Efficient Parallel Algorithms for String Comparison

Nikita Mishin*
Saint Petersburg State University
Saint Petersburg, Russia
mishinnikitam@protonmail.com

Daniil Berezun*
Saint Petersburg State University
Saint Petersburg, Russia
d.berezun@2009.spbu.ru

Alexander Tiskin
Saint Petersburg State University
Saint Petersburg, Russia
a.tiskin@spbu.ru

ABSTRACT

The longest common subsequence (LCS) problem on a pair of strings is a classical problem in string algorithms. Its extension, the semi-local LCS problem, provides a more detailed comparison of the input strings, without any increase in asymptotic running time. Several semi-local LCS algorithms have been proposed previously; however, to the best of our knowledge, none have yet been implemented. In this paper, we explore a new hybrid approach to the semi-local LCS problem. We also propose a novel bit-parallel LCS algorithm. In the experimental part of the paper, we present an implementation of several existing and new parallel LCS algorithms and evaluate their performance.

CCS CONCEPTS

• **Theory of computation → Parallel algorithms**; *Algorithm design techniques*; *Dynamic programming*; *Divide and conquer*.

KEYWORDS

string algorithms, longest common subsequence, semi-local string comparison, parallel algorithms, divide-and-conquer, dynamic programming, braid multiplication, parallel braid multiplication, bit-parallel algorithms

ACM Reference Format:

Nikita Mishin, Daniil Berezun, and Alexander Tiskin. 2021. Efficient Parallel Algorithms for String Comparison. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The longest common subsequence (LCS) problem on a pair of strings is a classical problem in string algorithms. Its standard solution is based on straightforward dynamic programming [27]. Its extension, the semi-local LCS problem, provides a more detailed comparison of the input strings, without any increase in asymptotic running time. Several semi-local LCS algorithms have been proposed previously based on computations with an algebraic structure known as *sticky braid*. In particular, a sticky braid corresponding to comparison of a pair of input strings can be constructed either iteratively (*iterative combing*) or recursively (*recursive combing*) [23]. However, to our knowledge, none of these algorithms have yet been implemented. In this paper, we explore a hybrid approach, combining iterative and recursive combing. We also propose a novel bit-parallel LCS algorithm, free of integer arithmetic and the associated carry propagation delays that are typical of existing bit-parallel LCS algorithms. Furthermore, we present an implementation of several LCS algorithms, including recursive and iterative combing, as well as their parallel versions using thread-level parallelism, and intra-processor SIMD subword and bit parallelism, with a number of optimizations.

For experimental evaluation of the presented algorithms we use two types of input: randomly generated strings and a real-life dataset of virus genomes. Our experiments show that the running times of our implementations of semi-local LCS algorithms correspond to their theoretical estimations with no extra overheads and are comparable to an implementation of standard LCS. Thus, the algorithms have acceptable running times and are practically applicable. Finally, we show that these algorithms have good potential for parallelization and for practical usage in real-life data analysis.

*Also with JetBrains Research.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 RELATED WORK

Classical dynamic programming algorithms for the LCS problem and the closely related edit distance problem were developed by Levenshtein [16], Wagner and Fischer [27], Hirschberg [11], Masek and Paterson [17]. Approximate pattern matching by edit distance, which is essentially a form of semi-local string comparison, was studied by Sellers [22], Landau and Vishkin [15], Cole and Hariharan [9]. Significant advances in approximate matching algorithms have been made recently by Charalampopoulos et al. [7].

The most natural approach to the parallelization of dynamic programming consists in iterating over the dynamic programming grid in anti-diagonals of independent cells. An alternative approach, presented by Aluru et al. [1], iterates over the grid in horizontal or vertical tiles of cells, which are updated by a parallel prefix subroutine. The ingenuous bit-parallel LCS algorithms by Crochemore et al. [10] and Hyyrö [12] also iterate in vertical or horizontal tiles, relying on efficient parallel hardware adders for a tile update; we note that the design of such adders can also be expressed as a parallel prefix computation, see e.g. [18].

A closely related topic to parallelilzation is cache-efficient computation. A cache-oblivious version of dynamic programming was proposed by Chowdhury and Ramachandran [8].

The type of algorithms presented in this work originates with the local LCS algorithm of Schmidt [21], which was adapted for the string-substring LCS problem by Alves at al. [4], who also developed coarse-grained parallel algorithms for this problem in [2, 3].

The connection between the semi-local LCS problem and the algebraic structure of sticky braids was exposed by Tiskin [23], and a fast algorithm for sticky braid multiplication was developed independently by Tiskin [24] and Sakai [20]. A detailed study of the semi-local LCS problem and its applications was presented by P. Krusche in his Ph.D. thesis [14], along with a detailed presentation of the author's implementations of several algorithms. However, these results are a bit out of date now, as the technology and the research field have moved forward. Moreover, while the focus of [14] is on applications of the semi-local LCS problem to various scientific problems, our study focuses on the development and optimization of semi-local LCS algorithms as such, as well as their comprehensive evaluation on strings of different length and composition.

Russo [19] explored further the problem of sticky braid multiplication, and evaluated several algorithms for that problem. Although some experimental analysis of the algorithms' implementation is present in that work, it is unclear how thoroughly the algorithms were optimized, and how their scalability and performance can be compared against other approaches to semi-local LCS.

Recently, Tiskin [25] presented several new parallel semi-local LCS algorithms in the bulk-synchronous parallelism (BSP) model due to Valiant [26]. These algorithms are based on a parallel version of sticky braid multiplication. In the current work, we explore whether this approach can compete with previous algorithms for semi-local LCS, and provide practical performance for comparison of large strings. We also study what optimizations and tradeoffs are possible in an implementation of semi-local LCS algorithms.

3 SEMI-LOCAL LCS

In this section we provide a brief explanation of the semi-local LCS problem. For a detailed study of mathematics behind semi-local LCS see for example [23].

Hereafter, we denote by m and n the lengths of strings a and b respectively. We denote by $a[i:j)$ a substring of string a of length $j-i$, that starts at position i and ends at j exclusive (thus $a[i]$ stands for $a[i:i+1)$), and by $LCS(a,b)$ the length of the longest common subsequence (*LCS score*) of a, b.

Definition 3.1. Matrix M is called a *(sub)permutation* matrix if all its elements are zeros or ones and there are exactly (at most) one non-zero in each row and each column.

Definition 3.2. The *semi-local LCS* problem asks for *LCS scores* as follows [23]:

- **string-substring**: whole a against every substring of b,
- **substring-string**: whole b against every substring of a,
- **prefix-suffix**: every prefix of a against every suffix of b,
- **suffix-prefix**: every prefix of b against every suffix of a.

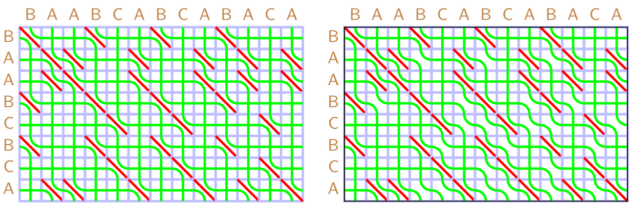
The solution of this problem is presented as square matrix $H_{a,b}$, called *LCS matrix*, where each quadrant contains a solution for each of the above sub-problems:

$$H_{a,b} = \begin{bmatrix} \textit{suffix-prefix} & \textit{substring-string} \\ \textit{string-substring} & \textit{prefix-suffix} \end{bmatrix} \quad (1)$$

This matrix is defined as follows:

Definition 3.3. [23] The *LCS matrix* $H_{a,b}[i,j]$ of size $(m+n+1)\times(m+n+1)$ is defined by:

$$H[i,j] = \begin{cases} LCS(a, b^{pad}[i:j+m)) & i < j+m \\ j+m-i & \text{otherwise} \end{cases}$$



(a) Initial unreduced sticky braid (b) Reduced sticky braid

Figure 1: Semi-local LCS solution represented by a sticky braid

where $i, j \in [0:m+n]$, $b^{pad} := ?^m b ?^m$ and ? is a wildcard character that matches any other character

Note that the naïve algorithm for solving the problem immediately follows from this definition — an independent computation of each matrix cell, which in the worst-case gives $O((n+m)^2) \times O((n+m)^2) = O((n+m)^4)$ complexity. Nonetheless, there are a number of algorithms that solve the semi-local LCS problem in time $O(mn)$, i.e. the same asymptotic time as the ordinary LCS problem. These algorithms are based on several fascinating mathematical properties of matrix $H_{a,b}$.

