A Scalable Approximation Algorithm for Weighted Longest Common Subsequence

Jeremy Buhler[1], Thomas Lavastida[2], Kefu Lu[3(✉)], and Benjamin Moseley[2]

[1] Washington University in St. Louis, St. Louis, MO, USA
jbuhler@wustl.edu
[2] Carnegie Mellon University, Pittsburgh, PA, USA
{tlavasti,moseleyb}@andrew.cmu.edu
[3] Washington and Lee University, Lexington, VA, USA
klu@wlu.edu

Abstract. This work introduces novel parallel methods for weighted longest common subsequence (WLCS) and its generalization, all-substrings WLCS. Previous work developed efficient algorithms for these problems via Monge matrix multiplication, which is a limiting factor for further improvement. Diverging from these approaches, we relax the algorithm's optimality guarantee in a controlled way, using a different, natural dynamic program which can be sketched and solved in a divide-and-conquer manner that is efficient to parallelize.

Additionally, to compute the base case of our algorithm, we develop a novel and efficient method for all-substrings WLCS inspired by previous work on unweighted all-substrings LCS, exploiting the typically small range of weights.

Our method fits in most parallel models of computation, including the PRAM and the BSP model. To the best of our knowledge this is the fastest $(1-\epsilon)$-approximation algorithm for all-substrings WLCS and WLCS in BSP. Further, this is the asymptotically fastest parallel algorithm for weighted LCS as the number of processors increases.

Keywords: Parallel approximation algorithms · Weighted LCS

1 Introduction

Technologies for sequencing DNA have improved dramatically in cost and speed over the past two decades [15], resulting in an explosion of sequence data that presents new opportunities for analysis. To exploit these new data sets, we must devise scalable algorithms for analyzing them. A fundamental task in analyzing DNA is comparing two sequences to determine their similarity.

A basic similarity measure is weighted longest common subsequence (WLCS). Given two strings x and y over a finite alphabet Σ (e.g. {A,C,G,T}), a *correspondence* between them is a set of index pairs $(i_1, j_1) \ldots (i_\ell, j_\ell)$ in x and y such that

B. Moseley, K. Lu and T. Lavastida were supported in part by a Google Research Award and NSF Grants CCF-1617724, CCF-1733873, and CCF-1725661.

L. Sousa et al. (Eds.): Euro-Par 2021, LNCS 12820, pp. 368–384, 2021.
https://doi.org/10.1007/978-3-030-85665-6_23

for all $k < \ell$, $i_k < i_{k+1}$ and $j_k < j_{k+1}$. A correspondence need not use all symbols of either string. We are given a non-negative scoring function $f : \Sigma \times \Sigma \to \mathbb{N}$ on pairs of symbols, and the goal is to find a correspondence (the WLCS) that maximizes the total weight $\sum_{k=1}^{\ell} f(x[i_k], y[j_k])$. We assume, consistent with actual bioinformatics practice [23], that the maximum weight σ returned by f for any pair of symbols is a small constant [10], so that the maximum possible weight for a correspondence between sequences is proportional to their length. WLCS is a special case of the weighted edit distance problem [14] in which match and mismatch costs are non-negative and insertion/deletion costs are zero. This problem is sufficient to model similarity scoring with a match bonus and mismatch and gap penalties, provided we can subsequently normalize alignment weights by the lengths of the two sequences [24]. If f scores +1 for matching symbol pairs and 0 for all others, the problem reduces to unweighted LCS.

A generalization of WLCS is the *all-substrings WLCS* or *AWLCS* problem. In this variant, the goal is to compute a matrix H such that $H[i, j]$ is the weight of a WLCS between the entire string x and substring $y[i..j]$. This "spectrum" of weights can be used to infer structure in strings, such as approximate tandem repeats and circular alignments [21]. Of course, H includes the weight of a WLCS between the full strings x and y as an entry.

Throughout the paper, we let $|x| = n$, $|y| = m$ and assume that $n \geq m$.

Sequential Methods. The WLCS problem, like the unweighted version, can be solved by dynamic programming in time $O(nm)$. In particular, the well-known Needleman-Wunsch algorithm [14] for weighted edit distance, which is the basis for many practical biosequence comparison tools [18–20,22], solves the WLCS problem as a special case. Sub-quadratic time algorithms are also known for the WLCS problem based on the "Four Russians" technique [13], which works for integer weights. In addition, there is the work of Crochemore et al. that works for unrestricted weights and also achieves sub-quadratic time [8]. At the same time, the sequential complexity of the LCS and WLCS problem is well understood - results in fine-grained complexity give strong lower bounds assuming the Strong Exponential Time Hypothesis [1,6,7].

Schmidt [21] showed that AWLCS, which naively requires much more computation than WLCS, can be solved in time $O(nm \log m)$ as a special case of all-substrings weighted edit distance. Alves et al. reduced this cost to $O(nm)$ for the special case of unweighted all-substrings LCS (ALCS) [4].

Parallel Methods. One way to solve large WLCS problems more efficiently is to parallelize their solution. Krusche and Tiskin [12] study parallelization of standard dynamic programming algorithms for LCS. However, the straightforward dynamic programming approaches for LCS and WLCS do not easily parallelize because they contain irreducible chains of dependent computations of length $\Theta(n+m)$. The fastest known parallel algorithms for these problems instead take a divide-and-conquer approach (such as [5]), combining the all-substrings generalization of LCS with methods based on max-plus matrix multiplication as we will describe.

Let x_1 and x_2 be two strings, and let H_1 and H_2 be AWLCS matrices on string pairs (x_1, y) and (x_2, y), respectively. Defining matrix multiplication over the ring $(\max, +)$, $H_1 \times H_2$ is the AWLCS matrix for strings $x_1 \cdot x_2$ and y [25]. Hence, we can compute the AWLCS matrix for the pair (x, y) on p processors by subdividing x into p pieces x_k, recursively computing matrices H_k for each x_k with y, and finally multiplying the H_k together. Given a base-case algorithm to compute AWLCS in time $B(m, n)$ and an algorithm to multiply two $m \times m$ AWLCS matrices in time $A(m)$, this approach will run in time $B(m, \frac{n}{p}) + A(m) \log p$.

Fast multiplication algorithms exist that exploit the *Monge property* of AWLCS matrices: for all $1 \leq i < k \leq m$ and $1 \leq j < \ell \leq n$, $H[i, j] + H[k, \ell] \leq H[i, \ell] + H[k, j]$. Tiskin [25] showed that for the special case of unweighted ALCS, $A(m) = O(m \log m)$, yielding an overall time of $O\left(\frac{mn}{p} + m \log m \log p\right)$. Leveraging related strategies yields other fast BSP algorithms for unweighted ALCS with improved per-processor memory and communication costs [3,11].

