Parallel Variable Gapped Longest Common Sequence and Incremental Range Maximum Query

SHIANG-YUN YANG and PANGFENG LIU, National Taiwan University
JAN-JAN WU, Institute of Information Science, Academia Sinica

The longest common subsequence problem with variable gapped constraints (VGLCS) is used in genes and molecular biology. An $O(nm)$ solution has been proposed in the previous study, by reduction to the efficient incremental suffix maximum query (ISMQ) problem. Algorithm for solving ISMQ supports appending a value to array and querying the suffix maximum value in amortized $O(1)$ time. However, we try to parallelize origin algorithm by wavefront method, but failed to achieve better performance. In this paper, our algorithm and data structure can achieve a better theoretical time complexity on both querying and appending. The VGLCS problem can be solved in $O(nm/p + n \log n)$, where p is the number of parallel running processors. And also, our dynamic tree computation technique can answer incremental range maximum query in amortized $O(1)$.

CCS Concepts: • **Computing methodologies** → *Shared memory algorithms*;

Additional Key Words and Phrases: range minimum query, incremental range maximum query, incremental suffix maximum query, longest common sequence, parallel, Cartesian tree

ACM Reference format:
Shiang-Yun Yang, Pangfeng Liu, and Jan-Jan Wu. 2017. Parallel Variable Gapped Longest Common Sequence and Incremental Range Maximum Query. *ACM Trans. Parallel Comput.* 9, 4, Article 39 (October 2017), 27 pages.
DOI: 0000001.0000001

1 INTRODUCTION

The *longest common subsequence* (LCS) [10] is a famous problem in string processing. For example, the `diff` utility show the difference between texts by finding their LCS. Revision control systems like SVN and Git use LCS to reconciling multiple changes. In bioinformatics, the best-known application of the LCS problem is the sequence alignment [1, 12], which identifies the region of similarity between the sequences of DNA, RND, or protein.

Iliopoulos and Rahman [15] introduced many constrained versions of LCS. For example, a *fixed gap LCS* (FGLCS) requires that the distance between consecutive characters in the LCS is *at most* $k + 1$ characters away. A fixed gap LCS can be found in time $O(nm)$, where n and m are the lengths of the two input strings [15]. On the other hand, a *variable gap LCS* (VGLCS) requires that each character has a *gap* value and two consecutive characters in LCS must be with distance of the gap of the *latter* character *plus* 1. One can think of the fixed gap LCS as a special case of variable gap LCS in which the gap values of all characters are k.

We use an example to illustrate the gap function and VGLCS. Let string A be `GCGCAATG` with gap values $(3, 1, 1, 2, 0, 0, 2, 1)$, and let string B be `GCCCTAGCG` with gap values $(2, 0, 3, 2, 0, 1, 2, 0, 1)$.

DOI: 0000001.0000001

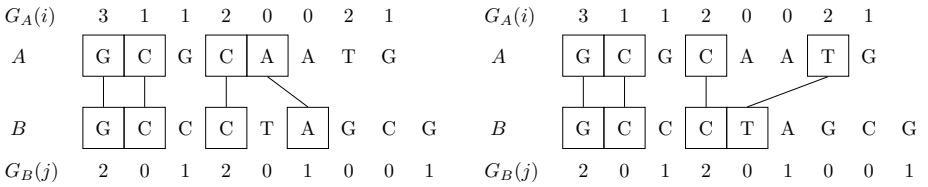


Fig. 1. A VGLCS example

Please refer to Figure 1 for an illustration. Now, the LCS GCCT is a VGLCS because every character in the LCS can find its predecessor in the LCS with distance at most its gap value plus 1.

This paper focuses on *efficient parallel algorithms* that find VGLCS. Peng [14] gives a $O(nm\alpha(n))$ algorithm that is easy to implement and an asymptotically better $O(nm)$ algorithm, where α is the inverse of Ackermann's function [2]. In this paper, we propose our $O(nm)$ algorithm which is *easy* to implement and runs *efficiently* in a *parallel* environment.

The parallelization of LCS on most multi-core platforms focuses on *wavefront* parallelism. The wavefront parallelism is motivated by the recursive solution of LCS. For example, Yang [18] introduced a new formulation to exploit more cache performance. Our algorithm uses a more powerful *sparse table* instead of the disjoint set in the Peng's serial algorithm and achieves better cache performance.

The remainder of the paper is organized as follows. In Section 2, we present previous parallel algorithms for finding VGLCS. In Section 3 and 4, we present our algorithm that is easy to parallelize, and has a time complexity $O(nm)$, which is better than previous works. In Section 5 and 6, we describe our optimized implementation and the results of our experiments. Section 8 and 9 conclude this paper with lessons learned and possible future works.

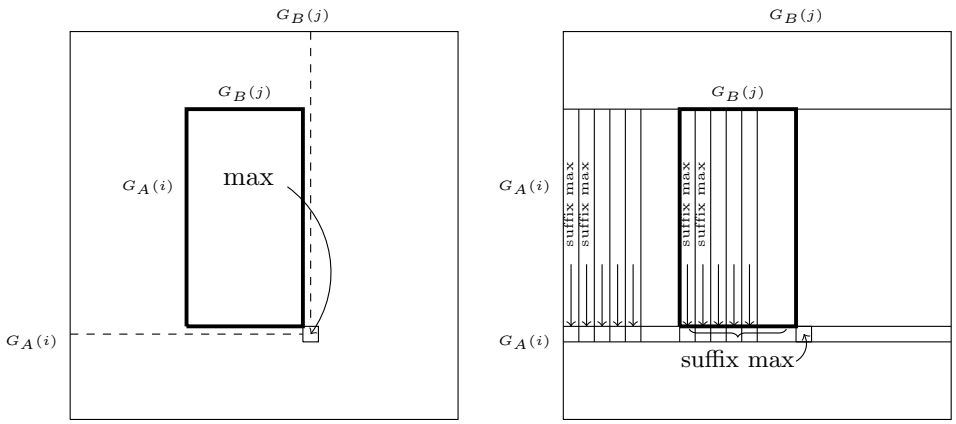
2 PARALLEL VGLCS ALGORITHM

2.1 Basic Dynamic Programming

We first describe a basic dynamic programming for VGLCS [14]. Let A and B denote two input strings of length n and m respectively, and G_A and G_B be the arrays of the variable gap constraints. We define $V[i][j]$ to be the *maximum* length of the variable gapped longest common subsequence between substring $A[1, i]$ and $B[1, j]$. It is easy to see that $V[i][j]$ is the *maximum* among $V[k][l]$, where k is between $i-1$ and $i-G_A(i)-1$, and l is between $j-1$ and $j-G_B(j)-1$, i.e., a rectangle within V on the left and upper of $V[i][j]$. Please refer to Figure 2(a) for an illustration.

The computation of V can be optimized as follows. Note that the computation of all $V[i][j]$'s with the same i has the *same* gap constraint $G_A(i)$, so the maximum within the rectangle can be computed in two steps. First, we compute the maximum of *every column* of this rectangle, and place them into another array R. Then we compute the maximum of the *suffix* of length G_B on R, which is exactly $V[i][j]$. Please refer to Figure 2(b) for an illustration.

We note that this optimization requires maximum queries on the suffix *incrementally* in the following sense. Recall that in the first step of the optimization, we need to compute the maximum of every column within the rectangle. This is just like finding the maximum of the *suffix* of every column in that rectangle. After we compute the i-th row of V and go to the next row to compute $V[i+1][*]$, we will then need the maximum of the suffix of every column of length $G_A(i+1)$. It will be beneficial if we put the V's in each column into a data structure that supports suffix maximum query. Similarly, the computation within the same row, that is, from $V[i][j]$ to $V[i][j+1]$,



(a) How to compute V.

(b) Compute V with incremental suffix maximum queries.

Fig. 2. The basic dynamic programming for VGLCS

also requires the maximum of the suffix on R. We will refer to this type of queries as *incremental suffix maximum query*, i.e., we would like to maintain a data structure form which we can find the maximum of its suffix efficiently while we add data at its end.

2.2 Peng's Algorithm

The sequential VGLCS algorithm 1 by Peng [14] applies the optimization and is shown as Algorithm 1. The outer loop goes through every row, and the inner loop goes through every element of a row from left to right. We use an array of C to answer incremental maximum queries on all columns. That is, we can think of $C[j]$ as a data structure that supports incremental suffix maximum queries on the j-th column of V. From the previous observation that all computation of elements in the i-th row of V share the same gap $G_A(i)$, we will query each C for the maximum in the suffix that ends at row $i-1$ with length $G_A(i)+1$, and place these maximums in another data structure R that also supports incremental suffix maximum queries. It is easy to see that the value of $V[i][j]$ can be obtained by querying R with a suffix maximum query of length $G_B(j)+1$ as shown in Figure 2(b).

We update V, C, and R as follows. If the i-th character of A matches the j-th character of B then $V[i][j]$ is the maximum among the rectangle plus 1, as shown in Figure 2(b). This maximum can be found by querying R for the maximum among the last $G_B(j)$ elements in it. Note that R contains the information of the previous row, up to the element of the $j-1$ element. After that, we add the maximum of last $G_B(j)+1$ in the j-the column into R, and the newly computed $V[i][j]$ into $C[j]$, which supports ISMQ on the j-the column, before going to column $j+1$. If the i-th character of A does *not* match the j-th character of B, we simply set $V[i][j]$ to 0 since it does not affect the answer, then again update R accordingly.