It is proved in [23], that matrix $H_{a,b}$ can be represented implicitly by a permutation matrix $P_{a,b}$, called *semi-local LCS kernel*. This allows us to store the solution of the semi-local LCS problem in linear memory, but the time complexity of accessing an arbitrary element of the output matrix grows from constant to polylogarithmic[1]. Kernel $P_{a,b}$ represents an implicit solution of semi-local LCS, and is associated with an algebraic object called *reduced sticky braid* of order $m+n$. In this paper, we will not consider the algebraic nature of sticky braids, and will instead use them as just a visual aid. A sticky braid consists of $m+n$ monotone curves, called *strands*. Any pair of neighboring strands can form a crossing; furthermore, a sticky braid is called *reduced*, if any given pair of strands cross at most once. Figure 1(a) shows original unreduced sticky braid for specific input strings a and b, while Figure 1(b) shows the corresponding reduced sticky braid, where the endpoints of each strand are represented by a nonzero in a permutation matrix. We will exploit this correspondence between permutation matrices and reduced sticky braids, by using these two terms interchangeably.

Let $a = a'a''$. Given kernels $P_{a',b}$, $P_{a'',b}$, kernel $P_{a,b}$ can be obtained via a special multiplication operation, sometimes called *Demazure multiplication*, defined for reduced sticky braids. Algorithms for sticky braid multiplication were developed in [19, 20, 24]. Such algorithms allow us to solve the semi-local LCS problem either iteratively, or recursively; the recursive solution makes use of the fast sticky braid multiplication algorithm of [24], running in time $O(N \log N)$ on sticky braids of order N. In the following subsections, we provide a description of both these approaches.

3.1 Iterative combing algorithm

A sticky braid corresponding to the input strings a, b is embedded in the $m \times n$ LCS grid. The braid consists of $m+n$ strands, m

[1] There are several structures for range counting in permutations [5, 6, 13]

Listing 1: Iterative combing

```
1  fun iterative_combing(String a,Int m,String b,Int n):Perm
2    // Initialization (phase 1)
3    for i in [0 .. m): h_strands[i] = i
4    for j in [0 .. n): v_strands[j] = j + m
5    // Braid combing (phase 2)
6    for i in [0 .. m): for j in [0 .. n):
7    | | h_index  = m - 1 - i
8    | | v_index  = j
9    | | h_strand = h_strands[h_index]
10   | | v_strand = v_strands[v_index]
11   | | if a[i] == b[j] or h_strand > v_strand
12   | | | // should not cross them
13   | | | swap(h_strands[h_index], v_strands[v_index])
14   // Resulting permutation construction (phase 3)
15   for l in [0 .. m): kernel[h_strands[l]] = n + l
16   for r in [0 .. n): kernel[v_strands[r]] = r
17   return kernel
```

beginning horizontally at the grid's left edge, and n vertically at the grid's top edge (see Figure 1). We call these strands horizontal and vertical ones, and denote them by identifiers beginning with h and v respectively. The strands are ordered so that the first strand begins at the left edge of the bottom-left cell, and the last strand begins at the top edge of the top-right cell. The strands enter and leave every grid cell in pairs; a pair of strands never cross within a match cell, and may or may not cross in a mismatch cell. While it is easy to produce a braid for a given pair of strings a, b, it will generally be unreduced; we need to transform it to an equivalent reduced one in order to obtain the kernel $P_{a,b}$.

The *iterative combing* algorithm obtains a reduced braid by beginning with a trivial reduced braid corresponding to an empty grid without any matches. The processing of a cell $c[i, j]$ refers to adding a match in that cell, if one exists between $a[i]$ and $b[j]$, and then deciding whether the strands that hit the cell's left and top edges should cross inside this cell. If either $a[i]$ and $b[j]$ match, or else if this pair of strands have crossed previously, then the strands should not cross in the current cell; otherwise they should. It is easy to check whether a pair of strands have crossed previously, given their starting indices (see 11th line in Listing 1). Thus, by processing the cells left to right and top to bottom (for example, in row-major order) we preserve the invariant that each pair of strands cross at most once and, thereby, a reduced sticky braid will be obtained at the end. The pseudocode of the algorithm is presented in Listing 1.

Iterative combing. First, we initialize our current braid as the reduced sticky braid in which no pair of strands cross each other (aka the identity braid). This initial braid satisfies trivially the invariant of being reduced. Then the processing of each cell is performed in row-major order[2]. Each cell is processed as described before. Finally, we obtain the kernel as a straightforward mapping between the initial and the final index of each strand.

3.2 Recursive combing

The second algorithm is based on the idea that one can split the LCS grid into smaller parts and solve the semi-local LCS problem independently in each part. Then, as each of these solutions represents a smaller reduced sticky braid, one can apply sticky braid multiplication to compose these smaller braids to get the solution for the original problem, making use of the following theorems.

Listing 2: Reduced braid multiplication (Steady Ant)

```
1  fun braid_mult(Perm P, Perm Q, Int n) : Perm
2    if (n == 1) return [1]
3    P1, P2, map_P = P.split_with_map()
4    Q1, Q2, map_Q = Q.split_with_map()
5    R1 = inverse_map_row(braid_mult(P1, Q1), map_P)
6    R2 = inverse_map_col(braid_mult(P2, Q2), map_Q)
7    fresh_nzs = ant_passage(R1, R2)
8    R1_good_nzs, R2_good_nzs = filter(fresh_nzs, R1, R2)
9    return R1_good_nzs + R2_good_nzs + fresh_nzs
```

Listing 3: Recursive combing

```
fun recursive_combing (String a, String b) : String = when
  a.len==1 and b.len==1 and a==b → return [[1,0], [0,1]]
  a.len==1 and b.len==1 and a≠ b → return [[0,1], [1,0]]
  _ → if flag = a.len > b.len then b, a = a, b // swap
    | b_left, b_right = b[:b.len/2], b[b.len/2:]
    | l = recursive_combing(b_left , a)
    | r = recursive_combing(b_right, a)
    | m = compose(l, r)
    | return flag ? m : transpose(m)
```

THEOREM 3.4. *(LCS kernel composition) [23] The semi-local LCS kernel for strings $a = a'a''$, b can be obtained via composition of $P_{a',b}$ and $P_{a'',b}$ by operation compose, which mainly uses braid multiplication:* $P_{a,b} = compose(P_{a',b}, P_{a'',b})$.

THEOREM 3.5. *(flip) [23] For strings a and b, and indices $i, j \in [0 : m + n - 1]$:* $P_{a,b} = P_{b,a}[n + m - 1 - i, m + n - 1 - j]$.

The pseudocode for sticky braid multiplication and the resulting algorithm for the semi-local LCS problem are presented in Listing 2 and Listing 3 respectively.