For AWLCS, $A(m) = O(m^2)$ using an iterated version of the SMAWK algorithm [2,17]. No faster multiplication algorithm is known for the general case. Practically subquadratic multiplication has been demonstrated for specific scoring functions f [17], but the performance of these approaches depend on f in a difficult-to-quantify manner. In [16] a complex divide-and-conquer strategy was used to achieve an optimal running time for the pairwise sequence alignment problem, which is similar but more general than our problem. In our work we use an alternative divide-and-conquer strategy to obtain a fast parallel algorithm.

Results. This paper introduces a new approach to parallelizing AWLCS and therefore WLCS. We introduce algorithms that are $(1 - \epsilon)$ approximate. Our algorithm's running time improves upon the best BSP algorithms for the problems and scales to $o(m^2)$ in the PRAM setting as the number of processors increases.

The new algorithm is our main contribution. Our algorithm sketches a sequential dynamic program and uses a divide-and-conquer strategy which can be parallelized. This sketch comes with a cost of approximating the objective to within a $1 - \epsilon$ factor for any parameter $\epsilon \in (0, 1)$. By relaxing the algorithm's optimality guarantee, we are able to obtain subquadratic-time subproblem composition by building on recent results on parallelizing dynamic programs for other problems [9]. Additionally, we develop and utilize a new base case algorithm for AWLCS that takes advantage of the small range of weights typically used [10]. The following theorem summarizes our main result.

Theorem 1. *Let W be the largest possible correspondence weight and let p be the number of processors. For any $\epsilon \in (0, 1)$, there is a BSP algorithm running in time $O(B(m, \frac{n}{p}) + m\frac{\log^2(W)\log^2(n)\log(p)}{\epsilon^2})$ and using $O(\frac{n}{p} + m\frac{\log^2(W)\log^2(n)}{\epsilon^2})$ local memory per processor that computes a $(1 - \epsilon)$-approximate solution to the WLCS problem.*

In the BSP model with p processors and using Schmidt's algorithm ($B(m,n) = mn \log m$) for the base case, we obtain a parallel algorithm with running time $O\left(\frac{mn \log m}{p} + m\frac{\log^2 W \log^2 n}{\epsilon^2} \log p\right)$, where W is the largest possible correspondence weight between strings(which, by our assumption of bounded weights, is $O(\min(n,m))$). As mentioned, this is the first parallel algorithm for weighted LCS for which the running time scales as $o(m^2)$, and also the fastest $(1-\epsilon)$-approximation algorithm for weighted LCS in BSP. In contrast, previous methods' running times have a $\Theta(m^2)$ term that does not diminish as the number of processors p increases. Our method uses $O\left(\frac{n}{p} + m\frac{\log^2 \sigma m \log^2 n}{\epsilon^2}\right)$ local memory per processor, where σ is the highest weight produced by the scoring function.

Using Schmidt's algorithm for the base case dominates the running time. We would like to improve the $O(mn \log m)$ running time of the base case to get as close to $O(mn)$ as possible. We develop an interesting alternative base case algorithm by extending Alves's $O(mn)$ algorithm for ALCS [4] to the weighted case of AWLCS. This is our second major contribution. This algorithm, like our overall divide-and-conquer strategy, exploits the small range of weights typically used by scoring functions for DNA comparison.

Theorem 2. *Let σ be the highest weight produced by the scoring function f. There is a sequential algorithm running in time $O(\sigma nm)$ time for computing an implicit representation of the AWLCS matrix using space $O(\sigma m)$.*

Using this algorithm as the base case in Theorem 1, we achieve an overall running time of $O(\frac{\sigma mn}{p} + m \log^2(\sigma m) \log^2(n) \log(p)/\epsilon^2)$.

Algorithmic Techniques. The algorithms developed in this paper leverage two main techniques. The first is *parallelizing a natural dynamic program* for a problem via sketching. Let $C(i,j)$ be the weight of a WLCS between $x[1:n]$ and $y[i:j]$; we want to compute this quantity for all $1 \le i < j \le m$. One may add a third index to specify $C_k(i,j)$, the weight of a WLCS between $x[1:k]$ and $y[i:j]$. C_k can be computed via the following recurrence: $C_k(i,j) = \max\{C_{k-1}(i,j), C_k(i,j-1), C_{k-1}(i,j-1) + f(x[k], y[j])\}$. But this recurrence is both inefficient, requiring time $O(nm^2)$, and difficult to parallelize, with dependent computation chains of size $\Omega(n+m)$.

To improve efficiency, we abandon direct computation of $C(i,j)$ and instead compute some $D(i,w)$ which is subsequently be used to derive the entries of $C(i,j)$. $D(i,w)$ is the least index j s.t. there exists a correspondence of weight at least w between $y[i:j]$ and $x[1:n]$. We compute and store $D(i,w)$ only for values w that are powers of $1+\epsilon'$ for some fixed $\epsilon' > 0$. This sketched version of D effectively represents the $O(m^2)$ sized matrix C using $O(m \log_{1+\epsilon'} m\sigma)$ entries. Although our sketching strategy is not guaranteed to find the optimal values $C(i,j)$, we show that it exhibits bounded error as a function of ϵ'.

A straightforward computation of $D(i,w)$ entails long chains of serial dependencies. Thus, we use a divide-and-conquer approach instead. Let $D_{r_1,r_2}(i,w)$

store the the minimum index j s.t. a correspondence of weight at least w exists between $y[i:j]$ and $x[r_1:r_2]$. We will show how to compute $D_{r_1,r_3}(i,w)$ given $D_{r_1,r_2}(i,w')$ and $D_{r_2+1,r_3}(i,w')$ for values $w' \leq w$. If we compute D matrices for non-overlapping substrings of x in parallel and double the range of x covered by each D matrix at each step, we can compute $D_{1,n}(i,w)$ in a logarithmic number of steps.

In realistic applications, we seek to compare sequences with millions of DNA bases; the number of available processors is small in comparison, that is, $p \ll \min(n,m)$. Speedup is therefore limited by the base-case work on each processor, which must sequentially solve an AWLCS problem of size roughly $m \times \frac{n}{p}$. Solving these problems using Schmidt's algorithm, which is insensitive to the magnitude of weights, takes time $O(\frac{n}{p} m \log m)$. However, Schmidt's algorithm involves building complex binary trees which proved to have high overhead in practice. Our second main technique is developing a weight-sensitive AWLCS algorithm utilizing an efficient and compact implicit representation.