2.3 Incremental Suffix Maximum Query

From the previous discussion of Peng's algorithm, we note that in order to find VGLCS efficiently, we need to address the *incremental suffix maximum query* (ISMQ) problem. A data structure that supports incremental suffix maximum queries should support the three operations. First, a MAKE

ALGORITHM 1: Peng's algorithm for finding VGLCS [14]

Input: A, B: the input string; G_A, G_B: the array of variable gapped constraints;
Output: Find the LCS with variable gap constraints.
Create an array of m data structures $C[m]$ that support ISMQ;
Create an empty table $V[n][m]$;
for $i \leftarrow 1$ *to* n **do**
  Create a data structure R that supports ISMQ;
  for $j \leftarrow 1$ *to* m **do**
    if $A[i] = B[j]$ **then**
      $t \leftarrow$ Query R for the maximum among the last $G_B(j) + 1$ elements ;
      $V[i][j] \leftarrow t + 1$;
    else
      $V[i][j] \leftarrow 0$;
    end
    $t \leftarrow$ Query $C[j]$ for the maximum among the last $G_A(i) + 1$ elements ;
    Append t to R ;
    Append $V[i][j]$ into $C[j]$;
  end
end
Retrieve the VGLCS by tracing $V[n][m]$;

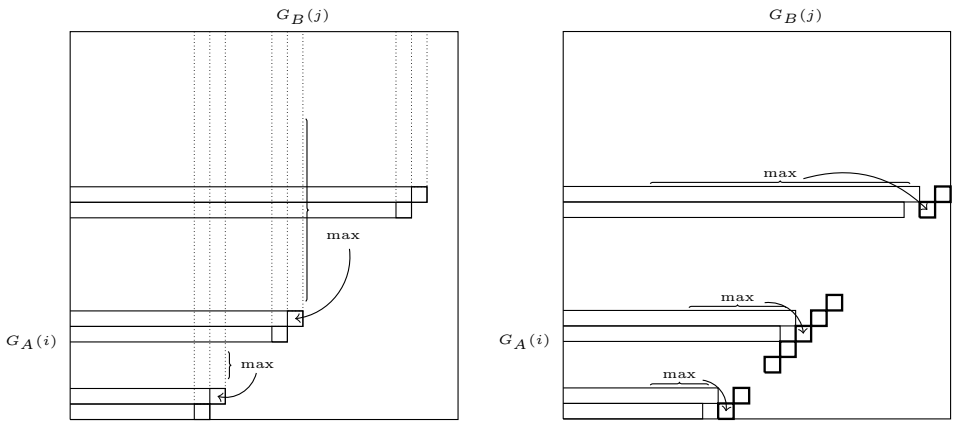
operation creates an empty array A. Second, an Append(V) operation appends a value V to array A. Finally, an ISMQ Query(x) finds the *maximum* value among those from x to the end of an array A.

Peng uses a *disjoint-set* data structure to answer incremental suffix maximum queries in his VGLCS algorithm. The disjoint-set data structure was proposed by Gabow [8] and Tarjan [16] to solves the *union-and-find problem*. The set of data is stored in a sequence of disjoint sets, and the *maximum* of each disjoint set is at the root of the tree, and these maximum are in *decreasing* order. When we add a value x, we make it as a disjoint set with a single element by itself, and as the *last* disjoint set in the sequence. Then we start joining (with union operation) from the last set to its previous set until the maximum of the previous set is *larger* than x. It is easy to see that the Query(x) operation is simply a *find* operation that finds the root, which has the *maximum*, of the tree that x belongs to. The amortized time per union/find operation is $O(\alpha(n))$.

2.4 A Parallel VGLCS Algorithm with Sparse Table

The sequential VGLCS algorithm 1 by Peng [14] and other variants of LCS are difficult to parallelize in a row-by-row manner. These algorithms use several states to determine a new state with a dynamic programming. This construction requires *heavy data dependency*, and is difficult to parallelize in a naive row-by-row manner because an element of the dynamic table needs the values of elements in the *same* row to compute its value.

It is also difficult to parallelize Peng's algorithm with the wavefront method because it requires *extra space* to keep track of row status, which is not required in a sequential algorithm. Recall that VGLCS requires a rectangle of data in V to compute a new element, if those V's want to compute on a diagonal wavefront, those rectangles of data will require extra bookkeeping since the gap constraints of those elements on the wavefront could be very different. Please refer to Figure 3 for an illustration of those data that must be present (in solid line). Also, the new elements computed must be appended to the data structures according to the wavefront, which incurs more bookkeeping and overheads.



(a) The first stage (b) The second stage

Fig. 3. The bookkeeping data of the wavefront method

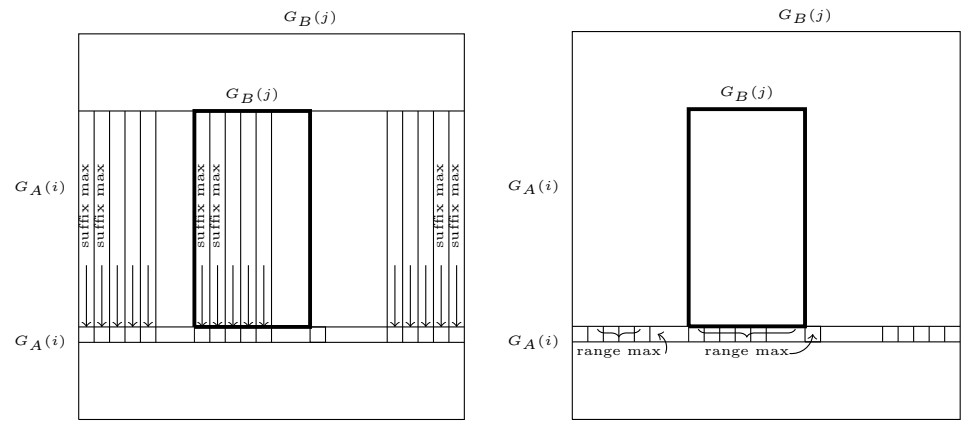
We propose a parallel VGLCS algorithm with an optimized row-by-row approach, which maintains only *one* row data structure that collects suffix maximum values from all columns. That is, our optimized row-by-row approach removes the data dependency among elements *within the same row*.

Our optimized row-by-row approach uses less space, has a more balanced workload, and a smaller thread synchronization overhead, than the wavefront method. We observe that the length of the critical path of a wavefront method is greater than that of a row-by-row method. In addition, if we can removes the data dependency among elements within the same row, then the computation on the elements of the same row can be *fully* and *evenly* parallelized. The result is a much more balanced workload distribution and a much easier synchronization among threads.

A sketch of our algorithm is as follows. Our algorithm computes V one row at a time. The computation of each row has two stages. In the first stage, the algorithm queries each data structure C for every column within the rectangle *in parallel*, so as to obtain the maximums of suffix of length $G_A(i) + 1$ of every column, and place them into an array R. Recall from Algorithm 1 that every column of V has a data structure C that supports incremental suffix maximum on V. Please refer to Figure 4 for an illustration.

In the second stage, our algorithm issues *range maximum queries*, one for each column, on R to compute all m elements of the i-th row of V *in parallel*. Note that unlike the sequential algorithm, we compute all elements in the i-th row of V in parallel, so we cannot query the *suffix* of R. Instead, we need to query a *range* of R for the maximum, where the range is the gap constraint on that column. Please refer to Figure 4 for an illustration. Note that we need to add the newly computed $V[i][j]$ into the C of the j-th column *incrementally*, so that they will contain the correct information for the computation of the $(i + 1)$-th row. Also, since the algorithm iterates in rows, these C's only need to support suffix maximum query. No range query on them is required. In contrast, we do need to support range maximum query on R, and these queries will be in parallel.

To resolve the data dependency, we need to consider a good data structure that can handle incremental suffix/range maximum query *in parallel*. We note that it is *not* feasible to parallelize the disjoint set implementation for three reasons. First, a query for disjoint set will change the data



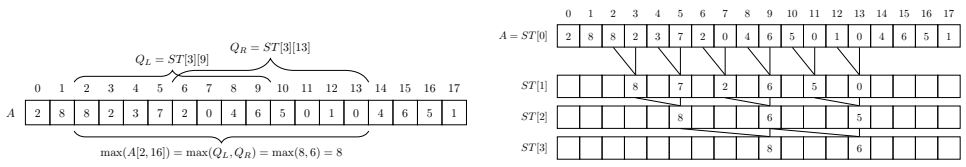
(a) The first stage (b) The second stage

Fig. 4. Two stages of the computation of one row of V.

structure because a lookup will *compress* the path to the root. It is difficult to maintain a consistent view of the data structure when multiple threads are compressing the path *simultaneously*. Second, when multiple threads are compressing different paths, the load among them could be very different, and this will incur load imbalance. Third, there will be a large number of threads working on different parts of the disjoint set, therefore it will be difficult to synchronize them efficiently.

2.4.1 Sparse Table. Since the disjoint set cannot be implemented efficiently in parallel, we use *sparse table* [3] to support incremental suffix/range maximum queries in our VGLCS algorithm. Sparse table [3] requires a $O(n \log n)$ preprocessing, and can support range maximum query in $O(1)$ time on one dimensional data. A sparse table is a two dimensional array. The element of a sparse table in the j-th row and i-th column is the maximum among the i-th elements and its $2^j - 1$ predecessors in the input array.

We give an example of the sparse table (Figure 5). The input is in array A. Then we build a sparse table T on A as described earlier. Now a ranged maximum query on A can be answered by at most *two* queries into the sparse table. For example, if the query is from 2 to 13, then the answer is the maximum of from 2 to 9 ($T[3][9]$), and from 6 to 13 ($T[3][13]$). Both are from the third level of the table since each has the maximum of $2^3 = 8$ elements in the input.



(a) Array (b) Sparse table

Fig. 5. A sparse table example

It is easy to see that one can build a sparse table in parallel efficiently. Please refer to Algorithm 2 for details. Algorithm 2 builds sparse table in parallel and in $O(n \log n/p + \log n)$ time, where n is the number of elements and p is the number of processors. This algorithm is very easy to parallelize and implement.

ALGORITHM 2: A Parallel Sparse Table Building Algorithm

Input: $A[0..N-1]$: the input array
Output: T: a sparse table for the input array A
Create a two-dimensional array $T[\log N][N]$;
Copy A to $T[0]$;
for $j \leftarrow 0$ *to* $\log N$ **do**
  for $i \leftarrow 2^j$ *to* N **do in parallel**
    $T[j][i] \leftarrow \max(T[j-1][i-2^j], T[j-1][i])$;
  end
end

2.4.2 A Parallel VGLCS with Sparse Table. The operations on a sparse table are much easily to parallelize than those on a disjoint set, which is used within the inner loop of Peng's sequential VGLCS algorithm. The inner loop of Peng's algorithm alternates between append and query operations on R. Please refer to Algorithm 1 for details. This alternation between appending and querying incurs heavy data dependency. In addition, the parallelism of operations on a disjoint tree is limited by the length of path under compression. The length is usually very short and provides very limited parallelism.

The pseudo code of our parallel VGLCS algorithm with sparse table is given in Algorithm 3. The algorithm computes V one row at a time. The computation of each row has two stages. In the first stage, the algorithm queries the C's *in parallel* to obtain the maximums of suffix of length $G_A + 1$ and place them into an array R. Then we build a sparse table T with the data of R. Then in the second stage, the algorithm queries T to find the range maximum in R to compute all elements in the i-th row of V *in parallel*.

The implementation of the two stages have different challenges. The first stage is easier to parallelize because the operations on individual columns are *independent*. However, it will insert new data into C, and still needs to answer suffix queries efficiently in order to build the R array. The second stage does *not* requires insertion so it is more static. However, since we compute all V's in the same row in parallel, it requires *ranged queries*, instead of suffix queries, on the sparse table T. The next two sections will describe our approaches to address these challenges of two stages. For ease of presentation we will describe our approach for the second stage in Section 3 first. Then we address the first stage in Section 4.

