Parallel Longest Common SubSequence Analysis In Chapel

Soroush Vahidi* , Baruch Schieber*, Zhihui Du+, David A. Bader+
*Department of Computer Science
+Department of Data Science
New Jersey Institute of Technology
Newark, NJ, USA
{sv96,sbar,zd4,bader}@njit.edu

Abstract—One of the most important problems in the field of string algorithms is the longest common subsequence problem(LCS). The problem is NP-HARD for an arbitrary number of strings but can be solved in polynomial time for a fixed number of strings. In this paper, we select a typical parallel LCS algorithm and integrate it into our large-scale string analysis algorithm library to support different types of large string analysis. Specifically, we take advantage of the high-level parallel language, Chapel, to integrate Lu and Liu's parallel LCS algorithm into Arkouda, an open-source framework. Through Arkouda, data scientists can handle large string analytics efficiently on the back-end high-performance computing resources from the front-end Python interface easily. The Chapel-enabled parallel LCS algorithm can identify the longest common subsequences of two strings, and some experimental results are also given to show its performance.

Index Terms—string algorithms - parallel computing - Chapel programming language

I. Introduction

The longest Common Subsequence(LCS) of a couple of strings is the longest string which is a subsequence of all of them. For example, the LCS of strings *abccb*,*abba* and *acbb* is *abb*. The finding of the LCS of some strings has applications, in particular, in the context of bioinformatics, where strings represent DNA or protein sequences [6].

Using a simple dynamic programming approach, one can find the LCS of two strings with lengths m and n in $\mathcal{O}(mn)$ time on one processor. For long strings, computing the LCS can take a long time, people have tried to find faster algorithms. One way to increase the speed of the algorithm is to use an approximation algorithm instead of an exact algorithm. In this way, some methods, such as [12], [1] and [2], have introduced approximation algorithms for the LCS problem and some of its variations.

Another way to solve the LCS problem with a higher speed is to develop parallel algorithms. In Lu and Lin's work [8], two algorithms are suggested for finding the LCS of two strings in parallel, such that one of them has a time complexity of $\mathcal{O}(\log^2(m) + \log(n))$ with $mn/\log(m)$ processors, and the other one has time complexity $\mathcal{O}(\log^2(m)\log\log(m))$ with $mn/(\log^2(m)\log\log(m))$ processors.

In this work, we have implemented the first algorithm with a little change and have measured its average running time for different test cases. To the best of our knowledge, it is the first parallel implementation of LCS in Chapel. The main contributions in this paper are as follows:

1) A typical longest common subsequence algorithm is implemented in Chapel to support high-performance string analysis.
2) Experimental results are given to show how the performance of the parallel algorithm will change with the size of two strings.
3) This work is based on an open-source framework Arkouda [9]. It means that data scientists can take advantage of the user-friendly Python language supported by Arkouda to conduct large-scale string analysis efficiently on the back-end high-performance computing resource with terabyte data or beyond.

II. Algorithm Description and Parallel Implementation

A. Basic Idea of the Algorithm

We implement Lu and Liu's parallel method [8] in Chapel [3]. The general idea of the algorithm is to reduce the LCS problem to the problem of finding the maximum weighted path between two special vertices in a grid graph. The algorithm is a recursive algorithm such that, in summary, for finding the maximum weighted path between vertices a and b, it finds vertex c such that the sum of the weight of the path between vertices a and c and the weight of the path between vertices c and b are maximized. To do it, it must find the maximum weighted path between a and each vertex c and between each vertex c and b. This is the reason why it is a recursive algorithm. Like many other recursive algorithms, to prevent exponential time complexity, we use dynamic programming to solve this recursive statement.

B. Recursive and Parallel Methods

1) Definitions and the recursive formula: If vertices a and b are in two consecutive rows of the grid graph, our program computes the maximum weighted path between them non-recursively using a parallel prefix min algorithm.

