string. There are two important steps in dynamic approach and they are as follows.

a) Computing dynamic matrix b) Traceback

We compute the dynamic LCS matrix using below theorem

$$LCS(X_i, Y_j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i-1}, Y_{j-1}) \frown x_i & \text{if } x_i = y_j \\ \text{longest}\,(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & \text{if } x_i \neq y_j \end{cases}$$

This dynamic programming theorem says, for the given two strings X = DGAFB and Y=DAB, arrange one string along the column and another string along the row like shown in Fig 1. Set first row and column to null values (i.e. zeroes) and start computing other remaining matrix values by applying theorem rules.

	Ø	D	G	A	F	B
Ø	0	0	0	0	0	0
D	0					
A	0					
B	0					

Fig 1. LCS dynamic matrix creation

Once the matrix computation is performed, the value we get in the last cell of the matrix represents the length of the one of the longest common sequences as shown below.

	Ø	D	G	A	F	B
Ø	0	0	0	0	0	0
D	0	1	1	1	1	1
A	0	1	1	2	2	2
B	0	1	1	2	2	**3**

Fig 2. Computing the length of LCS

The bold highlighted in Fig. 2 entry is what we have computed after applying the theorem, which is the length of the longest common sequence. In order to get the LCS, we need to trace the output back to where we started like shown below.

	Ø	D	G	A	F	B
Ø	0	0	0	0	0	0
D	0	1	1	1	1	1
A	0	1	1	2	2	2
B	0	1	1	2	2	**3**

Fig 3. LCS Traceback

As shown in Fig.3 the trace back procedure follows the arrows backwards, starting from the last cell in the table. The arrow

which is diagonal leaving the cell represents the letter to consider for LCS. In this case, DAB is the LCS with length=3.

B. Parallel LCS implementation

Our approach is to implement LCS using MPI based on master-slave notion where if we have two given strings X and Y, then the string X is broadcasted to all the slaves by master node and string Y is divided into blocks of characters and then scattered across all the slaves (i.e. nodes).

Each slave will then perform the computation of an intermediate dynamic matrix by accepting the result from previous slave and forward the computed matrix to the next slave. This process continues till the last node returns the final matrix to the master node.

This concept is useful to speed up the running time of the computation of LCS for any two given strings which uses load balancing and dynamic data allocation which are two important factors in cloud computing.

C. Algorithm using parallel computing

The number of slaves (nodes) is denoted by no_of_proc (number of processors) and for given two strings X and Y, we compute the LCS using the steps below:

1) The master broadcasts the string x to all slaves.
2) The master reads from the file the first p blocks of the string y. The size of each block (local_Y) is greater than or equal to len(y)/no_of_proc.
3) The master scatters the string Y across the slaves using MPI_scatter.
4) Each node creates its own intermediate matrix like shown in fig.2 using Dynamic Programming (DP) algorithm including(optional) master node.
5) As every node needs the local LCS value of another node, they wait to get the local LCS value from the previous nodes as well as they send the local LCS value to the next node.
6) Step 5 continues until one result from all the processes reach the node 0 which performs its own traceback to compute final output as shown in fig. 3.

Pseudo code for above Algorithm:

```
//distribute the data
define arrays to store blocks of string y
if(rank == 0):
        initialize arrays
        divide the string y into number_of_nodes
        MPI_Bcast(string x)
        create 2D matrices to store intermediate results
        MPI_Barrier(MPI.COMM_WORLD)
        //to ensure synchronization

        if(number_of_nodes > 1):
                MPI_send(send intermediate result to
                the next node)
                //trace the output
if (rank == 0):
        MPI_recv(result from other nodes)
elseif(rank == number_of_nodes-1):
        MPI_send(result to previous node)
else:
        MPI_recv(result from next node)
        MPI_send(result to previous node)
        trace_output(string x, part_of_y)
def trace_output():
        return sequence
```

a) Intermediate Matrix formation using DP

We will consider the same example illustrated in fig 1 and apply the above algorithm to compute LCS. We will assume that we have 3 processors namely P0, P1 and P2 and we will divide string Y into 3 parts. (D, A and B)

Process P0

	Ø	D	G	A	F	B
Ø	0	0	0	0	0	0
D	0	1	1	1	1	1

Fig 4. String X with first part of string Y

Process P1

	Ø	D	G	A	F	B
Ø	0	0	0	0	0	0
A	0	0	0	1	1	1

Fig 5. String X with second part of string Y

Process P2

	Ø	D	G	A	F	B
Ø	0	0	0	0	0	0
B	0	0	0	0	0	1

Fig 6. String X with third part of string Y

b) Process communication using send() and recv()

Process P2 will then send its result to process P1. Process P1 will perform the computation and forward the result to preceding process i.e. P0. Process P0 will perform its own traceback to return the LCS.

III. Running Time

After running this algorithm, it has been observed that the running time when implemented using MPI is tremendously less than that of running it normally without using MPI.

Strings with different sizes were tested and time taken to run the program for strings with up to 1500 characters was only in milliseconds vs running it without parallel implementation.

While running time of algorithm itself doesn't change (i.e. O(mn)), running it parallel using multiple nodes does the trick.

IV. Methodology and contribution

Our study contributes parallel implementation of LCS dynamic programming algorithm and below is the individual contribution of the group members.

Gopal Vishwakarma is responsible for implementing distribution of data from root node, writing pseudo code completing the PPT and report for the same part.

Harshal More is responsible for Implementing computation of LCS on every node and traceback, completing the PPT and report for the same part and pointing out limitations and improvements.

V. LIMITATION and improvements

a) Algorithm needs to be checked for larger string size

This algorithm needs to be checked for even more larger strings to be able to use in version control systems where file size could be in megabytes containing a large amount of data/text.

b) Not been able to run on multiple nodes

For some reason, this algorithm is unable to run on multiple nodes however, successfully running on a single node. This suggests that if we can make it run on multiple nodes, then the running time would be even more less.

c) Scope for optimization

Knowing that there could be multiple ways to write a parallel computing program, this algorithm could be optimized to get better results.

d) Can be implemented for more than 2 input strings

Algorithm can be improved to find LCS of more than 2 input strings.

VI. Conclusion

Finding longest common subsequence is very useful in many applications where you need to compare two large strings or collection of strings.

Implementing LCS using parallel computing makes it even more useful as the running time is exceptionally less for large input and hence can actually be used in real time applications.

VII. References

[1] https://www.researchgate.net/publication/234032723_A_Parallel_Implementation_for_Finding_the_Longest_Common_Subsequence

[2] Communication of Generic Python Objects https://mpi4py.readthedocs.io/en/stable/tutorial.html

[3] https://stackoverflow.com/questions/21088420/mpi4py-send-recv-with-tag

[4] ICS 161: Design and Analysis of Algorithms Lecture notes for February 29, 1996 https://www.ics.uci.edu/~eppstein/161/960229.html

[5] William R Pearson An introduction to Sequence Similarity (Homology) Searching https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3820096