Efficient Parallel Output-Sensitive Edit Distance

Xiangyun Ding
University of California, Riverside

Xiaojun Dong
University of California, Riverside

Yan Gu
University of California, Riverside

Youzhe Liu
University of California, Riverside

Yihan Sun
University of California, Riverside

Abstract

In this paper, we study efficient parallel edit distance algorithms, both in theory and in practice. Given two strings $A[1..n]$ and $B[1..m]$, and a set of operations allowed to edit the strings, the edit distance between A and B is the minimum number of operations required to transform A into B. In this paper, we use edit distance to refer to the Levenshtein distance, which allows for unit-cost single-character edits (insertions, deletions, substitutions). Sequentially, a standard Dynamic Programming (DP) algorithm solves edit distance with $\Theta(nm)$ cost. In many real-world applications, the strings to be compared are similar to each other and have small edit distances. To achieve highly practical implementations, we focus on output-sensitive parallel edit-distance algorithms, i.e., to achieve asymptotically better cost bounds than the standard $\Theta(nm)$ algorithm when the edit distance is small. We study four algorithms in the paper, including three algorithms based on Breadth-First Search (BFS), and one algorithm based on Divide-and-Conquer (DaC). Our BFS-based solution is based on the Landau-Vishkin algorithm. We implement three different data structures for the longest common prefix (LCP) queries needed in the algorithm: the classic solution using parallel suffix array, and two hash-based solutions proposed in this paper. Our DaC-based solution is inspired by the output-insensitive solution proposed by Apostolico et al., and we propose a non-trivial adaption to make it output-sensitive. All of the algorithms studied in this paper have good theoretical guarantees, and they achieve different tradeoffs between work (total number of operations), span (longest dependence chain in the computation), and space.

We test and compare our algorithms on both synthetic data and real-world data, including DNA sequences, Wikipedia texts, GitHub repositories, etc. Our BFS-based algorithms outperform the existing parallel edit-distance implementation in ParlayLib in all test cases. On cases with fewer than 10^5 edits, our algorithm can process input sequences of size 10^9 in about ten seconds, while ParlayLib can only process sequences of sizes up to 10^6 in the same amount of time. By comparing our algorithms, we also provide a better understanding of the choice of algorithms for different input patterns. We believe that our paper is the first systematic study in the theory and practice of parallel edit distance.

2012 ACM Subject Classification Theory of computation → Parallel algorithms

Keywords and phrases Edit Distance, Parallel Algorithms, String Algorithms, Dynamic Programming, Pattern Matching

Digital Object Identifier 10.4230/LIPIcs.ESA.2023.46

Related Version *Full Version*: `https://arxiv.org/abs/2306.17461` [16]

Supplementary Material *Source Code*: `https://github.com/ucrparlay/Edit-Distance` [17]

Funding This work is supported by NSF grants CCF-2103483, CCF-2238358, and IIS-2227669, and UCR Regents Faculty Fellowships.

1 Introduction

Given two strings (sequences) $A[1..n]$ and $B[1..m]$ over an alphabet Σ and a set of operations allowed to edit the strings, the ***edit distance*** between A and B is the minimum number of operations required to transform A into B. WLOG, we assume $m \leq n$. The most commonly used metric is the ***Levenshtein distance*** which allows for unit-cost single-character edits (insertions, deletions, substitutions). In this paper, we use *edit distance* to refer to the Levenshtein distance. We use k to denote the edit distance for strings A and B throughout this paper. Edit distance is usually used to measure the similarity of two strings (a smaller distance means higher similarity).

Edit distance is a fundamental problem in computer science, and is introduced in most algorithm textbooks (e.g., [14, 15, 23]). In practice, it is widely used in version-control software [54], computational biology [12, 31, 39], natural language processing [10, 29], and spell corrections [28]. It is also closely related to other important problems such as longest common subsequence (LCS) [50], longest increasing subsequence (LIS) [34], approximate string matching [56], and multi-sequence alignment [59]. The classic dynamic programming (DP) solution can compute edit distance in $O(nm)$ work (number of operations) between two strings of sizes n and m. This complexity is impractical if the input strings are large. One useful observation is that, in real-world applications, the strings to be compared are usually *reasonably similar*, resulting in a relatively small edit distance. For example, in many version-control softwares (e.g., Git), if the two committed versions are similar (within a certain number of edits), the "delta" file is stored to track edits. Otherwise, if the difference is large, the system directly stores the new version. Most of the DNA or genome sequence alignment applications also only focus on when the number of edits is *small* [39]. We say an edit distance algorithm is *output-sensitive* if the work is $o(nm)$ when $k = o(n)$. Many more efficient and/or practical algorithms were proposed in this setting with cost bounds parameterized by k [19, 20, 21, 22, 26, 35, 36, 37, 46, 47, 49].

Considering the ever-growing data size and plateaued single-processor performance, it is crucial to consider parallel solutions for edit distance. Although the problem is simple and well-studied in the sequential setting, we observe a *huge gap* between theory and practice in the parallel setting. The few implementations we know of [7, 55, 58] simply parallelize the $O(nm)$-work sequential algorithm and require $O(n)$ span (longest dependence chain), which indicates low-parallelism and redundant work when $k \ll n$. Meanwhile, numerous theoretical parallel algorithms exist [1, 3, 20, 37, 41, 48], but it remains unknown whether these algorithms are practical (i.e., can be implemented with reasonable engineering effort), and if so, whether they can yield high performance. *The goal of this paper is to formally study parallel solutions for edit distance. By carefully studying existing theoretical solutions, we develop* ***new output-sensitive parallel solutions with good theoretical guarantees and high performance in practice****. We also conduct in-depth experimental studies on existing and our new algorithms.*

The classic dynamic programming (DP) algorithm solves edit distance by using the states $G[i, j]$ as the edit distance of transforming $A[1..i]$ to $B[1..j]$. $G[i, j]$ can be computed as:

$$G[i,j] = \begin{cases} G[i-1,j-1] & \text{if } A[i] = B[j] \text{ and } i > 0, j > 0 \\ 1 + \min(G[i-1,j], G[i-1,j-1], G[i,j-1]) & \text{otherwise} \end{cases}$$

$$G[i,j] = \max(i,j) \qquad \text{if } i = 0 \text{ or } j = 0$$

A simple parallelization of this computation is to compute all states with the same $i + j$ value in parallel, and process all $i + j$ values in an incremental order [7, 55, 58]. However,

Algorithm	Work	Span	Space*	Algorithm	Work	Span	Space*
BFS-SA	$O(n+k^2)$	$\tilde{O}(k)$	$O(n)$	**BFS-Hash***	$O(n+k^2\log n)$	$\tilde{O}(k)$	$O(n)$
DaC-SD	$O(nk\log k)$	$\tilde{O}(1)$	$O(nk)$	**BFS-B-Hash***	$O(n+k^2b\log n)$	$\tilde{O}(kb)$	$O(n/b+k)$

Table 1 Algorithms in this paper. k is the edit distance. b is the block size. *: Monte Carlo algorithms due to the use of hashing. "Space*" means auxiliary space used in addition to the input. Here we assume constant alphabet size for BFS-SA.

this approach has low parallelism as it requires $n+m$ rounds to finish. Later work [1, 3, 41] improved parallelism using a ***divide-and-conquer (DaC)*** approach and achieved $\tilde{O}(n^2)$ work and polylog(n) span. These algorithms use the monotonicity of the DP recurrence, and are complicated. There are two critical issues in the DaC approaches. First, to the best of our knowledge, there exist no implementations given the sophistication of these algorithms. Second, they are not output-sensitive ($\tilde{O}(nm)$ work), which is inefficient when $k \ll n$.

Alternatively, many existing solutions, both sequentially [19, 20, 21, 22, 26, 35, 36, 46, 47] and in parallel [20, 37] use output-sensitive algorithms, and achieve $\tilde{O}(nk)$ or $\tilde{O}(n+k^2)$ work and $\tilde{O}(k)$ span. These algorithms view DP table as a grid-like DAG, where each state (cell) (x,y) has three incoming edges from $(x-1,y)$, $(x,y-1)$, and $(x-1,y-1)$ (if they exist). The edge weight is 0 from $(x-1,y-1)$ to (x,y), when $A[x]=B[y]$, and 1 otherwise. Then edit distance is equivalent to the shortest path from $(0,0)$ to (n,m). An example is given in Fig. 1. Since the edge weights can only be 0 or 1, we can use ***breadth-first search (BFS)*** from the cell $(0,0)$ until (n,m) is reached. Ukkonen [56] further showed that using ***longest common prefix (LCP)*** queries based on suffix trees or suffix arrays, the work can be improved to $O(n+k^2)$. Landau and Vishkin [37] parallelized this algorithm (see Sec. 3). While the sequential output-sensitive algorithms have been widely used in practice [21, 26, 36, 46, 47], we are unaware of any existing implementations for the parallel version.