In the implementation, we define a couple of matrices; however, for most of them, we do not compute the matrix

Algorithm 1: Find ColMins Function

```
1  findColMins(dgu, dgl, vertex : int, left : int, right :
     int, top : int, bottom : int, mins, firstind : int)
2  var cols = right − left + 1
3  if (cols < 1) then
4  |  return
5  end
6  var midCol = ceil((right + left)/2) : int
7  var minIndex = findMinIndex(dgu, dgl, vertex :
     int, midCol, top, bottom)
8  mins[firstind + midCol − left] = minIndex
9  if (find_cell(dgu, dgl, vertex, minIndex, midCol) ≠
     infin) then
10 |  cobegin{
11 |      findColMins(dgu, dgl, vertex :
   |  int, left, midCol − 1, top, minIndex, mins, firstind)
   |
12 |      findColMins(dgu, dgl, vertex : int, midCol +
   |  1, right, minIndex, bottom, mins, firstind +
   |  midCol − left + 1)
13 |  }
14 |  else
15 |  |  findColMins(dgu, dgl, vertex :
   |  |  int, left, midCol −
   |  |  1, top, bottom, mins, firstind);
16 |  end
17 end
```

Algorithm 2: Find Min Index Function

```
1  findMinIndex(dgu, dgl, vertex : int, col : int, top :
     int, bottom : int)
2  var listsize = bottom − top + 1
3  var exp : int = 1
4  var expm1, expnot : int
5  var prefix : [0..listsize − 1]int
6  var minIndex : [0..listsize − 1]int
7  forall (i in 0..listsize-1) do
8  |  prefix[i] = find_cell(dgu, dgl, vertex, i + top, col)
9  |  minIndex[i] = i
10 end
11 while (exp < listsize) do
12 |  expm1 = exp − 1
13 |  expnot = exp
14 |  forall (j in 0..listsize-1) do
15 |  |  if (j&exp! = 0) then
16 |  |  |  if (prefix[j&expnot|expm1] <= prefix[j] )
   |  |  |  then
17 |  |  |  |  prefix[j] = prefix[j&expnot|expm1]
18 |  |  |  |  minIndex[j] =
   |  |  |  |  minIndex[j&expnot||expm1]
19 |  |  |  end
20 |  |  end
21 |  end
22 |  exp = exp << 1
23 end
24 return minIndex[listsize − 1] + top
```

explicitly, and when we need to know the amount of a specific cell, we only compute the amount of that cell.

The most important matrix is named D_G. Cell(i, j) of D_G shows the column of the leftmost vertex in the row of the grid graph G that can be reached by a path with weight j from the vertex in the ith column of the first row. If matrix D_G has more than two rows, we have the following formula.

$$D_G(i,j) = min(D_{G_U}(i,j), D_{G_L}(i,j), D_{G_L}(D_{G_U}(i,k), j-k)$$

for $1 \leq k \leq j$, where D_{G_U} is the upper half of D_G, and D_{G_L} is the lower half of D_G.

To understand this formula, note that $D_{G_U}(i,k)$ is the leftmost vertex in the down row of G_U such that we can reach it from the ith vertex of the first row of G_U with a path of weight k. Therefore, $D_{G_L}(D_{G_U}(i,k))(j-k)$ is the leftmost vertex in the down row of G_U we can reach from the ith vertex of the first row of G such that the weight of this path is j, and the sum of the weight of the edges of this path which are in G_U is k, and the sum of the weights of the edges of this path which are in G_L is $j-k$.

We define a vertex v in the bottom row of a grid graph G as the jth breakout vertex of the vertex $G(1,i)$ if v is the leftmost vertex in the bottom row, such that there is a path with cost j from vertex $G(1,i)$ to v. For example, in Fig. 1, vertices (9, 2), (9, 3), (9,4), (9, 5), and (9, 13) are the first, second, third, fourth, and fifth breakout vertices of the vertex (1,1). There is no 5^{th} breakout for vertex (1,8), or we can say the 5th breakout of vertex(1,8) is ∞.

A monotone matrix is a matrix in which if we have two consecutive columns c_1 and c_2 and c_1 on the left of c_2, then the cell that has the minimum cell of c_2 is not in a row higher than the row of the cell that has the minimum amount in c_1.

2) Functions: A crucial part of computing matrix D_G(which also can be called the cost matrix of G) is Alg. 1, which recursively finds the index of the minimum element in each column of a monotone matrix and saves it in array *mins*. It means that $mins[i]$ saves the index of the minimum element of the column i. The variables $left, right, top$ and *bottom* show the first and last columns and the first and last rows of the matrix, respectively.

