CS 309 – GROUP ASSIGNMENT

Efficient Parallel Algorithms for String Comparison

Authors: Nikita Mishin, Daniil Berezun, Alexander Tiskin

Submitted By:
Deepkamal Singh Jassal (190001011)
Anikeit Sethi (190001003)
Devansh Sahu (190001012)
Nimish Bansal (190001040)

Submitted To:
Dr. Chandresh Kumar Maurya

ABSTRACT

The longest common subsequence (LCS) problem on a pair of strings is a classical problem in string algorithms. Its extension, the semilocal 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. In the experimental part of the paper, we present an implementation of 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.

Introduction:

LCS:
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 . 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. Here, we present an implementation of iterative combing, Parallel Iterative Combing and Parallel Hybrid Combing using sticky braids.

SEMI-LOCAL LCS :
Let **m** and **n** be 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.
The semi-local LCS problem asks for LCS scores as follows -

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

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

It is proved that matrix Ha,b can be represented implicitly by a permutation matrix Pa,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 polylogarithmic1 . Kernel Pa,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.

Sticky Braid:

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 , Pa ″ ,b , kernel Pa,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.

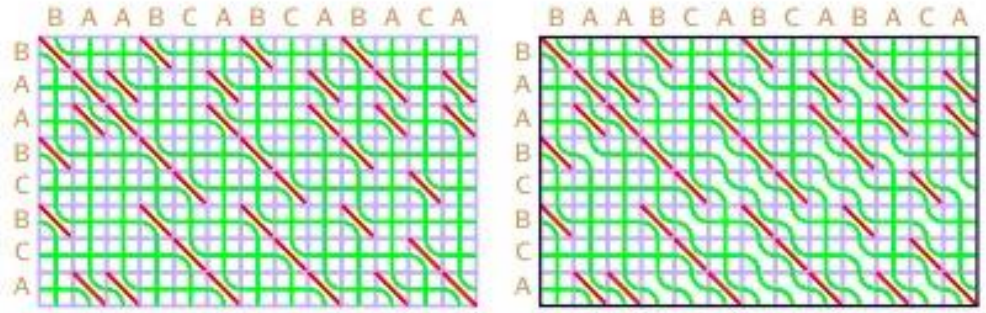


Figure 1: Semi-local LCS solution represented by a sticky braid

Iterative combing

In this method, 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.

Algorithm:

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.

Pseudocode:

```
fun iterative_combing(String a,Int m,String b,Int n):Perm
  // Initialization (phase 1)
  for i in [0 .. m): h_strands[i] = i
  for j in [0 .. n): v_strands[j] = j + m
  // Braid combing (phase 2)
  for i in [0 .. m): for j in [0 .. n):
  | | h_index  = m - 1 - i
  | | v_index  = j
  | | h_strand = h_strands[h_index]
  | | v_strand = v_strands[v_index]
  | | if a[i] == b[j] or h_strand > v_strand
  | | | // should not cross them
  | | | swap(h_strands[h_index], v_strands[v_index])
  // Resulting permutation construction (phase 3)
  for l in [0 .. m): kernel[h_strands[l]] = n + l
  for r in [0 .. n): kernel[v_strands[r]] = r
  return kernel
```

Parallel Iterative combing

A sticky braid corresponding to the input strings a, b is embedded in the m × n LCS grid. The braid consists of m + n strands, m beginning horizontally at the grid's left edge, and n vertically at the grid's top edge.

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.

Algorithm:
We present a parallel version of iterative combing, running in time O(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. 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.

Using braid algebra, we can consider the three sub-braids 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 sub-braids 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.

Pseudocode:

```
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
```

Parallel Hybrid Combing

The iterative and the recursive combing algorithms can be combined to form a **hybrid algorithm**. So, for a recursive combining algorithm we require a parallel version of braid multiplication algorithm (Steady Ant).

Pseudocode for 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
```

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.

Pseudocode for 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)
```

Evaluation

Data:

For semi local lcs naive iterative, runtime are as follows :-

Length of string :	16,000	57,000	91,000	150,000
Runtime	1813	20653	52605	142375

For semi local lcs parallel iterative, runtime are as follows :-

Length of string : number of threads	16,000	57,000	91,000	150,000
2	1701	19616	50614	138942
4	924	10304	26172	74732
8	664	7596	21125	58275

For semi local lcs parallel hybrid, runtime are as follows :-

Length of string : number of threads	16,000	57,000	91,000	150,000
2	1533	17812	45921	123721
4	756	8023	22732	62752
8	538	6893	19364	52892