3 RANGE MAXIMUM QUERY

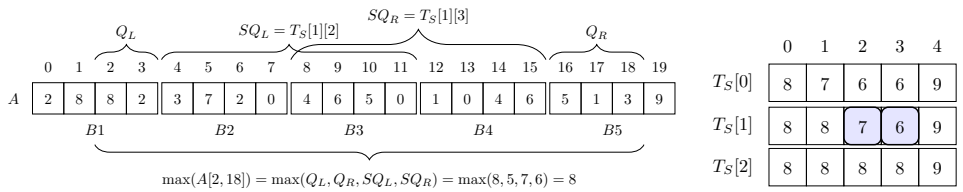
In this section, we will describe our approach to address the challenges in the second stage of Algorithm 3, i.e., an efficient *range maximum query*. The range maximum query problem is more complicated than the previous incremental suffix maximum query problem because the suffix maximum query is a special case of the range maximum query.

The operations to support range maximum query is similar to those for suffix maximum query. A Make operation creates an empty array A, an Append(V) operation appends a value V to the end of an array A. Finally, a Query(L, R) operation finds the *maximum* value among the L-th value to the R-th value of an array A.

ALGORITHM 3: Parallel Algorithm for Finding VGLCS

Input: A, B: the input string; G_A, G_B: the array of variable gapped constraints;
Output: Find the LCS with variable gapped constraints
Create an array of m data structure $C[m]$ to support ISMQ ;
Create an empty table $V[n][m]$;
for $i \leftarrow 1$ *to* n **do**
  for $j \leftarrow 1$ *to* m **do in parallel** *// The first stage*
    $M[j] \leftarrow$ suffix maximum of length $G_A(i) + 1$ from $C[j]$;
  end
  Build a sparse table T with data of M in parallel with Algorithm 2;
  for $j \leftarrow 1$ *to* m **do in parallel** *// The second stage*
    if $A[i] = B[j]$ **then**
      $t \leftarrow$ the range maximum among the $G_B(j) + 1$ elements before $M[j]$ by querying T ;
      $V[i][j] \leftarrow t + 1$;
      Append $V[i][j]$ to $C[j]$;
    end
  end
end
Retrieve the VGLCS by tracing $V[n][m]$;

3.1 Blocked Sparse Table



(a) Array (b) Blocked sparse table

Fig. 6. A Blocked Sparse Table

We improve the efficiency of our parallel VGLCS algorithm with a *blocked* sparse table proposed by Fischer [6]. The blocked approach first partitions the data into blocks of size s, then it computes the maximum of each block, and compute a sparse table T_s for these maximums. Recall that the *unblocked* sparse table in Section 2.4.1 does *not* partition data into blocks, but builds a table of $\log n$ rows of maximum for different lengths of intervals.

The blocked approach answers a range maximum query as follows. We consider two types of queries – *super block* query and *in-block* query. A super block query queries the answer for *consecutive* blocks, and an in-block query queries a segment *within* a block. It is easy to see that we can answer a super block query by querying T_s at most *twice*, as described in Section 2.4.1. We can also answer an in-block queries answered by a *single* lookup into an *least common ancestor table*. We will provide more details on this table later. Since we can split *any* range query into at most *two* queries into T_s and *two* in-block queries, we need at most *four* memory access to answer any range maximum query. Also since the super block query is easy to answer with T_s, we will now focus on in-block query.

Fischer's algorithm [6] scans through the data within a block and places them into a *Cartesian tree*. Each node of this Cartesian tree stores a data and the index of this data within the block. One can think of this Cartesian tree as a *heap* where the data are in heap order, and the indexes of the data are in *sorted binary search tree order*. Please refer to Figure 7 for an illustration. Note that in Figure 7 the horizontal position of tree nodes reflects their position in the data block. As a result, the answer to an in-block range maximum query from the i-th to the j-th element of a block is located at their *least common ancestor* in the Cartesian tree. For example, the maximum from the fourth (with data 3) to the six elements (with data 2) is located at the fifth element (with data 7).

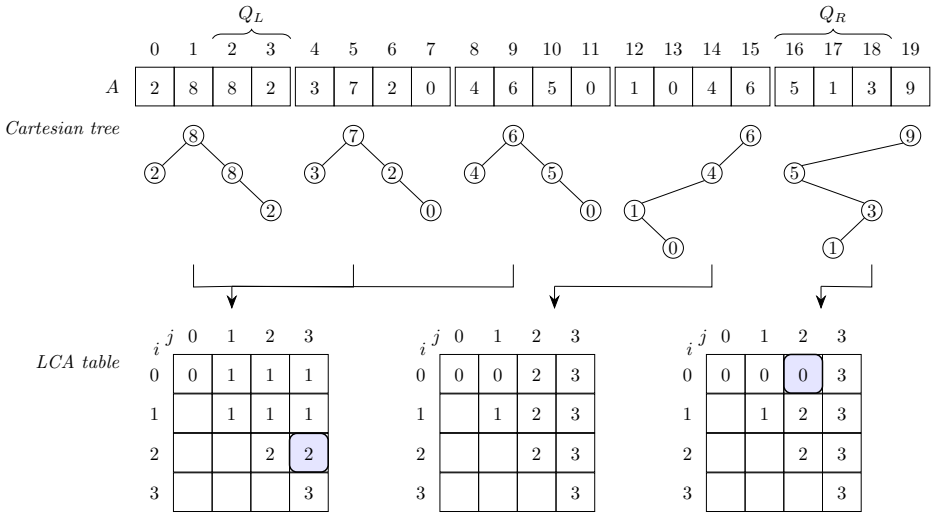


Fig. 7. Least common ancestor tables

To answer in-block queries, Fischer's algorithm computes a *least common ancestor table* for every block. After scanning the data in a block, Fischer's algorithm builds a Cartesian tree and its least common ancestor (LCA) table. An LCA table stores all $((i, j), k)$'s where the k-th data is the common ancestor for the i-th and j-th data in a block. Please refer to Figure 7 for an illustration. Now we can answer an in-block range maximum query from the i-th to the j-th element simply by look into the LCA table and return k, least common ancestor table of this block. One can think of the LCA table as a mapping table from an in-block query (i, j) to its answer k.

Note that the algorithm does *not* maintain the value of the k-th element. Instead, it stores the *index*, i.e., k, of the least common ancestor so that two blocks with the *same relative key order* can *share* a least common ancestor table. For example, the first three blocks in Figure 7 can share the same LCA table because they have the same Cartesian tree. Consequently, an in-block range maximum query $(1, 3)$ to *any* of these three blocks will return the *same* answer 2.

The main idea of Fischer's algorithm is to compute an LCA table for every block, and answer an in-block query directly by looking into its LCA table. It is easy to see that there are C_s different Cartesian trees, where C_s is number of different binary trees of s nodes. It is also easy to see that each block can be identified by the shape of its Cartesian tree, so it can be represented by an index. For ease of notation we will refer to this index as its *Catalan index*. By knowing the Catalan index of a block, we can answer an in-block range maximum query by looking into its corresponding LCA table. Please refer to Figure 7 for an illustration.

Fischer's algorithm [6] builds least ancestor tables by choosing $s = \frac{\log n}{4}$ as the block size for performance reason. Recall that $C_s = \frac{1}{s+1}\binom{2s}{s} = O(\frac{4^s}{s^{3/2}})$. As a result the time to scan data and to build Cartesian tree and LCA tables is $O(n)$, and the space requirement is $O(s^2 \frac{4^s}{s^{3/2}}) = O(n)$. That is, a sequential Fischer's algorithm requires $O(n)$ time in preprocessing, and $O(1)$ time to answer a query. It is easy to see that both preprocessing time and query answering are optimal.

One drawback of Fischer's algorithm is that it causes *serious cache miss* when the number of data is large. Fischer's algorithm will construct LCA tables for blocks *on-demand*. When the algorithm finds that the corresponding LCA table is *not* in memory, it will build the LCA table, which will be cached and this may *evict* other LCA tables from cache. This LCA table building process repeats for as many times as the number of blocks, and may cause serious cache misses.

In order to reduce cache miss, Demaine [4] proposed *cache-aware* operations on Cartesian tree [17]. Demaine's algorithm does not check if an LCA table is in memory – instead it builds LCA table for *every possible* block. To do so, Demaine's algorithm uses a binary string of length $2s$ to identify a block and its Cartesian tree. The binary string is encoded in such a way that one can answer an in-block range maximum query by examining this binary string only. However, this examination requires counting the number of 1's *between* the last two 0's, which is *hard* to implement efficiently in a modern computer.

3.2 Rightmost-pops Encoding

We propose a new encoding for blocks, called *rightmost-pops*, instead of the binary string by Demaine [4], in order to improve the performance of range maximum query. This rightmost-pops encoding is inspired by Demaine's algorithm and Cartesian tree.

The rightmost-pops encoding encodes a Cartesian tree by keeping only the *rightmost* path of the Cartesian tree in a *stack*, and keeping track of the *number of pops* from the stack when we add data into the Cartesian tree. Please refer to Figure 8 for an illustration. To maintain the heap property of the Cartesian tree, when we add the i-th data a_i into the Cartesian tree, we need to *pop* the data in the stack, which stores the rightmost path of the Cartesian tree, that are *smaller* than a_i. We keep popping data until the top of stack is no less than a_i, then we push a_i into the stack. Let t_i be the number of nodes that need to be popped, and it is easy to see that $0 \le t_i < s$, where s is the block size. We use these t_i, the number of pops on the right most path, to encode a Cartesian tree.

Consider the example in Figure 8. When we insert $a_1 = 0$, we just insert it into the stack since $a_0 = 1$, no data is popped, and t_1 is 0. When we insert $a_2 = 4$ we need to pop both a_0 and a_1 out of the stack, since they are smaller than a_2, so t_2 is 2. As defined earlier, the contents of the stack is exactly the rightmost path of the Cartesian tree, and we keep these t_i's to represent a Cartesian tree.

The key idea of rightmost-pops encoding is that we can use t_i's to *implicitly identify* the Cartesian tree of this block of data, and that we can answer in-block range queries simply by examining these t_i's. Suppose we want to answer an in-block maximum query that ranges from l to r with these t_i's. We maintain the number of times data are *popped* from the stack in a variable x, and initialize x to 0. We then loop through t_l to t_r and let index j run from l to r. Every iteration adds 1 to x then subtracts t_j from x. We need to remember the index j when x becomes smaller or equal to 0. Finally, we report the *last* index j when x becomes smaller or equal to 0, as the answer. The pseudo code of the query answering algorithm is in Algorithm 4. All the operations of Algorithm 4 map directly to machine instructions so that unlike Demaine's algorithm, Algorithm 4 is extremely intuitive to implement.

