

Approximate Longest Common Substring of Multiple Strings: Experimental Evaluation

Hamed Hasibi*, Neerja Mhaskar*, and William F. Smyth

Algorithms Research Group,
Department of Computing & Software
McMaster University, Canada
hasibih@mcmaster.ca, pophlin@mcmaster.ca, smyth@mcmaster.ca

Abstract. Determining an *Approximate Longest Common Substring* (ALCS) among a collection of strings $\mathbf{S} = \{s_1, s_2, \dots, s_m\}$, $m \geq 2$, is especially significant in computational biology — for instance, when tracking related mutations across several genomic sequences. To address this, the Restricted k - t Longest Common Substring (*Rkt*-LCS) problem was introduced by Hasibi *et al.* in [12], along with several efficient solutions under various distance metrics. In this paper, we describe and evaluate two parallel strategies for computing the *Rkt*-LCS of a set \mathbf{S} of strings under the Hamming distance metric. The first strategy parallelizes the computation of the core data structure introduced in [12], while the second makes use of a Graphics Processing Unit (GPU) that we demonstrate executes over a hundred times faster than its CPU counterpart.

Keywords: Approximate longest common substring, Hamming distance, parallel computation, longest common prefix, biological sequences.

1 Introduction

The Longest Common Substring (LCS) problem has been extensively studied for decades, with applications ranging from genome sequence comparison and plagiarism detection to compression algorithms. However, evolutionary variation necessitates algorithms that can tolerate mismatches. Hence, the approximate longest common substring (ALCS) problem — which seeks the longest substring appearing in strings while permitting a fixed number of operations such as mismatches, insertions, or deletions — is well-suited to comparative genomics. The distance metric d_δ measures the minimum number of allowed operations required to transform a string u to another string u' , and denoted as $d_\delta(u, u')$. Recently, Hasibi *et al.* in [12] introduced several new variants of the ALCS problem for a set of strings \mathbf{S} . One of the variants is stated as follows:

Problem 1. [Restricted k - t Longest Common Substring (*Rkt*-LCS) [12]] Given integers $k, t, m \in \mathbb{N}$ with $1 \leq t \leq m$ and a set $\mathbf{S} = \{s_1, s_2, \dots, s_m\}$ of strings, find a longest substring u taken from any string in \mathbf{S} such that there exist t distinct strings $s'_1, \dots, s'_t \in \mathbf{S}$ with corresponding substrings u_1, \dots, u_t satisfying $d_\delta(u, u_j) \leq k$ for every $j = 1, \dots, t$.

In Problem 1, the distance metric $\delta = \{H, L, E\}$ denotes the Hamming, Levenshtein, and edit distances, respectively, which are defined in the next section. The authors present two $\mathcal{O}(N^2)$ and $\mathcal{O}(mN \log^k \ell)$ time sequential algorithms for $\delta = H$

* Corresponding author

and two $\mathcal{O}(k\ell N^2)$ and $\mathcal{O}(mN \log^k \ell)$ time sequential algorithms for $\delta \in \{L, E\}$, where ℓ is the length of each string and N is the total length of all strings.

In this paper, we present two parallel implementations for the *Rkt*-LCS problem under the Hamming distance metric ($\delta = H$), each designed for speed and scalability on large datasets. The first implementation uses p processors to lower the sequential $\mathcal{O}(N^2)$ runtime by a factor of p , whereas the second accelerates the computation still further via GPU¹ threads².

In Section 2, we introduce the relevant terminology. Section 3 reviews the LCS literature. Section 4 presents detailed descriptions of each implementation, while Section 5 provides a comparative analysis of their run times. Finally, we summarize in Section 6.

2 Preliminaries

A **string** s is a sequence of $n \geq 0$ **letters** drawn from a finite ordered **alphabet** Σ of size $\sigma = |\Sigma|$. A string s of **length** n is represented as an array $s[1..n]$ with elements from Σ . The length of a string s is denoted by $|s| = n$. A **substring** of s is defined as a contiguous sequence of characters within s . Specifically, given integers $1 \leq i \leq j \leq n$, a substring of s is denoted as $s[i..j] = s[i]s[i+1]\dots s[j]$. We say string s_1 **occurs** in string s_2 if there exists a substring $s_2[i..j]$ such that $s_2[i..j] = s_1$. A **prefix** of a string s is a substring that starts at position 1, i.e., $s[1..j]$ for some $1 \leq j \leq n$. Similarly, a **suffix** of a string s is a substring that ends at position n , i.e., $s[i..n]$ for some $1 \leq i \leq n$. Given two strings s_1 and s_2 , each of length n , the **Hamming distance** $d_H(s_1, s_2)$ is the number of positions at which s_1 and s_2 differ. For any two strings s_1 and s_2 of arbitrary lengths, the **edit distance** $d_E(s_1, s_2)$ is the minimum cost over all sequences of edit operations that transform s_1 to s_2 . In the case that each edit operation has unit cost, the edit distance is called **Levenshtein distance** and denoted $d_L(s_1, s_2)$. For $1 \leq i' \leq |s_1|$ and $1 \leq j' \leq |s_2|$, $LCP_{(s_1, s_2)}^{H, k}[i', j']$ is defined as the length of the longest common prefix (*LCP*) between the suffixes $s_1[i'..|s_1|]$ and $s_2[j'..|s_2|]$, allowing for at most k mismatches under Hamming distance. $MaxLCP_{(s_i, s_j)}^{H, k}$ is defined as an array of length $|s_i|$, where each entry $MaxLCP_{(s_i, s_j)}^{H, k}[i']$ stores the maximum value of $LCP_{(s_i, s_j)}^{H, k}[i', j']$ over all $1 \leq j' \leq |s_j|$. In other words, $MaxLCP_{(s_i, s_j)}^{H, k}$ stores the maximum value in each row of the $LCP_{(s_i, s_j)}^{H, k}$ table. The values of $LCP_{(s_1, s_2)}^{H, k}$ and $MaxLCP_{(s_i, s_j)}^{H, k}[i']$ for $s_1 = \text{ACGTA}$ and $s_2 = \text{ACGACA}$ with $k = 1$ are shown in Table 1.