Braid multiplication. As is proved in [24], the multiplication of two braids can be performed by the algorithm in Listing 2, also called the *steady ant* algorithm. This algorithm is based on a divide-and-conquer approach. Let P and Q be $n \times n$ permutation matrices (assume that n is even). We split each of them into matrices of size $\frac{n}{2} \times \frac{n}{2}$ as follows. Matrix P is split vertically into a pair of $n \times \frac{n}{2}$ subpermutation matrices. Then the zero rows are deleted from each of these matrices to obtain $\frac{n}{2} \times \frac{n}{2}$ permutation matrices P_1 and P_2 (Line 3). Matrix Q is split horizontally in an analogous way. In order to reinsert the deleted rows (columns) after the recursive calls, we need to keep the mapping between old and new row (column) indices. The algorithm solves recursively the two resulting subproblems of size $\frac{n}{2}$. Then reverse index mapping is performed to restore the $n \times n$ matrices.

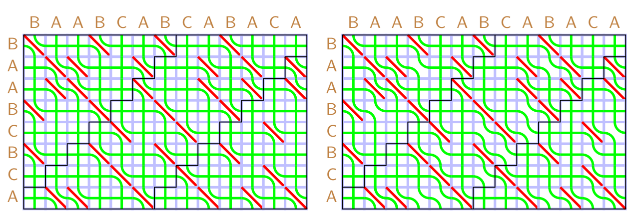
Although intuitively, it might seem that in order to obtain the final result, it would be sufficient to merge the nonzeros of matrices R_1 and R_2 obtained as the solutions to the subproblems, this is not quite true. To obtain the nonzeros that are still missing (*fresh nonzeros*), a procedure called *ant passage* [24] needs to be performed on R_1 and R_2 (Line 7). The nonzeros of R_1 and R_2 then need to be filtered separating *good nonzeros*, remaining in the solution, from *bad nonzeros*, which are deleted.

Recursive combing. The algorithm in Listing 3 follows the divide-and-conquer approach. The recursion base is a pair of strings of length 1. A match yields the identity kernel $\left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$ and a mismatch the *zero kernel* $\left(\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right)$ The recursive step splits the LCS grid into two subgrids vertically if $a.len < b.len$, or horizontally otherwise.

[2]The processing could be performed in any order compatible with the top-to-bottom, left-to-right dependencies of the cells

Listing 4: Parallel iterative combing

```
fun parallel_iterative_combing (String a, Int m, String
    b, Int n) : Perm
 fun inloop (Int up_bound,Int h_index,Int v_index): Void
 | # pragma parallel loop
 | for j in [0,up_bound):
 | | h_strand = h_strands[h_index + j]
 | | v_strand = v_strands[v_index + j]
 | | p = a_reverse[h_index + j] == b[v_index + j] ||
      (h_strand > v_strand)
 | | cond_if_store(h_strands[h_index + j], v_strand, p)
 | | cond_if_store(v_strands[v_index + j], h_strand, p)
 | # pragma sync
 // init phase
 a_reverse = reverse(a)
 # pragma parallel loops
 for i in [0 .. m): h_strands[i] = i
 for i in [0 .. n): v_strands[i] = i + m
 # pragma sync
 // 1st phase
 for anti_d_len in [0 .. m-1): inloop(anti_d_len +1,
      m-1-anti_d_len, 0)
 //2nd phase
 for k in [0 .. full_len_diags): inloop(m, 0, k)
 //3rd phase
 v_index = full_len_diags
 for anti_d_len in [m-1 .. 1]:
 | inloop(anti_d_len, 0, v_index++)
 // building of perm
 # pragma parallel loops
 for l in [0 .. m): perm[h_strands[l]] = n + l
 for r in [0 .. n): perm[v_strands[r]] = r
 # pragma sync
 return perm
```



(a) Unreduced braid as composition of subbraids (b) Reduced subbraids

Figure 2: Load-balanced iterative combing

Then for each subgrid, a recursive call is performed. The resulting kernels are composed via the *compose* routine that calls the braid multiplication algorithm. Note that when the grid is split vertically, a solution for $P_{b,a}$ will be obtained, so we need to flip $P_{b,a}$ to get $P_{a,b}$ via Theorem 3.5.

4 PARALLEL ALGORITHMS

In this section, we describe parallel versions of the discussed algorithms.

4.1 Parallel iterative combing

In Listing 4 we present a parallel version of iterative combing, running in time $O(\frac{mn}{t})$ on a machine that allows data-parallel operations on a vector of length t. The initialization and the construction of the output kernel are trivially parallelized, so we concentrate on parallelizing the braid combing itself.

Given indices i, j, the processing of cell $c[i, j]$ depends on the processing of cell $c[i, j-1]$ to the left and the cell $c[i-1, j]$ above. Thus, the cells within a given anti-diagonal are independent of each other, so the computation can proceed in anti-diagonals in parallel using thread-level parallelism, potentially speeding up the computation by the number of threads allowed in the system[3]. Note that after the processing of each anti-diagonal, a synchronization of worker threads is required, which may introduce its own overhead.

Without loss of generality, let $m \leq n$. Then the computation is split into three phases (see Figure 2). The length of the anti-diagonal increases from 1 to $m-1$ in the first phase, stays equal to m in the second phase, and decreases from $m-1$ to 1 in the third phase.

Unless the strings' lengths are very different, variable anti-diagonal length in different iterations may result in poor load balancing. Using braid algebra, we can consider the three subbraids each corresponding to a different phase of computation. We reorder the iterations within each phase to improve load balancing, and then we compose the resulting subbraids to get the final result. More precisely, we can compute in parallel the first and the third braid, so that in each iteration exactly m cells are processed. Besides improving the algorithm's load balancing, this approach also reduces the number of synchronizations between the threads.

Now we describe the inner loop (routine `inloop` in Listing 4). Characters of the input strings and elements of arrays `h_strands`, `v_strands` are read consecutively in adjacent iterations[4]. Conditional branching within the inner loop prevents the full application of SIMD parallelization. Moreover, its presence creates problems with branch prediction. However, branching may also have a positive effect by reducing the number of memory writes significantly. Therefore, it is a priori unclear how an elimination of branching would affect the performance in various scenarios.

Conditional branching can be eliminated using integer arithmetic as follows:

$$\texttt{h\_strands[i]} = \texttt{h\_strand} * (1-\texttt{p}) + \texttt{p} * \texttt{v\_strand}$$
$$\texttt{v\_strands[j]} = \texttt{v\_strand} * (1-\texttt{p}) + \texttt{p} * \texttt{h\_strand}$$

This solution allows us to employ fully SIMD parallelism but requires the use multiplication and addition operations. Given that p can only take two values (0 and 1) and using bitwise Boolean logic on integers, we can eliminate branching as follows[5]:

$$\texttt{h\_strands[i]} = (\texttt{h\_strand} \ \& \ (\texttt{p}-1)) \mid ((-\texttt{p}) \ \& \ \texttt{v\_strand})$$
$$\texttt{v\_strands[j]} = (\texttt{v\_strand} \ \& \ (\texttt{p}-1)) \mid ((-\texttt{p}) \ \& \ \texttt{h\_strand})$$

Such an approach allows us to replace multiplication instructions with bitwise instructions that are far more effective.

Finally, we can optimize SIMD parallelism utilization in case $m + n \leq 2^{16}$. We use 16-bit machine words for strand numbers.

4.2 Parallel recursive combing

We now present a parallel version of the recursive combing algorithm. Since function `compose`, which is the heart of this algorithm, relies on braid multiplication we fist give a parallel version of the braid multiplication algorithm.