We systematically study parallel output-sensitive edit distance, using both the BFS-based and the DaC-based approaches. Our first effort is to implement the BFS-based Landau-Vishkin algorithm with our carefully-engineered parallel suffix array (SA) implementation, referred to as BFS-SA. Although suffix array is theoretically efficient with $O(n)$ construction work, the hidden constant is large. Thus, we use hashing-based solutions to replace SA for LCP queries to improve the performance in practice. We first present a simple approach BFS-Hash in Sec. 3.2 that stores a hash value for all prefixes of the input. This approach has $O(n)$ construction work, $O(\log n)$ per LCP query, and $O(n)$ auxiliary space. While both BFS-SA and BFS-Hash take $O(n)$ extra space, such space overhead can be significant in practice—for example, BFS-Hash requires n 64-bit hash values, which is 4× the input size considering characters as inputs, and 32× with even smaller alphabet such as molecule bases (alphabet as $\{A,C,G,T\}$). To address the space issue, we proposed BFS-B-Hash using *blocking*. Our solution takes a user-defined parameter b as the block size, which trades off between space usage and query time. BFS-B-Hash limits extra space in $O(n/b)$ by using $O(b\log n)$ LCP query time. Surprisingly, despite a larger LCP cost, our hash-based solutions are consistently faster than BFS-SA in all real-world test cases, due to cheaper construction. All of our BFS-based solutions are simple to program.

We also study the DaC-based approach and propose a parallel output-sensitive solution. We propose a non-trivial adaption for the AALM algorithm [1] to make it output-sensitive. Our algorithm is inspired by the BFS-based approaches, and improves the work from $\tilde{O}(nm)$ to $\tilde{O}(nk)$, with polylogarithmic span. The technical challenge is that the states in the computation are no longer a rectangle, but an irregular shape (see Fig. 1 and 3). We then present a highly non-trivial implementation of this algorithm. Among many key challenges, we highlight our solution to avoid dynamically allocating arrays in the recursive execution.

While memory allocation is mostly ignored theoretically, in practice it can easily be the performance bottleneck in the parallel setting. We refer to this implementation as DaC-SD, with details given in Sec. 4 and 5.2 and the full version of this paper [16].

The bounds of our algorithms (BFS-SA, BFS-Hash, BFS-B-Hash, and DaC-SD) are presented in Tab. 1. We implemented them and show an experimental study in Sec. 6. We tested both synthetic and real-world datasets, including DNA, English text from Wikipedia, and code repositories from GitHub, with string lengths in 10^5–10^9 and varying edit distances, many of them with real edits (e.g., edit history from Wikipedia and commit history on GitHub). In most tests, our new BFS-B-Hash or BFS-Hash performs the best, and their relative performance depends on the value of k and the input patterns. Our BFS-based algorithms are faster than the existing parallel output-insensitive implementation in ParlayLib [7], even with a reasonably large $k \approx 10^5$. We believe that our paper is the first systematic study in theory and practice of parallel edit distance, and we give the first publicly available parallel edit distance implementation that can process *billion-scale strings* with small edit distance and our code at [17]. Due to page limit, some details are provided in the full version of this paper [16]. We summarize our contributions as follows:

1. Two new BFS-based edit distance solutions BFS-Hash and BFS-B-Hash using hash-based LCP queries. Compared to the existing SA-based solution in Landau-Vishkin, our hash-based solutions are simpler and more practical. BFS-B-Hash also allows for tradeoffs between time and auxiliary space.
2. A new DaC-based edit distance solution DaC-SD with $O(nk \log k)$ work and polylogarithmic span.
3. New implementations for four output-sensitive edit distance algorithms: BFS-SA, BFS-Hash, BFS-B-Hash and DaC-SD. Our code is publicly available[17].
4. Experimental study of the existing and our new algorithms on different input patterns.

2 Preliminaries

We use $O(f(n))$ *with high probability* (*whp*) (in n) to mean $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \geq 1$. We use $\tilde{O}(f(n))$ to denote $O(f(n) \cdot \text{polylog}(n))$. For a string A, we use $A[i]$ as the i-th character in A. We use *string* and *sequence* interchangeably. We use $A[i..j]$ to denote the i-th to the j-th characters in A, and $A[i..j)$ the i-th to the $(j-1)$-th characters in A. Throughout the paper, we use "auxiliary space" to mean space used in addition to the input.

String Edit Distance. Given two strings $A[1..n]$ and $B[1..m]$, Levenshtein's Edit Distance [38] between A and B is the minimum number of operations needed to convert A to B by using insertions, deletions, and substitutions. We also call the operations ***edits***. In this paper, we use ***edit distance*** to refer to Levenshtein's Edit Distance. The classic dynamic programming (DP) algorithm for edit distance uses DP recurrence shown in Sec. 1 with $O(mn)$ work and space.

Hash Functions. For the simplicity of algorithm descriptions, we assume a perfect hash function for string comparisons, i.e., a function $h : S \to [1, O(|S|)]$ such that $h(x) = h(y) \iff x = y$. For any alphabet Σ with size $|\alpha|$, we use a hash function $h(A[l..r]) = \sum_{i=l}^{r} A[i] \times p^{r-i}$ for some prime numbers $p > |\alpha|$, which returns a unique hash value of the substring $A[l..r]$. The hash values of two consecutive substrings S_1 and S_2 can be concatenated as $h([S_1, S_2]) = h(S_1) \cdot p^{|S_2|} + h(S_2)$, and the inverse can also be computed as $h(S_2) = h([S_1, S_2]) - p^{|S_2|} \cdot h(S_1)$. For simplicity, we denote concatenation and its inverse operation as $\oplus$ and $\ominus$, respectively, as $h([S_1, S_2]) = h(S_1) \oplus h(S_2)$ and $h(S_2) = h([S_1, S_2]) \ominus h(S_1)$. We assume perfect hashing for

theoretical analysis. In practice, we use p as a large prime and modular arithmetic to keep the word-size hash values. In our experiment, we compare different approaches and validate that our implementations are correct in all test cases. However, collisions are possible for other datasets, since different strings may be mapped to the same hash value. If such cases arise, one can either use multiple hash functions for a better success rate in practice, or use the idea of Hirschberg's algorithm [27] to generate the edit sequence and run a correctness check (and restart with another hash function if failed).

Longest Common Prefix (LCP). For two sequences $A[1..n]$ and $B[1..m]$, the Longest Common Prefix (LCP) query at position x in $A[1..n]$ and position y in $B[1..m]$ is the longest substring starting from $A[x]$ that match a prefix starting from $B[y]$. With clear context, we also use the term "LCP" to refer to the length of the LCP, i.e., $LCP(A, B, x, y)$ is the length of the longest common prefix substring starting from $A[x]$ and $B[y]$ for A and B.

Computational Model. We use the ***work-span model*** in the classic multithreaded model with ***binary-forking*** [2, 8, 9]. We assume a set of threads that share the memory. Each thread acts like a sequential RAM plus a `fork` instruction that forks two child threads running in parallel. When both child threads finish, the parent thread continues. A parallel-for is simulated by `fork` for a logarithmic number of steps. A computation can be viewed as a DAG (directed acyclic graph). The ***work W*** of a parallel algorithm is the total number of operations, and the ***span (depth) S*** is the longest path in the DAG. The randomized work-stealing scheduler can execute such a computation in $W/P + O(S)$ time *whp* in W on P processors [2, 9, 25].

Suffix Array. The suffix array (SA) [42] is a lexicographically sorted array of the suffixes of a string, usually used together with the longest common prefix (LCP) array, which stores the length of LCP between every adjacent pair of suffixes. The SA and LCP array can be built in parallel in $O(n)$ work and $O(\log^2 n)$ span *whp* [32, 53].

In edit distance, we need the LCP query between $A[x..n]$ and $B[y..m]$ for any x and y. This can be computed by building the SA and LCP arrays for a new string $C[1..n+m]$ that concatenates $A[1..n]$ and $B[1..m]$. The LCP between any pair of suffixes in C can be computed by a range minimum query (RMQ) on the LCP array, which can be built in $O(n+m)$ work and $O(\log(n+m))$ span [8]. Combining all pieces gives the following theorem:

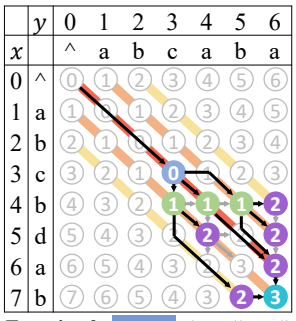
▶ **Lemma 1.** *Given two strings $A[1..n]$ and $B[1..m]$, using a suffix array, the longest common prefix (LCP) between any two substrings $A[x..n]$ and $B[y..m]$ can be reported in $O(1)$ work and span, with $O(n+m)$ preprocessing work and $O(\log^2(n+m))$ span* whp.