3 Literature Review

The LCS problem has been investigated under four categories:

1. Exact LCS (ELCS) of two strings
2. Exact LCS of multiple strings

¹ A Graphics Processing Unit (GPU) is a high-throughput, multi-core processor designed for parallel computation, ideal for tasks like machine learning, data-parallel algorithms, and high performance computing [8].

² A GPU thread is a lightweight execution unit that runs a single instance of a kernel, working in parallel with thousands of other threads to process different pieces of data simultaneously [8].

	A	C	G	A	C	A	$MaxLCP_{(s_1,s_2)}^{H,1}$
A	4	1	1	3	1	1	4
C	1	3	1	1	2	1	3
G	1	1	2	1	1	1	2
T	1	1	2	1	2	1	2
A	1	1	1	1	1	1	1

Table 1: $LCP_{(s_1,s_2)}^{H,1}$ for $s_1 = \text{ACGTA}$ (rows) and $s_2 = \text{ACGACA}$ (columns). For example, element $[2,2]=3$ is the length of the longest common prefix of suffixes $s_1[2..] = \text{CGTA}$ and $s_2[2..] = \text{CGACA}$ with up to 1 mismatch. The array to the right is $MaxLCP_{(s_1,s_2)}^{H,1}$.

3. Approximate LCS (ALCS) of two strings
4. Approximate LCS of multiple strings

The LCS investigation started with the ELCS problem for two or more strings [2,5,7,15,19]. The ELCS for two strings is defined as follows: given two strings s_1 and s_2 of length n , locate the longest substring that appears in both strings. For a constant-size alphabet, Weiner obtained a linear-time solution [19]. Farach later proved that, even when the alphabet is unbounded, a suffix tree can be built in overall $\mathcal{O}(n)$ time plus the time to sort the characters, furnishing an $\mathcal{O}(n)$ algorithm on the word-RAM for polynomially bounded integer alphabets [7]. Building on Farach's construction, Charalampopoulos *et al.* showed that the problem over an alphabet $[0, \sigma)$ can be solved in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time and $\mathcal{O}(n / \log_\sigma n)$ space whenever $\log \sigma = o(\sqrt{\log n})$ [5].

The ELCS for multiple strings is defined as follows: given a string set $\mathbf{S} = \{s_1, s_2, \dots, s_m\}$, m positive integers x_1, x_2, \dots, x_m , and t where $1 \leq t \leq m$, find a longest substring u of any string in \mathbf{S} for which there are at least t strings $s_{i_1}, s_{i_2}, \dots, s_{i_t}$ ($1 \leq i_1 < i_2 < \dots < i_t \leq m$) such that u occurs at least x_{i_j} times in s_{i_j} for each j with $1 \leq j \leq t$ [2]. Lee and Pinzon solved the problem in $\mathcal{O}(N)$ time, where $N = \sum_{i=1}^m |s_i|$ [15]. Later, Arnold and Ohlebusch showed an $\mathcal{O}(N)$ time solution for this problem for all $1 \leq t \leq m$ [2]. Recently, the decision version of the ELCS problem for two strings has been studied in the quantum model by Jin and Nogler [13]. Given two strings, they propose a quantum algorithm that decides whether there exists an ELCS of length d in $\tilde{\mathcal{O}}(n^{2/3}/d^{1/6-o(1)})^3$ time complexity.

The *ALCS under Hamming distance for two strings* is defined as follows: look for the longest substring that appears in both s_1 and s_2 while permitting at most k mismatches. For $k = 1$, Babenko and Starikovskaya achieved an $\mathcal{O}(n^2)$ -time, $\mathcal{O}(n)$ -space algorithm (2011) [3]; Flouri *et al.* improved this to $\mathcal{O}(n \log n)$ time with the same space (2015) [9]. Leimeister and Morgenstern (2014) were the first to study the case $k > 1$, proposing a greedy heuristic solution [16]. Subsequent work yielded faster worst-case bounds: Flouri *et al.* (2015) gave a concise $\mathcal{O}(n^2)$ algorithm using only constant extra space; Grabowski (2015) produced two output-dependent algorithms running in $\mathcal{O}(n((k+1)(\ell_0+1))^k)$ and $\mathcal{O}(n^2 k / \ell_k)$ time, where ℓ_0 is the ELCS length and ℓ_k is the ALCS length [11]. Abboud *et al.* (2015) developed a randomized algorithm whose running time is $k^{1.5} n^2 / 2^{\Omega(\sqrt{(\log n)/k})}$ [1]. Thankachan *et al.* (2016) achieved $\mathcal{O}(n \log^k n)$ time and $\mathcal{O}(n)$ space [18]. Charalampopoulos *et al.* (2018) showed that if the answer

³ $\tilde{\mathcal{O}}(\cdot)$ hides a polylog(n) factor.

length is $\Omega(\log^{2k+2} n)$, the problem can be solved in linear time and space [4]. Kociumaka *et al.* (2019) proved, under the Strong Exponential Time Hypothesis, that no strongly subquadratic algorithm exists for $k = \Omega(\log n)$ [14]. Finally, Charalampopoulos *et al.* (2021) obtained an $\mathcal{O}(n \log^{k-1/2} n)$ -time, $\mathcal{O}(n)$ -space algorithm [5].

The *ALCS under edit distance* problem for two strings resembles the Hamming distance variant, but also permits insertions and deletions in addition to substitutions. Abboud *et al.* (2015) gave a randomized solution with run time $k^{1.5} n^2 / 2^{\Omega(\sqrt{(\log n)/k})}$ [1]. Thankachan *et al.* (2018) later proposed an $\mathcal{O}(n \log^k n)$ -time, $\mathcal{O}(n)$ -space algorithm for this edit-distance variant [17].

4 Parallel Implementations of *Rkt-LCS*

In this section, we first give a brief summary of the data structure computed to solve the *Rkt-LCS* problem in [12], and then present the details of our two new parallel implementations. Finally, we review the feasibility of one of the existing relevant implementations for computing *Rkt-LCS*.