We always give the variable $firstind$ an amount equal to the variable $left$. Cases in which $left \neq firstind$ occurs in recurrences, and the user does not become involved with them. In the pseudo codes, the commands $forall$ and *cobegin* show the situations that all the commands inside them will be done in parallel, and for does the commands inside its loop sequentially.

Alg. 1 shows the pseudo-code of $ColMin$. Inside Alg. 1, we have used Alg. 2, which finds the index of the minimum value in a column of a matrix, and it is a parallel algorithm. Every recursive relation has some initial values.

3) Computing D_{G_H}: In the recursive computation of D_G, we do not recursively compute the cost matrix of G

if G has two rows, and we obtain it differently. We consider the input strings a and b and define the cost matrix of the grid graph of G consisting of rows number h and $h+1$ of it by D_{G_h}.

In our implementation, we considered that D_{G_H} has only two rows. The first row of it shows the 0^{th} breakout for each vertex, and we can easily define the 0^{th} breakout of the i^{th} vertex of G_h as i. In [8], the 0^{th} breakout is not defined; however, we defined it because defining it could help us simplify the implementation. Therefore, from now, for simplicity of notation, we assume that D_{G_h} has only one row that shows the first breakout of each vertex in the upper row of the grid graph consisting of rows number h and $h+1$ of G.

We show the i^{th} letter of the string s by s_i and $i \geq 1$. For computing D_{G_H}, we need to calculate $j_1, j_2, j_3, \ldots, j_r$ such that $j_1, < j_2 < j_3 < \cdots < j_r$, and $b_{j_i} = a_h$ for $1 \leq i \leq r$. Finding $j_1, j_2, j_3, , j_r$ can be done in $\mathcal{O}(\log n)$ using n processors, when n is the size of b. After that, we assign $j_k - j_{k-1}$ to $D_{G_H}(j_{k-1}+1)$ for $1 < k \leq r$, and assign $j_1 + 1$ to $D_{G_h}(1)$.

For example, after doing these steps for computing D_{G_1} in Fig. 1, we will have $j = (3, 4, 5, 7)$ and $D_{(G_1)} = (4, x, x, 1, 1, 2, x, x, x, x, x, x)$, where x shows the amounts we have not computed yet. After that, we will put $D_{G_h}(k) = \sum_{j=1}^{k} D_{G_h}(j)$ for $1 \leq k \leq j_r + 1$. At the end of this step, we will have $D_{G_1} = (4, 4, 4, 5, 6, 8, 8, 8, 8, 8, 8, 8)$. In the last step of computing D_{G_h}, we will assign ∞ to the entries $j_r + 2$ to n of D_{G_h}. Therefore, we will have $D_{G_1} = (4, 4, 4, 5, 6, 8, 8, \infty, \infty, \infty, \infty, \infty)$.

Computing D_{G_h} for all amounts of h can be done in $\mathcal{O}(\log n)$ using $mn/\log(n)$ processors [8].

C. Finding the Most Weighted Path

After computing matrix D_G, which in fact, shows the weight of some paths, we should extract the vertices of the maximum weighted path from the upper left vertex(named as source) of G to the downright vertex(called sink) of it. For a maximum-cost path from the source to the sink in G, like $P = v_1, v_2, \ldots, v_l$, there could be more than one vertex in P that is in a specific row in G.

A vertex like v_i, in P, is a cross-vertex if v_i is the leftmost vertex of P in its row. We denote the cross-vertex on the jth row of G with the notation $v[j]$ to distinguish cross-vertices from other vertices on P. Clearly, $v_1 = v[1]$ if we consider row number 1(not 0) as the first row.

D. Eliminating LCS from D_G

Now, we should solve two subproblems: Identify the cross vertices of P, and identify the other vertices of P. We start with the first subproblem:

All cross-vertices on a maximum-cost path can be obtained as the side effect of computing the cost matrix D_G. Suppose that we are computing $D_G(i, j)$, that is, finding in G the jth breakout vertex of $x = G(1, i)$, and show it by y. Let p be the maximum-cost path from x to y, and let the vertex q be the cross-vertex of p on the boundary between G_U and G_L. It means that $q = v[m/2 + 1]$.