The correctness proof of Algorithm 4 is in Theorem 3.1. The intuition is that we only need to know the root of the tree when the last element in range was inserted.

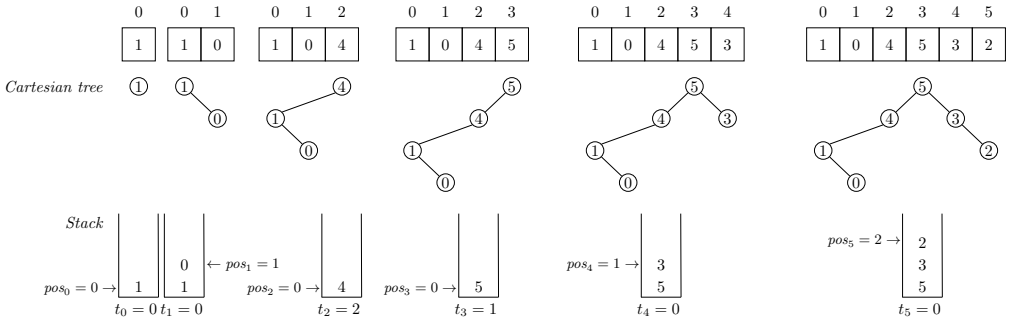


Fig. 8. Rightmost path in stack

ALGORITHM 4: Range Minimum Query in 64-bits Cartesian Tree

Input: *tmask*: 64-bits Cartesian tree; $[l, r]$: query range
Output: *minIdx*: the index of the minimum value in interval

```
minIdx ← l, x ← 0 ;
for l ← l + 1 to r do
    x ← x + 1 − ((tmask ≫ (l ≪ 2)) & 15) ;
    if x ≤ 0 then
        minIdx ← l, x ← 0 ;
    end
end
return minIdx ;
```

Theorem 3.1. *Algorithm 4 correctly answers an in-block range maximum query.*

Proof. When x, the number of poppings, is *smaller than or equal* to 0, it means the added data has become the *root* of the tree of the queried range. As a result, when we add a data and it became the root of the tree for the *last* time, the added data is indeed the maximum among this interval, because according to the heap property, the root is the maximum among all nodes within this tree. □

We also note that since all t_i's are small than 16, we can represent each of them as a 4-bit integer. We then concatenate sixteen of these 4-bit integers into a 64-bit integer to present a Cartesian tree for a block of sixteen data. The pseudo code on how to build a 64-bit integer to represent a block of 16 data is in Algorithm 5, which runs in time $O(s)$. where s is the block size. Note that all the operations, e.g., shift, addition, subtraction, in Algorithm 5 map directly to machine instructions and are straightforward to implement.

We choose the block size as $s = \frac{\log n}{4} = 16$ for performance reason. Modern CPUs support 64-bit register and fast operation on them. When we pack 16 t_i in to a 64-bit integer, we can leverage fast 64-bit instructions and improve performance. In addition, we do *not* build least common ancestor tables *explicitly* since we implicitly maintain the Cartesian tree information within these 64-bit t_i. This approach reduces memory usage and improves cache performance, it can also efficiently answer one-dimension range maximum query for up to $n = 2^{64}$ data.

Note that our rightmost-pops encoding does improve cache performance, but will increase the time complexity in answering queries. Algorithm 4 accesses data in a very regular manner and

ALGORITHM 5: Encode a data block of sixteen data with rightmost-pops encoding into a 64-bits integer.

Input: $A[1..16]$: input data block; s: the fixed block size as $\frac{\log n}{4} = 16$
Output: t: a 64 bit rightmost-pops encoding of A
Create an array D of size $s + 1$;
$p \leftarrow 0, D[0] \leftarrow \infty$;
$t \leftarrow 0$;
for $i \leftarrow 1$ *to* s **do**
    $v \leftarrow A[i],\ c \leftarrow 0$;
    while $D[p] < v$ **do**
        $p \leftarrow p - 1, c \leftarrow c + 1$;
    end
    $p \leftarrow p + 1$;
    $D[p] \leftarrow v$;
    $t \leftarrow t \mid (c \ll ((i - 1) \ll 2))$;
end
return t ;

has a better data locality than Fischer's algorithm. The preprocessing time is $O(n)$, as same as in Fischer's algorithm. However, a single query now needs $O(s)$ time, where s is the block size $\frac{\log n}{4}$. This is acceptable in practice since we set the block size s to 16 for n up to 2^{64}, so s is a small constant. The space complexity is $O(n)$ as in Fischer's algorithm. The overall time complexity of the parallel version of our VGLCS algorithm becomes $O(n^2 \log n/p + n \log n)$. Note that the $\log n$ comes from the block size $s = O(\log n)$

4 MAXIMUM INTERVAL QUERY ON INCREMENTAL DATA

This section describes our approach to address the challenges in the first stage of Algorithm 3, where we compute the suffix maximum on every *column* of V while new data are added. Here we generalize our technique so that we can also answer incremental *range* maximum queries on incrementally added data, so that our technique can be applied to other cases that require queries on an interval.

4.1 Build Least Common Ancestor Table

Recall from the discussion of Cartesian tree in Section 3 that finding the *least common ancestor* is important for answering range maximum queries. Here we need to address two issues – how to label a binary tree with a *Catalan index* and how to find the least common ancestor of two nodes in a given Cartesian tree.

4.1.1 Cartesian Tree Labeling. Our Cartesian tree labeling will enumerate all binary search trees in *lexicographical order* from 0 to the n-th Catalan number minus 1. This lexicographical order among binary search trees of the *same* number of nodes is defined *recursively* as follows. A binary tree x appears *before* another binary tree y if any of the following condition is true. Figure 9 shows our lexicographical labeling of binary search trees of sizes 1, 2 and 3.

- x has more nodes than y in the left subtree.
- x and y have the same number of nodes in the left subtree, and x's left subtree appears before y's left subtree in lexicographical order.

- x and y has the same left subtree, and x's right subtree appears before y's right subtree in lexicographical order.

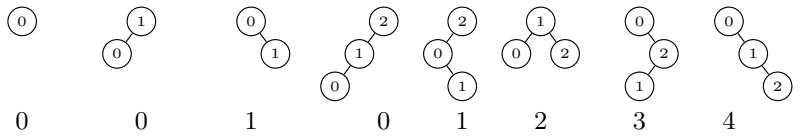


Fig. 9. The labeling of binary search trees of sizes 1, 2, and 3.

4.1.2 Least Common Ancestor. We also need to determine the *least common ancestor* efficiently for answering range maximum queries when data are incrementally added. Let t be the Catalan index of the binary search tree from our lexicographical labeling, so t is between 0 and $C_s - 1$, where s is the number of search trees. Let $\mathcal{A}(s, t, p, q)$ denote the least common ancestor of the node p and q within a binary search tree of size s that has a Catalan index t. For example, $\mathcal{A}(3, 2, 0, 2) = 1$ from Figure 9. Now let s_l denote the size of the left subtree, s_r denote the size of the right subtree, t_l be the Catalan index of its left subtree, and t_r be the Catalan index of its right subtree. With these notations we can define the least common ancestor $\mathcal{A}$ *recursively* as in Equation 1. A Pseudo code for computing the LCA table $\mathcal{A}$ is given in Algorithm 6.

$$\mathcal{A}(n, t, p, q) = \begin{cases} \mathcal{A}(s_r, t_r, p - s_l - 1, q - s_l - 1) + s_l + 1 & s_l \le p \le q < n \\ \mathcal{A}(s_l, t_l, p, q) & p \le q < s_l \\ s_l & 0 \le p \le s_l, s_l \le q \le i \\ -1 & \text{otherwise} \end{cases} \tag{1}$$

ALGORITHM 6: A parallel algorithm that computes the least common ancestor table $\mathcal{A}$

Input: s: the maximum tree size

```
1  for n ← 1 to s do
2      for t ← 0 to C_n − 1 do in parallel
3          for p ← 0 to n − 1 do in parallel
4              Compute s_l, t_l, s_r, and t_r ;
5              for q ← p to n − 1 do
6                  Compute A[n][t][p][q] according to Equation 1 ;
7              end
8          end
9      end
10 end
```

We first analyze the space complexity of Algorithm 6. The table $\mathcal{A}$ keeps all LCA information for binary trees of sizes from 1 to s. When tree size is m, the number of different binary trees is the m-th Catalan number C_m, which is $\frac{1}{m+1}\binom{2m}{m} = O(\frac{4^m}{m^{3/2}})$. For each binary tree of size m, we store the least common ancestor of *every* pair of nodes into the table, so the size of the table is $O(m^2)$. Therefore, the space complexity is $O(s\ \frac{1}{s+1}\binom{2s}{s}\ s^2)$, where s is the number of elements in a block. When we set s to $\frac{\log n}{4}$, the space complexity is $O(\sqrt{n}\ \log^{3/2} n)$. Note that the size of range query is up to tree size s. However, we do need the space for tables of *smaller* tree sizes as intermediate data to compute the table of tree size s.

We now analyze the time complexity of the parallel version of Algorithm 6. The time complexity of a sequential version of Algorithm 6 is $O(\frac{s^3}{s+1}\binom{2s}{s}))$ because the number of operations in Equation 1 is $O(1)$. The time complexity hence becomes $O(\sqrt{n}\ \log^{3/2} n)$ when we set s to $\frac{\log n}{4}$.

Now for the parallel version, we observe that the computation of the C_m trees of size m are *independent*, hence can be done in parallel. However, the time to find the sizes and Catalan indices of subtrees (in line 4) is $O(m)$ for a tree of size m. Since both line 4 and 5 are in the same loop body, it is not necessary to parallelize line 4 because the loop starting at line 5 will dominate the time. As a result, we only parallelize the double loops in line 2 and 3 in Algorithm 6, and the time complexity of our parallel algorithm is $O(\frac{s^3}{s+1}\binom{2s}{s}/p + s^2) = O(\sqrt{n}\ (\log^{3/2} n)/p + \log^2 n)$, where p is the number of processors.

4.2 Catalan Index Computation

Note that Algorithm 6 requires Catalan index t, so we need to determine t efficiently when given a block of data. There are two possible approaches – build the tree or keep only the rightmost path.

4.2.1 Build the Tree. In order to find the Catalan index of a block of data, we can build a Cartesian tree corresponding to the data of the block, and then find the index of the Cartesian tree. That is, we build the tree and compute it from the the sizes and indices of the left and right subtrees. This requires a recursive traversal on the tree and has a $O(n)$ time complexity, where n is the number of tree nodes. Let $\mathcal{T}$ denote the Catalan index. After knowing these information, we can compute the Catalan index $\mathcal{T}$ by Equation 2. Recall that s_l denotes the size of the left subtree, s_r denotes the size of the right subtree, t_l is the Catalan index of the left subtree, and t_r is the Catalan index of the right subtree. Recall that C_i denotes the Catalan number of tree of size i.

$$\mathcal{T}(s_l, t_l, s_r, t_r) = t_l\ C_{s_r} + t_r + \sum_{i=0}^{s_l-1} C_i\ C_{s_l+s_r-i} \tag{2}$$

We can further optimize Equation 2 by precomputing the *prefix sum* of the Catalan number products. Then we store these sums in memory, so that we can use them directly, instead of recomputing them as in Equation 2.