The sequential algorithm proposed in [12] compares the longest common prefix of every pair of suffixes across all strings by using a data structure called *LengthStat* (see Definition 2), solving the *Rkt-LCS* problem under Hamming distance in $\mathcal{O}(N^2)$ time, where N is the total length of the input strings (see [12], Theorem 3).

LengthStat is a lightweight, statistical data structure that summarizes the distribution of k -approximate substring matches at different lengths, enabling rapid filtering of candidate substrings for *Rkt-LCS* [12]. It is defined to facilitate computing the answer to the *Rkt-LCS* problem: each entry records whether a length- l prefix of a suffix of s_i occurs in another string s_j with up to k mismatches.

Definition 2 (LENGTHSTAT [12]). *Let $\mathbf{S} = \{s_1, s_2, \dots, s_m\}$ be a set of strings. For every (i, x) pair with $1 \leq i \leq m$ and $1 \leq x \leq |s_i|$, define the $LengthStat_{(i,x)}^k$ table as follows:*

$$LengthStat_{(i,x)}^k[l, j] = \begin{cases} 1 & \text{if for some } y \ (1 \leq y \leq |s_j|), \ LCP_{(s_i, s_j)}^{H,k}[x, y] \geq l \\ 0 & \text{otherwise} \end{cases}$$

where $1 \leq j \leq m$ indexes the strings \mathbf{S} and $1 \leq l \leq |s_i| - x + 1$ is the prefix length.

The matrix is augmented with a final column $LengthStat_{(i,x)}^k[l, m+1]$ storing, for each row l , the sum of its first m entries, i.e. the number of strings in \mathbf{S} that share with $s_i[x..]$ a prefix of length at least l under k -mismatch Hamming distance.

To reduce the storage requirements of the LCP_k tables, in both our implementations, we derive *LengthStat* from the $MaxLCP_{(s_i, s_j)}^{H,k}$ arrays as follows:

$$LengthStat_{(i,x)}^k[l, j] = \begin{cases} 1, & \text{if } MaxLCP_{(s_i, s_j)}^{H,k}[x] \geq l \\ 0, & \text{otherwise} \end{cases}$$

In this section, we present two parallel implementations for computing the *Rkt-LCS* of the string set $\mathbf{S} = \{s_1, s_2, \dots, s_m\}$. Each algorithm returns the *Rkt-LCS* when its length is at least the user-specified threshold τ ; otherwise it outputs `Null`. Although the *LengthStat* definition is for strings of fixed length ℓ , our implementations allow for different length strings in \mathbf{S} . For ease of explanation, however, we provide results here only for the fixed length case.

1. **Parallel CPU algorithm.** We parallelize the $\mathcal{O}(N^2)$ sequential algorithm proposed by [12] across p processors, achieving an expected runtime of $\mathcal{O}(\frac{N^2}{p})$. Compared with the sequential algorithm, doubling the number of processors approximately yields a two-fold speed-up; furthermore, unlike the implementation of Chockalingam *et al.* [6] discussed later in this section, our runtime is not exponential in k , the number of permissible mismatches.
2. **Parallel GPU-accelerated algorithm.** This GPU implementation further reduces the effective run time of CPU implementation and scales well for large N . By offloading the computation of multiple $MaxLCP^{H,k}$ arrays to the GPU, this implementation achieves over hundred times improvement in runtime over the parallel CPU version.

4.1 Parallel CPU algorithm

In this parallel implementation, the *Rkt*-LCS computation for the string set $\mathbf{S} = \{s_1, s_2, \dots, s_m\}$, where each string has length ℓ , is distributed across p processors. We assume, for simplicity, that the m input strings are evenly divisible among the p processors (i.e., $p \mid m$), though this is not strictly required. Each processor P_r ($1 \leq r \leq p$) is assigned exactly $\frac{m}{p}$ consecutive strings of \mathbf{S} ; that is, the set of strings $\mathbf{S}_r = \{s_i \mid s_i \in \mathbf{S} \wedge i = (r-1) \cdot \frac{m}{p} + t', 1 \leq t' \leq \frac{m}{p}\}$ are assigned to processor P_r .

Processor P_r computes all the longest substrings in any $s_i \in \mathbf{S}_r$ that occur with at most k mismatches in at least t strings in entire \mathbf{S} . For each $s_i \in \mathbf{S}_r$, this is accomplished implicitly via the computation of the $LengthStat_{(p,i)}^k[l, j]$ tables (see Algorithm 1). Then, for each $s_i \in \mathbf{S}_r$, processor P_r computes the longest substring of s_i that satisfies the *Rkt*-LCS criteria (i.e., occurring in at least t other strings with at most k mismatches), identifying it as a *candidate Rkt*-LCS, denoted as C_i (as shown in Algorithm 2). Thus, each processor computes $\frac{m}{p}$ candidates, and across all processors, a total of m candidates for *Rkt*-LCS, C_1, C_2, \dots, C_m , are generated. The candidate with the greatest length among them is reported as the global *Rkt*-LCS. Under ideal load balancing and negligible communication overhead, this strategy reduces the sequential $\mathcal{O}(N^2)$ running time to $\mathcal{O}(N^2/p)$.

By Definition 2, the last column of the $LengthStat_{(i,p)}^k[l, m+1]$ table stores the count of strings in \mathbf{S} that contain a k -mismatch occurrence of $s_i[p, p+l-1]$. As shown in Algorithm 1, for efficiency, we do not store the entire $LengthStat$ table. Instead, the last column of this table is stored as a key-value pair data structure named **LS**, where the tuple (i, p, l) is the key and **count** is the value. The elements of the tuple i , p , and l refer to the string index of the string $s_i \in \mathbf{S}$, the starting position of s_i , and the prefix length of the p -th suffix of s_i , respectively. For instance, the entry $((1, 2, 4), 5)$ in **LS** states that the substring $s_1[2..2+4-1]$ occurs with at most k mismatches in five strings of the set \mathbf{S} . Storing **LS** for all strings requires $\mathcal{O}(m\ell^2)$ space, whereas retaining the entire $LengthStat$ tables in memory requires $\mathcal{O}(m^2\ell^2)$ space, where ℓ is the length of each string in \mathbf{S} .