We show that if the scoring function f assigns weights at most σ to symbol pairs, the matrix $C(i,j)$ can be represented implicitly using only $O(m\sigma)$ storage rather than $O(m^2)$. Moreover, we can compute this representation in sequential time $O(\frac{n}{p} m\sigma)$. The algorithm computes and stores values of the form $h_s(j)$, which is the least index i such that $C(i,j) \geq C(i,j-1)+s$, for $1 \leq s \leq \sigma$. These h values indicate where there is an increase of s in the optimal correspondence weight when the index j increases. The key to the technique is showing that these values contain information for reconstructing C and how to compute them efficiently without complex auxiliary data structures.

Roadmap. Section 3.1 presents the main dynamic program which can be parallelized via a divide-and-conquer strategy, while Sect. 3.2 shows how to use sketching to make this step time and space efficient while retaining $(1-\epsilon)$-approximate solutions. Section 3.3 presents our new algorithm for AWLCS which we use as an efficient local base case algorithm on each processor. Finally, Sect. 4 completes our analysis.

2 Preliminaries

We denote by $x[i:j]$ the contiguous substring of x that starts at index i and ends at index j. The goal of AWLCS is to find correspondences of maximum weight between $x[1:n]$ and $y[i:j]$ for all $1 \leq i \leq j \leq m$. We develop a method to obtain the *weights* of the desired correspondences; the alignments can be recovered later by augmenting the recurrence to permit traceback of an optimal solution. However, for AWLCS, the weights alone suffice for many applications [21]. Finally, we denote by W the highest possible weight of a WLCS between x and y, which we assume to be $O(\sigma \min(n,m))$. Here $\sigma = \max_{c,c' \in \Sigma} f(c,c')$ is the maximum possible weight of matching two characters. We note that in practice, σ is a constant and typically less than 20.

We now define two key matrices utilized in the design of our algorithms. $C(i,j)$ will denote the maximum weight of a correspondence between x and

$y[i:j]$. The AWLCS problem seeks to compute $C(i,j)$ for all $1 \leq i < j \leq m$. An alternative way to view these weights is via the matrix D, where we swap the entry stored in the matrix with one of the indices. Let $D(i,w) = \min\{j \mid C(i,j) \geq w\}$. If no such j exists, we define $D(i,w) = \infty$. D stores essentially the same information as C; a single entry of $C(i,j)$ can be queried via the matrix D in time $O(\log W)$ by performing a binary search over possible values of w. However, the matrix D will be a substantially more compact representation than C once we introduce our sketching strategy.

3 All-Substrings Weighted Longest Common Subsequence

Here we present our algorithm for AWLCS. Following the divide-and-conquer strategy of prior work, we initially divide the string x equally among the processors, each of which performs some local computation using a base-case algorithm to solve AWLCS between y and its portion of x, yielding a solution in the form of the D matrix defined above. We then combine pairs of subproblem solutions iteratively to arrive at a global solution. We first describe the algorithm's divide and combine steps while treating the base case as a black box, then discuss the base-case algorithm.

3.1 Divide-and-Conquer Strategy

Let D_{r_1,r_2} be the D matrix resulting from the AWLCS computation between strings $x[r_1:r_2]$ and y. Our goal is to compute $D_{1,n}$, which encompasses all of x and y.

Our algorithm first divides x into p substrings of length $\frac{n}{p}$, each of which is given to one processor along with the entire string y. We assume that consecutive substrings of x are given to consecutive processors in some global linear processor ordering. If a processor is given a substring $x[r_1:r_2]$, it computes the subproblem solution D_{r_1,r_2} using our new local, sequential base-case algorithm described in Sect. 3.3. It then remains to combine the p subproblem solutions to recover the desired solution $D_{1,n}$. We compute $D_{1,n}$ in $O(\log p)$ rounds. In the j'th round, the algorithm computes $O(2^{\log(p)/j})$ subproblem solutions, where each solution combines two sub-solutions from adjacent sets of 2^{j-1} consecutive processors.

Let D_{r_1,r_2} and D_{r_2+1,r_3} be adjacent sub-solutions obtained from previous iterations. We combine these solutions to obtain D_{r_1,r_3}. To compute $D_{r_1,r_3}(i,w)$, we consider all possible pairs w_1, w_2 for which $w = w_1 + w_2$. For each possible w_1, we use the solution of the first subproblem to find the least index j' for which there exists a correspondence of weight w_1 between $x[r_1:r_2]$ and $y[i:j']$. We then use the solution of the second subproblem to find the least j such that a correspondence of weight $w_2 = w - w_1$ exists between $x[r_2+1:r_3]$ and $y[j'+1:j]$. (Clearly, $j \geq j'$.) These two correspondences use non-overlapping substrings of x and y and can be combined feasibly. The exact procedure can be found in Algorithm 1.

Algorithm 1. Combining Subproblems

```
procedure COMBINE(D_{r1,r2}, D_{r2+1,r3})
    for i = 1 to m do
        for w = 0 to W do
            D_{r1,r3}(i, w) ← ∞
            for w1 = 0 to w do
                w2 ← w − w1
                j' ← D_{r1,r2}(i, w1)
                j ← D_{r2+1,r3}(j' + 1, w2)
                D_{r1,r3}(i, w) = min(D_{r1,r3}(i, w), j)
```

3.2 Approximation via Sketching

Algorithm 1 solves the AWLCS problem exactly; the cost to combine two subproblems is $O(mW^2)$. For unweighted ALCS, $W = m$; the combine step is $O(m^3)$. To overcome this cost, we *sketch* the values of w. Sketching reduces the number of distinct weights considered from W to $O(\log W)$ and hence reduces the cost to combine two subproblems from $O(mW^2)$ to $O(m \log^2 W)$. We analyze its precise impact on solution quality and overall running time in Sect. 4.

Our sketching strategy fixes a constant $\epsilon > 0$ and sets $\beta = 1 + \frac{\epsilon}{\log n}$. Define $D^*(i, s)$ to be the least j such that there exists a correspondence between x and $y[i : j]$ with weight $w \geq \lfloor \beta^s \rfloor$. Define $D^*_{r_1,r_2}$ analogously to D_{r_1,r_2} for substrings of x. To compute $D^*_{r_1,r_3}$ from $D^*_{r_1,r_2}$ and $D^*_{r_2+1,r_3}$, we modify the algorithm described above as follows. For each power s s.t. $\lfloor \beta^s \rfloor \leq W$, we consider each power $s_1 \leq s$ and compute the least s_2 such that $\lfloor \beta^{s_1} \rfloor + \lfloor \beta^{s_2} \rfloor \geq \lfloor \beta^s \rfloor$. Let $j' = D^*_{r_1,r_2}(i, s_1)$ and $j = D^*_{r_2+1,r_3}(j' + 1, s_2)$. Then there exist non-overlapping correspondences with weights at least β^{s_1} and β^{s_2}, and hence a combined correspondence of weight at least β^s, between $x[r_1 : r_3]$ and $y[i : j]$. We take $D^*_{r_1,r_3}(i, s)$ to be the least j' that results from this procedure. In Sect. 4, we formally show that this sketching strategy preserves $(1 - \epsilon)$-approximate solutions and analyze the runtime and space usage of our algorithm.

