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 critical 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 easily handle large string analytics on the back-end high-performance computing resources from the front-end Python interface. The Chapel-enabled parallel LCS algorithm can identify the longest common subsequences of two strings, and experimental results are given to show how the number of parallel resources and the length of input strings can affect the algorithm's performance.**

***Index Terms*—string algorithms, parallel computing, Chapel programming language**

I. Introduction

The longest Common Subsequence (LCS) of a set 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, particularly in the context of bioinformatics, where strings represent DNA or protein sequences [7].

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, and researchers 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 [14], [1] and [4], 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 [11], 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 a variant of the first algorithm 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 and the number of parallel resources.
3) This work is based on an open-source framework Arkouda [12]. 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

Lu and Liu's parallel method, as presented in their work [11], offers a novel approach to solving the Longest Common Subsequence (LCS) problem. The central idea behind their method is to transform the LCS problem into a search for the maximum weighted path between two specially designated vertices within a grid graph.

In essence, this algorithm operates recursively. To elucidate, when tasked with discovering the maximum weighted path between vertices a and b, it seeks out a strategic intermediary vertex, denoted as c. The objective is to maximize the combined weight of the path from a to c and the path from c to b. Achieving this necessitates the determination of two critical components: the maximum weighted path from a to c and from c to b. This recursive nature stems from the need to address these intermediate paths.

Like numerous other recursive algorithms, there exists a risk of exponential time complexity. To mitigate this concern, Lu and Liu's method employs dynamic programming techniques to efficiently tackle the recursive challenges posed by the problem. This pragmatic approach helps maintain computational tractability while deriving optimal solutions for the LCS problem.

B. Recursive and Parallel Methods

1) Recursive Formula: If vertices a and b are in two consecutive rows of the grid graph, the maximum weighted path between them can be calculated using a parallel prefix min algorithm.

In our implementation, we define several matrices, but for most of them, we compute only specific cells when needed. The most crucial matrix is denoted as D_G. Cell (i, j) of D_G indicates the column of the leftmost vertex in row i of the grid graph G that can be reached by a path with weight j from the vertex in the i-th column of the first row. If matrix D_G has more than two rows, we employ 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} represents the upper half of D_G, and D_{G_L} represents the lower half of D_G.

To better understand this formula, consider that $D_{G_U}(i,k)$ represents the leftmost vertex in the bottom row of G_U that can be reached from the i-th vertex of the first row of G_U with a path of weight k. Therefore, $D_{G_L}(D_{G_U}(i,k))(j-k)$ represents the leftmost vertex in the bottom row of G_U that can be reached from the i-th vertex of the first row of G such that the weight of this path is j. The sum of the weight of the edges in G_U for this path is k, and the sum of the weights of the edges in G_L for this path is $j-k$. An example of computing a cell of D_G is shown in Figure 2.

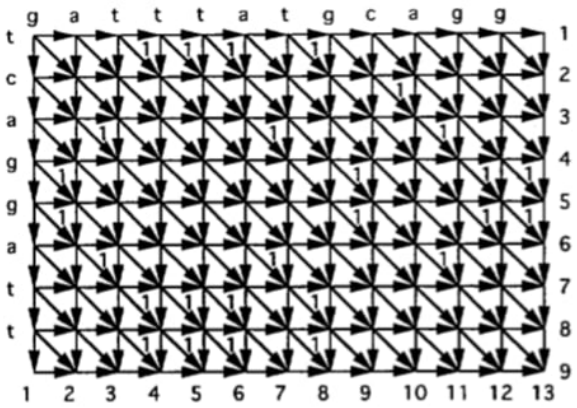

Fig. 1: Finding the LCS of *gatttatgcagg* and *tcaggatt* is equal to finding the maximum weighted path in this graph from the upper left vertex to downright vertex. This figure is copied from [11]. It's worth noting that in our implementation, we exclude diagonal edges with a weight of 0.

We define a vertex v in the bottom row of a grid graph G as the j-th breakout vertex of the vertex $G(1,i)$ if v is the leftmost vertex in the bottom row and there exists a path with cost j from vertex $G(1,i)$ to v. For instance, in Figure 1, vertices (9, 2), (9, 3), (9, 4), (9, 5), and (9, 13) represent 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 that the 5th breakout of vertex (1,8) is ∞. An example of computing a cell of D_G can be seen in Figure 2.