[3] The algorithm is structurally very similar to the classical dynamic programming LCS algorithm with the only difference that cell processing depends on fewer previous cells. For dynamic programming LCS there is an additional dependency on cell $c[i-1, j-1]$ above and to the left.

[4] If we reverse a and store it in $a\_reverse$ then access to the latter one would be consecutive

[5] The standard representations of -1 is a machine word with all bits set to one

Listing 5: Parallel Steady Ant

```
fun parallel_steady_ant (Perm P,Perm Q,Memory used_block,
    Memory free_block,Map[Pair[Perm,Perm],Perm] precalc,
    Int k, Memory memory_block_indices) : Perm
 if (n <= k) return precalc[(P,Q)]
 P1,P2=split_with_mapping_on_prealloc_memory(P,free_space)
 Q1,Q2=split_with_mapping_on_prealloc_memory(Q,free_space)
 # parallel task
 R1 = parallel_steady_ant(P1, Q1,free_space,used_block,
       precalc, k, memory_block_indices + some_shift)
 # parallel task
 R2 = parallel_steady_ant(P2,Q2,free_space+n,used_space+n,
       precalc,k, memory_block_indices+some_other_shift)
 # task wait
 P, Q = inverse_mapping(R1, R2)
 fresh_nzs  = ant_passage(P, Q)
 P_good_nzs = filter(fresh_nzs, P)
 Q_good_nzs = filter(fresh_nzs, Q)
 return P_good_nzs + Q_good_nzs + fresh_nzs
```

Listing 6: Parallel hybrid combing

```
fun parallel_rec_combing (String a, String b,
    Map[Pair(Perm, Perm), Perm] precalc, Int threshold)
    : Perm = when
| a.len + b.len <= threshold →
| | return parallel_iterative_combing(a,a.len,b,b.len)
| a.len < b.len →
| | b_left, b_right = b[:b.len/2], b[b.len/2:]
| | l = parallel_rec_combing(b_left ,a,precalc,threshold)
| | r = parallel_rec_combing(b_right,a,precalc,threshold)
| | m = parallel_compose(l, r, precalc)
| | // need to flip to get kernel for a against b
| | return get_permutation_ab(m)
| default →
| | a_left, a_right = a[:a.len/2], a[a.len/2:]
| | l = parallel_rec_combing(a_left ,b,precalc,threshold)
| | r = parallel_rec_combing(a_right,b,precalc,threshold)
| | return parallel_compose(l, r, precalc)
```

4.2.1 Parallel braid multiplication. The parallel version of braid multiplication is presented in Listing 5. There are three potential bottlenecks in parallelizing the algorithm in Listing 2.

(1) In contrast with the iterative combing algorithm, fine-grained parallelism (thread-level and SIMD) is no longer applicable: both the mapping stage and the ant passage are strictly sequential, since neither the indices of zeros rows and columns, nor the steps of the ant, can be determined beforehand.
(2) Deep recursion may affect the algorithm's performance.
(3) The recursion requires $O(n \log n)$ memory since in each level it allocates $O(n)$ memory for permutations and index mappings.

We first note that the problem with lack of fine-grained parallelism is not critical, since there is sufficient coarse-grained (processor-level) parallelism in the algorithm: the subtasks in the same recursion level are independent of each other.

Recursion depth can be reduced by pre-computation as follows. We cut off several levels at the bottom of the recursion tree by pre-computing products of small matrices. Given some N there are $N!$ distinct permutation matrices of size N, thus, there are $N! \times N!$ possible pairs. For small N we can pre-compute the product for each possible pair and store it in one machine word[6]. Then we use these pre-computed products as the base for the recursion.

To reduce the memory requirements for permutation matrices one may use memory preallocation as follows. Let P and Q be the input permutation matrices each of size N for the algorithm's recursive call. Since each matrix requires memory[7] exactly $2N$, they both are stored in a memory block of size $4N$ denoted by `memory_block`. The resulting matrices P_1, Q_1, P_2, Q_2 are placed, in that order, in a memory block denoted by `free_space` of size $4N$. Since the information from P and Q is no longer required, the associated memory block may be split into two smaller blocks each of size $2N$ and used by the lower recursion levels. Thus, the memory requirement for storing permutation matrices is indeed reduced to exactly $8N$. Additionally, we can preallocate a memory block for the mappings in order to reduce the number of calls to the memory manager. Nonetheless, the memory requirement for memory mappings is still $O(N \log N)$ since we need to know all previous mappings.

4.2.2 Parallelizing the outer recursion. Similarly to parallel braid multiplication, coarse-grained parallelism can also be used for the outer recursion of the recursive combing algorithm, by using the recursion to generate the required number of independent subproblems.

4.3 Parallel hybrid combing

The iterative and the recursive combing algorithms can be combined to form a hybrid algorithm presented in Listing 6. The algorithm follows the structure of recursive combing up to some fixed recursion level, and then switches to iterative combing. Thus, we enable both coarse-grained and fine-grained parallelism.

We can apply several optimizations to this algorithm (Listing 7). First, we eliminate outer recursion in order to avoid the associated overhead (especially since the braid multiplication subprocedure is itself recursive). We also implement a flexible partition scheme. Note that in each step of the reduction, we need to decide on performing either a horizontal or a vertical compose. The order in which the compose operations are performed can affect the overall running time, since the compose operation itself is log-linear, rather than just linear. We use the following heuristic: always merge by the longest axis of a subgrid, so that the subgrid sizes are always approximately balanced.

The second optimization refers to the aforementioned use of 16-bit machine words for strand indices in the iterative combing algorithm. In order to apply this optimization, we need to partition the grid in a such way that the total numbers of strands within each lowest-level sub-grid does not exceed 2^{16}.

4.4 Bit-parallel iterative combing

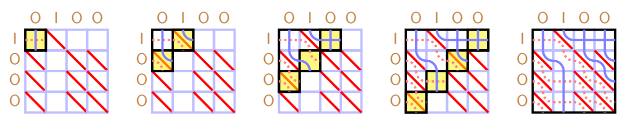
In some situations, only the global LCS score is of interest, and the alphabet is small (e.g. binary). In such cases, we can develop a bit-parallel algorithm based on the algorithm of Listing 4. The algorithm runs in $O(\frac{mn}{w})$ bit operations, where w is the machine word size in bits. In contrast to bit-parallel LCS algorithms by Crochemore et al. [10] and Hyyrö [12], which iterate over the grid in vertical or horizontal tiles, our algorithm iterates over the grid

[6] In our implementation we precalculate all the $(5!)^2 = 14400$ products of 5×5 permutation matrices, as well as the products of all smaller matrices. Such a matrix is stored in a 32-bit machine word as a top-left corner submatrix of an 8×8 permutation matrix. The latter is represented as an array of 8 tetrades, where the k-th tetrade provides the column index of the nonzero in row k. In principle, it could be just feasible to also precompute all the $(6!)^2 = 518400$ products of 6×6 permutation matrices, but probably not any larger ones.

[7] The permutation matrix of size N can be represented as two lists of size N.