4.2.2 Keep the Rightmost Path. The previous computation of Catalan index requires building the tree to obtain subtree information, and may not be efficient. We propose a more efficient method that determines the Catalan index by keeps only the *rightmost path* in a *stack*, without building the entire tree. This technique is similar to the *rightmost-pops encoding* in Section 3.2. After knowing the Catalan index t, we can compute LCA and answer queries with Algorithm 6 and Equation 2.

We compute the Catalan index t efficiently by the maintaining its *rightmost path* in a *stack*. That is, we will *not* build these left subtrees along the rightmost path. Instead, we keep their Catalan indices and sizes in the stack (denoted by D). That is, the stack D has the Catalan *indices* and *sizes* of every *left subtree* along the rightmost path. Please refer to Algorithm 7 for the details on the stack. In Algorithm 7 each node of the stack D has three members – v as the data, s as the size of its subtree, and t as the index of its left subtree. We also use a pointer p to point to the top of the stack.

Algorithm 7 computes the Catalan index for a given block of data with a stack D. In the first double loop, the outer loop goes through every input and the inner loop inserts a data at the *end* of the rightmost path, which is at the top of the stack $D[p]$, and traverse towards the root by popping any *smaller* data out of the stack D. When we rotate nodes along the rightmost path to update the Cartesian tree, we compute the new *index* t and size s of the new left subtree whenever the newly inserted data replaces a node on the rightmost path. As a result, the new Catalan index t can be

ALGORITHM 7: Catalan index computation for a data block

Input: $A[1..n]$: input data block; n: the number of elements;
Output: t: The Catalan index of the input data block
Create a stack D of $n+1$ elements. Every element has s, t, and v ;
$p \leftarrow 0$, $D[0] \leftarrow \langle 0, 0, \infty \rangle$;
for $i \leftarrow 1$ *to* n **do**
    $v \leftarrow A[i]$;
    $s \leftarrow 0$, $t \leftarrow 0$;
    while $D[p].v < v$ **do**
        $t \leftarrow \mathcal{T}(D[p].s, D[p].t, s, t)$;
        $s \leftarrow s + D[p].s + 1$;
        $p \leftarrow p - 1$;
    end
    $p \leftarrow p + 1$;
    $D[p] \leftarrow \langle s, t, v \rangle$;
end
$s \leftarrow 0$, $t \leftarrow 0$;
while $p > 0$ **do**
    $t \leftarrow \mathcal{T}(D[p].s, D[p].t, s, t)$;
    $s \leftarrow s + D[p].s + 1$;
    $p \leftarrow p - 1$;
end
return t ;

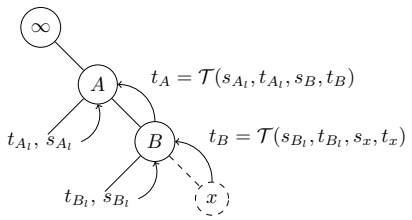


Fig. 10. Compute Catalan index for a tree. A_l and B_l denote the left subtrees of A and B respectively.

recomputed with Equation 2 by the indices and sizes of the left and right subtrees in the stack. Please refer to the first while loop of Algorithm 7 and Figure 10 for an illustration. Note that A_l and B_l denote the left subtrees of A and B respectively. After popping all smaller data in the stack the while loop stops and the size, index, and input are pushed into the new top of stack. Finally, we pop all data out of the stack and compute the Catalan index for the entire block in the last loop of Algorithm 7.

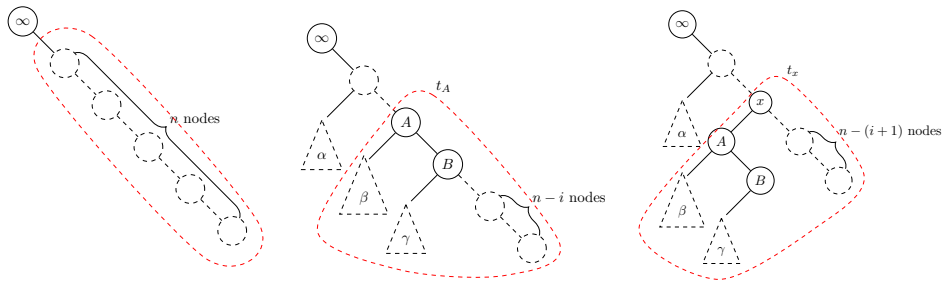
Algorithm 7 can compute any Catalan index for Cartesian trees with the entire block of data and the block size. The algorithm runs in $O(s)$ time since an element is pushed/popped at most *once*.

4.3 Dynamic Catalan Index Computation

Several encoding methods were proposed for indexing Cartesian trees. Fischer [6] introduced the first encoding method and Masud [9] presents a new encoding method that reduces the number of instructions. Unfortunately all these algorithms are *off-line*, i.e., they assume all data are given

in advance, as a result, they cannot cope with incrementally added data. In addition, they require a preprocessing of time $O(n)$, where n is the number of data. The preprocessing requires extra memory to process the input data block, or reads external files from disk.

We generalize our Cartesian tree labeling technique for incrementally added data by a *normalization*, which keeps the size of tree as a constant n. That is, suppose we want to encode a series of Cartesian trees of *increasing* sizes up to n, we can append a path of $n - i$ *right-child-only* nodes to the rightmost path of the existing tree of size i, so that the total number of node is *always* n. That is, when the computation starts without any existing tree nodes, we will start with a path of n right-child-only nodes. Please refer to Figure 11(a) for an illustration. After we have added i nodes, we will have a path of $n - i$ right-child-only nodes attached to the rightmost path, as illustrated by Figure 11(b). For ease of explanation we will refer to this added path as the *virtual path*.



(a) Initiation $t_{root_0} = C_n - 1$ (b) t_{root_i} (c) $t_{root_{i+1}} = t_{root_i} + t_x - t_A$

Fig. 11. Normalization of Cartesian trees of increasing sizes by adding a virtual path.

There are two advantages of this tree normalization. First, this normalization simplifies the computation of Catalan index. That is, we can update the Catalan index with ease whenever we insert a new data. Second, this normalization simply adds increasingly *larger* data to the end of a data block, so that the answers to the interval maximum queries on actual data are *not* affected. As a result we can *dynamically* maintain a lookup table to obtain the maximum value within a range, so as to answer a range query from this dynamic Cartesian tree encoding.

The dynamic Catalan index computation with tree normalization works as follows. We will maintain the Catalan index of the current Cartesian tree (as t^*), and a stack of the rightmost path of the current tree (as D). Then we will *update* these information every time we add a new data x into the tree, by calling Algorithm 8, which is based on Algorithm 6 and Equation 2. At any given time i, where i is the index of the data, we can compute the Catalan index of the current tree, which has i nodes, and use it to answer a range maximum query.

For ease of notation, we use t and s to denote the Catalan index and the size of the Cartesian tree *without* the virtual path. These notations are similar to those in Algorithm 7. We also use t' and s' to maintain the index and size of the tree *with* the virtual path. Please refer to Figure 11 for an illustration, where the tree enclosed by the red dotted line does include the virtual path.

Algorithm 8 first initializes s and t as Algorithm 7 does, since it will also work from the end of the rightmost path and start with an empty tree. Then Algorithm 8 initializes s' to $n - i$ since we will append the a virtual path of length $n - i$ to the end of the rightmost path. Consequently, t' is $C_{s'} - 1$ since it has the largest Catalan index of that size.

Algorithm 8 then inserts the i-th element x in two stages. The line 4 and 5 of Algorithm 8 pops the elements smaller than x from the stack D, using p as the stack pointer of D. Note that t and

s are the index and size of the last subtree *without* the virtual path. Consequently, we popped smaller elements from the stack and update s and t just as in Algorithm 8. In line 6 and 7 of the loop Algorithm 8 updates s' and t' similarly. However it must maintain a virtual path of $n - s'$ right-child-only nodes to the right of the rightmost path. Please refer to the Algorithm 8 and Figure 11 for details.

ALGORITHM 8: Online Type of Cartesian Tree

Input: x: the added data ;
t^*: The Catalan index of the tree before adding x ;
i: the index of x ;
D: A stack where an element has s and t ;
Output: t^*: The Catalan index of the tree after adding x ;
D: A updated stack where an element has s and t

```
1  s ← 0, t ← 0 ;
2  s' ← n − i + 1, t' ← C_{s'} − 1 ;
3  while D[p].v < x do
4      t ← T(D[p].s, D[p].t, s, t) ;
5      s ← s + D[p].s + 1 ;
6      t' ← T(D[p].s, D[p].t, s', t') ;
7      s' ← s' + D[p].s + 1 ;
8      p ← p − 1 ;
9  end
10 p ← p + 1 ;
11 D[p] ← ⟨s, t, x⟩ ;
12 t* ← t* + t' − T(s, t, s − i, C_{s−i} − 1) ;
13 return t* ;
```

Now we are ready to update the stack D and return the overall Catalan index t^*. The new top of stack is now x, with left subtree of size s and index t, so we push them into the stack. Please refer to Figure 11(c) for an illustration. Next we update the overall Catalan index t^*. After we popped all elements smaller than x, the original subtree of these popped nodes on the rightmost path, should be inserted as the left subtree of x. Please compare Figure 11(b) (before) and Figure 11(c) (after).

We observe that the only difference between Figure 11(b) and Figure 11(c) is the area enclosed by the red dotted line. If we compute the *difference* between the Catalan indices of these two red dotted line area, we can compute the final t^* (Figure 11(c)) by *patching* the Catalan index in Figure 11(b) with this difference. Fortunately our tree lexicographical ordering considers left subtree *before* right subtree, so the difference of the Catalan indices between Figure 11(b) and Figure 11(c) is $t' - \mathcal{T}(s, t, s - i, C_{s-i} - 1)$. As a result, the new t^* is the old value of t^*, plus $t' - \mathcal{T}(s, t, s - i, C_{s-i} - 1)$.

Note that Algorithm 8 does *not* increase the amortized time complexity to compute the Catalan index, even when we dynamically add data into the tree. As described earlier, the time complexity of the static index computation (Algorithm 7) is $O(n)$ because each element is popped exactly once. Similarly, the total cost of calling Algorithm 8 n times is also $O(n)$, since each operation within the loop is $O(1)$. As a result, the amortized cost for an in-block query, including the computation of Catalan index, is also $O(1)$.