3 BFS-based Algorithms

3.1 Overview of Existing Sequential and Parallel BFS-based Algorithms

Many existing output-sensitive algorithms [19, 20, 21, 22, 26, 35, 36, 37, 46, 47] are based on breadth-first search (BFS). These algorithms view the DP matrix for edit distance as a DAG, as shown in Fig. 1. In this section, we use x and y to denote the row and column ids of the cells in the DP matrix, respectively. Each state (cell) (x, y) has three incoming edges from $(x-1, y)$, $(x, y-1)$, and $(x-1, y-1)$ (if they exist). The edge weight is 0 from $(x-1, y-1)$ to (x, y) when $A[x] = B[y]$, and 1 otherwise. Then edit distance is equivalent to the shortest distance from $(0, 0)$ to (n, m).

Since the edge weights are 0 or 1, we can use a special breadth-first search (BFS) to compute the shortest distance. In round t, we process states with edit distance t. The



Frontier 0: `f₀[0]=3`, the cell on diagonal 0 is `(3,3)`
Frontier 1: `f₁[-1]=4`, `f₁[0]=4`, `f₁[1]=4` `[(4,3),(4,4),(4,5)]`
Frontier 2: `f₂[-2]=4`, `f₂[-1]=5`, `f₂[0]=6`, `f₂[1]=5`, `f₂[2]=7`
Frontier 3: `f₃[0]=3`, the others are invalid

Figure 1 BFS-based edit distance on $A[1..n]$ and $B[1..m]$. A more detailed description is in the full version [16]. $f_t[i]$ is the row-id of the last cell on diagonal i with edit distance t (frontier t), representing cell $(f_t[i], f_t[i]-i)$.

Algorithm 1 BFS-based parallel edit distance [37]

```
1  f_0[0] ← LCP(A[1..n], B[1..m])   // Starting point
2  t ← 0
3  while f_t[n − m] ≠ n do
4      t ← t + 1
       // Find new frontier for diagonal i
5      parallel-for-each −t ≤ i ≤ t do
6          f_t[i] ← f_{t−1}[i]      // Start from the last cell
7          foreach ⟨dx, dy⟩ ∈ {⟨0,1⟩, ⟨1,0⟩, ⟨1,1⟩} do
               // The previous cell is from diagonal j
               // j = (x − dx) − (y − dy) = i − dx + dy
8              j ← i − dx + dy
9              if |j| ≤ t − 1 then
10                 The row id x ← f_{t−1}[j] + dx
11                 The column id y ← x − i
                   // Skip the common prefix
12                 x ← x + LCP(A[x + 1..n], B[y + 1..m])
                   // Keep the largest row id
13                 f_t[i] ← max(f_t[i], x)
14 return t
```

algorithm terminates when we reach cell (n, m). First observed by Ukkonen [56], in the BFS-based approach, not all states need to be visited. For example, all states with $|x-y| > k$ will not be reached before we reach (n, m) with edit distance k, since they require more than k edits. Thus, this BFS will touch at most $O(kn)$ cells, leading to $O(kn)$ work.

Another key observation is that starting from any cell (x, y), if there are diagonal edges with weight 0, we should always follow the edges until a unit-weight edge is encountered. Namely, we should always find the longest common prefix (LCP) from $A[x+1]$ and $B[y+1]$, and skip to the cell at $(x+p, y+p)$ with no edit, where p is the LCP length. This idea is used in Landau and Vishkin [37] on parallel approximate string matching, and we adapt this idea to edit distance here. Using the modified parallel BFS algorithm by Landau-Vishkin [37] (shown in Alg. 1), only $O(k^2)$ states need to be processed—on each diagonal and for each edit distance t, only the last cell with t edits needs to be processed (see Fig. 1). Hence, the BFS runs for k rounds on $2k+1$ diagonals, which gives the $O(k^2)$ bound above. In the BFS algorithm, we can label each diagonal by the value of $x-y$. In round t, the BFS visits a *frontier* of cells $f_t[\cdot]$, where $f_t[i]$ is the cell with edit distance t on diagonal i, for $-t \le i \le t$. We present the algorithm in Alg. 1 and an illustration in Fig. 1. Note that in the implementation, we only need to maintain two frontiers (the previous and the current one), which requires $O(k)$ space. We provide more details about this algorithm in the full version [16]. If the LCP query is supported by suffix arrays, we can achieve $O(n+k^2)$ work and $O(\log n + k \log k)$ span for the edit distance algorithm.

Algorithm Based on Suffix Array (BFS-SA). Using the SA algorithm in [32] and the LCP algorithm in [53] for Landau-Vishkin gives the claimed bounds in Tab. 1. We present details about our SA implementation in Sec. 5.1.

3.2 Algorithm Based on String Hashing (BFS-Hash)

Although BFS-SA is theoretically efficient with $O(n)$ preprocessing work to construct the SA, the hidden constant is large. For better performance, we consider string hashing as an alternative for SA. Similar attempts (e.g., locality-sensitive hashing) have also been used in approximate pattern matching problems [43, 44]. In our pursuit of exact output-sensitive

edit distance computation, we draw inspiration from established string hashing algorithms, such as the Rabin-Karp algorithm (also known as rolling hashing) [33]. We will first present a simple hash-based solution BFS-Hash with $O(n)$ preprocessing cost and $O(n)$ auxiliary space. Then later in Sec. 3.3, we will present BFS-B-Hash, which saves auxiliary space by trading off more work in LCP queries.

As mentioned in Sec. 2, the hash function $h(\cdot)$ maps any substring $A[l..r]$ to a unique hash value, which provides a fingerprint for this substring in the LCP query. The high-level idea is to binary search the query length, using the hash value as validation. We precompute the hash values for all prefixes, i.e., $T_A[x] = h(A[1..x])$ for the prefix substring $A[1..x]$ (similar for B). They can be computed in parallel by using any scan (prefix-sum) operation [6] with $O(n)$ work and $O(\log n)$ span. We can compute $h(A[l..r])$ by $T_A[r] \ominus T_A[l-1]$.

With the preprocessed hash values, we dual binary search the LCP of $A[x..n]$ and $B[y..m]$. We compare the hash values starting from $A[x]$ and $B[y]$ with chunk sizes of $1, 2, 4, 8, \ldots$, until we find value l, such that $A[x..x+2^l) = B[y..y+2^l)$, but $A[x..x+2^{l+1}) \neq B[y..y+2^{l+1})$. By doing this with $O(\log n)$ work, we know that the LCP of $A[x..n]$ and $B[y..m]$ must have a length in the range $[2^l, 2^{l+1})$. We then perform a regular binary search in this range, which costs another $O(\log n)$ work. This indicates $O(\log n)$ work in total per LCP query. Combining the preprocessing and query costs, we present the cost bounds of BFS-Hash:

▶ **Theorem 2.** *BFS-Hash computes the edit distance between two sequences of length n and $m \leq n$ in $O(n + k^2 \log n)$ work, $\tilde{O}(k)$ span, and $O(n)$ auxiliary space, where k is the output size (fewest possible edits).*

BFS-Hash is simple and easy to implement. Our experimental results indicate that its simplicity also allows for a reasonably good performance in practice for most real-world input instances. However, this algorithm uses n 64-bit integers as hash values, and such space overhead may be a concern in practice. This is more pronounced when the input is large and/or the alphabet is small (particularly when each input element can be represented with smaller than byte size), as the auxiliary space can be much larger than the input size. This concern also holds for BFS-SA as several $O(n)$-size arrays are needed during SA construction. Note that for shared-memory parallel algorithms, space consumption is also a *key constraint*—if an algorithm is slow, we can wait for longer; but if data (and auxiliary data) do not fit into the memory, then this algorithm is not applicable to large input at all. In this case, the problem size that is solvable by the algorithm is limited by the space overhead, which makes the improvement from parallelism much narrower. Below we will discuss how to make our edit distance algorithms more space efficient.

3.3 Algorithm Based on Blocked-Hashing (BFS-B-Hash)

In this section, we introduce our BFS-B-Hash algorithm that provides a more space-efficient solution by trading off worst-case time (work and span). Interestingly, we observed that on many data sets, BFS-B-Hash can even outperform BFS-Hash and other opponents due to faster construction time, and we will analyze that in Sec. 6.

To achieve better space usage, we divide the strings into blocks of size b. As such, we only need to store the hash values for prefixes of the entire blocks $h(A[1..b]), h(A[1..2b]), \cdots, h(A[1..\lfloor(n/b)\rfloor \cdot b])$. Our idea of blocking is inspired by many string algorithms (e.g., [4]). Using this approach, we only need auxiliary space to store $O(n/b)$ hash values, and thus we can control the space usage using the parameter b. To compute these hash values, we will first compute the hash value for each block, and run a parallel scan (prefix sum on

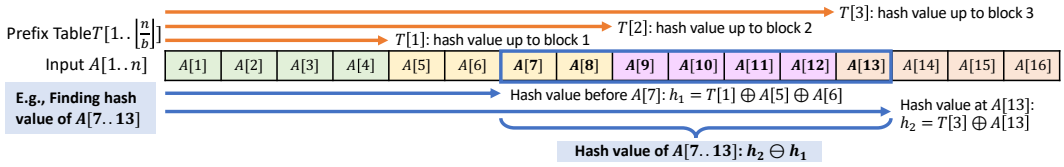


Figure 2 The illustrations of prefix table values and one specific query, with key concepts shown when computing the hash value of a range using a prefix table.