A matrix is considered monotone if, given two consecutive columns c_1 and c_2 with c_1 to the left of c_2, the cell with the minimum value in c_2 is not in a row higher than the row containing the cell with the minimum value in c_1.

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

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

Fig. 2: D_{G_U}, D_{G_L} and computing $D_G(1,3)$ for the graph in Fig 1. This figure is copied from [11].

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 = ⌈(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
```

2) Key Functions: A critical component of the matrix computation for D_G (the cost matrix of G) lies in Algorithm 1. This algorithm recursively identifies the minimum element index in each monotone matrix column and stores these indices in an array called $mins$. Specifically, $mins[i]$ preserves the index of the minimum element in column i. The variables $left$, $right$, top, and $bottom$ correspond to the first and last columns and the matrix's first and last rows, respectively.

Algorithm 2: Find Min Index Function

```
findMinIndex(dgu, dgl, vertex : int, col : int, top :
  int, bottom : int)
var listsize = bottom − top + 1
var exp : int = 1
var expm1, expnot : int
var prefix : [0..listsize − 1]int
var minIndex : [0..listsize − 1]int
forall (i in 0..listsize-1) do
    prefix[i] = find_cell(dgu, dgl, vertex, i + top, col)
    minIndex[i] = i
end
while (exp < listsize) do
    expm1 = exp − 1
    expnot = ẽxp
    forall (j in 0..listsize-1) do
        if (j&exp ≠ 0) then
            if (prefix[j&expnot|expm1] ≤ prefix[j] )
              then
                prefix[j] = prefix[j&expnot|expm1]
                minIndex[j] =
                  minIndex[j&expnot‖expm1]
            end
        end
    end
    exp = exp << 1
end
return minIndex[listsize − 1] + top
```

We consistently initialize the variable $firstind$ to match the value of $left$. Cases where $left \neq firstind$ arise in recursive processes, but these intricacies do not require user intervention. Within the pseudocode, the commands $forall$ and $cobegin$ signify situations where all enclosed commands will be executed in parallel, while for executes commands within its loop sequentially.

Algorithm 1 presents the pseudocode for $ColMin$. Inside Algorithm 1, we rely on Algorithm 2, which is responsible for determining the index of the minimum value within a column of a matrix. Notably, Algorithm 2 operates in parallel.

It's important to note that every recursive relation possesses its own set of 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 only two rows; instead, we approach it differently. We consider the input strings a and b and define the cost matrix of the grid graph G consisting of rows numbered h and $h+1$ as D_{G_h}.

In our implementation, we assume that D_{G_H} has only two rows. The first row of this matrix represents the 0^{th} breakout for each vertex, and we define the 0^{th} breakout of the i^{th} vertex of G_h as i. While [11] does not define the 0^{th} breakout, we introduce this definition for simplifying the implementation. Therefore, for the sake of simplicity in notation, we assume that D_{G_h} consists of only one row, representing the first breakout of each vertex in the upper row of the grid graph comprising rows h and $h+1$ of G.

We represent the i^{th} letter of the string s as s_i, where $i \geq 1$. To compute D_{G_H}, we need to determine values $j_1, j_2, j_3, \ldots, j_r$ such that $j_1 < j_2 < j_3 < \ldots < j_r$, and $b_{j_i} = a_h$ for $1 \leq i \leq r$. Finding these values can be accomplished in $\mathcal{O}(\log n)$ using n processors, where n represents the size of string b. Afterward, we assign $j_k - j_{k-1}$ to $D_{G_H}(j_{k-1}+1)$ for $1 < k \leq r$, and set $D_{G_h}(1)$ to $j_1 + 1$.

For instance, after performing these steps to compute D_{G_1} in Fig. 1, we obtain $j = (3, 4, 5, 7)$ and $D_{(G_1)} = (4, x, x, 1, 1, 2, x, x, x, x, x, x)$, where x represents values that have not yet been computed. Subsequently, we set $D_{G_h}(k) = \sum_{j=1}^{k} D_{G_h}(j)$ for $1 \leq k \leq j_r + 1$. At the conclusion of this step, $D_{G_1} = (4, 4, 4, 5, 6, 8, 8, 8, 8, 8, 8, 8)$. In the final step of computing D_{G_h}, we assign ∞ to the entries $j_r + 2$ to n of D_{G_h}. Consequently, we arrive at $D_{G_1} = (4, 4, 4, 5, 6, 8, 8, \infty, \infty, \infty, \infty, \infty)$.