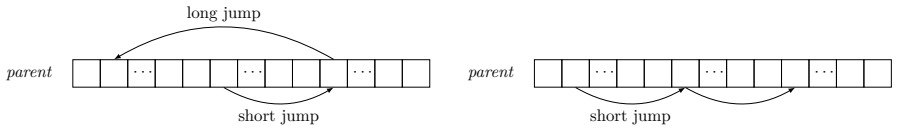
5 IMPLEMENTATION OPTIMIZATION

This section describes optimization in our implementations of VGLCS algorithms, both in sequential and parallel environments. Note that some of these techniques address hardware characteristics, e.g., memory cache behavior, and are not confined to asymptotically analysis only.

5.1 The Strategy of Disjoint Set Implementation

We now describe the optimization in the implementations of disjoint sets, whose applications include VGLCS.

5.1.1 Cache Performance. Memory cache is essential to efficient implementation of disjoint sets. Patwary, Blair, and Manne [13] conducted experiments on disjoint-set and showed that different implementations have different impact on different levels of caches misses. In practice, the cache miss is strongly related to how we go from a child to its ancestors through pointer chasing during path compression. Usually, an algorithm with a lower time complexity, e.g., will have more "long jumps" than an algorithm with a higher time complexity. Here the long jump means the pointer chasing will go from a memory address to a far away memory address. Please refer to Figure 12 for an illustration.



(a) An algorithm with a lower time complexity (b) An algorithm with a higher time complexity

Fig. 12. The parent jump in disjoint set

The Rem's algorithm [5] achieves better cache performance by a *merge-by-index* technique. Traditional disjoint set merging techniques are *merge-by-rank* and *merge-by-size* [16], which determine the root of new tree by the ranks and the sizes of the two trees respectively. Despite that they do have asymptotically better time complexity, their performance in practice is not so impressive due to the previously mentioned cache issue. In contrast, Rem's algorithm assigns an *index* to each node, and merges two disjoint sets according the index of the roots of the two trees being merged. That is, the root of the new merged tree will be the root with the "larger" index.

In our experiments, we still use merge-by-rank to merge disjoint set trees. However, when we have two *equally ranked* disjoint set trees to merge, we will apply the *merge-by-index* technique to break the tie, so as to improve cache performance. Figure 13 compares the cache miss rate of merge-by-index tie-breaking and the usual random tie-breaking approach in both merge-by-rank and merge-by weight, and under different application contexts. The experiments show that the merge-by-index tie-breaking technique in average reduces cache miss rate by 3% of the usual random tie-breaking approach.

5.1.2 Application on VGLCS. It is possible to further optimize the implementation of Peng's sequential algorithm (Algorithm 1) by a *lazy insertion* technique. Recall that in Algorithm 1, when the i-th character from the first input string does not match the j-th character of the second input string, we need to place a zero into the dynamic programming table at position $V[i][j]$. In practice, this mismatch will happen *very frequently*, and will cause frequent insertions of zeros into the

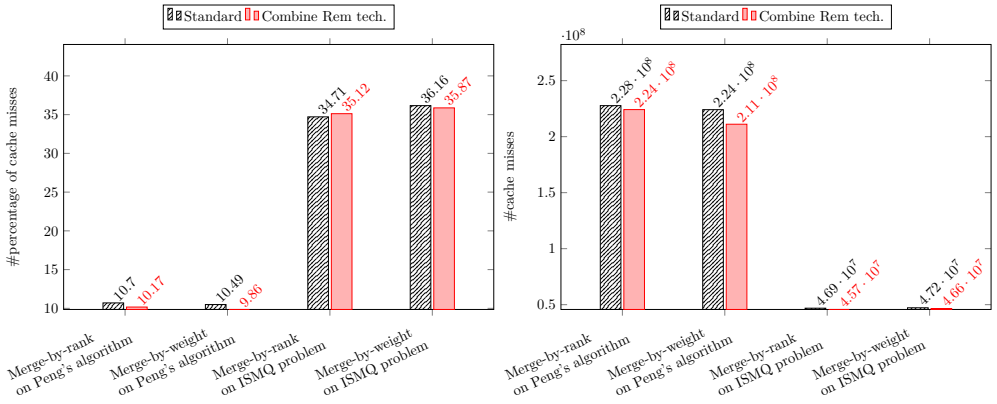


Fig. 13. The cache miss rate of the *merge-by-index tie-breaking* for different applications and merge techniques in disjoint set data structure on an E5-2620 server. The statistics are measured by the `perf` analyzing tool in Linux.

disjoint sets. Each insertion then links the newly inserted singleton node of zero to the node of zero inserted one iteration ago.

We implemented an optimization that resolves this repeated insertion, and linking, of zeros. Our implementation will scan through a series of zeros, and locate the next non-zero V element (denoted as v) in the same column, and insert them as a batch. Since all values in V are non-negative, we can link all zeros we have seen *directly* to v. From the experiments, we observe that this technique causes less pointer chasing and updates and does improve performance.

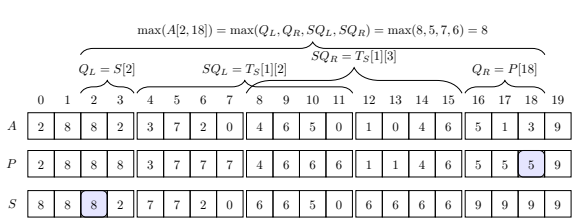
Note that we do *not* apply the lazy insertion optimization in our *parallel* implementation. For a multi-core platform, the efficiency of thread synchronization is essential to the performance. Since the threads of a parallel VGLCS algorithm needs to synchronize at every column of the dynamic programming table in order to process a row, the lazy insertion, which can ignore this dependency in a sequential environment, is not beneficial, and is not adopted in our parallel implementation.

5.2 Parallel Range Query in VGLCS

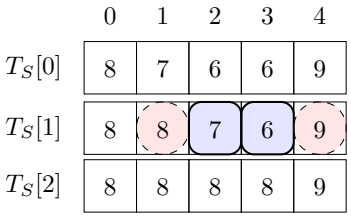
We further improve the performance of range query by maintaining two extra tables in the blocked sparse table approach described in Section 3.1. There are now three tables – a sparse table on the block maximum T_S, a *prefix maximum table* P, and a *suffix maximum table* S. As described in Section 3.1, the sparse table T_S can easily answer range query of consecutive blocks on the input A. The prefix maximum table P maintains the maximum of the prefix *within* a block, and similarly the suffix maximum table S maintains the maximum for prefix. Please refer to Figure 14 for an illustration.

It is easy to see that any range query can be answered by *two* queries into sparse table T_S, *one* query into prefix maximum table P, and *one* into suffix maximum table S. For example, in Figure 14, the query from index 2 to 18 can be answered by two queries into the sparse table T_S – one from block 1 to block 2, and one from 2 to 3, a query for the suffix of length 2 into the first block of S, and a query for the prefix of length 3 into the last block of P. Please refer to Figure 14 for an illustration.

We argue that the *order* to access these maximum tables is important. Our implementation accesses the block maximum T_S *first*, then the prefix maximum P *second*, then the suffix maximum S *last*. The reasoning for this order is as follows. Since we need to access two elements in the sparse table T_S in *the same level*, it is very likely that they will be in the same cache line, so access the



(a) The prefix/suffix maximum tables for blocks



(b) A Sparse Table for T_S

Fig. 14. Block maximum T_S, prefix maximum P, and suffix maximum S.

first will bring in the other automatically by the hardware caching mechanism. For example, in Figure 14(b), we will access both $T_S[1][2]$ and $T_S[1][3]$ in order to find the maximum from block 1 to 3.

Our implementation improves performance by "peeking" into two neighboring elements in the T_S table. To be more specific, we peek the element *before* the first element, and the one *after* the second element, which we just accessed from T_S.

For example, in Figure 14(b), we will peek into $T_S[1][1]$ and $T_S[1][4]$, which are very likely also present in cache because they are in the same level of the sparse table as the previously accessed elements $T_S[1][2]$ and $T_S[1][3]$. The idea is that if the maximum from the block maximum, i.e., the maximum of $T_S[1][2]$ and $T_S[1][3]$, is already *larger* than the overall block maximum where the prefix belongs, i.e., in $T_S[1][4]$, then we do *not* need to check the P table. Similarly, if the maximum from the block maximum *and* the prefix maximum is larger than the block maximum where the suffix belongs, i.e., in $T_S[1][1]$, then we do not need to check the S table either. This peeking is efficient since the data being peeked are most likely in the same cache line, and can save unnecessary access to P and S tables.

Algorithm 9 shows the access order for the range maximum query to reduce the cache miss. Note that we check the prefix maximum P *before* the suffix maximum S since the value of the dynamic programming table in the same row is *increasing*, so it is more likely that a prefix maximum, which has a larger column index, can save a check into the suffix table, which has a smaller row index.

Our experiments show that despite that these maximum arrays require additional $O(n)$ memory, the peeking technique improves overall performance by up to 8% when $n = 10^4$, and it improves the overall performance of query operations by up to 35% when $n = 10^7$. Please refer to Figure 15 for an illustration.

6 EXPERIMENT

We conduct three sets of experiments. The first set compares the performance of blocked and unblocked spare tables in supporting parallel range maximum query. The second set evaluates the performance of incremental suffix maximum query using various data structures, especially the blocked sparse tables. Finally, the third experiment evaluates the effects of different data structures on the overall performance of VGLCS computation.

We conduct experiments on an Intel Xeon E5-2620 2.4 Ghz processor with 384K bytes of level 1 cache, 1536K bytes of level 2 cache, and 15M bytes of shared level 3 cache. The Intel CPU supports hyper-threading, and each processor has six cores. The operating system is Ubuntu 14.04. We

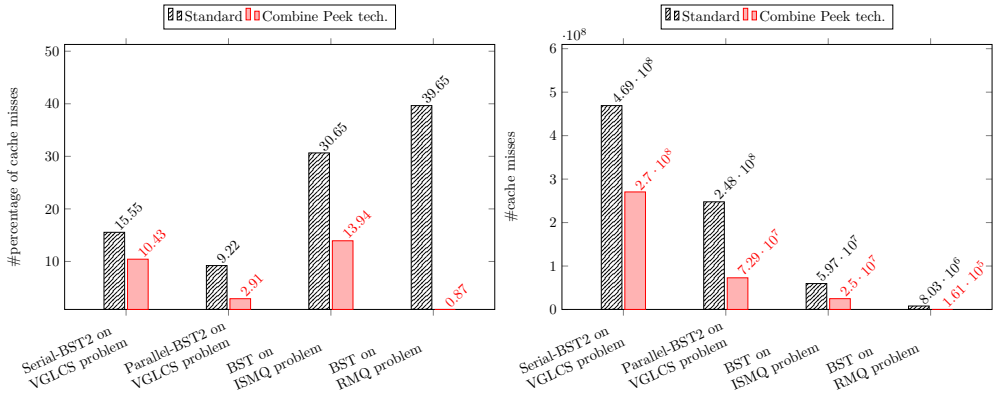


Fig. 15. The cache miss rates of the "peeking" for different applications and environments in blocked sparse table on an E5-2620 server. The cache miss rate is measured by the `perf` analyzing tool in Linux.