Listing 7: Parallel hybrid combing (Optimized)

```
fun parallel_rec_combing (String a, String b,
    Map[Pair(Perm, Perm), Perm] precalc) : Perm
 m_outer, n_outer = optimal_split(a.len,b.len,n_thds)
 sub_grids = new array[m_outer][n_outer]
 #pragma omp parallel master taskloop
 for (i,j) in [0 .. m_outer)×[0 .. n_outer):
 | sub_grids[i][j] =
    iterative_combing_wrapper(a,b,i,j,m_outer,n_outer)
 #pragma sync
 steps = ceil(log2(m_outer)) + ceil(log2(n_outer))
 m_inner, n_inner = a.len / m_outer, b.len / n_outer
 new_m_outer, new_n_outer = m_outer, n_outer
 repeat steps times:
 | row_reduction = m_outer < n_outer
 // composing braids by subgrid's longest side
 | if n_outer > 1 && m_outer > 1
 | then row_reduction = m_inner >= n_inner
 | if row_reduction
 | then n_inner *= 2, new_n_outer = ceil(n_outer/2)
 | else m_inner *= 2, new_m_outer = ceil(m_outer/2)
 | #pragma omp parallel master taskloop
 | for (i,j) in [0 .. new_m_outer)×[0 .. new_n_outer):
 | | if (row_reduction)
 | | // compose subgrid pairs on common vertical side
 | | then reduction_in_row(i,j, precalc)
 | | // compose subgrid pairs on common horizontal side
 | | else reduction_in_col(i,j, precalc)
 | #pragma sync
 | n_outer,m_outer = new_n_outer, new_m_outer
 return sub_grids[0][0] // resulting reduced braid
```



(a) $h = 1111$ $v = 0000$ (b) $h = 0011$ $v = 0011$ (c) $h = 0111$ $v = 0001$ (d) $h = 0011$ $v = 0101$ (e) $h = 0001$ $v = 1101$

Figure 3: Snapshots of grid processing for string a = "1000", b = "0100"

in anti-diagonal blocks[8]. Furthermore, while the aforementioned algorithms use integer addition for propagating a strand as a carry across the tile, our algorithm uses only Boolean logic and shifts. Also, no precomputed table is needed.

We consider strings a, b over a binary alphabet. For simplicity, let us assume m, n are both multiple of w.

We partition strings a, b into groups of w characters. For string a, both the groups and the characters within each group are stored in reverse order (most significant bit first), and for string b in normal order (least significant bit first). Similarly, the array of horizontal strands is stored in reverse order, and vertical strands in normal order. The initialization of strands is similar to that of iterative combing (Listing 4) but differs in that all the horizontal strands' indices are set to ones while all vertical ones to zeros. So to check if a pair of strands have previously crossed we need to check if the horizontal strand's index is less than the vertical one's.

Next, we need to implement the inner loop logic of iterative combing (Listing 4). The storage of horizontal strands and characters of string a in reverse order within machine words allows us to implement this logic via shifts and Boolean operators. As we process an antidiagonal block, shifts are used to align characters of string a against string b, and horizontal strands against vertical ones. After the alignment, we use combing logic on binary strand indices, which is analogous to the combing logic on integers.

Consider an example. Let a = "1000", b = "0100", $w = 4$, so each string can be represented by a single machine word. First, we encode both strings into machine words: $a' = 1000_2, b' = 0010_2$ and process each antidiagonal of a grid (see the visualization on Figure 3). After the initialization step we have initialized v (vertical) and h (horizontal) strands: $h = 1111_2, v = 0000_2$.

Consider the processing of the second antidiagonal of the grid (see Figure 3(b)). At this stage $h = 1111_2, v = 0000_2$, so we need to process two cells on this antidiagonal. We proceed as follows:

- compare string characters: $s = !((a' \gg 2) \oplus b)$
- calculate mask to select the active bits: $mask = 0011_2$
- evaluate combing condition: $c = mask \,\&\, (s \mid (!(h \gg 2) \,\&\, v))$
- save $v' = v$
- update $v = (!c \,\&\, v) \mid (c \,\&\, (h \gg 2))$
- update $c = c \ll 2$
- update $h = (!c \,\&\, h) \mid (c \,\&\, (v' \ll 2))$

Upon processing all the antidiagonals, the LCS score can be obtained via Kernighan's Algorithm that counts all the set bits in horizontal strands: $|a|$ – *set bits in h*. The resulting algorithm is presented in Listing 8.

We can apply several optimizations to the above algorithm. First, it is clear that when processing some sub-grid of size $w \times w$, we do not need to load associated words for every antidiagonal of this sub-grid i.e *h_vec*, *v_vec*, *a_vec*, *b_vec* . It suffices to load all required data only once to the registers, then to write back the updated data after processing the sub-grid. This optimization is already presented in Listing 8.

Second, by studying the truth table for the inner loop logic we can find alternative Boolean formulas that have the same truth table and require fewer operations. We believe the following formula for an update of v to be optimal in terms of the number of operations[9]:

$$v = ((h \gg k) \mid !mask) \,\&\, (v \mid (!((a \gg k) \oplus b) \,\&\, mask))$$

Instead of updating h by a similar formula, we can observe that h is determined uniquely, given the new v and old v' indices of vertical strands and the old value for h. Indeed, v and h are updated by swapping bits between them in certain positions, therefore the update for h can be performed by filling in the "missing bits": $h = h \oplus (v \ll k) \oplus (v' \ll k)$. Therefore, applying this optimization reduces the number of operations from 18 to 12 inside the loop[10].

The third optimization consists in eliminating one operation when comparing characters of the input strings. Since $!(a \oplus b)$ is the same as $!a \oplus b$, we can store $!a$ instead of a and eliminate one operation from the formula.

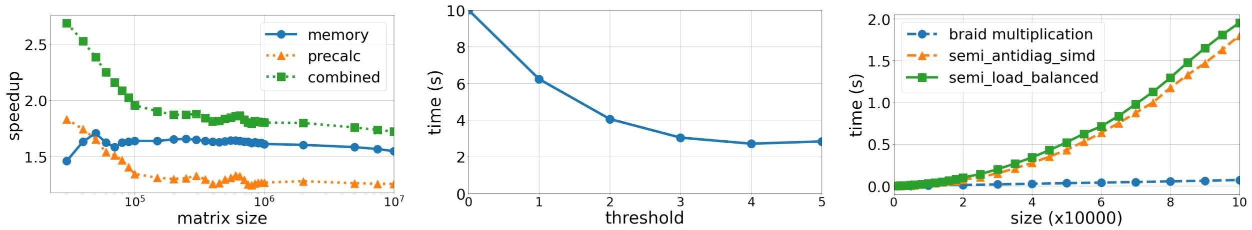
5 EVALUATION

In this section we use the following notation for implementations of various algorithms. For braid multiplication algorithms: `precalc` — with the precalc optimization, `memory` — with memory preallocation and optimized memory management, as described previously,

[8]Thus, our shift in computation pattern with respect to traditional bit-parallel techniques is, in a way, the opposite of the shift in [1] from the anti-diagonal to the vertical or horizontal pattern for standard dynamic programming.

[9]Note that this formula for the upper-left of sub-grid; for the lower-right part of the sub-grid formula is the same but $\ll$ is used

[10]*mask* as well as *!mask* is a compile-time constant

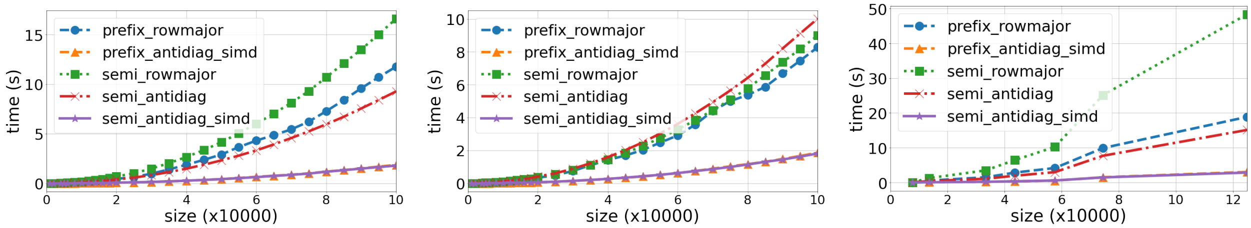


(a) Speedup of sequential braid multiplication optimizations relative to unoptimized version