Algorithm 2 The prefix table for finding the longest common prefix of $A[1..n]$ and $B[1..m]$

```
// Table construction
Function Construct(A, B)
    T_A[·] ← Build(A)
    T_B[·] ← Build(B)
// The prefix table building process
Function Build(A)
    w ← ⌊|A|/b⌋
    T[0] ← 0
    parallel-for-each j ← 1 to w do
        T[j] ← h(A[(j − 1)b + 1 .. jb])
    Scan(T)
    return T[·]
// Get hash value for prefix sub-
// sequence A[1..x]
Function GetHash(A, T_A, x)
    if x = 0 then return 0
    r ← ⌊(x − 1)/b + 1⌋
    h̄ ← T_A[r]
    for i ← r · b + 1 to x do
        h̄ ← h̄ ⊕ h(A[i])
    return h̄
// Compare the subsequences A[x..x+l] and B[y..y+l]
Function Compare(A, B, x, y, l)
    h_A ← GetHash(A, T_A, x+l) ⊖ GetHash(A, T_A, x−1)
    h_B ← GetHash(B, T_B, y+l) ⊖ GetHash(B, T_B, y−1)
    return h_A = h_B
// Longest Common Prefix from A[x] and B[y]
Function LCP(A, B, x, y)
    l_1 ← 0
    // Find l_1, s.t. the LCP is between 2^{l_1} to 2^{l_1+1}
    blocks
    while x + 2^{l_1} < n and y + 2^{l_1} < m do
        if Compare(A, B, x, y, l_1) = false then break
        l_1 ← l_1 + 1
    // Trivial binary search process on the range
    [2^{l_1}, 2^{l_1+1})
    s ← 2^{l_1}, t ← 2^{l_1+1}
    while s < t do
        if Compare(A, B, x, y, ⌊(s + t)/2⌋) = false then
            t ← ⌊(s + t)/2⌋
        else s ← ⌊(s + t)/2⌋ + 1
    return s
```

$\oplus$) on the hash values for all the blocks. Similar to the above, we refer to these arrays as $T_A[i] = h(A[1..ib])$ (and $T_B[i]$ accordingly), and call them ***prefix tables***.

We now discuss how to run LCP with only partial hash values available. The *LCP* function in Alg. 2 presents the process to find the LCP of $A[x..n]$ and $B[y..m]$ using the prefix tables. We present an illustration in Fig. 2. We will use the same dual binary search approach to find the LCP of two strings. Since we do not store the hash values for all prefixes, we use a function $GetHash(A, T_A, x)$ to compute $h(A[1..x])$. We can locate the closest precomputed hash value and use r as the previous block id before x. Then the hash value up to block r is simply $\bar{h} = T_A[r]$. We then concatenate the rest characters to the hash value (i.e., return $\bar{h} \oplus h(A[rb + 1]) \oplus \cdots \oplus h(A[x])$). In this way, we can compute the hash value of any prefixes for both A and B, and plug this scheme into the dual binary search in BFS-Hash. In each step of dual binary search, the concatenation of hash value can have at most b steps, and thus leads to a factor of b overhead in query time than BFS-Hash.

▶ **Theorem 3.** *BFS-B-Hash computes the edit distance between two sequences of length n and $m \le n$ in $O(n + k^2 \cdot b \log n)$ work and $\tilde{O}(kb)$ span, using $O(n/b + k)$ auxiliary space, where k is the output size (fewest possible edits).*

The term k in space usage is from the BFS (each frontier is at most size $O(k)$). $O(b \log n)$ is the work for each LCP query. Note that this is an upper bound—if the LCP length L is small, the cost can be significantly smaller (a tighter bound is $O(\min(L, b \log L))$). Sec. 6 will show that for normal input strings where the LCP lengths are small in most queries,

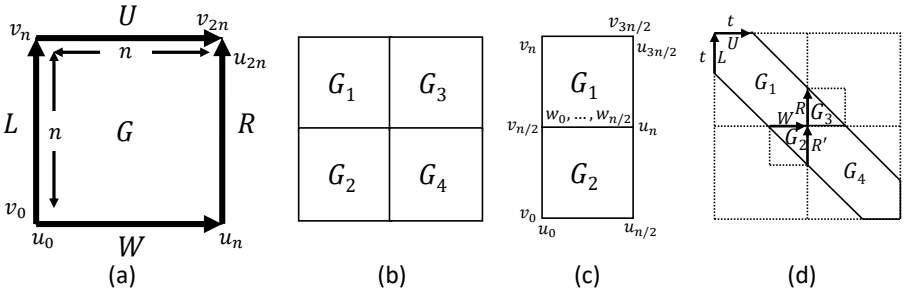


Figure 3 The illustrations of the key concepts and notation in the AALM algorithm described in Sec. 4.

the performance of BFS-B-Hash is indeed the fastest, although for certain input instances when the worst case is reached, the performance is not as good.

4 The Divide-and-Conquer Algorithms

Our parallel output-sensitive algorithm DAC-SD is inspired by the AALM algorithm [1], and also uses it as a subroutine. We first overview the AALM algorithm, and introduce our algorithm in details. We assume $m = n$ is a power of 2 in this section for simple descriptions, but both our algorithm and AALM work for any n and m.

The AALM Algorithm. As described above, the edit distance problem can be considered as a shortest distance (SD) problem from the top-left cell $(0, 0)$ to the bottom-right cell (n, n) in the DP matrix G. Instead of directly computing the SD from $(0, 0)$ to (n, n), AALM computes pairwise SD between any cell on the left/top boundaries and the bottom/right boundaries (i.e., those on $L \cup U$ to $W \cup R$ in Fig. 3(a)). We relabel all cells in $L \cup U$ as a sequence $v = \{v_0, v_1, \ldots v_{2n}\}$ (resp., $W \cup R$ as $u = \{u_0, u_1 \cdots, u_{2n}\}$), as shown in Fig. 3. Therefore, for the DP matrix G, the pairwise SD between v and u forms a $(2n+1) \times (2n+1)$ matrix. We call it the ***SD matrix*** of G, and denote it as D_G. AALM uses a divide-and-conquer approach. It first partitions G into four equal submatrices G_1, G_2, G_3, and G_4 (See Fig. 3(b)), and recursively computes the SD matrices for all G_i. We use D_i to denote the SD matrix for G_i. In the "conquer" step, the AALM algorithm uses a *Combine* subroutine to combine two SD matrices into one if they share a common boundary (our algorithm also uses this subroutine). For example, consider combining G_1 and G_2. We still use v_i and u_j to denote the cells on the left/top and bottom/right boundaries of $\begin{pmatrix} G_1 \\ G_2 \end{pmatrix}$ (see Fig. 3(c)), and denote the cells on the common boundary of G_1 and G_2 as $w_1, \cdots, w_{n/2}$, ordered from left to right. For any pair v_i and u_j, if they are in the same submatrix, we can directly get the SD from the corresponding SD matrix. Otherwise, WLOG assume $v_i \in G_1$ and $u_j \in G_2$, then we compute the SD between them by finding $\min_l D_1[i, l] + D_2[l, j]$, i.e., for all w_l on the common boundary, we attempt to use the SD between v_i to w_l, and w_l to u_j, and find the minimum one. Similarly, we can combine D_3 with D_4, and $D_{1\cup2}$ with $D_{3\cup4}$, and eventually get D_G. We note that the *Combine* algorithm, even theoretically, is highly involved. At a high level, it uses the Monge property of the shortest distance (the monotonicity of the DP recurrence), and we refer the readers to [1] for a detailed algorithm description and theoretical analysis. In Sec. 5.2, we highlight a few challenges and our solutions for implementing this highly complicated algorithm. Theoretically, combining two $n \times n$ SD matrices can be performed in $O(n^2)$ work and $O(\log^2 n)$ span, which gives $O(n^2 \log n)$ work and $O(\log^3 n)$ span for AALM.

Algorithm 3 Divide-and-Conquer edit distance algorithm on $A[1..n]$ and $B[1..n]$.