ALGORITHM 9: The process of answering a range query. Note that it accesses T_S, then P, then S.

Input: A: the input array;
P, S: the prefix/suffix maximum arrays for each block ;
$\mathcal{T}$: the Catalan index array for each block ;
T_s: the blocked sparse table ;
$[l, r]$: the bound of the ranged query ;
Output: v: the maximum value in $A[l..r]$;
if *l and r are in the same block b* **then**
  $i \leftarrow$ Query the ranged maximum query $[l, r]$ on $\mathcal{T}_k$, where k is the Catalan index of the b ;
  return $A[i]$;
end
$v \leftarrow -\infty$;
$l' \leftarrow$ The starting index of the next block from the position l ;
$r' \leftarrow$ The ending index of the previous block from the position r ;
$t \leftarrow \lfloor \log_2(r' - l' + 1) \rfloor$;
if $l' \le r'$ **then**
  $v \leftarrow \max(T_s[t][l' + 2^t - 1], T_s[t][r'])$;
end
if $T_s[t][l' + 2^t - 2] > v$ **then** *// Access P only when necessary*
  $v \leftarrow \max(v, P[r])$;
end
if $T_s[t][r' + 1] > v$ **then** *// Access S only when necessary*
  $v \leftarrow \max(v, S[l])$;
end
return v ;

implemented all algorithms in C++ and OpenMP and compiled them using gcc with `-O2` and `-fopenmp` flag.

6.1 Blocked and Unblocked Sparse Table

We first compare the performance of *unblocked* sparse table (Section 2.4.1) and *blocked* sparse tables (Section 3.1) for supporting parallel range maximum query. For the blocked sparse table, we use the rightmost-pops encoding instead of the LCA tables, since we will show that the rightmost-pops encoding is more efficient than LCA table in the next set of experiments. We test all possible query range sizes and the number of range queries is set to be the number of elements N.

Table 1 compares the parallel range query time of unblocked sparse table and blocked sparse table, and we observe that the blocked sparse table using rightmost-pops sparse table is more efficient than the unblocked sparse table when N increases. For example, the blocked sparse table is 1.4 times faster than the unblocked sparse table when N reaches 10^5.

We believe that the maximum length of interval query (L in Table 1) affects the cache performance of accessing a sparse table. In addition, even when we fix the number of data N, the speedup of blocked sparse table over unblocked one does increase as L increases. We believe that the reason of this speedup is as follows. Even with a very large query length, a blocked sparse table will only need to access a limited range of memory. For example, the rightmost-pops encoding only needs to access two elements that are $\log N/s$ apart in memory (the sparse table for the block maximums) for super block query, and two in-block queries $O(s)$ in consecutive memory. In contrast, the unblocked sparse table can have up to $\log N$ levels, and different queries of different sizes will need to access different levels of the sparse table. This does not help cache locality because consecutive queries will unlikely to access consecutive memory. When N is small this will not be problem for unblocked sparse table since its size will be small. When N increases the cache locality of blocked sparse table becomes more significant.

Table 1. Total running time (ms) for finding RMQ of different sizes N and maximum interval sizes L.

	$N = 30000$			$N = 50000$			$N = 100000$			
Method \ L	2^{10}	2^{12}	2^{14}	2^{10}	2^{12}	2^{14}	2^{10}	2^{12}	2^{14}	2^{16}
$ST_{standard}$	10.0	11.7	12.9	13.7	15.7	17.5	23.0	26.1	30.1	35.1
$BST_{rightmost-pops}$	9.6	9.8	11.3	13.8	13.5	13.6	23.8	22.8	22.9	24.9
speedup	1.01	1.19	1.14	0.99	1.16	1.28	0.96	1.14	1.31	1.40

6.2 Rightmost-pops and LCA Tables

We now compare the performance of *four* data structures for supporting *incremental suffix* query. The four data structures are *disjoint set*, *unblocked* sparse table, blocked table with *rightmost-pops encoding*, and blocked sparse table with LCA. Note that since we are testing suffix queries so we can use disjoint set. Please refer to Section 3 for details. For disjoint set, we implemented both *path compression* and *merge-by-rank* strategies so that the amortized time $O(\alpha(n))$. In our implementation of blocked sparse tables, we set the block size s to 8. All sparse tables are allocated in memory in a row-major manner to reduce cache miss. Please refer to Algorithm 3 and Figure 5 for details.

We first tried a simple scenario in which we alternate between *appending* a data and *querying* a range. Both the length and the position of the range query are uniformly distributed among all possibilities. Figure 16 shows the performance of the four data structures for supporting incremental suffix maximum queries under this simple scenario. We note that our rightmost-pops encoding implementation runs faster than all other implementations. For example, it runs faster than the

disjoint set implementation when the number of data n reaches 10^6. In particular, when n is greater than 10^7, the rightmost-pops encoding runs 1.8 times faster than the disjoint set.

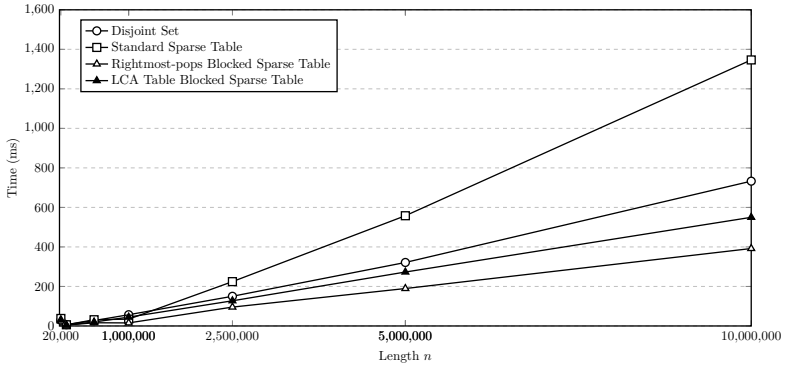


Fig. 16. The performance of the different data structures for supporting incremental suffix maximum query on an E5-2620 server

Now we conduct another experiment in a more complex scenario. In a dynamic programming various factors affect performance, these factors include the distribution of values being inserted, the maximum interval being queried, and the ratio between the numbers of insertion and queries. For ease of notation, we use p to denote the probability of insert a larger next element, q for the probability of inserting a zero, and L for the maximum interval length being queried. In our experiments, we set the number of data N to be 10^7, and vary the maximum interval sizes L from 4 to 16, and vary p and q from 0 to 100%. We also set the number of the queries to be *ten* times of the number of insertions.

From Figure 16, we observe that blocked sparse table implementations outperform disjoint set and unblocked sparse table, so we will focus on comparing the two implementations for blocked sparse table, i.e., rightmost-pops and LCA tables. Note that it is easy to extend rightmost-pops encoding for incremental data since we only need to maintain the number of pops incrementally for the last block.

Table 2 compares the time for answering incremental suffix maximum query using rightmost-pops and LCA tables. Despite that the LCA table method has a theoretically better amortized $O(1)$ query time, we observe that the rightmost-pops runs up to 1.5 times faster than the LCA table. We believe that there are two reasons for this. First, the LCA table implementation requires more instructions to compute the Catalan index. In contrast, the rightmost-pops encoding does not require the computation for tree index. Instead it uses the stored number of pops to answer the query directly. Second, despite that in Theory [6] the optimal block size is $\frac{\log n}{4}$, this block size is usually too large for LCA table implementations. That is, we will need to build a very large LCA lookup table. Even worse, we will *not* access all of it, which means it is very unlikely that we will access contiguous memory, which causes serious cache misses.

We also observe that the probability p affects the performance gain of our "peeking" technique. When the probability p is close to 0 or 1, the peeking achieves excellent performance gain. The performance gain of the peeking operation also depends the block size. In addition, the best block size for LCA sparse table is different from the best block size for the rightmost-pops sparse table.

We observe that the performance of the rightmost-pops sparse table is better than that of LCA sparse table, but the performance gain decreases when q is close to 1. LCA sparse table requires the

Table 2. The timing (in seconds) of answering incremental suffix maximum query using rightmost-pops sparse table and the theoretically better LCA table sparse table (in bold font).

	$L = 4$					$L = 8$					$L = 16$					
q \ p	0%	25%	50%	75%	100%	0%	25%	50%	75%	100%	0%	25%	50%	75%	100%	speedup
0%	**1.15**	**0.89**	**0.86**	**0.88**	**0.91**	**0.88**	**0.87**	**0.87**	**0.85**	**0.87**	**1.02**	**1.00**	**0.99**	**1.00**	**1.02**	1.56
	1.30	1.05	1.05	1.05	1.05	1.32	1.32	1.32	1.32	1.32	1.35	1.34	1.34	1.34	1.34	
20%	**0.98**	**0.95**	**0.98**	**0.99**	**0.96**	**1.16**	**1.16**	**1.19**	**1.19**	**1.18**	**1.24**	**1.28**	**1.31**	**1.25**	**1.21**	1.26
	1.09	1.09	1.09	1.09	1.09	1.40	1.40	1.40	1.40	1.40	1.53	1.53	1.53	1.53	1.53	
40%	**1.01**	**1.01**	**1.02**	**1.02**	**0.99**	**1.23**	**1.24**	**1.25**	**1.24**	**1.21**	**1.39**	**1.43**	**1.45**	**1.31**	**1.26**	1.28
	1.13	1.13	1.13	1.13	1.12	1.46	1.46	1.47	1.47	1.45	1.62	1.62	1.62	1.62	1.61	
60%	**1.01**	**1.02**	**1.04**	**1.02**	**0.99**	**1.23**	**1.25**	**1.27**	**1.26**	**1.20**	**1.44**	**1.48**	**1.51**	**1.34**	**1.26**	1.28
	1.13	1.14	1.15	1.14	1.12	1.47	1.48	1.50	1.49	1.45	1.63	1.64	1.66	1.65	1.61	
80%	**0.99**	**1.01**	**1.03**	**1.01**	**0.97**	**1.20**	**1.22**	**1.25**	**1.23**	**1.15**	**1.38**	**1.43**	**1.49**	**1.31**	**1.19**	1.31
	1.11	1.12	1.15	1.12	1.10	1.44	1.46	1.49	1.47	1.41	1.59	1.61	1.65	1.63	1.56	
100%	**0.94**	**0.96**	**1.00**	**0.97**	**0.91**	**0.96**	**1.01**	**1.01**	**0.99**	**1.03**	**1.09**	**1.12**	**1.16**	**1.34**	**1.20**	1.39
	1.04	1.06	1.10	1.07	1.03	1.34	1.36	1.39	1.36	1.33	1.39	1.41	1.44	1.41	1.39	

Catalan index computation, which is not required in the rightmost-pops sparse table implementation. When the probability q is close to 1, LCA sparse table will require less instructions on computing Catalan index, so its performance will be close to that of the rightmost-pops sparse table.