(b) Scalability of parallel braid multiplication

(c) Braid multiplication as subroutine of iterative combing

Figure 4: Experimental results for braid multiplication

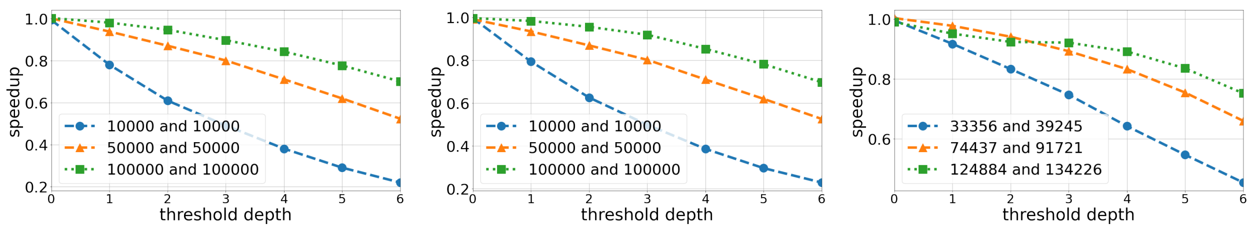


(a) Random strings of equal lengths, $\sigma = 1$

(b) Random strings of equal lengths, $\sigma = 26$

(c) Real-life strings

Figure 5: Comparison of the running time of sequential implementations

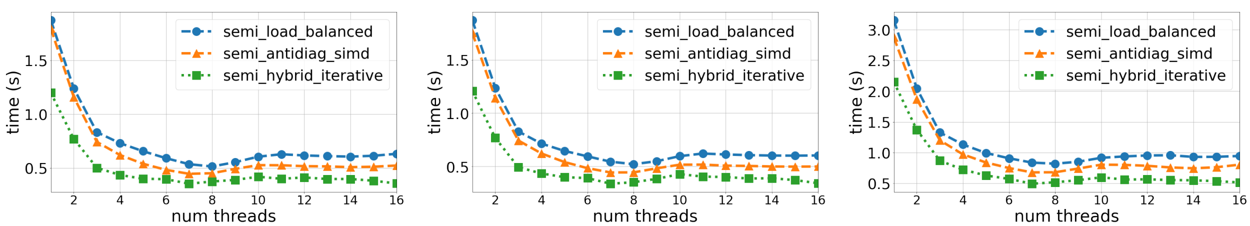


(a) Random strings, $\sigma = 1$

(b) Random strings, $\sigma = 26$

(c) Real-life strings

Figure 6: Tradeoff between sequential performance and parallelization potential



(a) Random strings, $\sigma = 1$, $m = n = 100000$

(b) Random strings, $\sigma = 26$, $m = n = 100000$

(c) Real-life strings, $m = 124884$, $n = 134226$

Figure 7: Comparison of running time of implementations depending on the number of threads

`combined` — with both optimizations above. For linear space dynamic programming LCS algorithms:

- `prefix_rowmajor` — row-major computation order,
- `prefix_antidiag_SIMD` — anti-diagonal computation order and SIMD parallelism.

For semi-local LCS algorithms:

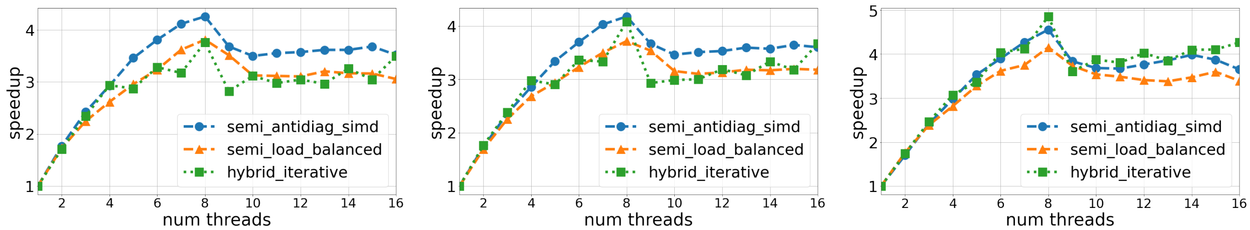
- `semi_rowmajor` — Algorithm 1,
- `semi_antidiag` — Algorithm 4, anti-diagonal computation order,
- `semi_antidiag_SIMD` — as `semi_antidiag` but with SIMD parallelism instead of branching, as described previously,
- `semi_load_balanced` — as `semi_antidiag_SIMD` but in three independent phases followed by braid multiplication, as described previously,
- `semi_hybrid` — Algorithm 6,
- `semi_hybrid_iterative` — Algorithm 7 with 16-bit strand indices.

For bit-parallel LCS algorithms:

- `bit_old` — Algorithm 8,
- `bit_new_1` — Algorithm 8 with memory access optimization and original Boolean formula,
- `bit_new_2` — Algorithm 8 with all described optimizations including an optimized formula.

We have implemented these algorithms[11] in the OpenMP framework that supports multi-processor and multi-threaded programs with shared memory. The implementations were compiled via `G++`

[11] https://github.com/NikitaMishin/semilocal



(a) Random strings, $\sigma = 1$, $m = n = 100000$

(b) Random strings, $\sigma = 26$, $m = n = 100000$

(c) Real-life strings, $m = 124884$, $n = 134226$

Figure 8: Performance of semi-local LCS algorithms

Listing 8: Bit-parallel algorithm

```
fun bp_iterative_combing (String a, String b): Int
 a_bin,b_bin = encode_reverse(a), encode(b)
 h, v = init_all_ones(), init_all_zeroes()
 // up_bound is current antidiag len
 fun inloop(Int up_bound,Int h_index,Int v_index): Void
 | #pragma parallel loop
 | for j in [0 .. up_bound):
 | | h_vec, v_vec = h[h_index+j],v[v_index+j]
 | | a_vec, b_vec = a_bin[h_index+j],b_bin[v_index+j]
 | | mask = 1 // lsb set to one, remaining bits to 0
 | #pragma unroll // upper left triangle
 | for shift in [strands_per_word - 1 .. 0):
 | | h_s,v_s = h_vec >> shift, v_vec << shift
 | | // compute strand combing condition
 | | cond = mask &
            (~((a_vec >> shift) ^ b_vec) | (~h_s & v_vec))
 | | inv_cond = ~cond
 | | // perform combing
 | | v_vec = (inv_cond & v_vec) | (cond & h_s)
 | | cond <<= shift
 | | inv_cond = ~cond
 | | h_strand = (inv_cond & h_vec) | (cond & v_s)
 | | mask = (mask << 1) | Input(1)
 | ... // process main antidiagonal, no shifts needed
 | #pragma unroll // lower-right triangle
 | for shift in [1 .. strands_per_word):
 | | mask <<= 1
 | | ... // same logic as in 1st loop but symmetric case
 | h[h_index + j], v[v_index + j] = h_vec, v_vec
 // phase 1: upper left triangle
 for diag_len in [0 .. m/w-1):
 | inloop(diag_len + 1, m/w-1-diag_len, 0)
 // phase 2: main parallelogram
 for k in [0 .. full_len_diags/w): inloop(m / w, 0, k)
 start_j = full_len_diags/w
 // phase 3: lower-right triangle
 for diag_len in [m/w-1..1]: inloop(diag_len,0,start_j++)
 return a.len - count_ones(h)
```

`10.2.0` with `"-std=c++17 -fopenmp -march=native -O3"` options. The experiments were performed on a workstation with `AMD Ryzen-7-3800X` processor with 8 cores and 16 threads and `Manjaro Linux 21.0.1` operating system. Both synthetic and real-life strings were used as input. Synthetic strings were obtained as randomly generated integer sequences of length up to 10^6, with characters sampled from a normal distribution with zero mean and standard deviation σ, and then rounded towards zero (so that, for example, the proportion of zero characters for $\sigma = 1$ is $\frac{1}{2}\left(\mathrm{erfc}(-1/\sqrt{2}) - \mathrm{erfc}(1/\sqrt{2})\right) \approx 0.683$). By varying parameter σ we can emulate different scenarios that include high, medium, and low matching frequency. The real-life strings represent genome sequences of various viruses of lengths up to 134 000 from National Center for Biotechnology Information[12], mostly from project PRJNA485481.