Line #5 of Algorithm 1 calls the `COMPUTE_MAXLCP_K(s_i, s_j, k, τ)` function that computes $MaxLCP_{(s_i, s_j)}^{H,k}[0..|s_i| - \tau]$. The implementation of this function is adapted from Flouri *et al.* [9] with a single modification: once the entire matrix $LCP_{(s_i, s_j)}^{H,k}$ has been computed, the $MaxLCP_{(s_i, s_j)}^{H,k}$ array is obtained by taking the maximum value in

each row of $LCP_{(s_i, s_j)}^{H,k}$. This reduces the peak memory usage by freeing each $LCP^{H,k}$ table after computing its corresponding $MaxLCP^{H,k}$.

Algorithm 1 COMPUTE_LS_KEYVALUES($s_i, \{s_1, \dots, s_m\}, k, \tau$)

```

1: Define LS: (key : i, position, length, value : count)
2: Initialize empty array MaxLCP of length  $|s_i| - \tau + 1$ 
3: for  $j \leftarrow 1$  to  $m$  do
4:   if  $j = i$  then skip ▷ Skip comparing  $s_i$  to itself
5:   MaxLCP  $\leftarrow$  COMPUTE_MAXLCP_K( $s_i, s_j, k, \tau$ )
6:   for  $p \leftarrow 0$  to  $|s_i| - \tau$  do
7:      $\ell \leftarrow MaxLCP[p]$ 
8:     for  $l \leftarrow \tau$  to  $\ell$  do
9:       entry.key  $\leftarrow (i, p, l)$ 
10:      found  $\leftarrow$  false
11:      if entry.key exists in LS then
12:        entry.count++
13:        found  $\leftarrow$  true
14:        break
15:      if not found then
16:        Add (entry.key, 1) to LS
17: return LS

```

To determine the global *Rkt*-LCS, each processor P_r examines LS for each string $s_i \in \mathcal{S}_r$ and extracts the pairs (i, p, l) with the largest l such that the associated **count** satisfies **count** $\geq t$. As mentioned above, we refer to such a pair as a *candidate Rkt*-LCS originating from s_i (C_i). This procedure is detailed in Algorithm 2.

As noted earlier, to enhance scalability and performance, the candidate computation procedure (Algorithm 2) is parallelized. Each processor is assigned a subset of indices i — that is, a consecutive range of strings — and independently computes the *Rkt*-LCS candidates for each strings in \mathcal{S}_r ; that is, processor P_r invokes Algorithm 2 exactly $|\mathcal{S}_r|$ times, once for each string in \mathcal{S}_r . Finally, all candidates C_1, C_2, \dots, C_m are examined, and the longest of them is returned as the final *Rkt*-LCS. Since these computations are fully independent, the overall parallel time complexity is:

$$\mathcal{O}\left(\frac{m}{p} \cdot m\ell^2\right) = \mathcal{O}\left(\frac{N^2}{p}\right),$$

where N is the total length of the strings in \mathcal{S} and ℓ is the length of each string.

Algorithm 2 COMPUTE_CANDIDATE_RKT_LCS($s_i, \{s_1, s_2, \dots, s_m\}, k, t, \tau$)

```

1: results  $\leftarrow$  COMPUTE_LS_KEYVALUES( $s_i, \{s_1, s_2, \dots, s_m\}, k, \tau$ )
2: if results = null then
3:   return (i = -1, position = -1, length = -1)
4:  $C_i \leftarrow (-1, -1, -1)$ 
5: max.length  $\leftarrow$  0
6: for each entry in results do
7:   if entry.count  $\geq t$  and entry.key.length  $>$  max.length then
8:      $C_i \leftarrow$  entry.key
9:     max.length  $\leftarrow$  entry.key.length
10: return  $C_i$ 

```

4.2 Parallel GPU-accelerated algorithm

This implementation focuses on improving the computation of the $MaxLCP^{H,k}$ tables using GPU threads. Algorithms 1 and 2 are then used to compute the *Rkt*-LCS solution for the set \mathbf{S} .

In our parallel scheme, the strings in \mathbf{S} are evenly divided among the p processors. Let \mathbf{S}_r be the set of substrings assigned to processor P_r (as in Section 4.1). Then, each processor P_r off-loads every string in its assigned subset of strings \mathbf{S}_r to the GPU for further processing. We compute $MaxLCP^{H,k}(s_i, S_{buffer})$ for every $s_i \in \mathbf{S}_r$, where $S_{buffer} = s_1 \cdot s_2 \cdots s_{i-1} \cdot s_{i+1} \cdots s_m$ is the flattened concatenation of all strings in \mathbf{S} except s_i . In other words, $MaxLCP^{H,k}_{(s_i, S_{buffer})}$ is the concatenation of the $MaxLCP^{H,k}_{(s_i, s_j)}$ arrays for all $1 \leq j \leq m$ with $j \neq i$. The workload distribution, together with the launch time of every GPU task, is illustrated below. We explain below why $\mathcal{O}(m\ell)$ threads are created for each s_i .

$$\left\{ \begin{array}{l} P_1 \text{ invokes } \mathcal{O}(m\ell) \text{ threads for } MaxLCP^{H,k}_{(s_1, S_{buffer})} \text{ at } t_1^1 \\ P_1 \text{ invokes } \mathcal{O}(m\ell) \text{ threads for } MaxLCP^{H,k}_{(s_2, S_{buffer})} \text{ at } t_2^1 \\ \vdots \\ P_1 \text{ invokes } \mathcal{O}(m\ell) \text{ threads for } MaxLCP^{H,k}_{(s_{m/p}, S_{buffer})} \text{ at } t_{m/p}^1 \end{array} \right.$$

$$\vdots$$

$$\left\{ \begin{array}{l} P_p \text{ invokes } \mathcal{O}(m\ell) \text{ threads for } MaxLCP^{H,k}_{(s_{m-m/p+1}, S_{buffer})} \text{ at } t_1^p \\ P_p \text{ invokes } \mathcal{O}(m\ell) \text{ threads for } MaxLCP^{H,k}_{(s_{m-m/p+2}, S_{buffer})} \text{ at } t_2^p \\ \vdots \\ P_p \text{ invokes } \mathcal{O}(m\ell) \text{ threads for } MaxLCP^{H,k}_{(s_m, S_{buffer})} \text{ at } t_{m/p}^p \end{array} \right.$$