3.3 Base Case Local Algorithm

We now describe a sequential algorithm, inspired by the work of [3], to obtain the initial matrices $D_{r_1,r_2}(i, w)$ for each individual processor.

In theory, one could continue the divide and conquer approach on each local machine until the entry to compute is of the form $D_{r_1,r_1+1}(i, w)$, yielding a simple base case to solve. However, this procedure proves computationally inefficient with a fixed number p of processors. Instead, we propose a different base case algorithm for computing $D_{r_1,r_2}(i, w)$ which better fits our setting.

For this section, we will drop the indices r_1 and r_2 and create an algorithm for computing D for strings x and y. Each processor applies this same algorithm, but to different substrings $x[r_1 : r_2]$.

The algorithm works in two steps. First, we calculate two sequences of indices, referred to as the h- and v-indices. Then, we use these indices to compute $D(i, w)$ for all desired i, w. Intuitively, these indices give compact information about the structure of the C matrix (and hence the D matrix), specifically the magnitude of change between weights in adjacent rows and columns of C.

Definition of the Indices. Recall the definition of the AWLCS matrix C, and let C^ℓ be the C matrix corresponding to the strings $x[1:\ell]$ and y. Before proceeding with the definition of the h- and v-indices, we note a lemma concerning the *Monge* properties of C^ℓ. These properties are well-known; see, e.g., [3].

Lemma 1. *For any triple of indices i, j, ℓ, $C^\ell(i-1, j-1) + C^\ell(i, j) \geq C^\ell(i-1, j) + C^\ell(i, j-1)$, and $C^\ell(i-1, j) + C^{\ell-1}(i, j) \geq C^{\ell-1}(i-1, j) + C^\ell(i, j)$.*

The following corollaries result from rearranging terms in the previous lemma.

Corollary 1. *For any i, j, ℓ, $C^\ell(i, j) - C^\ell(i, j-1) \geq C^\ell(i-1, j) - C^\ell(i-1, j-1)$.*

Corollary 2. *For any i, j, ℓ, $C^\ell(i, j) - C^{\ell-1}(i, j) \leq C^\ell(i-1, j) - C^{\ell-1}(i-1, j)$.*

We now consider the implications of Corollary 1. Fix i, j and ℓ with $C^\ell(i, j) - C^\ell(i, j-1) = s$ for some s. This s is the difference in WLCS weight if the second string is allowed one extra character at its end ($y[j]$), since it is comparing $x[1:\ell]$ with either $y[i:j]$ or $y[i:j-1]$. The corollary states that this difference is only greater for a substring of y that starts at $i' > i$ instead of i. Therefore, for each pair of fixed j, ℓ, there exists some minimal i such that $C^\ell(i, j) - C^\ell(i, j-1)$ is *first* greater than s, as it will be true for all $i' > i$. For different values of s, there are possibly different corresponding i which are minimal. Similar implications can be derived from Corollary 2.

Using this insight, we can define the h-indices and v-indices. These values $h_1, \ldots, h_\sigma$ and $v_1, \ldots, v_\sigma$ are the key to our improved base case algorithm. For $s \in [\sigma]$, $h_s(\ell, j)$ is the smallest index i such that $C^\ell(i, j) \geq C^\ell(i, j-1) + s$. That is, each $h_s(\ell, j)$ for a fixed ℓ and j marks the row of C^ℓ where we start to get a horizontal increment of s between columns $j-1$ and j. The v-indices are slightly different; $v_s(\ell, j)$ is the smallest index i such that $C^\ell(i, j) < C^{\ell-1}(i, j) + s$. The v-indices mark the row where we stop getting a vertical increment of s in column j between $C^{\ell-1}$ and C^ℓ. The entire matrix C^ℓ can be computed recursively as a function of the indices as follows:

$$C^\ell(i,j) = \begin{cases} C^\ell(i, j-1) & i < h_1(\ell, j) \\ C^\ell(i, j-1) + s & h_s(\ell, j) \leq i < h_{s+1}(l, j) \\ C^\ell(i, j-1) + \sigma & h_\sigma(\ell, j) \leq i \end{cases} \tag{1}$$

$$C^\ell(i,j) = \begin{cases} C^{\ell-1}(i, j) + \sigma & i < v_\sigma(\ell, j) \\ C^{\ell-1}(i, j) + s & v_{s+1}(\ell, j) \leq i < v_s(l, j) \\ C^{\ell-1}(i, j) & v_1(\ell, j) \leq i \end{cases} \tag{2}$$

The h and v-indices provide an efficient way to compute the entries in $D(i, w)$. If we can compute $C^\ell(i, j)$ for all i, j, then $D(i, w)$ is the smallest j for which $C^\ell(i, j) \geq w$. The indices actually correspond to a recursive definition of the values of $C^\ell(i, j)$.

The following intuition may help to interpret the h-indices. consider $h_1(\ell, j)$ for a fixed ℓ and j. This is the smallest value of i for which $C^\ell(i, j)$ exceeds $C^\ell(i, j-1)$ by at least 1. Suppose we compare the best WLCS of x and $y[i : j-1]$ against that of x and $y[i : j]$. There is a gain of one character (the last one) in the second pair of strings, so the second WLCS might have more weight. There is a unique value $h_1(\ell, j)$ of i for which the difference in weight first becomes ≥ 1. The uniqueness of this value can be inferred from Corollary 1.

The increment in the WLCS weight due to adding $y[j]$ may become greater as i increases, i.e., as we allow fewer opportunities to match x to earlier characters in y. However, the increment cannot exceed σ, the greatest possible weight under f of a match to $y[j]$. Note that if $h_s(\ell, j) \geq j$, there is no index fulfilling the condition since the substring $y[h_s(\ell, j) : j]$ has no characters.

Given the h-indices for every ℓ, j, we may compute $C^\ell(i, j)$ for any fixed ℓ as follows. $C^\ell(i, i) = 0$ by definition, and $C^\ell(i, j+1)$ can be computed from $C^\ell(i, j)$ by comparing i against each possible $h_s(\ell, i+1)$. One may then compute $D(i, w)$ from $C^n(i, j)$. However, one may directly compute D from the h-indices more efficiently using an approach described in Sect. 3.3.