Notes: We assume both A and B has size $n = 2^c$ for simple description. Our algorithm also works for strings with different lengths with minor changes.
Function DAC-SD
  $t \leftarrow 1$
  while *true* **do**
    $D \leftarrow$ *Check*(t)
    if $D[t][t] \leq t$ **then break**
    $t \leftarrow \min(2t, n)$
  return $D[t][t]$
// Find the SD in the DP matrix from $(0,0)$ to (n,n) by restricting in the diagonal stripe with width t
Function *Check*(t)
  $D \leftarrow$ GETDISTANCE$(0, 0, n, t)$

Function GETDISTANCE(i, j, n, t)
  if $n/2 < t$ **then**
    Computed D by the AALM algorithm
    return D
  // Compute the SD matrices for G_1, G_2, G_3, G_4 (as shown in Fig. 3 (d)).
  $D_1 \leftarrow$ GETDISTANCE$(i, j, n/2, t)$
  Compute D_2 and D_3 by the AALM algorithm
  $D_4 \leftarrow$ GETDISTANCE$(i + n/2, j + n/2, n - n/2, t)$
  // use the same Combine function as AALM
  $D_{1\cup2} \leftarrow$ *Combine*(D_1, D_2)
  $D_{3\cup4} \leftarrow$ *Combine*(D_3, D_4)
  $D \leftarrow (D_{1\cup2}, D_{3\cup4})$
  return D

Our algorithm. The AALM algorithm has $\tilde{O}(n^2)$ work ($\tilde{O}(nm)$ if $n \neq m$) and polylogarithmic span, which is inefficient in the output-sensitive setting. As mentioned in Sec. 3.1, only a narrow width-$O(k)$ diagonal area in G is useful (Fig. 3(d)). We thus propose an output-sensitive DAC-SD algorithm adapted from the AALM algorithm. We follow the same steps in AALM, but restrict the paths to the diagonal area, although the exact size is unknown ahead of time. We first present the algorithm to compute the shortest distance on the diagonal region with width $2t + 1$ as function *Check*(t) in Alg. 3, which restricts the search in diagonals $-t$ to t. First, we divide such a region into four sub-regions (see Fig. 3(d)). Two of them (G_1 and G_4) are of the same shape, and the other two of them (G_2 and G_3) are triangles. For G_2 and G_3, we use the AALM algorithm to compute their SD matrices by aligning them to squares. For G_1 and G_4, we process them recursively, until the base case where the edge length of the matrix is smaller than t and they degenerate to squares, in which case we apply the AALM algorithm. Note that even though the width-$(2t+1)$ diagonal stripe is not a square (G_1 and G_4 are also of the same shape), the useful boundaries are still the left/top and bottom/right boundaries ($L \cup U$ and $W \cup R$ in Fig. 3(d)). Therefore, we can use the same *Combine* algorithm as in AALM to combine the SD matrices. For example, in Fig. 3(d), when combining G_1 with G_2, we obtain the pairwise distance between $L \cup U$ and $R \cup R'$ using the common boundary W. We can similarly combine all G_1, G_2, G_3, and G_4 to get the SD matrix for G.

However, the output value k is unknown before we run the algorithm. To overcome this issue, we use a strategy based on prefix doubling to "binary search" the value of k without asymptotically increasing the work of the algorithm. We start with $t = 1$, and run the *Check*(t) in Alg. 3 (i.e., restricting the search in a width-$(2t + 1)$ diagonal). Assume that the *Check* function returns σ edits. If $\sigma \leq t$, we know that σ is the SD from $(0, 0)$ to (n, n), since allowing the path to go out of the diagonal area will result in an answer greater than t. Otherwise, we know $\sigma > t$, and σ is not necessarily the shortest distance from the $(0, 0)$ to (n, n), since not restricting the path in the t-diagonal area may allow for a shorter path. If so, we double t and retry. Although we need $O(\log k)$ searches before finding the final answer k, we will show that the total search cost is asymptotically bounded by the last search. In the last search, we have $t < 2k$.

We first analyze the cost for *Check*(t). It contains two recursive calls, two calls to AALM, and three calls to the *Combine* function. Therefore, the work for *Check*(t) is $W(n) =$

$2W(n/2) + O(t^2 \log t)$, with base cases $W(t) = t^2 \log t$, which solves to $W(n) = O(nt \log t)$. For span, note that there are $\log(n/t)$ levels of recursion before reaching the base cases. In each level, the *Combine* function combines $t \times t$ SD matrices with $O(\log^2 t)$ span. In the leaf level, the base case uses AALM with $O(\log^3 t)$ span. Therefore, the total span of a *Check* is:

$$O(\log n/t \cdot \log^2 t + (\log n/t + \log^3 t)) = O(\log^2 t \cdot (\log n/t + \log t)) = O(\log^2 t \log n) \quad (1)$$

We will apply *Check*$(\cdot)$ for $O(\log k)$ times, with $t = 1, 2, 4, \ldots$ up to at most $2k$. Therefore, the total work is dominated by the last *Check*, which is $O(nk \log k)$. The span is $O(\log n \log^3 k)$.

▶ **Theorem 4.** *The DaC-SD algorithm computes the edit distance between two sequences of length n and $m \leq n$ in $O(nk \log k)$ work and $O(\log n \log^3 k)$ span, where k is the output size (fewest possible edits).*

Compared to the BFS-based algorithms with $\tilde{O}(k)$ span, our DaC-SD is also output-sensitive and achieves polylogarithmic span. However, the work is $\tilde{O}(kn)$ instead of $\tilde{O}(n + k^2)$, which will lead to more running time in practice for a moderate size of k.

5 Implementation Details

We provide all implementations for the four algorithms as well as testing benchmarks at [17]. In this section, we highlight some interesting and challenging parts of our implementations.

5.1 Implementation Details of BFS-based Algorithms

For the suffix array construction in BFS-SA, we implemented a parallel version of the DC3 algorithm [32]. We also compared our implementation with the SA implementation in ParlayLib [7], which is a highly optimized version of the prefix doubling algorithm with $O(n \log n)$ work and $O(\log^2 n)$ span. On average, our implementation is about $2\times$ faster than that in ParlayLib when applied to edit distance. We present some results for their comparisons in the full version [16]. For LCP array construction and preprocessing RMQ queries, we use the implementation in ParlayLib [7], which requires $O(n \log n)$ work and $O(\log^2 n)$ span. With them, the query has $O(1)$ cost.

In our experiments on both synthetic and real-world data, we observed that the LCP length is either very large when we find two long matched chunks, or in most of the cases, very short when they are not corresponding to each other. This is easy to understand—for genomes, text or code with certain edit history, it is unlikely that two random starting positions share a large common prefix. Based on this, we add a simple optimization for all LCP implementations such that we first compare the leading eight characters, and only when they all match, we use the regular LCP query. This simple optimization greatly improved the performance of BFS-SA, and also slightly improved the hash-based solutions.

5.2 Implementation Details of the DaC-SD Algorithm

Although our DaC-SD algorithm given in Alg. 3 is not complicated, we note that implementing it is highly non-trivial in two aspects. First, in Sec. 4, we assume both strings A and B have the same length n, which is a power of two. However, handling two strings with different lengths makes the matrix partition more complicated in practice. Another key challenge is that the combining step in the AALM algorithm is recursive and needs to allocate memory with varying sizes in the recursive execution. While memory allocation is mostly ignored theoretically, frequent allocation in practice can easily be the performance bottleneck in the parallel setting. We discuss our engineering efforts as follows.

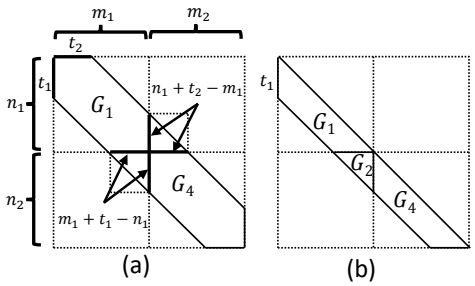


Figure 4 The illustrations of our output-sensitive DaC-SD algorithm. (a) Two parameters t_1 and t_2 are needed to denote the lengths of the diagonal area on each side. (b) The case that $t_2 = 0$ and G_3 degenerates.

Data	Alias	$\|A\|$	$\|B\|$	k	$\|\Sigma\|$
Wikipedia pages [45]	Wiki v1	0.56M	0.56M	439	256
	Wiki v2	0.56M	0.56M	5578	256
	Wiki v3	0.56M	0.55M	15026	256
Linux kernel code [40]	Linux v1	6.47M	6.47M	236	256
	Linux v2	6.47M	6.47M	1447	256
	Linux v3	6.47M	6.46M	9559	256
DNA sequences [5]	DNA 1	42.3M	42.3M	928	4
	DNA 2	42.3M	42.3M	9162	4
	DNA 3	42.3M	42.3M	91419	4

Table 2 Real-world datasets in our experiments, including input sizes $|A|$ and $|B|$, number of edits k, and alphabet sizes $|\Sigma|$.