The second subproblem is simple. Suppose that $v[i]$ and $v[i+1]$ are vertex $G(i, j_1)$ and vertex $G(i+1, j_2)$ respectively. Then vertices on the ith row of G from $G(i, j_1+1)$ to $G(i, j_2-1)$ must be all the vertices between $v[i]$ and $v[i+1]$ in p, considering that we have not considered diagonal edges with weight 0. Therefore, once all cross-vertices have been identified through the first stage, there should be no difficulty in writing all of the vertices of p in an array, by using a parallel PrefixSum function with time complexity of $\mathcal{O}(\log n)$ that uses n processors.

E. Identifying the LCS

In the last stage of the algorithm, we check the cost of each edge $e = (v[k], v[k+1])$. Symbol a_i is to be marked if we find that the edge e has a cost of 1 and vertex $v[k]$ has the column index i. The LCS of a and b that correspond to p can be obtained by sorting those marked symbols. Since the number of edges on p is bounded by $n+m$, and because checking the cost on an edge takes constant time, marking symbols in a can be done in constant time with n processors or in $\mathcal{O}(\log n)$ with $n/(\log n)$ processors.

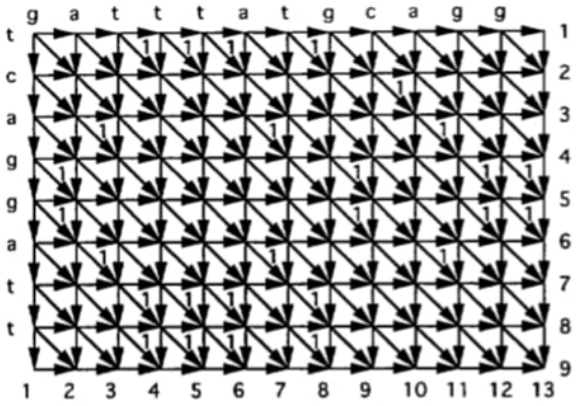

Fig. 1: Finding the LCS of "gatttatgcagg" and "tcaggatt" is equal to finding the maximum weighted path in this graph from the upper left vertex to downright vertex. The image is from [8]. In our implementation, we do not consider diagonal edges with weight 0.

III. Experimental Results

In this section, we will give the average running time of our program for some strings with different lengths. We reached these results by using 32 processors. In each figure, the length of the smaller string, which is m is written on the caption of the figure, and the length of the longer string, defined by n, is a power of 2, greater than m and less than 129. In each figure, m is constant and for different amounts of n, the average running time of the program is shown.

$$D_{G_U} = \begin{pmatrix} 2 & 7 & 9 & 12 \\ 3 & 7 & 9 & 12 \\ 4 & 7 & 9 & 12 \\ 5 & 7 & 9 & 12 \\ 6 & 7 & 9 & 12 \\ 7 & 9 & 11 & 12 \\ 8 & 9 & 11 & 12 \\ 9 & 11 & 12 & \infty \\ 10 & 11 & 12 & \infty \\ 11 & 12 & \infty & \infty \\ 12 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{pmatrix} \quad D_{G_L} = \begin{pmatrix} 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & \infty \\ 4 & 5 & \infty & \infty \\ 5 & 6 & \infty & \infty \\ 6 & 8 & \infty & \infty \\ 7 & 8 & \infty & \infty \\ 8 & 11 & \infty & \infty \\ 9 & 11 & \infty & \infty \\ 11 & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty \\ 13 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{pmatrix}$$

$D_G(1,3) = \min(D_{G_L}(1,3) = 4, D_{G_U}(1,3) = 9, D_{G_L}(D_{G_U}(1,1),2)=4, D_{G_L}(D_{G_U}(1,2),1)=8)=4$

Fig. 2: D_{G_U}, D_{G_L} and computing $D_G(1,3)$ for the graph in Fig 1

In the figures, the x axis shows the logarithm of n, and the y axis shows the average running time of our program in seconds to compute the LCS. Since we have always considered $n > m$, in each figure, the running time assigned to $n \leq m$ is 0.