[12]https://www.ncbi.nlm.nih.gov/, last access 23.04.2021

5.1 Braid Multiplication

We have implemented the proposed sequential algorithm for braid multiplication with the previously described optimizations. To see how these optimizations affect the running time of the algorithm, we have tested them on randomly generated input permutation matrices of sizes up to 10^7. The relative speedup of each optimization is presented in Figure 4(a). While both optimizations improve the overall running time, their speedups decrease with increasing matrix size and converge to a constant. For the precalc optimization this is due to the fact that the linear time saving provided by the precomputation becomes dominated by the superlinear growth in main computation. The memory preallocation optimization is less sensitive to matrix size. For matrix size 10^7 (the largest size in our experiments), these optimizations together improve the speed of the algorithm approximately by a factor of 1.75.

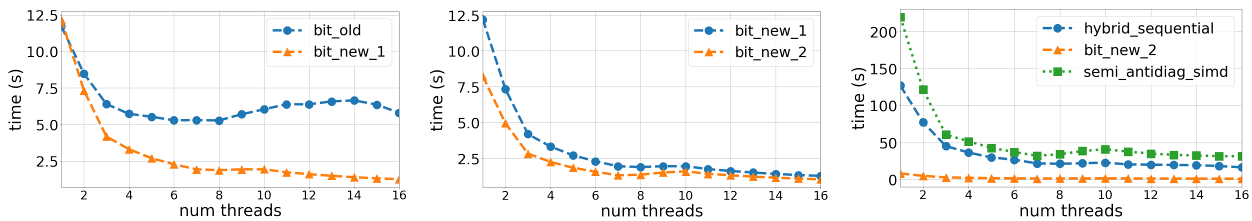
Further, we have implemented a parallel version of the braid multiplication algorithm proposed in Listing 5. To see how well it scales we have tested our implementation on permutation matrices of fixed size 10^7, while varying the threshold at which the algorithm switches to sequential computation, in the range from zero (no switching) to six (matrices at recursion level 6 are multiplied sequentially). Figure 4(b) shows that the optimal threshold value is 4, which provides a speedup of 3.7.

We have also compared the running time of two sequential versions of iterative combing: the basic and the load-balanced one. Figure 4(c) shows that the performance of these two versions is quite similar: this is expected, since load balancing only becomes useful for parallel computation. The same Figure also shows the performance of braid multiplication, which is used as a subroutine by the load-balanced iterative combing algorithm. We see that braid multiplication contributes a small fraction to the overall running time.

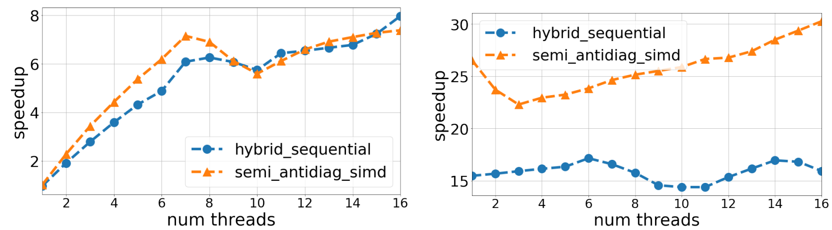
To summarize, we see that the braid multiplication algorithm performs well as a sequential algorithm and has moderate parallel scalability.

5.2 Semi-local LCS algorithms

Although, theoretically, algorithms for the LCS and the semi-local LCS problems have the same asymptotic running time, the practical behavior of semi-local LCS algorithms has been unclear. The results in Figure 5 demonstrate that the iterative combing algorithm is of comparable running time with prefix LCS and thus is applicable in practice. Even though the branchless version of iterative combing might have a higher number of memory writes, the introduction of SIMD instructions together with the elimination of



(a) Memory access optimization for bit-parallel algorithm (b) Boolean formula optimization for bit-parallel algorithm (c) Running times of different algorithms



(d) Performance of semi-local LCS algorithms (e) Relative performance of bit-parallel algorithm against semi-local LCS

Figure 9: Performance of different algorithms on binary sequences of length 10^6

branch prediction makes a significant impact. As our results show (see Figure 5) such an approach is the fastest on both the synthetic and the real-life dataset. Moreover, the effect of optimizations is greater on semi-local LCS than on prefix LCS due to better data locality since the latter has an additional dependency on the penultimate antidiagonal. The performance disadvantage of the branching version relative to the SIMD version is partially compensated by the fact that the former makes fewer memory writes, especially when the input strings are dissimilar. Overall, SIMD parallelism provides a speedup by a factor of 5.5 to 6 relative to the version with branching.

The coarse-grained parallelization potential of Algorithm 6 is expressed in the depth of the threshold at which the computation switches from recursive to iterative combing. A threshold of 0 indicates pure iterative combing with no recursion. Increasing the threshold depth creates independent subproblems, which can be executed in parallel; however, that affects negatively the implementation's sequential performance, and thus this threshold has to be chosen carefully. Figure 6 demonstrates this tradeoff between coarse-grained parallelization potential and sequential performance for various input string lengths. For example, for string lengths under 10^5, the appropriate threshold depth is 3 or less. Moreover, we see that as the input strings become longer, the appropriate threshold becomes deeper.

Figure 7 demonstrates that the load-balancing optimization has the opposite effect to what was expected, slowing down the computation. This is because synchronizations, in fact, require much less time compared to braid multiplication. However, it is possible that with a longer string b and more threads being used, this optimization can become useful. Figure 7 also shows that the hybrid algorithm performs better than iterative combing.

Figure 8 demonstrates the scalability of several algorithms. The maximum speedup is by a factor of 4, achieved on synthetic strings of length 10^5 with seven threads, which is one fewer than the number of cores on the testing machine. For real-life strings, similar results with five-fold speedup are observed. Note that since our grid partitioning heuristic does not always provide the best possible partition and composition order, the performance of the hybrid version can be quite erratic (see, for example, the speedup on five threads).

Figure 9 demonstrates results for large binary strings of lengths 10^6. First, Figure 9(a) shows that memory access optimization noticeably improves the running time of bitwise algorithm, especially when it works in multithreaded mode. This can be explained by a reduction in false-sharing among threads and the resulting significant drop in the number of synchronizations needed. In fact, on 16 threads, this optimization improves the running time by as much as a factor of 4.5. Second, optimized Boolean formula, as expected, improves the running time by a factor of 1.48 (Figure 9(b)). Third, although the semi-local LCS algorithm demonstrates a speedup by a factor of 4 to 5 on strings of length up to 10^5, Figure 9 shows that both implementations nearly reached an optimal speedup of 8 on long synthetic strings (Figures 9(d), 9(c)). For example, the parallel hybrid algorithm runs at a 7.95 speedup against its sequential version. And last but not least, Figures 9(c), 9(e) demonstrate that our bit-parallel algorithm is faster than hybrid and iterative combing by a factor of approximately 16 and 29, respectively.