Irregularity. The general case, when n and m are not powers of two and not the same, is more complicated than the case in Alg. 3. In this case, all four subproblems G_1, G_2, G_3, and G_4 will have different sizes. While theoretically, we can always round up, for better performance in practice, we need to introduce additional parameters to restrict the search within the belt region as shown in Fig. 4. Therefore, we use two parameters t_1 and t_2, to denote the lengths of the diagonal area on each side. We show an illustration in Fig. 4(a) along with how to compute the subproblem sizes. In extreme cases, t_1 or t_2 can degenerate to 0, which results in three subproblems (Fig. 4(b)). In such cases, we will first merge G_2 and G_4, then merge G_1 and $G_{2\cup 4}$.

The Combining Step. As mentioned in Sec. 4, achieving an efficient combining step is highly non-trivial. The straightforward solution to combine two matrices is to use the Floyd-Warshall algorithm [18], but it incurs $O(n^3)$ work and will be a bottleneck. The AALM algorithm improves this step to $O(n^2)$ by taking advantage of the Monge property of the two matrices. For page limit, we introduce the details of the combining algorithm in the full version [16]. However, the original AALM algorithm is based on divide-and-conquer and requires memory allocation for every recursive function call. This is impractical as frequent parallel memory allocation is extremely inefficient. To overcome this challenge, we redesign the recursive solution to an iterative solution, such that we can preallocate the memory space before the combining step. No dynamic memory allocation is involved during the computation. We provide the details of this approach in the full version [16].

6 Experiments

Setup. We implemented all algorithms in C++ using ParlayLib [7] for fork-join parallelism and some parallel primitives (e.g., reduce). Our tests use a 96-core (192 hyperthreads) machine with four Intel Xeon Gold 6252 CPUs, and 1.5 TB of main memory. We utilize `numactl -i all` in tests with more than one thread to spread the memory pages across CPUs in a round-robin fashion. We run each test three times and report the median.

Tested Algorithms and Datasets. We tested five algorithms in total: four output-sensitive algorithms in this paper (BFS-SA, BFS-Hash, BFS-B-Hash, DaC-SD), and a baseline algorithm from ParlayLib [7], which is a parallel output-insensitive implementation with $O(nm)$ work. The ParlayLib implementation is intended to showcase the simplicity of parallel algorithms, and as a result, it may not be well-ptimized. We are unaware of other

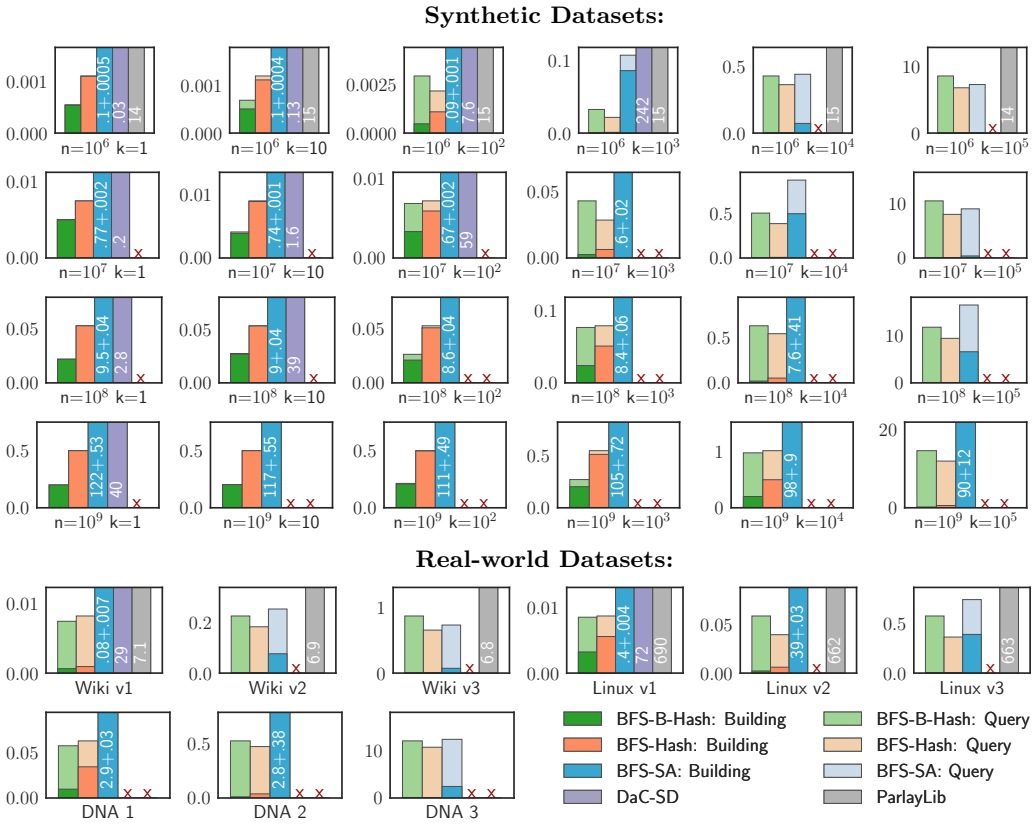
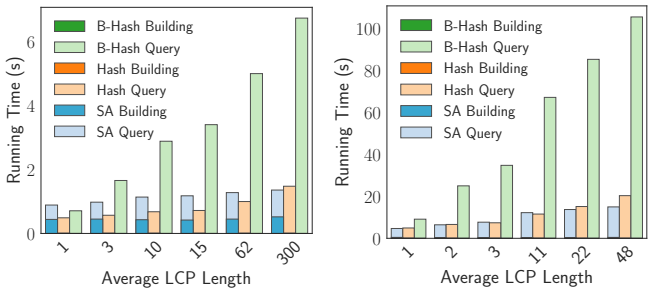


Figure 5 Running time (in seconds) of synthetic and real-world datasets for all algorithms. Lower is better. We put an "×" if the algorithm does not finish within 1000 seconds. For BFS-based algorithms, we separate the time into building time (constructing the data structure for LCP queries) and query time (running BFS). All bars out of the range of the y-axis are annotated with numbers. The number is the total running time for DaC-SD and ParlayLib, and is in the format of $a + b$ for BFS-SA, where a is the building time and b is the query time. Full results are presented in the full version [16].

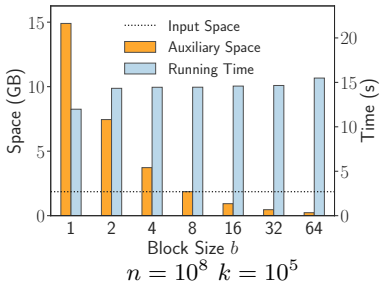
parallel implementations that provide output-sensitive cost bounds. We use $b = 32$ for our BFS-B-Hash. As we will show later, the running time is generally stable with $4 \leq b \leq 64$. We tested the algorithms on both synthetic and real-world datasets. For synthetic datasets, we generate random strings with different string lengths $n = 10^i$ for $6 \leq i \leq 9$ and k (number of edits) varying from 1 to 10^5, and set the size of the alphabet as 256. We create strings A and B by generating n random characters, and applying k edits. The k edits are uniformly random for insertion, deletion and substitution. For $k \ll n$, we have $m \approx n$. All values of k shown in the figures and tables are approximate values. Our real-world datasets include Wikipedia [45], Linux kernel [40], and DNA sequences [51]. We compare the edit distance between history pages on Wikipedia and history commits of a Linux kernel file on GitHub. We also compare DNA sequences by adding valid modifications to them to simulate DNA damage or genome editing techniques, as is used in many existing papers [11, 13, 30, 57]. We present the statistics of the real-world datasets in Tab. 2.

Overall Performance on Synthetic Data. We present our results on synthetic data in the upper part of Fig. 5. We also present the complete results in the full version [16]. For BFS-based algorithms, we also separate the time for ***building*** the data structures for LCP queries, and the ***query*** time (the BFS process). ParlayLib cannot process instances with $n > 10^6$ due to its $O(nm)$ work bound.



(a) $n = 10^7$ $k = 10000$ (b) $n = 10^7$ $k = 50000$

Figure 6 Performance of BFS-based algorithms vs. average LCP length. Some building times are invisible because they are too small.



$n = 10^8$ $k = 10^5$

Figure 7 Time-space trade-off in BFS-B-Hash. The space shown is the memory required for the prefix tables. The dotted line is the input size. Note that by setting $b = 1$, the algorithm is equivalent to BFS-Hash.

We first *compare our solutions with ParlayLib* [7]. Since ParlayLib is not output-sensitive, its running time remains the same regardless of the value of k. Among the tests that ParlayLib can process ($n = 10^6$), our output-sensitive algorithms are much faster than ParlayLib, especially when k is small (up to $10^5\times$). For $n = 10^6$, all our BFS-based algorithms are at least $1.7\times$ faster than ParlayLib even when $k \approx n/10$.