For example, in the bar for $m = 16$, because we have considered $n > m$, it shows the running time for $n \leq m$ equal to 0. You can obviously see that for small amounts of n, the running time increases by a constant proportion of $\log(n)$, and after the number of needed processors is greater than 32, it increases linearly by increasing n. This observation verifies our complexity analysis.

To be precise, for small values of n, the running time does not increase by a proportion to $\log(n)$ but it increases with a proportion to $\log^3(n)$, which is the complexity of our algorithm; however, because in our test cases, n is not very large, it is not very transparent. The important point which verifies the correctness of the program is that until those 32 processors are enough for parallel processing, the running time is much better than the running time of a sequential algorithm, and it is logarithmic.

Because we implemented a simpler version of the first algorithm in [8], the complexity of the approach is not exactly the same as the complexity of the algorithm in [8].

IV. Related Work

In [14], Yang et al. has developed an efficient parallel algorithm on GPUs for the LCS problem. They have proposed a new technique that changes the data dependency in the score table used by dynamic programming algorithms to enable higher degrees of parallelism. In [5], Dhraief et al. have studied some languages for parallel development on GPU (CUDA and OpenCL). Then, they presented a parallelization approach to solving the LCS problem on GPU. They have evaluated their proposed algorithm on an NVIDIA platform using CUDA, OpenCl. Nandan Babu et al. [10] has introduced an algorithm with

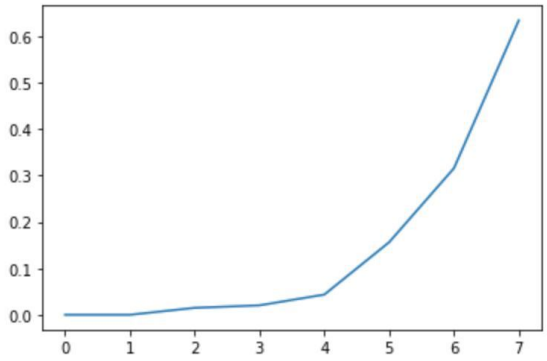

Fig. 3: running times for m=2

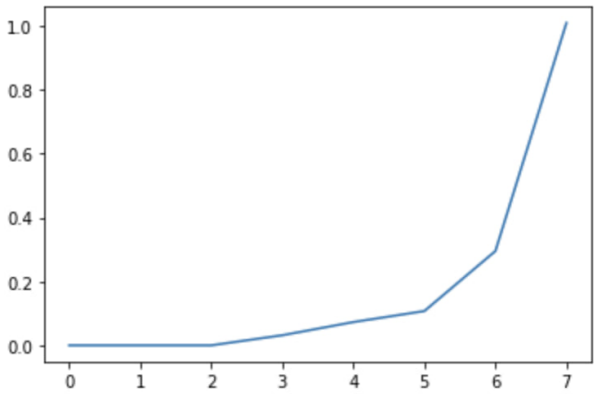

Fig. 4: running times for m=4

$\mathcal{O}(\log m)$ time using mn processors when m is the length of the shorter string and n is the length of the longer string. Some parallel algorithms [7], [13] and [4] are represented to find the LCS of multiple strings. Nguyen et al. [11] has introduced the basics of parallel prefix scans.

V. Conclusion and Future work

In this paper, we presented an implementation of a parallel algorithm for computing the LCS of two strings in Chapel, and showed the average running time of it for strings with different lengths.
For future research, we will create a library for computing the LCS in Chapel, and extend it by adding other parallel methods for computing the LCS.

Acknowledgment

We thank the Chapel and Arkouda communities for their support as well as the NSF funding support through grant CCF-2109988. We also appreciate Nese L. Us for helping us to debug the code and Jose L. Mojica Perez for helping us in debugging the code and installing Chapel.

References

[1] Shyan Akmal and Virginia Vassilevska Williams. Improved approximation for longest common subsequence over small alphabets, 2021.