Each processor P_r invokes $\mathcal{O}(m\ell)$ GPU threads for each $s_i \in \mathbf{S}_r$ in order to compute $MaxLCP^{H,k}_{(s_i, S_{buffer})}$. These threads are created for a specific string $s_i \in \mathbf{S}_r$ and execute in parallel. However, the set of GPU threads created by P_r for the next string $s_{i+1} \in \mathbf{S}_r$ can only be created after the GPU threads created for the previous string s_i finish their execution. Let t_c^r denote the launch time of the c -th GPU task issued by processor P_r ($1 \leq r \leq p$ and $1 \leq c \leq m/p$); these times then satisfy

$$t_1^1 < t_2^1 < \cdots < t_{m/p}^1, \dots, t_1^p < t_2^p < \cdots < t_{m/p}^p.$$

Recall that in Section 4.1 the array $MaxLCP^{H,k}$ was computed pairwise for two strings s_i and s_j ($1 \leq j \leq m$). In the GPU version, we replace the second string by the entire S_{buffer} , processing each s_i against all strings in \mathbf{S} except s_i . This choice maximizes GPU throughput on a large, contiguous workload and avoids repeated host-device transfers.

In Algorithm 3, each GPU thread is determined by the string index j — where $j \in \{1..m\} \setminus \{i\}$ denotes the index of a string in S_{buffer} — and the starting position $start$ in s_i . Thread $(j, start)$ computes $MaxLCP^{H,k}_{(s_i, s_j)}[start]$. In lines 8-15, the kernel first compares substrings of length τ , allowing at most k mismatches. In lines 18-24,

if this initial match is valid, the comparison continues beyond τ , extending as far as possible while ensuring that the mismatch count does not exceed k , and the substring bounds are not crossed. The longest valid match found by the thread is tracked in variable *longest*. If the matched length is at least τ , the result is stored as a tuple $(j, start, longest)$. This tuple is identical to (i, p, l) tuple stored in the *LS* key-value data structure in the CPU implementation (See Section 4.1).

Each processor P_r calls Algorithm 3 $|\mathcal{S}_r|$ times, and each algorithm call creates $c(m-1)(\ell-\tau+1) = \mathcal{O}(m\ell)$ GPU threads⁴. In other words, in order to compute $MaxLCP_{(s_i, S_{buffer})}^{H,k}$, we spawn $\mathcal{O}(m\ell)$ threads for each string s_i ($1 \leq i \leq m$).

Over the entire computation, $\mathcal{O}(m^2\ell)$ GPU threads are created to obtain the *Rkt*-LCS. After computing all $MaxLCP^{H,k}$ arrays, we compute the candidate *Rkt*-LCS set $\{C_1, C_2, \dots, C_m\}$, as shown in Algorithm 2 (which begins by executing Algorithm 1).

Algorithm 3 compute_MaxLCP_k_Kernel (CUDA)

```

1: Input:  $s_i, S_{buffer}, k, \tau$ 
2: Create GPU thread for each  $(j, start)$  pair
3: for each GPU thread  $(j, start)$  do
4:    $longest \leftarrow 0$ 
5:   compute  $S_{offset}$  ▷ Starting position of string  $j$  in  $S_{buffer}$ 
6:   for  $q_1 = 0$  to  $\ell - 1$  do
7:      $mismatch\_count \leftarrow 0$ 
8:     for  $q_2 = 0$  to  $\tau - 1$  do
9:       if  $start + q_2 \geq \ell$  or  $q_1 + q_2 \geq \ell$  then
10:        break
11:       if  $s_i[start + q_2] \neq S_{buffer}[S_{offset} + q_1 + q_2]$  then
12:         if  $mismatch\_count = k$  then
13:           break
14:         if  $mismatch\_count < k$  then
15:            $mismatch\_count \leftarrow mismatch\_count + 1$ 
16:       if  $q_2 < \tau$  then
17:         continue
18:       while  $start + q_2 < \ell$  and  $q_1 + q_2 < \ell$  do
19:         if  $s_i[start + q_2] \neq S_{buffer}[S_{offset} + q_1 + q_2]$  then
20:           if  $mismatch\_count = k$  then
21:             break
22:           if  $mismatch\_count < k$  then
23:              $mismatch\_count \leftarrow mismatch\_count + 1$ 
24:          $q_2 \leftarrow q_2 + 1$ 
25:       if  $q_2 > longest$  then
26:          $longest \leftarrow q_2$ 
27:       if  $longest \geq \tau$  then
28:         store  $(j, start, longest)$ 

```

Table 2 shows the threads created for the computation of $MaxLCP_{(s_i, S_{buffer})}^{H,k}$ where $s_i = \text{ATTTTCG}$ and $S_{buffer} = \{\text{GTGGGA, AGGGGAT, AGCGGA}\}$.

Finally, we review a closely related implementation by Chockalingam *et al.* [6], discuss its potential applicability to the *Rkt*-LCS problem and explain why our parallel approaches are significantly more effective for solving the *Rkt*-LCS problem. Chockalingam *et al.* in [6], provide a $\mathcal{O}(\frac{N}{p} \log N + occ) \log^k N$ running time parallel

⁴ The constant parameter c is the thread-amplification factor: each task is fanned out into c parallel GPU threads that run together in a single group.