We then *compare our DaC- and BFS-based solutions.* DaC-SD has the benefit of polylogarithmic span, compared to $\tilde{O}(k)$ span for the BFS-based algorithm. Although this seems to suggest that DaC-SD should have better performance when k is large, the result shows the opposite. The reason is that DaC-SD has $\tilde{O}(nk)$ work, compared to $\tilde{O}(n + k^2)$ cost of the BFS-based algorithms. When k becomes larger, the overhead in work is also more significant. On the other hand, when k is small, the $O(nk)$ work becomes linear, which hides the inefficiency in work. Therefore, the gap between DaC-SD and other algorithms is smaller when k is small, but DaC-SD is still slower than BFS-based algorithms in all test cases, especially when k is large. This experiment reaffirms the *importance of work efficiency* on practical performance for parallel algorithms.

Finally, we *compare all our BFS-based solutions.* Our hash-based solutions have significant advantages over the other implementations when k is small, since the pre-processing time for hash-based solutions is much shorter. When k is large, pre-processing time becomes negligible, and BFS-Hash seems to be the ideal choice since its query is also efficient. In particular, for $n \approx m \approx 10^9$, hash-based algorithms use about 1 second for pre-processing while BFS-SA uses about 100 seconds. Although BFS-SA also has $O(n)$ construction time, the constant is much larger and its memory access pattern is much worse than the two hash-based solutions. We note that in some cases, the query time of BFS-SA can still be faster than BFS-Hash and BFS-B-Hash, especially when k is large, which is consistent with the theory ($O(1)$ vs. $O(\log n)$ or $O(b \log n)$ per LCP query).

In theory, BFS-B-Hash reduces space usage in BFS-Hash by increasing the query time. Interestingly, when k is small, BFS-B-Hash can also be faster than BFS-Hash by up to $2.5\times$. This is because BFS-B-Hash incurs fewer writes (and thus smaller memory footprints) in preprocessing that leads to faster building time. When k is small, the running time is mostly dominated by the building time, and thus BFS-B-Hash can perform better. When k is relatively large and k^2 is comparable to n, BFS-Hash becomes faster than BFS-B-Hash due to better LCP efficiency. In fact, when k is large, the running time is mainly dominated by the query (BFS), and all three algorithms behave similarly. It is worth noting that in these experiments with $|\Sigma| = 256$ and random edits, in most of the cases, the queried LCP is

small. Therefore, the $O(\log n)$ or $O(b \log n)$ query time for BFS-Hash and BFS-B-Hash are not tight, and they have much better memory access patterns than BFS-SA in LCP queries. As a result, they can have matching or even better performance than BFS-SA. Later we will show that under certain input distributions where the average LCP length is large, BFS-SA can have some advantage over both BFS-Hash and BFS-B-Hash.

Real-World Datasets. We now analyze how our algorithms perform on real-world string and edit patterns. The results are shown in the lower part of Fig. 5. The results are mostly consistent with our synthetic datasets, where BFS-B-Hash is more advantageous when k is small, and BFS-Hash performs the best when k is large. When k is large, BFS-SA can also have comparable performance to the hash-based solutions.

LCP Length vs. Performance. It seems that for both synthetic and real-world data shown above, our hash-based solutions are always better than BFS-SA. It is worth asking, whether BFS-SA can give the best performance in certain cases, given that it has the best theoretical bounds (see Tab. 1). By investigating the bounds carefully, BFS-SA has better LCP query cost as $O(1)$, while the costs for BFS-Hash and BFS-B-Hash are $O(\log L)$ and $O(b \log L)$, respectively, where L is the LCP length. This indicates that BFS-SA should be advantageous when k and L are both large. To verify this, we artificially created input instances with medium to large values of k and controlled average LCP query lengths, and showed the results in Fig. 6 on two specific settings.

The experimental result is consistent with the theoretical analysis. The running time for BFS-Hash increases slowly with L, while the performance of BFS-B-Hash grows *much faster*, since it is affected by a factor of $O(b)$ more than BFS-Hash. The query time for BFS-SA almost stays the same, but also increases slightly with increasing L. This is because in general, with increasing L, the running time for all three algorithms may increase slightly due to worse cache locality in BFS due to more long matches. In Figure 6(a), the building time for both BFS-Hash and BFS-B-Hash are negligible, while BFS-SA still incurs significant building time. Even in this case, with an LCP length of 300, the query time of the hash-based solutions still becomes larger than the *total* running time of BFS-SA. In Figure 6(b) with a larger k, the building time for all three algorithms is negligible. In this case, BFS-SA always has comparable performance with BFS-Hash, and may perform better when $L > 20$. However, such extreme cases (both k and L are large) should be very rare in real-world datasets - when k is large enough so that the query time is large enough to hide SA's building time, L is more likely to be small, which in turn is beneficial for the query bounds in hash-based solutions. Indeed such cases did not appear in our 33 tests on both synthetic and real data.

Parallelism. We test the self-relative speedup of all algorithms. We present speedup numbers on two representative tests with different values of n and k in Tab. 3. For BFS-based algorithms, we separate the speedup for building and query. All our algorithms are highly parallelized. Even though BFS-SA and DaC-SD have a longer running time, they still have a 48–68× speedup, indicating good scalability. Our BFS-Hash algorithm has about 40–50× speedup in building, and BFS-B-Hash has a lower but decent speedup of about 20–40×. When k is small, the frontier sizes (and the total work) of BFS are small, and the running time is also negligible. In this case, we cannot observe meaningful speedup. For larger $k = 10^5$, three BFS-based algorithms achieve 27–48× speedup both in query and entire edit distance algorithm.

Space Usage. We study the time-space tradeoff of our BFS-B-Hash with different block sizes b. We present the *auxiliary space* used by the prefix table in BFS-B-Hash along with running time in Fig. 7 using one test case with $n = 10^8$ and $k = 10^5$ in our synthetic dataset.

n	k	BFS-B-Hash			BFS-Hash			BFS-SA			DaC-SD
		Build	Query	Total	Build	Query	Total	Build	Query	Total	Total
10^8	10	20.4	-	19.9	46.6	-	46.5	49.6	-	49.4	68.2
10^9	10^5	24.2	36.4	36.3	42.7	46.8	46.6	51.2	27.1	48.3	t.o.

Table 3 Self-relative speedup of each implementation in each step. "Build" = constructing the data structure for LCP queries. "Query" = the BFS process. "t.o." = timeout. We omit query speedup when $k = 10$ because there is little parallelism to be explored for BFS with small k, and the BFS time is also small and hardly affects the overall speedup. 192 hyperthreads are used for parallel executions.

The dotted line shows the input size. Note that when $b = 1$, it is exactly BFS-Hash. Since the inputs are 8-bit characters and the hash values are 64-bit integers, BFS-Hash incurs 8× space overhead than the input size. Using blocking, we can avoid such overhead and keep the auxiliary space even lower than the input. The auxiliary space decreases linearly with the block size b. Interestingly, although blocking itself incurs time overhead, the impact in time is small: the time grows by 1.19× from $b = 1$ to 2, and grows by 1.08× from $b = 2$ to 64. This is mostly due to two reasons: 1) as mentioned, with 8-bit character input type and random edits, the average LCP length is likely short and within the first block, and therefore the query costs in both approaches are close to $O(L)$ for LCP length L, and 2) the extra factor of b in queries (Line 17) is mostly cache hits (consecutive locations in an array). This illustrates the benefit of using blocking in such datasets, since blocking saves much space while only increasing the time by a small fraction.

7 Conclusion and Discussions

We proposed output-sensitive parallel algorithms for the edit-distance problem, as well as careful engineering of them. We revisited the BFS-based Landau-Vishkin algorithm. In addition to using SA as is used in Landau-Vishkin (our BFS-SA implementation), we also designed two hash-based data structures to replace the SA for more practical LCP queries (BFS-Hash and BFS-B-Hash). We also presented the first output-sensitive parallel algorithm based on divide-and-conquer with $\tilde{O}(nk)$ work and polylogarithmic span. We have also shown the best of our engineering effort on this algorithm, although its performance seems less competitive than other candidates due to work inefficiency.

We implemented all these algorithms and tested them on synthetic and real-world datasets. In summary, our BFS-based solutions show the best overall performance on datasets with real-world edits or random edits, due to faster preprocessing time and better I/O-friendliness. BFS-Hash performs the best in time when k is large. BFS-B-Hash has better performance when k is small. The blocking scheme also greatly improves space efficiency without introducing much overhead in time. In very extreme cases where both k and the LCP lengths are large, BFS-SA can have some advantages over the hash-based solutions, while BFS-B-Hash can be much slower than BFS-Hash. However, such input patterns seem rare in the real world.

All our BFS-based solutions perform better than the output-insensitive solution in ParlayLib, and the DaC-based solution with $\tilde{O}(nk)$ work and polylogarithmic span, even for large $k > \sqrt{n}$. The results also imply the importance of work efficiency in parallel algorithm designs, consistent with the common belief in the literature [52, 24]. Because the number of cores in modern multi-core machines is small (usually hundreds to thousands) compared to the problem size, an algorithm is less practical if it blows up the work significantly, as parallelism cannot compensate for the performance loss due to larger work.