The maximum length of interval query (L in Table 2) affects the performance of interval query in a rightmost-pops sparse table, but not the performance of the LCA sparse table. Note that in Algorithm 4 the block size s is set to 16, which is the maximum number of times the loop in Algorithm 4 will iterate for an query. Therefore, the query time will increase when L increases. On the other hand, the LCA sparse table answers a query by a single lookup into the LCA table, so its performance is not affected by L.

6.3 Variable Gapped Longest Common Subsequence

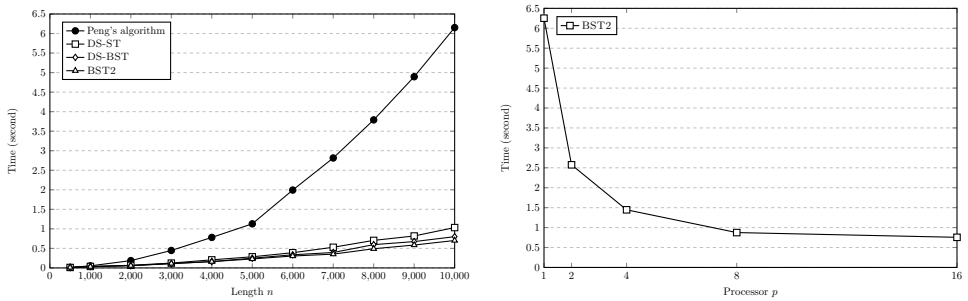
We implemented *four* combinations of data structures in solving the VGLCS problem and evaluated their performance on VGLCS. The first combination is a *sequential* implementation of Peng's algorithm using disjoint set on *both* the first stage and second stage. Note that since the disjoint set only supports suffix query, so the implementation must be sequential if the second stage uses the disjoint set. On the other hand, the other three combinations are all implemented in parallel. The *DS-ST* combination uses disjoint set in the first stage and the standard *unblocked* sparse table in the second stage. The *DS-BST* combination uses disjoint set in the first stage and *blocked* sparse table with rightmost-pops encoding in the second stage. The reason we use rightmost-pops encoding, instead of LCA tables, was describe earlier. Finally, the *BST2* combination uses blocked sparse table with rightmost-pops encoding in both stages.

Figure 17(a) compares the execution time of all four data structure combinations under different lengths of inputs. The input strings are generated *randomly* from the alphabet $\{A, T, C, G\}$, as this alphabet is most popular in bioinformatics. We note that, *BST2* outperforms all other parallel implementations.

Figure 17(b) shows the scalability of the best parallel combination *BST2*. It is *eight* times faster than a serial implementation on our server with 6 cores and hyper-threading.

7 RELATED WORK

Most of the parallelization of LCS on most multi-core platforms focuses on *wavefront* parallelism. The wavefront method keeps the computation as a wavefront that sweep through the entire dynamic programming tables. The wavefront computation is *not* cache-friendly, i.e., the wavefront algorithm cannot effectively keep the required data in cache. To address this cache issue, Maleki et al. [11] developed a technique to exploit more parallelism in the dynamic programming.



(a) Execution time of four data structure combinations (b) Scalability of the sparse table with rightmost-pops encoding

Fig. 17. The execution time and scalability results of our parallel implementations on an E5-2620 server with 6 cores and hyper-threading

The alternative to wavefront method is the traditional row-by-row approach, in which the dynamic programming tables is built in a row-by-row manner. For example, Peng [14] gives a $O(nm\alpha(n))$ VGLCS algorithm that is easy to implement and an asymptotically better $O(nm)$ algorithm, where α is the inverse of Ackermann's function [2].

It is difficult to parallelize traditional row-by-row approach for VGLCS due to the difficulty in efficient suffix and range query in a parallel environment. Peng's sequential VGLCS algorithm uses disjoint sets by Gabow [8] and Tarjan [16] for suffix maximum query. We instead use *sparse table* [3] to support incremental suffix/range maximum queries in our VGLCS algorithm. The sparse table is simple to implement and provides sufficient parallelism for good performance in a parallel environment.

Fischer [6] proposed blocked sparse table for better performance than the unblocked sparse table [3]. We also adopted blocked sparse table and tested its implementation in our experiments. Fischer's algorithm [6] builds least ancestor tables for answering range maximum query. We instead use a *rightmost-pops* encoding for Cartesian trees.

Demaine [4] also proposed *cache-aware* operations on Cartesian tree [17], to address the cache miss issues in Fischer's least common ancestor table building [6]. Masud [9] presents a new encoding method that reduces the number of instructions. Our rightmost-pops encoding for Cartesian trees also reduces cache misses, and with a much simpler implementation than Demaine's encoding.

Finally the authors would like to point out that to the best of our knowledge, we are not aware of any *dynamic* encoding for Cartesian trees. All previous works are *off-line*, i.e., they assume all data are given in advance, as a result, they cannot cope with incrementally added data. In contrast, our dynamic Catalan index computation technique in Section 4.3 does support efficient range query on an incremental data sets.

8 CONCLUSION

In this paper, we propose a two-stages row-by-row parallel VGLCS algorithm and optimization for both stages. The proposed two-stage algorithm is much more efficient than the traditional wavefront method since it is much regular and much easy to parallelize.

We use *sparse table* to solve the variable gapped longest common subsequence problem. The sparse table implementation runs efficiently in parallel because it improves the thread synchronize and workload imbalance than the disjoint set. In particular, we present a rightmost-pops encoding

that is both easy-to-implement and efficient in a parallel environment. Our VGLCS algorithm using right-most-pops encoding sparse table runs in $O(n^2 s/p + n \max(\log n, s))$, where n is the number of data, p is the number of processors, and s is the block size.

We also present a tree labeling technique that order trees lexicographically so that we can encode every binary search tree into a Catalan index. Then, we apply this technique in building Cartesian tree building so that the insertion takes amortized $O(1)$ time. As a result the VGLCS problem can be solved in $O(n^2/p + n \log n)$ time with this labeling technique.

Finally, we present a dynamic Catalan index computation algorithm for sparse table. We can answer the incremental ranged maximum query run in amortized $O(1)$ by our sparse table, and it can be applied to the incremental suffix maximum query as well. The time complexity of our VGLCS algorithm is $O(n^2/p + n \log n)$. Note that this dynamic Catalan index computation technique is very general and can be applied to other circumstances where data is constantly inserted into a binary tree.

We observed two interesting results from our experiments. First blocked sparse table outperforms unblocked sparse tables in a parallel environment. When the block size is properly chosen, the performance of blocked version runs more efficiently than the unblocked version. Second, a asymptotically better algorithm may not perform better in practice. For example, from our experiments we conclude that blocked sparse table with rightmost-pops, which has a query time $O(s)$, actually performs better than a sparse table with $O(1)$ amortized query time. We believe that an easy and straightforward implementation is the key of good performance, especially in a parallel environment.

9 FUTURE WORK

Many studies focused on how to minimize the amount of computation in the encoding process, such as changing the recursion definition formula. We can use the transformation of several register state instead of the memory access. These methods are used off-line encoding. In the case of the maximum value of the suffix, our approach still does not apply all the transformation in registers. If we can change the mathematical definition to reduce memory access, the performance will be more significantly improved.

REFERENCES

[1] Hsing-Yen Ann, Chang-Biau Yang, Yung-Hsing Peng, and Bern-Cherng Liaw. 2010. Efficient algorithms for the block edit problems. *Inf. Comput.* 208 (2010), 221–229.

[2] Lech Banachowski. 1980. A Complement to Tarjan's Result about the Lower Bound on the Complexity of the Set Union Problem. *Inf. Process. Lett.* 11 (1980), 59–65.

[3] Omer Berkman and Uzi Vishkin. 1993. Recursive Star-Tree Parallel Data Structure. *SIAM J. Comput.* 22 (1993), 221–242.

[4] Erik D. Demaine, Gad M. Landau, and Oren Weimann. 2009. On Cartesian Trees and Range Minimum Queries. In *Algorithmica*.

[5] Edsger Dijkstra. 1976. *A discipline of programming*. Prentice-Hall, Englewood Cliffs, N.J.

[6] Johannes Fischer and Volker Heun. 2006. Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. In *CPM*.

[7] Johannes Fischer and Volker Heun. 2007. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In *ESCAPE*.

[8] Harold N. Gabow and Robert E. Tarjan. 1983. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. *J. Comput. Syst. Sci.* 30 (1983), 209–221.

[9] Masud Hasan, Tanaeem M. Moosa, and Mohammad Sohel Rahman. 2010. Cache Oblivious Algorithms for the RMQ and the RMSQ Problems. *Mathematics in Computer Science* 3 (2010), 433–442.

[10] Daniel S. Hirschberg. 1975. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Commun. ACM* 18 (1975), 341–343.

[11] Saeed Maleki, Madan Musuvathi, and Todd Mytkowicz. 2016. Efficient parallelization using rank convergence in dynamic programming algorithms. *Commun. ACM* 59 (2016), 85–92.

[12] David Mount. 2001. *Bioinformatics : sequence and genome analysis.* Cold Spring Harbor Laboratory Press, Cold Spring Harbor, N.Y.
[13] Md. Mostofa Ali Patwary, Jean R. S. Blair, and Fredrik Manne. 2010. Experiments on Union-Find Algorithms for the Disjoint-Set Data Structure. In *SEA*.
[14] Yung-Hsing Peng and Chang-Biau Yang. 2011. The Longest Common Subsequence Problem with Variable Gapped Constraints.
[15] Mohammad Sohel Rahman and Costas S. Iliopoulos. 2006. Algorithms for Computing Variants of the Longest Common Subsequence Problem. In *ISAAC*.
[16] Robert E. Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22 (1975), 215–225.
[17] Jean Vuillemin. 1980. A Unifying Look at Data Structures. *Commun. ACM* 23 (1980), 229–239.
[18] Jiaoyun Yang, Yun Xu, and Yi Shang. 2010. An Efficient Parallel Algorithm for Longest Common Subsequence Problem on Gpus.

Received February 2007; revised March 2009; accepted June 2009