6 CONCLUSION

In this paper, we have presented what is, to our knowledge, the first implementation of several semi-local LCS algorithms, including recursive and iterative combing, and their experimental evaluation. We have implemented both sequential and parallel algorithms, with a number of optimizations; for parallel algorithms, we have used both coarse-grained and fine-grained parallelism. In the experiments, we have used two input datasets: a synthetic dataset of randomly generated strings, and a real-life dataset of virus genomes.

The experiments show that semi-local LCS algorithms have comparable running time with the standard dynamic programming LCS algorithm on both synthetic and real-life data. For input sequences of length up to 10^6, the application of 8-fold SIMD parallelism (a 256-bit AVX vector of 32-bit integers) gave a speedup by approximately a factor of 5.5 to 6 for the branchless version of iterative combing algorithm on both datasets. Experiments show that most of the suggested optimizations have a positive impact on the parallel performance of the algorithms.

New hybrid approach (algorithm from Listing 7) for semi-local LCS has been presented. We have shown that the hybrid algorithm performs better than the iterative combing algorithm. For the LCS problem on a binary alphabet, we have developed a novel bit-parallel algorithm that is based on the iterative combing approach. In contrast with the existing algorithms [10, 12], our algorithm uses only Boolean operations and shifts (no integer arithmetic), and requires no precomputed tables. Our new algorithm demonstrates a speedup against the hybrid and the iterative combing algorithms by a factor of 16 and 29 respectively.

Furthermore, we have implemented the sequential sticky braid multiplication algorithm presented in [24] with a number of optimizations, as well as its coarse-grained parallel version, and evaluated them on randomly generated input permutation matrices of size up to 10^7. The optimizations of the sequential algorithm give a speedup by approximately a factor of 1.75, and the parallel version has moderate scalabilty, with maximum speedup by approximately a factor of 3.7. We have also established that the braid multiplication algorithm performs well and can be used successfully as a subroutine of semi-local LCS algorithm to improve its running time.

Further work is possible in several directions. First, we would like to exploit the new opportunities provided by recent developments in intra-pocessor SIMD parallelism, in particular Intel's AVX-512 processor architecture. This new architecture provides several sources of potential speedup for our algorithms: apart from increasing the SIMD vector registers' size to 512, it also introduces (as part of the AVX-512BW instruction subset) SIMD arithmetic on 8-bit and 16-bit integers. In some applications of semi-local LCS, representing strand indices with such reduced precision is feasible, resulting in $512/8 = 64$-fold parallelism, and therefore much higher potential speedups. Besides these quantitative improvements, there is an important qualitative one: the AVX-512 instruction set contains new instructions for masked pairwise minimum and maximum evaluation on a pair of SIMD vectors. This is a perfect match to the logic of the inner loop in the iterative combing algorithm, that should enable its elegant and efficient implementation.

The second direction of future work is further study and evaluation of the bit-parallel LCS algorithm presented in this paper. It is yet unclear how well this algorithm can be generalized to an arbitrary alphabet and how well it would perform relative to state-of-the-art bit-parallel algorithms. The workload imbalance introduced by the antidiagonal computation pattern appears to be a bottleneck that should be studied, since its elimination could theoretically provide a boost by a factor of 2.

Our experiments could also be extended by implementing the algorithms on other popular parallel platforms, including GPU and FPGA. It would be especially interesting to measure performance of our algorithm against state-of-the-art bit-parallel algorithms on FPGA, since this platform is particularly sensitive to additional memory requirements and to delays induced by carry propagation in arithmetic operations.

Finally, our techniques could be used for analysis of patterns in real-life data, for example, in time series data.

REFERENCES

[1] Srinivas Aluru, Natsuhiko Futamura, and Kishan Mehrotra. 2003. Parallel biological sequence comparison using prefix computations. *J. Parallel and Distrib. Comput.* 63, 3 (2003), 264–272.

[2] C E R Alves, E N Cáceres, F Dehne, and S W Song. 2002. Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In *Proceedings of ACM SPAA*. New York, New York, USA, 275–281.

[3] C E R Alves, E N Cáceres, and S W Song. 2006. A Coarse-Grained Parallel Algorithm for the All-Substrings Longest Common Subsequence Problem. *Algorithmica* 45, 3 (2006), 301–335.

[4] C E R Alves, E N Cáceres, and S W Song. 2008. An all-substrings common subsequence algorithm. *Discrete Applied Mathematics* 156, 7 (2008), 1025–1035.

[5] Jon Louis Bentley. 1980. Multidimensional divide-and-conquer. *Commun. ACM* 23, 4 (1980), 214–229.

[6] Timothy M Chan and Mihai Pătraşcu. 2010. Counting inversions, offline orthogonal range counting, and related problems. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 161–173.

[7] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. 2020. Faster Approximate Pattern Matching: A Unified Approach. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. 978–989.

[8] R A Chowdhury and V Ramachandran. 2006. Cache-oblivious dynamic programming. In *Proceedings of SODA*. 591–600.

[9] R Cole and R Hariharan. 2002. Approximate String Matching: A Simpler Faster Algorithm. *SIAM J. Comput.* 31 (2002), 1761–1782.

[10] Maxime Crochemore, Costas S Iliopoulos, Yoan J Pinzon, and James F Reid. 2001. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Inform. Process. Lett.* 80, 6 (2001), 279–285.

[11] D S Hirschberg. 1975. A linear space algorithm for computing maximal common subsequences. *Commun. ACM* 18, 6 (1975), 341–343.

[12] Heikki Hyyrö. 2004. Bit-parallel LCS-length computation revisited. In *Proc. 15th Australasian Workshop on Combinatorial Algorithms (AWOCA 2004)*. Citeseer, 16–27.

[13] Joseph JáJá, Christian W Mortensen, and Qingmin Shi. 2004. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *International Symposium on Algorithms and Computation*. Springer, 558–568.

[14] Krusche. 2010. Parallel String Alignments: Algorithms and Applications. Warwick University.

[15] G M Landau and U Vishkin. 1989. Fast parallel and serial approximate string matching. *Journal of Algorithms* 10, 2 (1989), 157–169.

[16] V Levenshtein. 1965. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission* 1 (1965), 8–17.

[17] William J. Masek and Michael S. Paterson. 1980. A faster algorithm computing string edit distances. *J. Comput. System Sci.* 20, 1 (1980), 18–31.

[18] Aradhana Raju, Richi Patnaik, Ritto Kurian Babu, and Purabi Mahato. 2016. Parallel prefix adders — A comparative study for fastest response. In *2016 International Conference on Communication and Electronics Systems (ICCES)*. 1–6.

[19] Luís MS Russo. 2010. Multiplication algorithms for Monge matrices. In *International Symposium on String Processing and Information Retrieval*. Springer, 94–105.

[20] Yoshifumi Sakai. 2011. A fast algorithm for multiplying min-sum permutations. *Discrete Applied Mathematics* 159 (2011), 2175–2183.

[21] J P Schmidt. 1998. All Highest Scoring Paths in Weighted Grid Graphs and Their Application to Finding All Approximate Repeats in Strings. *SIAM J. Comput.* 27, 4 (1998), 972–992.

[22] Peter H. Sellers. 1980. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms* 1 (1980), 359–373.

[23] Alexander Tiskin. 2008. Semi-local string comparison: Algorithmic techniques and applications. *Mathematics in Computer Science* 1, 4 (2008), 571–603.

[24] Alexander Tiskin. 2015. Fast Distance Multiplication of Unit-Monge Matrices. *Algorithmica* 71 (2015), 859–888.

[25] Alexander Tiskin. 2020. Communication vs Synchronisation in Parallel String Comparison. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*. 479–489.

[26] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[27] Robert A. Wagner and Michael J. Fischer. 1974. The String-to-String Correction Problem. *J. ACM* 21, 1 (1974), 168–173.