Speed up for parallel iterative:

Length of string : number of threads	16,000	57,000	91,000	150,000
2	1.066	1.053	1.039	1.025
4	1.962	2.004	2.010	1.905
8	2.730	2.719	2.490	2.443

Speed up for parallel hybrid:

Length of string : number of threads	16,000	57,000	91,000	150,000
2	1.183	1.159	1.146	1.151
4	2.398	2.574	2.314	2.269
8	3.370	2.996	2.717	2.692

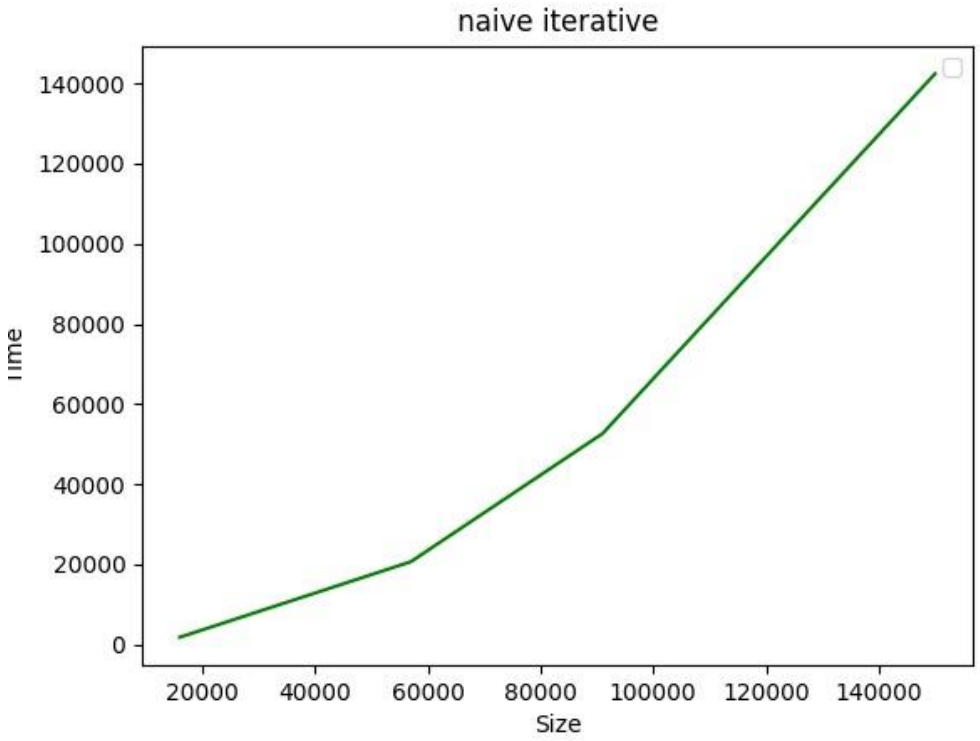
Average speed up for parallel iterative:

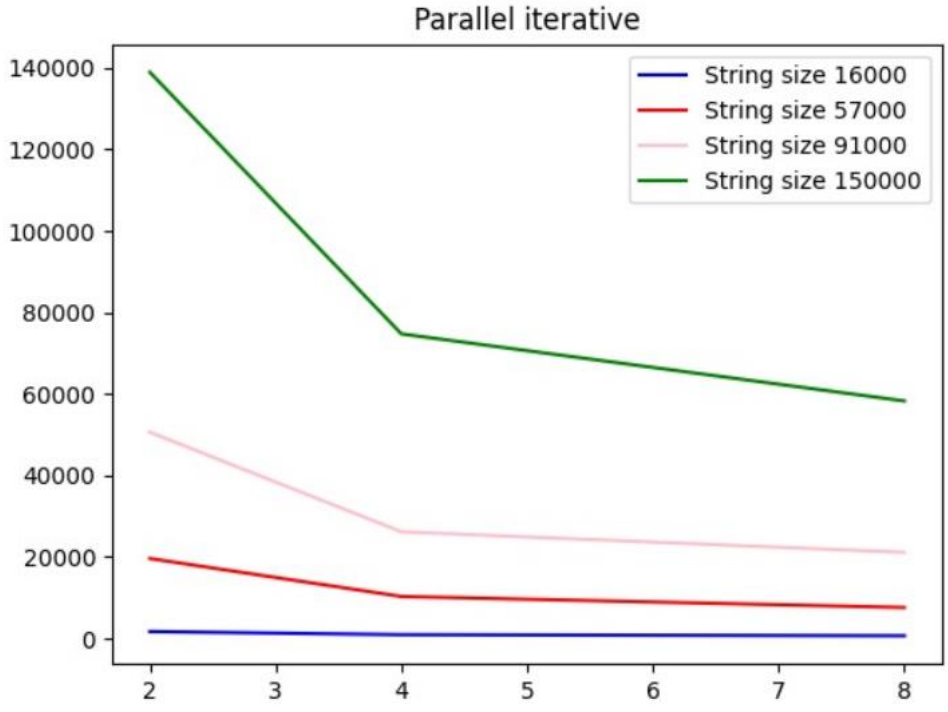
Number of threads:	speed up
2	1.046
4	1.970
8	2.596