The computation of D_{G_h} for all values of h can be achieved in $\mathcal{O}(\log n)$ using $mn/\log(n)$ processors [11].

C. Finding the Maximum Weighted Path

After computing matrix D_G, which represents the weights of various paths, we need to extract the vertices of the maximum weighted path from the upper-left vertex (referred to as the source) of G to the lower-right vertex (referred to as the sink). For a maximum-cost path $P = \langle v_1, v_2, \ldots, v_l \rangle$ from the source to the sink in G, there can be multiple vertices in P that belong to the same row in G.

A vertex v_i in P is considered a cross-vertex if it is the leftmost vertex of P within its respective row. We use the notation $v[j]$ to represent a cross-vertex on the j-th row of G, distinguishing it from other vertices in P. It is evident that $v_1 = v[1]$, assuming row number 1 (not 0) as the first row.

D. Eliminating LCS from D_G

Now, we need to address two subproblems: identifying the cross-vertices of P and identifying the other vertices of P. Let's start with the first subproblem:

All cross-vertices on a maximum-cost path can be determined as a byproduct of computing the cost matrix D_G. Suppose we are computing $D_G(i, j)$, which corresponds to finding in G the j-th breakout vertex of $x = G(1, i)$, denoted as y. Let p be the maximum-cost path from x to y, and let vertex q be the cross-vertex of p on the boundary between G_U and G_L. This implies that $q = v[m/2 + 1]$.

The second subproblem is straightforward. If $v[i]$ and $v[i+1]$ represent vertex $G(i, j_1)$ and vertex $G(i+1, j_2)$, respectively, then the vertices on the i-th row of G from $G(i, j_1 + 1)$ to $G(i, j_2 - 1)$ must all be part of the vertices between $v[i]$ and $v[i+1]$ in p, considering that diagonal edges with weight 0 are not considered. Therefore, once all cross-vertices have been identified in the first stage, there should be no difficulty in listing all the vertices of p in an array. This can be accomplished using a parallel PrefixSum function with a time complexity of $\mathcal{O}(\log n)$, employing n processors.

E. Identifying the LCS

In the final stage of the algorithm, we examine the cost of each edge $e = (v[k], v[k+1])$. Symbol a_i is marked if we find that the edge e has a cost of 1 and vertex $v[k]$ has a column

index of i. The LCS of strings a and b corresponding to path p can be obtained by sorting these marked symbols. Given that the number of edges on p is bounded by $n+m$, and checking the cost of an edge takes constant time, marking symbols in a can be accomplished in constant time using n processors or in $\mathcal{O}(\log n)$ using $n/(\log n)$ processors.

III. Experimental Results

A. Experimental System

We conducted our experiments on a system with 2.00GHz Intel(R) Xeon(R) Gold 6330 CPUs. Our program was executed using Chapel version 1.31.0.

In our Chapel configuration, we set the `CHPL_TASKS` variable to ``qthreads'', and `CHPL_LLVM` was configured as ``bundled''. The number of cores we utilized was controlled using the command `export CHPL_RT_NUM_THREADS_PER_LOCALE=x`, where x represents the desired number of cores.

B. Performance

In this section, we embark on an in-depth exploration of the multifaceted performance characteristics exhibited by the proposed parallel algorithm. Our initial focus is on examining how the execution time is influenced by varying the lengths of input strings, with one of them held constant. The comprehensive results of these investigations are meticulously presented in Fig. 3.

Fig. 3 eloquently illustrates a series of experiments where we meticulously maintain the length of one string at values of 2 and 4, while systematically extending the size of the other string from 2 to 8192. These empirical investigations were conducted with 32 processing cores.

Our observations from this figure reveal a striking pattern of nearly exponential growth in the total execution time required to determine the longest common subsequence. This growth is prominently evident when we hold the length of one string constant and progressively vary the length of the other. Specifically, when one string size is fixed at 2, our rigorous analysis yields a precise regression equation of $time = 5 \times 10^{-05} \times e^{0.8552 \times \text{size}}$, accompanied by an R^2 value of 0.9046. Similarly, for the scenario where one string length remains constant at 4, our analysis furnishes the regression equation as $time = 7 \times 10^{-05} \times e^{0.9556 \times \text{size}}$, accompanied by a notably higher R^2 value of 0.9646.