The v-indices can be interpreted similarly, though the ordering of $v_1 \ldots v_\sigma$ is reversed. Consider $v_\sigma(\ell, j)$ for some fixed ℓ and j. This is the smallest value of i for which $C^\ell(i, j)$ does *not* exceed $C^{\ell-1}(i, j)$ by at least σ. Here, the comparison is between the WLCS of $x[1 : l]$ and $y[i : j]$ and that of $x[1 : l-1]$ and $y[i : j]$. The first WLCS might have more weight, and so there is unique index where the difference in weight first becomes less than σ. In this case, due to Corollary 2, the difference between $C^\ell(i, j)$ and $C^{\ell-1}(i, j)$ can only be less than the difference between $C^\ell(i-1, j)$ and $C^{\ell-1}(i-1, j)$. Now $v_\sigma(\ell, j)$ is the unique value of i after which the difference can be no more than $\sigma - 1$. Similar intuition applies to all the other v-indices.

We note that the v-indices are not explicitly involved in the procedure for computing entries of D; however, they are necessary in computing the h-indices.

Recursive Computation of the Indices. We now show how to compute the h-indices $h_s(\ell, j)$ for all ℓ, j. We first show a general recursive formula for these indices, then show a more efficient strategy to compute them.

In the formula, h_s will always refer to $h_s(\ell-1, j)$ unless indices are specified. Similarly, v_s will always refer to $v_s(\ell, j-1)$ unless indices are specified.

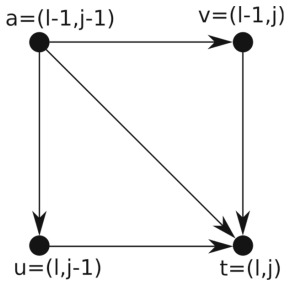

Fig. 1. Relationship between four points.

Let $d = f(x_\ell, y_j)$, where f is the scoring function. For the general case $\ell, j > 0$:

$$h_s(\ell, j) = \begin{cases} \text{if } d < s: \min\limits_{z \in [s,\sigma]} \left(\max(h_z, v_{z-(s-1)})\right) \\ \text{if } d \geq s: \\ \min\left(\min\limits_{z \in [d+1,\sigma]} \left(\max(h_z, v_{z-(s-1)})\right), v_{d-(s-1)}\right) \end{cases} \tag{3}$$

$$v_s(\ell, j) = \begin{cases} \text{if } d < s: \max\limits_{z \in [s,\sigma]} \left(\min(h_{z-(s-1)}, v_z)\right) \\ \text{if } d \geq s: \\ \max\left(\max\limits_{z \in [d+1,\sigma]} \left(\min(h_{z-(s-1)}, v_z)\right), h_{d-(s-1)}\right) \end{cases} \tag{4}$$

The base cases are $h_s(0, j) = j$ and $v_s(\ell, 0) = 0$ for all s. The first corresponds to an empty substring of x, which has an empty WLCS with any substring of y. The second corresponds to an empty substring of y, which has an empty WLCS with any substring of x. Recurrences (3) and (4) generalize the recurrences for h and v for unweighted LCS in [3], which can be recovered as a special case of our recurrence for $\sigma = 1$.

We now describe the calculations for these indices, beginning with the $h_s(\ell, j)$ calculations. To calculate $h_s(\ell, j)$, we use the entries $v_s(\ell, j-1)$ and $h_s(\ell-1, j)$ (for all possible s) in addition to the value of d. It is useful to visualize the situation using Fig. 1. In the figure, a represents the weight of the WLCS between $x[1 : l-1]$ and $y[i : j-1]$ for some i. Similarly, u represents the weight of the WLCS between $x[1 : l]$ and $y[i : j-1]$, and v is the WLCS between $x[1 : l-1]$ and $y[i : j]$. Finally, t represents the WLCS between $x[1 : l]$ and $y[i : j]$. The edges represent the relationship between the WLCS weights. First, u and v are both at least a. Further, one possible value for t could be $a + d$, since one may take the WLCS which corresponds to a and add in the match between $x[l]$ and $y[j]$, which has weight d. Alternatively, t could also be the same value as either u or v. If $t = u$, then $y[j]$ is unused in the WLCS; similarly, if $t = v$, the character $x[l]$ is unused.

The value of $h_s(\ell, j)$ has a natural interpretation: it is the first value of i for which the difference between t and u is at least s. Recall h_s will always refer to $h_s(\ell - 1, j)$ unless indices are specified; similarly, v_s will always refer to $v_s(\ell, j-1)$ unless indices are specified. Thus, h_s is exactly the minimum i where there is a difference of at least s between v and a. Similarly, v_s is the minimum i where there is a difference less than s between u and a. We will relate t and u by comparing both to a. Then we can determine the correct $h_s(\ell, j)$ where $t \geq u + s$ for any $i \geq h_s(\ell, j)$.

Suppose we seek to calculate $h_s(\ell, j)$ for a fixed $s > d$. One possible weight for t is $a + d$, but this edge cannot determine h_s as $u \geq a$, and hence $a + d$ is not greater than u by at least s. The WLCS which involves a never yields any information about the minimum i where $t \geq u + s$. However, consider an i such that $i \geq h_s(\ell - 1, j)$. The edge weight t can have value $a + s$. In this case, if $i \geq v_1(\ell, j - 1)$, then we know that $u = a$ and hence $t \geq u + s$. Therefore, if i is greater than both h_s and v_1, then the difference between t and u is at least s. Hence, it would seem that $h_s(\ell, j)$ is just equal to $\max(h_s, v_1)$. However, there are many other *pairs* which also fulfill this condition, e.g. if i is greater than both h_{s+1} and v_2. In general, if i is greater than any h_x and $v_{x-(s-1)}$ for some positive integer $\sigma \geq x > s$, then the difference between t and u is at least s. Therefore, in the case where $s > d$ the expression for $h_s(\ell, j)$ is the minimum of the pairwise maximum of such pairs. This is formalized in Eq. (3).

The other case is when we seek to compute h_s for a fixed $s \leq d$. Here, the weight $a + d$ is always possible for t. The expression in Eq. (3) is essentially a truncated version of the expression for $s > d$. Namely, we need not consider the pairs involving h_s where $s \leq d$. (If $i \geq v_{d-(s-1)}$ then immediately we already know that $t \geq u + s$ regardless of whether i is also greater than some h value.)