Thread	j	start	S_{buffer} string	s_i substring
0	0	0	GTGGA	ATTTCG
1	0	1	GTGGA	TTTCG
2	0	2	GTGGA	TTCG
3	0	3	GTGGA	TCG
4	1	0	AGGGGAT	ATTTCG
5	1	1	AGGGGAT	TTTCG
6	1	2	AGGGGAT	TTCG
7	1	3	AGGGGAT	TCG
8	2	0	AGCGGA	ATTTCG
9	2	1	AGCGGA	TTTCG
10	2	2	AGCGGA	TTCG
11	2	3	AGCGGA	TCG

Table 2: Thread assignment for `compute_MaxLCP_k_kernel` with $s_i = \text{ATTTCG}$ and $S_{buffer} = \{s_1, s_2, s_3\} = \{\text{GTGGA}, \text{AGGGGAT}, \text{AGCGGA}\}$ where $\tau = 3$. For example, Thread 2 computes $\text{MaxLCP}_{(s_i, s_1)}^{H,k}[3]$.

algorithm, where p is the number of processors and occ is the number of occurrences reported for the following problem:

Problem 3. (All-pair k -Mismatch Maximal Common Substrings). Given a collection $\mathcal{S} = \{s_1, s_2, \dots, s_m\}$ of m strings with total length N , a length threshold τ , and a mismatch threshold $k \geq 0$, report all k -mismatch maximal common substrings of length $\geq \tau$ between all pairs of strings in \mathcal{S} .

A pair of two equal-length substrings, $s_i[x..(x + \phi - 1)]$ and $s_j[y..(y + \phi - 1)]$, $1 \leq i, j \leq m$, form k -mismatch common substrings if the Hamming distance between them is at most k . Also, they are *maximal* if neither $s_i[(x - 1)..(x + \phi - 1)]$ and $s_j[(y - 1)..(y + \phi - 1)]$, nor $s_i[x..(x + \phi)]$ and $s_j[y..(y + \phi)]$, are a k -mismatch common substring pair. Coincidentally, the “longest” k -mismatch maximal common substring pair computed by this solution is an *Rkt*-LCS where $t = 2$. Unfortunately, using the approach and implementation of [6] to compute the *Rkt*-LCS is impractical for the following reasons:

1. For $k > 2$ and large N , their solution’s running time grows exponentially with k .
2. In addition, we need all k -mismatch common substrings (not only *maximal* ones) to compute *Rk*-LCS. Therefore, every substring of all maximal k -mismatch common substring needs to be evaluated (see Lemma 4).
3. The number of k -mismatch common substrings becomes extremely large even for a small number of relatively repetitive sequences over a small alphabet (see Lemma 5). On the other hand, $\text{LCP}_{(s_1, s_2)}^{H,k}$ needs $\mathcal{O}(\ell^2)$ space. Therefore, computing the k -mismatch common substrings is not a viable option for large strings.

Lemma 4. *Assume that substring u is an *Rkt*-LCS of the set \mathcal{S} , and substrings u'_1, \dots, u'_t are the corresponding k -mismatch occurrences of u in t strings of \mathcal{S} . Then at least one pair (u, u'_i) is a k -mismatch maximal common substring, $1 \leq i \leq t$.*

Proof. Assume, for the sake of contradiction, that all (u, u'_i) pairs are non-maximal. Then, each pair (u, u'_i) can be extended by at least one character to the left or right, yielding a longer *Rkt*-LCS. This contradicts the assumption that u is an *Rkt*-LCS. \square

Lemma 5. *The number of all k -mismatch common substrings between s_1 and s_2 , each of length ℓ , is $\mathcal{O}(\ell^3)$.*

Proof. Consider $s_1 = s_2 = a^\ell$, where $\ell > 0$. Let $\tau = 1$ and $k = 0$. Because every position $1..\ell$ in both strings contains the same symbol a , every pair of equal-length substrings is a 0-mismatch common substring and hence a k -mismatch common substring. For a fixed length ℓ' ($1 \leq \ell' \leq \ell$), both s_1 and s_2 have $\ell - \ell' + 1$ substrings of length ℓ' . Therefore, the number of ordered pairs of length- ℓ' substrings is $(\ell - \ell' + 1)^2$. Summing over all values of ℓ' , we get $\sum_{\ell'=1}^{\ell} (\ell - \ell' + 1)^2 = \frac{\ell(\ell+1)(2\ell+1)}{6} = \mathcal{O}(\ell^3)$. Hence, the total number of k -mismatch common substring pairs between two strings s_1 and s_2 is $\mathcal{O}(\ell^3)$. \square

In contrast, the practical time complexity of our implementations is independent of k and, while quadratic in N , scales robustly even for very large inputs with available hardware resources.

5 Experimental Evaluation

In this section, we evaluate the relative speedup of proposed implementations. Our experiments were conducted on a server equipped with two 4.1 GHz 16-core Intel Xeon Gold 6426Y processors, with 250 GB of main memory running on the REHL 9 operating system. The server also features four NVIDIA H100 GPUs, each with 80 GB of memory. The dataset consists of two files, each containing 1,077,820 nucleotide sequences ($\Sigma = \{A, T, C, G\}$) of uniform length 51, formatted in FASTQ. Sequences are taken from soil samples from unidentified taxa⁵ from an unpublished study. Our implementation is available online⁶. This repository is written using OpenMPI⁷ 5.0.6 and CUDA⁸ 12.8 libraries.

Table 3 shows the CPU and GPU runtime comparison for 5000 sequences; that is, for $m = 5000$. GPU implementation shows a 179 \times runtime improvement over CPU implementation for the setting $(p, k, t, \tau) = (4, 10, 100, 30)$.

In Table 3a and Table 3b, CPU implementation shows the expected behavior of improved runtime with more processors. CPU processors operate independently without competing for shared resources. Additionally, there is no memory transfer overhead, unlike in the GPU implementation. Finally, the workload is purely computational and scales well across multiple cores. This implementation uses a queue-based approach to compute the $MaxLCP^{H,k}$ table [9]. As k increases, the runtime of the CPU implementation also increases due to several reasons. One key factor is the overhead of queue operations. Each mismatch during sequence comparison results in an enqueue operation. Since the queue size scales with k , larger values of k lead to more frequent queue operations and increased memory handling costs. Another reason is the extended comparison length permitted by higher k values. While the theoretical time complexity might not directly depend on k , practical runtime is affected by more frequent queue operations, longer comparison sequences, and heavier memory

⁵ A taxonomic group of any rank, such as a species, family, or class.

⁶ <https://github.com/neerjamhaskar/Rkt-LCS>

⁷ OpenMPI is an open-source implementation of the MPI standard that enables parallel computing by coordinating communication between processes across distributed or shared memory systems [10].