These findings distinctly underscore the algorithm's remarkable sensitivity to input size. This sensitivity is vividly exemplified by the substantial and expedited growth in execution time experienced when dealing with larger strings.

Intriguingly, as we look at the results obtained with eight processing cores (as depicted in Fig. 4), we discern a similar trend. However, subtle differences emerge when examining the fitting equations. When one string length is kept at 2, our analysis yields a fitting equation of $time = 2 \times 10^{-05} \times e^{0.9535 \times \text{size}}$, resulting in an exceptionally high R^2 value of 0.9817. Similarly, for a fixed string length of 4, the regression equation is expressed as $time = 3 \times 10^{-05} \times e^{1.097 \times \text{size}}$, with an even higher R^2 value of 0.991.

These nuances in the results with eight cores highlight that (1) for the same fixed string size, increasing the size of the other string incurs a significantly faster growth in execution time. Notably, focusing on the exponent constants reveals that, for a fixed string size of 2, the execution time increase with 8 cores is approximately $0.4 \times e^{0.1283}$ times that of 32 cores. Similarly, for a fixed string size of 4, the execution time increase with 8 cores is roughly $\frac{3}{7} \times e^{0.1414}$ times that of 32 cores. These insights underscore the intriguing relationship between input length and core count, elucidating that increasing string length has a more profound impact on execution time than reducing the number of processing cores.

In Table I, we expand our testing to include various fixed sizes of strings. Then, we calculate the speedup of performance on 32 cores compared to that on 8 cores. The results underscore two key observations:

(1) Effective Parallelization: As we add processing cores, a clear reduction in total execution time becomes evident for identical string sizes. When both string lengths are larger, the evidence becomes more obvious. An average of $1.8\times$ speedup can be achieved. This demonstrates the tangible effectiveness of our parallel method, affirming its ability to optimize performance.

(2) Input Size Impact: It is noteworthy that amplifying the lengths of either string substantially impacts the total execution time. Specifically, an increase in the size of either string leads to a noticeable escalation in the overall execution duration.

These insights provide valuable confirmation of the efficacy of our parallel approach while highlighting the sensitivity of execution time to changes in input size.

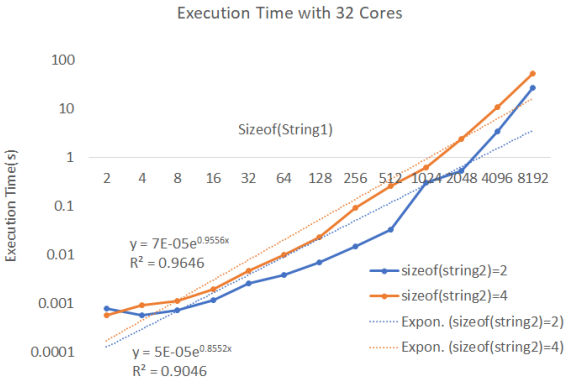

Fig. 3: The execution time experiences exponential growth as one string's size increases while the other remains fixed. This phenomenon occurs within the context of a computational environment equipped with a total of 32 cores.

IV. Related Work

In [17], Yang *et al.* developed an efficient parallel algorithm on GPUs for the LCS problem. They proposed a new technique that changes the data dependency in the score table used by

TABLE I: Algorithm execution time (seconds) and speedup for different number of cores and string sizes

number of cores	sizeof(string1)	sizeof(string2)				Speedup			
		2	4	8	16	2	4	8	16
32	1024	0.119569	0.62156	1.63747	3.2784	3.095577	2.555988	1.406334	1.449948
	2048	0.890801	2.4456	4.51969	12.2226	1.54443	2.583583	1.777177	1.575164
	4096	6.64456	11.1501	16.5999	51.9423	0.805853	1.902557	1.997831	1.40539
	8192	29.5529	53.5266	67.5972	190.575	0.971884	1.648791	2.152708	2.1932
8	1024	0.370135	1.5887	2.30283	4.75351				
	2048	1.37578	6.31841	8.03229	19.2526				
	4096	5.35454	21.2137	33.1638	72.9992				
	8192	28.722	88.2542	145.517	417.969				

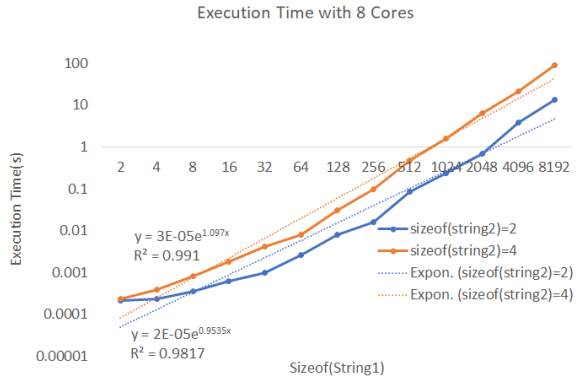

Fig. 4: The execution time experiences exponential growth as one string's size increases while the other remains fixed. This phenomenon occurs when we reduce the number of cores from 32 to 8.