Average speed up for parallel hybrid:

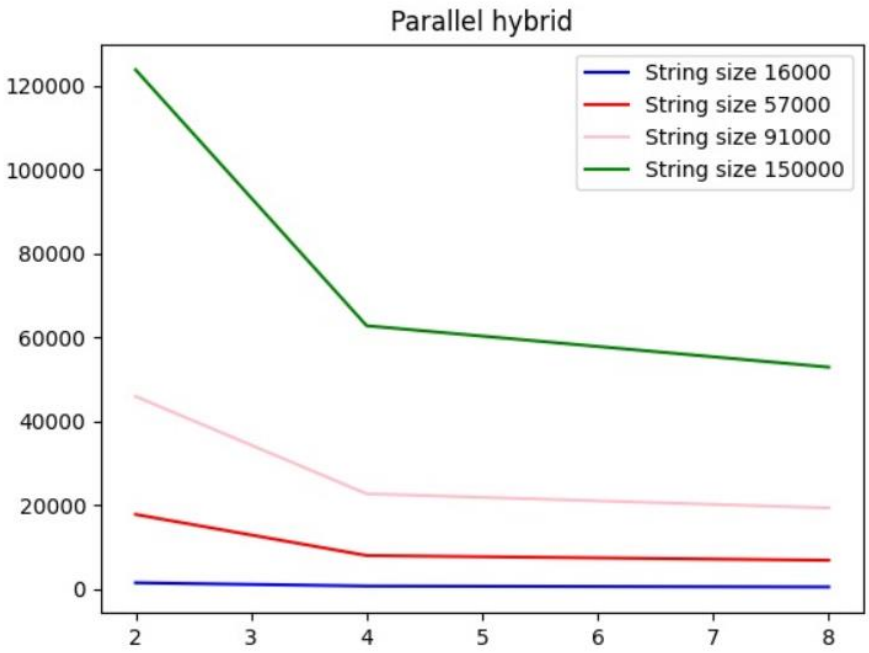
Number of threads:	speed up
2	1.160
4	2.389
8	2.944

Graphs:

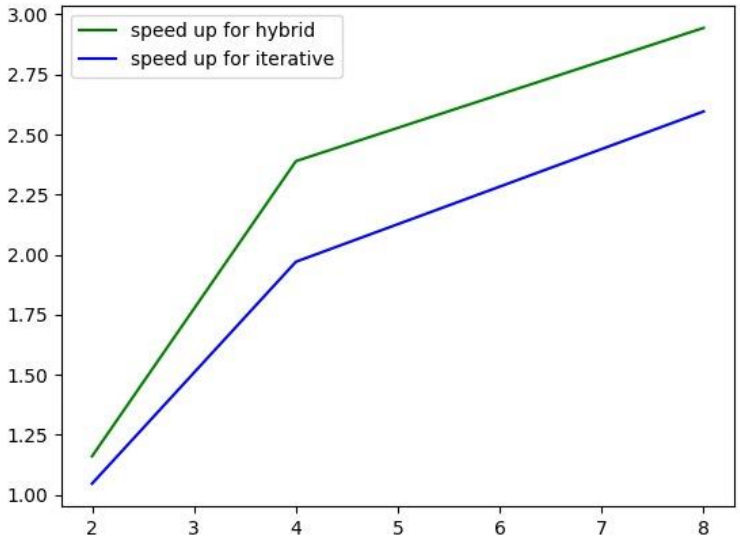




Runtime(y-axis) vs No. of Cores(x-axis)



Runtime(y-axis) vs No. of Cores(x-axis)



Speedup (y-axis) vs No. of cores (x-axis)

Comparison of speedup for Parallel Iterative combing and Parallel Hybrid Combing

Conclusion

We implemented numerous semi-local LCS algorithms in this project. There has been a new hybrid strategy for semi-local LCS presented. We've demonstrated that this hybrid method outperforms the iterative combing algorithm.