⁸ CUDA is NVIDIA's parallel-computing platform for general-purpose GPU programming [8].

Table 3: Runtime (in seconds) and Relative SpeedUp (RSU — relative to $Cores = 4$) comparison for $m = 5000$

(a) Parallel CPU, $t = 1000$, $\tau = 15$							(b) Parallel CPU, $t = 100$, $\tau = 30$						
Cores	$k = 1$		$k = 3$		$k = 10$		Cores	$k = 1$		$k = 3$		$k = 10$	
	Time	RSU	Time	RSU	Time	RSU		Time	RSU	Time	RSU	Time	RSU
4	138	1.00×	242	1.00×	705	1.00×	4	82	1.00×	146	1.00×	358	1.00×
8	71	1.94×	122	1.98×	352	2.00×	8	44	1.86×	75	1.94×	182	1.96×
16	40	3.45×	63	3.84×	181	3.89×	16	30	2.73×	39	3.74×	94	3.80×
32	19	7.26×	42	5.76×	112	6.29×	32	17	4.94×	26	5.61×	66	5.42×

(c) Parallel GPU, $t = 1000$, $\tau = 15$							(d) Parallel GPU, $t = 100$, $\tau = 30$						
Cores	$k = 1$		$k = 3$		$k = 10$		Cores	$k = 1$		$k = 3$		$k = 10$	
	Time	RSU	Time	RSU	Time	RSU		Time	RSU	Time	RSU	Time	RSU
4	4	1.00×	3	1.00×	72	1.00×	4	2	1.00×	2	1.00×	2	1.00×
8	5	0.80×	4	0.75×	37	1.94×	8	3	0.66×	3	0.66×	2	1.00×
16	6	0.66×	6	0.50×	22	3.27×	16	4	0.50×	5	0.40×	5	0.40×
32	12	0.33×	11	0.27×	21	3.42×	32	9	0.22×	9	0.22×	9	0.22×

management demands. These effects together reduce cache efficiency and increase the number of memory accesses.

As shown in Table 3c (except for $k = 10$) and Table 3d, the observed increase in runtime with more processors in the GPU implementation can be attributed to several key factors. Because each processor P_r launches its GPU tasks for each string in \mathcal{S}_r sequentially, at most $\mathcal{O}(pml)$ GPU threads are executed concurrently. Increasing the number of processors (p) increases GPU utilization but also increases the cost of coordination. Therefore, increasing p involves a trade-off between better GPU use and higher *work distribution overhead*. For a relatively small number of sequences and a large number of processors, work distribution overhead arises due to the use of OpenMPI for distributing work across processes. As the number of processes increases, each process handles fewer sequences, but the OpenMPI communication and coordination overhead grows proportionally. In addition, *memory transfer overhead* increases with more processes. Since each processor must transfer its data to the GPU, it results in greater cumulative overhead. However, as shown in Figure 2, increasing the number of processors for a larger number of sequences leads to lower runtime. In Table 3c, for $k = 10$, the runtime decreases as the number of processors increases, reflecting that larger k values incur greater computational work and thus benefit more from parallelization.

As shown in Figure 1, the GPU-accelerated implementation exhibits nearly constant runtime (for a relatively small number of sequences, perhaps 5k to 20k), while the CPU implementation shows a quadratic increase in runtime. As illustrated in Figure 2, by doubling the number of sequences, the quadratic runtime increase becomes more pronounced, and especially in a higher number of sequences (75k to 500k).

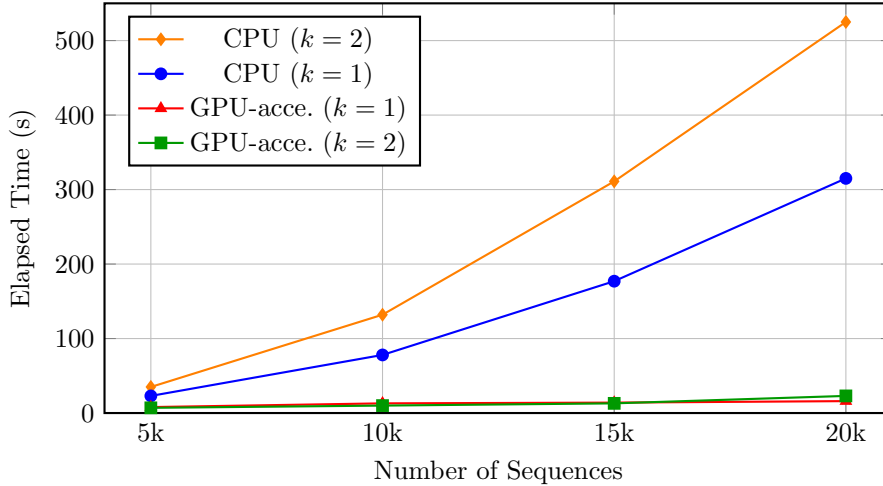


Figure 1: Runtime comparison for CPU and GPU-accelerated implementations with varying k on different sequence set sizes, $t = 1000$, $\tau = 15$, and $p = 32$

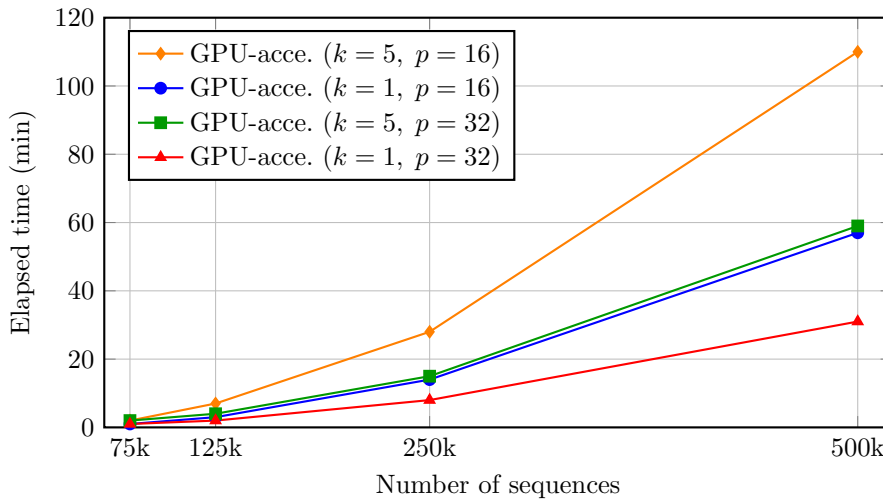


Figure 2: GPU-accelerated implementation runtime (in whole minutes) for different (k, p) settings, $t = 1000$, and $\tau = 15$