dynamic programming algorithms to enable higher degrees of parallelism. In [8], Garcia *et al.* introduce a coarse-grained multicomputer algorithm which works in $\mathcal{O}(N^2/P)$ time complexity with P processors and $\mathcal{O}(P)$ communication steps. Dhraief *et al.* [6] studied languages for parallel development on GPUs (CUDA and OpenCL) and presented a parallelization approach to solving the LCS problem on GPU. Their proposed algorithm was evaluated on an NVIDIA platform using CUDA and OpenCL. Babu *et al.* [9] introduced a parallel algorithm to compute the LCS using graphics hardware acceleration and multiple levels of parallelism. Babu and Saxena [3] introduced an algorithm with $\mathcal{O}(\log m)$ time complexity using mn processors, where m is the length of the shorter string and n is the length of the longer string. Several parallel algorithms (e.g., [10], [16], and [5]) have been proposed to find the LCS of multiple strings. Nguyen *et al.* [13] introduced the basics of parallel prefix scans. Tchendji *et al.* [15] provided a parallel algorithm to solve the LCS problem with constraints. Specifically, their problem is to find the longest common subsequence, which excludes some strings as its substrings. In [2], Alves, Caceres and Song introduce a parallel algorithm for the all-substrings longest common subsequence problem. In this problem, given two strings A and B, the goal is to compute the LCS of A and each substring of B denoted as B'.

V. Conclusions and Future Work

This paper introduces a parallel algorithm implementation for calculating the Longest Common Subsequence (LCS) of two strings using the Chapel programming language. It includes an analysis of the algorithm's average runtime across strings of varying lengths on different numbers of cores. Our source code is open source and available on GitHub at https://github.com/SoroushVahidi/parallel-longest-common-subsequence/

Our future research endeavors will focus on expanding the capabilities of this algorithm implementation. Specifically, we plan to develop a comprehensive library for LCS computation in Chapel, which will encompass additional parallel methods for solving the LCS problem.

Furthermore, an intriguing avenue for future research lies in the development of parallel algorithms tailored to address various LCS problems with specific constraints. These efforts aim to provide more versatile and efficient solutions for a wide range of real-world applications.

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 debug the code and Jose L. Mojica Perez for helping us debug the code and install Chapel.

References

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

[2] Carlos E.R. Alves, Edson N. Caceres, and Siang Wun Song. A coarse-grained parallel algorithm for the all-substrings longest common subsequence problem. *Algorithmica*, 45(3):301–335, Jul 2006.

[3] 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.

[4] L. Bergroth, H. Hakonen, and T. Raita. 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.

[5] 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.

[6] Amine Dhraief, Raik Issaoui, and Abdelfettah Belghith. Parallel computing the longest common subsequence (LCS) on GPUs: Efficiency and language suitability. In *The 1st International Conference on Advanced Communications and Computation (INFOCOMP)*, pages 143–148, 10 2011.

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

[8] T. Garcia, J.-F. Myoupo, and D. Semé. A coarse-grained multicomputer algorithm for the longest common subsequence problem. In *11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (Euro-PDP)*, page 349 – 356, 2003.

[9] John Kloetzli, Brian Strege, Jonathan Decker, and Marc Olano. Parallel Longest Common Subsequence using Graphics Hardware. In Jean M. Favre and Kwan-Liu Ma, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2008.

[10] 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.

[11] 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.

[12] 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.

[13] Hubert Nguyen. *GPU Gems 3*, chapter 39. Addison-Wesley Professional, first edition, 2007.

[14] 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.

[15] Vianney Kengne Tchendji, Armel Nkonjoh Ngomade, Jerry Lacmou Zeutouo, and Jean Frédéric Myoupo. Efficient cgm-based parallel algorithms for the longest common subsequence problem with multiple substring-exclusion constraints. *Parallel Computing*, 91:102598, 2020.

[16] 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.

[17] Jiaoyun Yang, Yun Xu, and Yi Shang. An efficient parallel algorithm for longest common subsequence problem on GPUs. In *World Congress on Engineering (WCE)*, London, England, June 2010.