References

1. Alberto Apostolico, Mikhail J Atallah, Lawrence L Larmore, and Scott McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM J. on Computing*, 19(5):968–988, 1990.
2. Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)*, 34(2):115–144, 2001.
3. K Nandan Babu and Sanjeev Saxena. Parallel algorithms for the longest common subsequence problem. In *IEEE International Conference on High Performance Computing (HiPC)*, pages 120–125. IEEE, 1997.
4. Michael A. Bender and Martin Farach-Colton. The lca problem revisited. In *Latin American Symposium on Theoretical Informatics (LATIN)*, pages 88–94. Springer, 2000.
5. Dennis A Benson, Mark Cavanaugh, Karen Clark, Ilene Karsch-Mizrachi, David J Lipman, James Ostell, and Eric W Sayers. Genbank. *Nucleic acids research*, 41(D1):D36–D42, 2012.
6. Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Comput.*, 38(11), 1989.
7. Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. Parlaylib — a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 507–509, 2020.
8. Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 89–102, 2020.
9. Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. on Computing*, 27(1), 1998.
10. Nicholas Boucher, Ilia Shumailov, Ross Anderson, and Nicolas Papernot. Bad characters: Imperceptible nlp attacks. In *IEEE Symposium on Security and Privacy (SP)*, pages 1987–2004. IEEE, 2022.
11. Dana Carroll. Focus: genome editing: genome editing: past, present, and future. *The Yale journal of biology and medicine*, 90(4):653, 2017.
12. Jung Hee Cheon, Miran Kim, and Kristin Lauter. Homomorphic computation of edit distance. In *International Conference on Financial Cryptography and Data Security*, pages 194–212. Springer, 2015.
13. Kendell Clement, Holly Rees, Matthew C Canver, Jason M Gehrke, Rick Farouni, Jonathan Y Hsu, Mitchel A Cole, David R Liu, J Keith Joung, Daniel E Bauer, et al. Crispresso2 provides accurate and rapid genome editing sequence analysis. *Nature biotechnology*, 37(3):224–226, 2019.
14. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009.
15. Sanjoy Dasgupta, Christos H Papadimitriou, and Umesh Virkumar Vazirani. *Algorithms*. McGraw-Hill Higher Education New York, 2008.
16. Xiangyun Ding, Xiaojun Dong, Yan Gu, Yihan Sun, and Youzhe Liu. Efficient parallel output-sensitive edit distance. *arXiv preprint:2306.17461*, 2023.
17. Xiangyun Ding, Xiaojun Dong, Yan Gu, Yihan Sun, and Youzhe Liu. Parallel implementations for output-sensitive edit distance. `https://github.com/ucrparlay/Edit-Distance`, 2023.
18. Robert W Floyd. Algorithm 97: shortest path. *Commun. ACM*, 5(6):345, 1962.
19. Zvi Galil and Raffaele Giancarlo. Improved string matching with k mismatches. *ACM SIGACT News*, 17(4):52–54, 1986.
20. Zvi Galil and Raffaele Giancarlo. Parallel string matching with k mismatches. *Theoretical Computer Science (TCS)*, 51(3):341–348, 1987.
21. Zvi Galil and Raffaele Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4(1):33–72, 1988.
22. Zvi Galil and Kunsoo Park. An improved algorithm for approximate string matching. *SIAM Journal on Computing*, 19(6):989–999, 1990.

23 Michael T Goodrich and Roberto Tamassia. *Algorithm design and applications*. Wiley Hoboken, 2015.

24 Yan Gu, Ziyang Men, Zheqi Shen, Yihan Sun, and Zijin Wan. Parallel longest increasing subsequence and van emde boas trees. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2023.

25 Yan Gu, Zachary Napier, and Yihan Sun. Analysis of work-stealing and parallel cache complexity. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 46–60. SIAM, 2022.

26 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984.

27 Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.

28 Daniel Hládek, Ján Staš, and Matúš Pleva. Survey of automatic spelling correction. *Electronics*, 9(10):1670, 2020.

29 Md Mosabbir Hossain, Md Farhan Labib, Ahmed Sady Rifat, Amit Kumar Das, and Monira Mukta. Auto-correction of english to bengali transliteration system using levenshtein distance. In *International Conference on Smart Computing & Communications (ICSCC)*, pages 1–5. IEEE, 2019.

30 Yoon-Seong Jeon, Kihyun Lee, Sang-Cheol Park, Bong-Soo Kim, Yong-Joon Cho, Sung-Min Ha, and Jongsik Chun. Ezeditor: a versatile sequence alignment editor for both rrna-and protein-coding genes. *International journal of systematic and evolutionary microbiology*, 64(Pt_2):689–691, 2014.

31 Tao Jiang, Guohui Lin, Bin Ma, and Kaizhong Zhang. A general edit distance between RNA structures. *Journal of Computational Biology*, 9(2):371–388, 2002.

32 Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*, pages 943–955. Springer, 2003.

33 Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM journal of research and development*, 31(2):249–260, 1987.

34 Peter Krusche and Alexander Tiskin. New algorithms for efficient parallel string comparison. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 209–216, 2010.

35 Gad M Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science (TCS)*, 43:239–249, 1986.

36 Gad M Landau and Uzi Vishkin. Fast string matching with k differences. *J. Computer and System Sciences*, 37(1):63–78, 1988.

37 Gad M Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, 1989.

38 Vladimir I Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

39 Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in bioinformatics*, 11(5):473–483, 2010.

40 Linux Kernel File dcn_1_0_sh_mask.h Commit History on GitHub. `https://github.com/torvalds/linux/blob/master/drivers/gpu/drm/amd/include/asic_reg/dcn/dcn_1_0_sh_mask.h`.

41 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.

42 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. on Computing*, 22(5):935–948, 1993.

43 Guillaume Marçais, Dan DeBlasio, Prashant Pandey, and Carl Kingsford. Locality-sensitive hashing for the edit distance. *Bioinformatics*, 35(14):i127–i135, 2019.

695 **44** Samuel McCauley. Approximate Similarity Search Under Edit Distance Using Locality-
696 Sensitive Hashing. In *24th International Conference on Database Theory (ICDT 2021)*, volume
697 186, pages 21:1–21:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.
698 **45** Municipal history of Quebec on Wikipedia. `https://en.wikipedia.org/wiki/Municipal_`
699 `history_of_Quebec`.
700 **46** Eugene W Myers. An o (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–
701 266, 1986.
702 **47** Eugene Wimberly Myers. *Incremental alignment algorithms and their applications*. University
703 of Arizona, Department of Computer Science, 1986.
704 **48** Jean-Frédéric Myoupo and David Seme. Time-efficient parallel algorithms for the longest
705 common subsequence and related problems. *J. Parallel Distrib. Comput.*, 57(2):212–223, 1999.
706 **49** Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys*
707 *(CSUR)*, 33(1):31–88, 2001.
708 **50** Mike Paterson and Vlado Dančík. Longest common subsequences. In *International Symposium*
709 *on Mathematical Foundations of Computer Science*, pages 127–142. Springer, 1994.
710 **51** Jane Peterson, Susan Garges, Maria Giovanni, Pamela McInnes, Lu Wang, Jeffery A Schloss,
711 Vivien Bonazzi, Jean E McEwen, Kris A Wetterstrand, Carolyn Deal, et al. The nih human
712 microbiome project. *Genome research*, 19(12):2317–2323, 2009.
713 **52** Zheqi Shen, Zijin Wan, Yan Gu, and Yihan Sun. Many sequential iterative algorithms can be
714 parallel and (nearly) work-efficient. In *ACM Symposium on Parallelism in Algorithms and*
715 *Architectures (SPAA)*, 2022.
716 **53** Julian Shun. Fast parallel computation of longest common prefixes. In *International Conference*
717 *for High Performance Computing, Networking, Storage, and Analysis (SC)*, pages 387–398.
718 IEEE, 2014.
719 **54** Diomidis Spinellis. Git. *IEEE software*, 29(3):100–101, 2012.
720 **55** Vianney Kengne Tchendji, Armel Nkonjoh Ngomade, Jerry Lacmou Zeutouo, and Jean Frédéric
721 Myoupo. Efficient cgm-based parallel algorithms for the longest common subsequence problem
722 with multiple substring-exclusion constraints. *Parallel Computing*, 91:102598, 2020.
723 **56** Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*,
724 64(1-3):100–118, 1985.
725 **57** Liang-Jiao Xue and Chung-Jui Tsai. Ageseq: analysis of genome editing by sequencing.
726 *Molecular plant*, 8(9):1428–1430, 2015.
727 **58** Jiaoyun Yang, Yun Xu, and Yi Shang. An efficient parallel algorithm for longest common
728 subsequence problem on GPUs. In *World Congress on Engineering*, volume 1, pages 499–504,
729 2010.
730 **59** Hongyu Zhang. Alignment of blast high-scoring segment pairs based on the longest increasing
731 subsequence algorithm. *Bioinformatics*, 19(11):1391–1396, 2003.