The $v_s(\ell, j)$ computations are similar. We are interested in the difference between t and v, so we will relate both t and v to a. First, consider computing $v_s(\ell, j)$ where $s > d$. In this case we can again ignore the case where $t = a + d$. Recall that $v_s(\ell, j)$ defines the value of i where if $i > v_s(\ell, j)$, then the $t < v + s$. Consider the case where only a single important pair of values exist, h_1 and v_s. If i is greater than v_s then $t = u < a + s$. A similar property holds if $i > h_1$. Hence, $v_s(\ell, j)$ is the value of the minimum of v_s and h_1 if that is the only pair. Once again, when there are multiple pairs of v_s, h_1 and v_{s+1}, h_2 and so on, the expression becomes more complex as it becomes the maximum of the minima of these pairs.

The case for computing $v_s(\ell, j)$ when $s \leq d$ is similar to the case for the $h_s(\ell, j)$ computations in Eq. (3) except where the formula is truncated; no pairs which involve v_s for $s \leq d$ are used. Since a weight of $a + d$ can always be attained, only $h_{d-(s-1)}$ needs to be checked for any of the lesser v_s pairs.

Equations (3) and (4) give a recursive computation for all of the h-indices and v-indices. There are $O(mn\sigma)$ total entries to compute, and following the two equations above yield a $O(mn\sigma^2)$ time algorithm for computing the h- and v-indices. However, using a clever observation, it is possible to compute these entries in $O(mn\sigma)$ time, which we show next.

Faster Computation of h- and v-Indices. Naively computing the recurrences 3 and 4 for each $1 \leq s \leq \sigma$ takes $O(\sigma^2)$ time. We show how to improve this to $O(\sigma)$ now.

We start with the following definitions. For $1 \leq s \leq \sigma$ define $z^*(s)$ to be the value such that the following holds: (1) $z < z^*(s) \implies v_{z-s+1} > h_z$ and (2) $z \geq z^*(s) \implies v_{z-s+1} \leq h_z$. Similarly, define $z^{\#}(s)$ to be the value such that (1) $z < z^{\#}(s) \implies h_{z-s+1} < v_z$ and (2) $z \geq z^{\#}(s) \implies h_{z-s+1} \geq v_z$. These values are well defined since the sequences h and v are respectively non-decreasing and non-increasing, so either the inequalities above trivially hold, or there is a point where the sequences cross. The existence of a crossing point is not affected by applying an offset to one of the sequences. We will simultaneously compute $z^*(s)$ and $z^{\#}(s)$ while computing new values of h_s and v_s.

To see why the above definitions are useful, consider substituting them into (3) and (4). First, consider the calculation of h_s when $d < s$:

$$\begin{aligned} h_s(\ell, j) &= \min_{z=s}^{\sigma} \left(\max(h_z, v_{z-s+1})\right) \\ &= \min\left(\min_{z<z^*(s)} \max\left(h_z, v_{z-s+1}\right), \min_{z\geq z^*(s)} \max\left(h_z, v_{z-s+1}\right)\right) \\ &= \min\left(\min_{z<z^*(s)} v_{z-s+1}, \min_{z\geq z^*(s)} h_z\right) = \min\left(v_{z^*(s)-s}, h_{z^*(s)}\right) \end{aligned}$$

where we again use the property that h and v are respectively non-decreasing and non-increasing. Similar calculations can be done with $z^{\#}(s)$ for computing $v_s(\ell, j)$ and for the case when $d \geq s$. This shows that given $z^*(s)$ and $z^{\#}(s)$, it is possible to compute $h_s(\ell, j)$ and $v_s(\ell, j)$ in constant time.

The only remaining task is to compute $z^*(s)$ and $z^{\#}(s)$ for each weight s. This can be done by sweeping through h and v in $O(\sigma)$ time. We may then compute $h_s(\ell, j)$ and $v_s(\ell, j)$ for each s in $O(\sigma)$ time.

Computing the D Matrix. We now show how to compute the entries $D(i, w)$ directly from the h-indices. The computation requires only the indices $h_s(n, j)$; in this section, we drop the n and refer to these indices simply as $h_s(j)$. We compute the entries of $D(i, w)$ row by row, iterating through one value of i at a time. At each iteration, we will keep T, a data structure storing pairs of the form $(j, h_s(j))$. During iteration i, we may insert pairs into T or delete pairs from T, maintaining the following invariant:

$$(j, h_s(j)) \in T \iff j > i \text{ and } h_s(j) \leq i. \tag{5}$$

The invariant guarantees two useful properties. First, all pairs in T have $h_s(j) \leq i$, the existence of such a pair in T means that the difference between $C(i, j)$ and $C(i, j-1)$ is s. Note that if $(j, h_s(j)) \in T$, then clearly $(j, h_{s'}(j)) \in T$ for all $s' < s$ since $h_{s'}(j) \leq h_s(j)$. Thus, one can think of each pair in T as representing an increase of 1.

Algorithm 2. Construct D matrix using the h-indices

```
T ← ∅
j ← 1, s ← 1
for i = 1, ..., m do
    while h_s(j) ≤ i do
        if j > i then                                  ▷ Insert pairs w/ j > i.
            T.insert((j, s, h_s(j)))
        end if
        s ← s + 1
        if s > σ then
            s ← 1
            j ← j + 1
        end if
    for k ∈ K do                                       ▷ Compute D(i, k) ∀k
        (j', s', h') ← T.search_by_rank(k)
        D(i, k) = j'
    Remove from T all (j, s, h) where j = i
```

Second, if the pairs in T are sorted increasingly by j, then $D(i,k)$ is exactly the j_k which corresponds to the pair of rank k within T. This can be shown as follows: Let $j_1, j_2, \ldots j_k$ denote the pairs of rank 1 through k (represented by say $(j_k, h_s(j_k))$ within T. Each fixed j_x among these means that there is a difference of 1 between $C(i, j_x - 1)$ and $C(i, j_x)$. There are k pairs here, each denoting a difference of 1 between some $C(i, j_x)$ and $C(i, j_x - 1)$ and there are a total of k such differences. Note by the invariant, $j_1 > i$. Furthermore, clearly $j_1 \leq j_k$. Thus, between $C(i,i)$ and $C(i, j_k)$ there are a total of k differences of 1 each. Since $C(i,i) = 0$ the difference between $C(i,i)$ and $C(i, j_k)$ is exactly k. Hence, j_k is exactly the value of $D(i,k)$.

This analysis yields Algorithm 2, where T is a balanced tree data structure.

4 Analysis of Approximation and Runtime

We now formally describe the sketching strategy for the D matrix and prove the claims about the performance of Algorithm 1 under our sketching procedure.