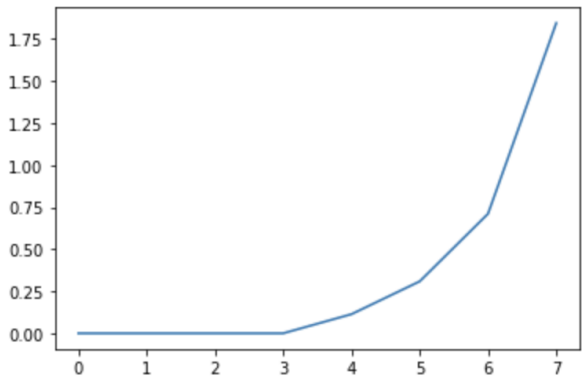

Fig. 5: running times for m=8

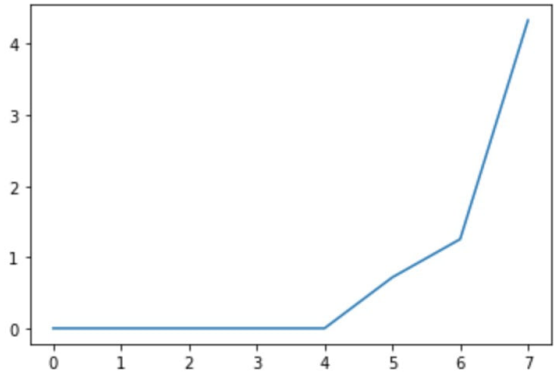

Fig. 6: running times for m=16

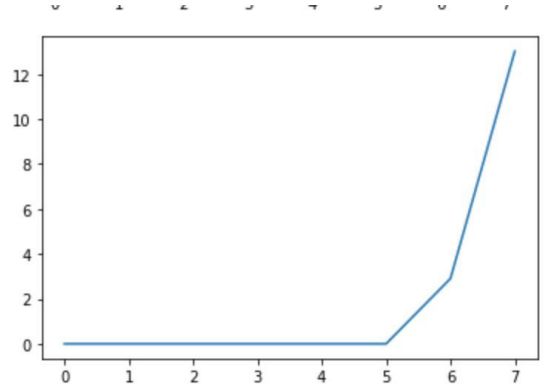

Fig. 7: running times for m=32

[2] L. Bergroth, H. Hakonen, and T. aRaita. New approximation algorithms for longest common subsequences. In Proceedings. String Processing and Information Retrieval: A South American Symposium (Cat. No.98EX207), pages 32–40, 1998.

[3] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. The International Journal of High Performance Computing Applications, 21(3):291–312, 2007.

[4] Yixin Chen, Andrew Wan, and Wei Liu. A fast parallel algorithm for finding the longest common sequence of multiple biosequences. BMC Bioinformatics, 7(S4), December 2006.

[5] Amine Dhraief, Raik Issaoui, and Abdelfettah Belghith. Parallel computing the longest common subsequence (lcs) on gpus: Efficiency and language suitability. 2011.

[6] Marko Djukanovic, Günther R. Raidl, and Christian Blum. Finding longest common subsequences: New anytime a∗ search results. Applied Soft Computing, 95:106499, 2020.

[7] Dmitry Korkin, Qingguo Wang, and Yi Shang. An efficient parallel algorithm for the multiple longest common subsequence (mlcs) problem. In 2008 37th International Conference on Parallel Processing, pages 354–363, 2008.

[8] Mi Lu and Hua Lin. Parallel algorithms for the longest common subsequence problem. IEEE Transactions on Parallel and Distributed Systems, 5(8):835–848, 1994.

[9] Michael Merrill, William Reus, and Timothy Neumann. Arkouda: interactive data exploration backed by Chapel. In Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop, pages 28–28, 2019.

[10] K. Nandan Babu and S. Saxena. Parallel algorithms for the longest common subsequence problem. In Proceedings Fourth International Conference on High-Performance Computing, pages 120–125, 1997.

[11] Hubert Nguyen. Gpu Gems 3, chapter 39. Addison-Wesley Professional, first edition, 2007.

[12] Aviad Rubinstein, Saeed Seddighin, Zhao Song, and Xiaorui Sun. Approximation algorithms for lcs and lis with truly improved running times. In 2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS), pages 1121–1145, 2019.

[13] Qingguo Wang, Dmitry Korkin, and Yi Shang. A fast multiple longest common subsequence (mlcs) algorithm. IEEE Transactions on Knowledge and Data Engineering, 23(3):321–334, 2011.

[14] Jiaoyun Yang, Yun Xu, and Yi Shang. An efficient parallel algorithm for longest common subsequence problem on gpus. 2010.