6 Conclusion

We introduced two implementations of the Rkt -LCS algorithm under Hamming distance: a CPU version with expected running time $\mathcal{O}(N^2/p)$ and a highly parallel GPU version, where both runtimes are independent of the mismatch parameter k . In our evaluations, we observed that off-loading the core computation of the $MaxLCP^{H,k}$ tables to GPU threads yields up to $179\times$ speed-up over the CPU implementation. This improvement is expected to be higher for larger number of sequences.

As future work, the input sequences could first be clustered — e.g., with machine-learning techniques — and then losslessly compressed before computing the Rkt -LCS. Conversely, the Rkt -LCS itself can help detect similar fragments, which may then be compressed by differential similarity encoding. Another promising direction is to adapt the GPU version to incorporate the $LCP^{H,k}$ computation scheme of Flouri *et al.* [9], combined with FFT (Fast Fourier Transform) computation techniques.

Acknowledgments

The authors wish to thank Shutong Wu for useful discussions of parallel CPU implementation. The second and third authors are supported by Grant Nos. RGPIN-2024-06915 and RGPIN-2024-05921, respectively, of the Natural Sciences & Engineering Research Council of Canada (NSERC).

References

1. A. ABBOUD, R. R. WILLIAMS, AND H. YU: *More applications of the polynomial method to algorithm design*, in Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, SIAM, 2015, pp. 218–230.
2. M. ARNOLD AND E. OHLEBUSCH: *Linear time algorithms for generalizations of the longest common substring problem*. Algorithmica, 60(4) 2011, pp. 806–818.
3. M. A. BABENKO AND T. STARIKOVSKAYA: *Computing the longest common substring with one mismatch*. Probl. Inf. Transm., 47(1) 2011, pp. 28–33.
4. P. CHARALAMPOPOULOS, M. CROCHEMORE, C. S. ILIOPOULOS, T. KOCIUMAKA, S. P. PISSIS, J. RADOSZEWSKI, W. RYTTER, AND T. WALLEN: *Linear-time algorithm for long LCF with k mismatches*, in Annual Symposium on Combinatorial Pattern Matching, CPM 2018, vol. 105 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 23:1–23:16.
5. P. CHARALAMPOPOULOS, T. KOCIUMAKA, S. P. PISSIS, AND J. RADOSZEWSKI: *Faster algorithms for longest common substring*, in 29th Annual European Symposium on Algorithms, ESA 2021, vol. 204 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 30:1–30:17.
6. S. P. CHOCKALINGAM, S. V. THANKACHAN, AND S. ALURU: *Sequential and parallel algorithms for all-pair k -mismatch maximal common substrings*. J. Parallel Distributed Comput., 144 2020, pp. 68–79.
7. M. FARACH: *Optimal suffix tree construction with large alphabets*, in 38th Annual Symposium on Foundations of Computer Science, FOCS '97, IEEE Computer Society, 1997, pp. 137–143.
8. R. FARBER: *CUDA Application Design and Development*, Elsevier, 2011.
9. T. FLOURI, E. GIAQUINTA, K. KOBERT, AND E. UKKONEN: *Longest common substrings with k mismatches*. Inf. Process. Lett., 115(6-8) 2015, pp. 643–647.
10. E. GABRIEL, G. E. FAGG, G. BOSILCA, T. ANGSKUN, J. J. DONGARRA, J. M. SQUYRES, V. SAHAY, P. KAMBADUR, B. BARRETT, A. LUMSDAINE, R. H. CASTAIN, D. J. DANIEL, R. L. GRAHAM, AND T. S. WOODALL: *Open MPI: goals, concept, and design of a next generation MPI implementation*, in Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, vol. 3241 of Lecture Notes in Computer Science, Springer, 2004, pp. 97–104.
11. S. GRABOWSKI: *A note on the longest common substring with k -mismatches problem*. Inf. Process. Lett., 115(6-8) 2015, pp. 640–642.
12. H. HASIBI, N. MHASKAR, AND W. F. SMYTH: *On the complexity of finding approximate LCS of multiple strings*, 2025, <https://arxiv.org/abs/2505.15992>.
13. C. JIN AND J. NOGLER: *Quantum speed-ups for string synchronizing sets, longest common substring, and k -mismatch matching*. ACM Trans. Algorithms, 20(4) 2024, pp. 32:1–32:36.
14. T. KOCIUMAKA, J. RADOSZEWSKI, AND T. STARIKOVSKAYA: *Longest common substring with approximately k mismatches*. Algorithmica, 81(6) 2019, pp. 2633–2652.
15. I. LEE AND Y. J. PINZÓN: *A simple algorithm for finding exact common repeats*. IEICE Trans. Inf. Syst., 90-D(12) 2007, pp. 2096–2099.
16. C. LEIMEISTER AND B. MORGENSTERN: *kmacs: the k -mismatch average common substring approach to alignment-free sequence comparison*. Bioinform., 30(14) 2014, pp. 2000–2008.
17. S. V. THANKACHAN, C. ALURU, S. P. CHOCKALINGAM, AND S. ALURU: *Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis*, in Research in Computational Molecular Biology - 22nd Annual International Conference, vol. 10812 of Lecture Notes in Computer Science, Springer, 2018, pp. 211–224.
18. S. V. THANKACHAN, A. APOSTOLICO, AND S. ALURU: *A provably efficient algorithm for the k -mismatch average common substring problem*. J. Comput. Biol., 23(6) 2016, pp. 472–482.
19. P. WEINER: *Linear pattern matching algorithms*, in 14th Annual Symposium on Switching and Automata Theory, IEEE Computer Society, 1973, pp. 1–11.