Recall that our sketching strategy fixes a constant $\epsilon > 0$ and sets $\beta = 1 + \frac{\epsilon}{\log n}$. Define $D^*(i,s)$ to be the least j such that there exists a correspondence between x and $y[i:j]$ with weight $w \geq \lfloor \beta^s \rfloor$. Define $D^*_{r_1,r_2}$ analogously to D_{r_1,r_2} for substrings of x. To compute $D^*_{r_1,r_3}$ from $D^*_{r_1,r_2}$ and $D^*_{r_2+1,r_3}$, we modify Algorithm 1 as follows. For each power s s.t. $\lfloor \beta^s \rfloor \leq W$, we consider each power $s_1 \leq s$ and compute the least s_2 such that $\lfloor \beta^{s_1} \rfloor + \lfloor \beta^{s_2} \rfloor \geq \lfloor \beta^s \rfloor$. Let $j' = D^*_{r_1,r_2}(i, s_1)$ and $j = D^*_{r_2+1,r_3}(j'+1, s_2)$. Then there exist non-overlapping correspondences of weights at least β^{s_1} and β^{s_2}, and hence a combined correspondence of weight at least β^s, between $x[r_1 : r_3]$ and $y[i:j]$. We take $D^*_{r_1,r_3}(i,s)$ to be the least j' that results from this procedure.

4.1 Quality of the Solution

In Sect. 3.2, we showed how to reduce the space and time requirement of computing the D matrix using Algorithm 1 via sketching. We consider only weights of the form $\lfloor \beta^s \rfloor$ for $\beta = 1 + \epsilon/\log n$ and so will not obtain the exact optimum. However, we show that we can recover a $(1-\epsilon)$ approximation to the optimum.

Computationally, we only need to construct the matrix D^* in order to extract solutions to the AWLCS and WLCS problems. However, for analysis purposes it is useful to define C^*, an approximate version of the matrix C. Let $C_{r_1,r_2}(i,j)$ be the optimal weight of a WLCS between $x[r_1 : r_2]$ and $y[i : j]$. Equivalently, we have $C_{r_1,r_2}(i,j) = \max\{w \mid D_{r_1,r_2}(i,w) \leq j\}$. This motivates defining C^* as follows. Let $C^*_{r_1,r_2}(i,j) = \max_s\{\lfloor \beta^s \rfloor \mid D^*_{r_1,r_2}(i,s) \leq j\}$.

We prove the following lemma which shows that the C^* matrices approximate the C matrices well, and hence the matrices D^* implicitly encode good solutions.

Lemma 2. *Let $x[r_1, r_2]$ be a substring of x considered by our algorithm in some step. Then for all i, j we have*

$$C^*_{r_1,r_2}(i,j) \geq (1-\epsilon)\, C_{r_1,r_2}(i,j)$$

Proof. We prove the following stronger claim by induction. Let ℓ be the level at which we combined substrings to arrive at $x[r_1 : r_2]$ in our algorithm. Then for all i, j we have

$$C^*_{r_1,r_2}(i,j) \geq \left(1 - \frac{\epsilon}{\log n}\right)^{\ell} C_{r_1,r_2}(i,j)$$

In particular this implies the lemma since we can use the inequality $(1-z)^\ell \geq 1 - \ell z$ for all $z \leq 1$ and $\ell \geq 0$.

$$\begin{aligned} C^*_{r_1,r_2}(i,j) &\geq \left(1 - \frac{\epsilon}{\log n}\right)^{\ell} C_{r_1,r_2}(i,j) \\ &\geq \left(1 - \frac{\epsilon\ell}{\log n}\right) C_{r_1,r_2}(i,j) \geq (1-\epsilon) C_{r_1,r_2}(i,j) \end{aligned}$$

Now to prove the claim by induction on the levels ℓ. Fix a pair of indices i, j in y. If $x[r_1, r_2]$ was considered at the first level, then we computed the exact matrix D_{r_1,r_2} using our base case algorithm, so the base case follows trivially. Now suppose that we considered $x[r_1, r_2]$ at some level ℓ after the first. Our algorithm combines the subproblems corresponding to $x[r_1, r']$ and $x[r'+1, r_2]$ that occur at level $\ell - 1$. where $r' = \lfloor (r_1 + r_2)/2 \rfloor$. To compute the solution for r_1, r_2 and i, j we concatenate solutions corresponding to $x[r_1, r']$ and $y[i : j']$ and $x[r'+1, r_2]$ and $y[j'+1 : j]$, for some j'. We get the sum of their weights, but rounded down due to sketching. Thus we have:

$$C^*_{r_1,r_2}(i,j) \geq \frac{C^*_{r_1,r'}(i,j') + C^*_{r'+1,r_2}(j',j)}{(1+\epsilon/\log n)}. \tag{6}$$

Now $C_{r_1,r_2}(i,j) = C_{r_1,r'}(i,j'') + C_{r'+1,r_2}(j''+1,j)$ for some j'', since the solution corresponding to $C_{r_1,r_2}(i,j)$ can be written as the concatenation of two subsolutions between $x[r_1 : r']$, $y[i : j'']$ and $x[r'+1 : r_2]$, $y[j''+1 : j]$. Note that our algorithm chooses j' such that $C^*_{r_1,r'}(i,j') + C^*_{r'+1,r_2}(j',j) \geq C^*_{r_1,r'}(i,j'') + C^*_{r'+1,r_2}(j''+1,j)$. Now applying the induction hypothesis to $C^*_{r_1,r'}(i,j'')$ and $C^*_{r'+1,r_2}(j''+1,j)$ we have:

$$\begin{aligned} C^*_{r_1,r'}(i,j') + C^*_{r'+1,r_2}(j',j) &\geq C^*_{r_1,r'}(i,j'') + C^*_{r'+1,r_2}(j''+1,j) \\ &\geq \left(1 - \frac{\epsilon}{\log n}\right)^{\ell-1} \left(C_{r_1,r'}(i,j'') + C_{r'+1,r_2}(j''+1,j)\right) \\ &= \left(1 - \frac{\epsilon}{\log n}\right)^{\ell-1} C_{r_1,r_2}(i,j) \end{aligned}$$

Now combining this with (6) we have:

$$C^*_{r_1,r_2}(i,j) \geq \frac{\left(1 - \frac{\epsilon}{\log n}\right)^{\ell-1} C_{r_1,r_2}(i,j)}{1 + \frac{\epsilon}{\log n}} \geq \left(1 - \frac{\epsilon}{\log n}\right)^{\ell} C_{r_1,r_2}(i,j),$$

which completes the proof of the general case.

4.2 Running Time

We now analyze the running time of our algorithm, starting with the combining procedure in Algorithm 1.

Lemma 3. *Let $n' = r_3 - r_1 + 1$ be the number of characters of the string x assigned to one call to Algorithm 1. Let $\epsilon' = \epsilon/2\log(n)$ and $t = \log_{1+\epsilon'}(\sigma \min(n', m))$. Then the procedure described in Algorithm 1 uses $O(mt)$ space and runs in $O(mt^2)$ time.*

Proof. First note that $\sigma \min(n', m)$ is an upper bound on the maximum weight for the instance passed to Algorithm 1. The matrix D contains an entry for each pair of starting indices and each weight. There are m starting indices. To prove the space bound, it suffices to show that the number of possible weights is $O(t)$. Recall that sketching the weights yields an entry for each weight of the form $(1 + \epsilon/2\log_2 n)^\ell = (1 + \epsilon')^\ell$ for integer ℓ, up to some upper bound on the weight. Hence, taking t as in the statement of the lemma implies that $(1 + \epsilon')^t \geq \sigma \min(n', m)$, so $O(t)$ different weights suffices.

The bound on the running time follows by noting that each of the $O(m)$ iterations of the algorithm iterates through all pairs of weights, yielding total time $O(mt^2)$.

$O(\log(p))$ rounds of combining are required to merge the p base-case results into the final D matrix, since each round reduces the number of remaining subproblems by half. To analyze the entire algorithm, we separately consider the two steps.

Lemma 4. *Let $B(m,n)$ be the running time of a base-case algorithm computing D. Then our algorithm runs in time $B(m,n/p) + O(m \log^2_{1+\epsilon'}(\sigma m) \log(p))$ on p processors.*

Proof. The base case algorithm is run on subproblems of size $n/p \times m$. Each of the $O(\log(p))$ rounds of merging has cost $O(m \log^2_{1+\epsilon'}(\sigma m))$ by the previous lemma.

Finally, since $\log_{1+\epsilon'}(\sigma m) = O(\log(\sigma m)/\epsilon') = O(\log_2(n) \log(\sigma m)/\epsilon)$, our algorithm achieves the claimed runtime and local memory per processor.

References

1. Abboud, A., Backurs, A., Williams, V.V.: Quadratic-time hardness of LCS and other sequence similarity measures. CoRR abs/1501.07053 (2015). http://arxiv.org/abs/1501.07053
2. Aggarwal, A., Klawe, M., Moran, S., Shor, P., Wilber, R.: Geometric applications of a matrix-searching algorithm. Algorithmica **2**, 195–208 (1987)
3. Alves, C., Cáceres, E., Song, S.: A coarse-grained parallel algorithm for the all-substrings longest common subsequence problem. Algorithmica **45**, 301–335 (2006). https://doi.org/10.1007/s00453-006-1216-z
4. Alves, C., Cáceres, E., Song, S.: An all-substrings common subsequence algorithm. Discrete Appl. Math. **156**(7), 1025–1035 (2008). https://doi.org/10.1016/j.dam.2007.05.056. http://www.sciencedirect.com/science/article/pii/S0166218X07002727
5. Apostolico, A., Atallah, M.J., Larmore, L.L., McFaddin, S.: Efficient parallel algorithms for string editing and related problems. SIAM J. Comput. **19**(5), 968–988 (1990). https://doi.org/10.1137/0219066
6. Bringmann, K., Chaudhury, B.R.: Sketching, streaming, and fine-grained complexity of (weighted) LCS. In: 38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2018, Ahmedabad, India, 11–13 December 2018, pp. 40:1–40:16 (2018). https://doi.org/10.4230/LIPIcs.FSTTCS.2018.40
7. Bringmann, K., Künnemann, M.: Multivariate fine-grained complexity of longest common subsequence. In: Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, 7–10 January 2018, pp. 1216–1235 (2018). https://doi.org/10.1137/1.9781611975031.79
8. Crochemore, M.M., Landau, G., Ziv-Ukelson, M.: A subquadratic sequence alignment algorithm for unrestricted scoring matrices. SIAM J. Comput. **32**, 1654–1673 (2003). https://doi.org/10.1137/S0097539702402007
9. Im, S., Moseley, B., Sun, X.: Efficient massively parallel methods for dynamic programming. In: Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC), Montreal, Canada, pp. 798–811 (2017)
10. Jones, N.: An Introduction to Bioinformatics Algorithms. MIT Press, Cambridge (2004)
11. Krusche, P., Tiskin, A.: New algorithms for efficient parallel string comparison. In: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Thira, Santorini, Greece, pp. 209–216 (2010)

12. Krusche, P., Tiskin, A.: Efficient longest common subsequence computation using bulk-synchronous parallelism. In: International Conference on Computational Science and its Applications (Proceedings, Part V), ICCSA 2006, Glasgow, UK, 8–11 May 2006, pp. 165–174 (2006). https://doi.org/10.1007/11751649_18
13. Masek, W.J., Paterson, M.: A faster algorithm computing string edit distances. J. Comput. Syst. Sci. **20**(1), 18–31 (1980). https://doi.org/10.1016/0022-0000(80)90002-1
14. Needleman, S., Wunsch, C.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. **48**, 443–453 (1970)
15. Program, N.G.S.: DNA sequencing costs: Data (2017). https://www.genome.gov/sequencingcostsdata/
16. Rajko, S., Aluru, S.: Space and time optimal parallel sequence alignments. IEEE Trans. Parallel Distrib. Syst. **15**(12), 1070–1081 (2004). https://doi.org/10.1109/TPDS.2004.86
17. Russo, L.: Monge properties of sequence alignment. Theor. Comput. Sci. **423**, 30–49 (2012). https://doi.org/10.1016/j.tcs.2011.12.068
18. http://jaligner.sourceforge.net/
19. https://github.com/mengyao/Complete-Striped-Smith-Waterman-Library
20. https://github.com/Martinsos/opal
21. Schmidt, J.: All highest scoring paths in weighted graphs and their applications to finding all approximate repeats in strings. SIAM J. Comput. **27**, 972–992 (1998)
22. Smith, T., Waterman, M.: Identification of common molecular subsequences. J. Mol. Biol. **147**(1), 195–197 (1981)
23. States, D., Gish, W., Altschul, S.: Improved sensitivity of nucleic acid database searches using application-specific scoring matrices. METHODS Companion Meth. Enzymol. **3**, 66–70 (1991)
24. Tiskin, A.: Semi-local string comparison: algorithmic techniques and applications, ch 6.1 (2013). https://arxiv.org/abs/0707.3619, v21
25. Tiskin, A.: Fast distance multiplication of unit-monge matrices. Algorithmica **71**(4), 859–888 (2015). https://doi.org/10.1007/s00453-013-9830-z