Nearly Optimal Parallel Algorithms for Longest Increasing Subsequence

Nairen Cao
caonc@bc.edu
Boston College
Boston, USA

Shang-En Huang
huangaul@bc.edu
Boston College
Boston, USA

Hsin-Hao Su
suhx@bc.edu
Boston College
Boston, USA

ABSTRACT

The paper presents parallel algorithms for multiplying implicit simple unit-Monge matrices (Krusche and Tiskin, PPAM 2009) of size $n \times n$ in the EREW PRAM model. We show implicit simple unit-Monge matrices multiplication of size $n \times n$ can be achieved by a deterministic EREW PRAM algorithm with $O(n \log n \log \log n)$ total work and $O(\log^3 n)$ span. This implies that there is a deterministic EREW PRAM algorithm solving the longest increasing subsequence (LIS) problem in $O(n \log^2 n \log \log n)$ work and $O(\log^4 n)$ span. Furthermore, with randomization and bitwise operations, implicitly multiplying two simple unit-Monge matrices can be improved to $O(n \log n)$ work and $O(\log^3 n)$ span, which leads to a randomized EREW PRAM algorithm obtaining LIS in $O(n \log^2 n)$ work and $O(\log^4 n)$ span with high probability. In the regime where the LIS has length $k = \Omega(\log^3 n)$, our results improve the span from $\tilde{O}(n^{2/3})$ (Krusche and Tiskin, SPAA 2010) and $O(k \log n)$ (Gu, Men, Shen, Sun, and Wan, SPAA 2023) to $O(\log^4 n)$ while the total work remains near optimal $\tilde{O}(n)$.

CCS CONCEPTS

• **Theory of computation → Design and analysis of algorithms**.

KEYWORDS

Longest increasing subsequence, Implicit simple unit-Monge matrix multiplication, Parallel algorithm

ACM Reference Format:
Nairen Cao, Shang-En Huang, and Hsin-Hao Su. 2023. Nearly Optimal Parallel Algorithms for Longest Increasing Subsequence. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3558481.3591078

1 INTRODUCTION

The *longest increasing subsequence* (LIS) problem is a classical problem in computer science, with many variants and applications such as text editing and dynamic time warps [28, 29], genome sequence alignments [5, 9, 10, 26], and quantum measurements [25]. Given a sequence of n numbers $A = (a_1, a_2, \ldots, a_n)$, the goal is to find the longest subsequence from A such that its values are (strictly) increasing. An $O(n \log n)$ time sequential algorithm is known, and an $\Omega(n \log n)$ lower bound is also known under the comparison model [11, 27]. In parallel settings, the earlier LIS algorithms focused on the Bulk-Synchronous Model [20, 21] and the Coarse-Grained Multicomputer models [14, 30]. However, the focus of the algorithms in these models lives in the regime that the number of processors p is much less than the input size n (i.e., $p \ll n$). Their results, when translated into the PRAM model, have either large work (e.g., $O(n^{1.5})$ in [20]) or large span (e.g., $\tilde{O}(n^{2/3})$ span in [21]).

Very recently, Shen, Wan, Gu, and Sun [31] proposed the *phase-parallel* framework solving dynamic programs whose parallel runtime depends substantially on the dependency chain length[1]. As a consequence, they proposed the first nearly work-optimal algorithm that computes LIS with $\tilde{O}(k)$ span, where k is the length of the LIS. This improves the results of Nakashima and Fujiwara's algorithm [24] from $\tilde{O}(k^2)$ span to $\tilde{O}(k)$ span. Later, Gu, Men, Shen, Sun, and Wan [15] further improved their LIS algorithm with $O(n \log k)$ optimal work and $O(k \log n)$ span.

However, given that a random permutation has its LIS length $\tilde{\Theta}(\sqrt{n})$ with high probability [4, 12], the span of $\tilde{O}(k)$ translates to $\tilde{O}(\sqrt{n})$ in most of the input permutations. Therefore, a natural question arises: Does there exist an efficient polylog(n) span algorithm, such that the total work is also nearly optimal?

In this paper, we give an affirmative answer to the above question. We propose the first polylogarithmic span algorithm that computes LIS with nearly optimal total work on an EREW PRAM, for which simultaneous access to any memory location by different processors is forbidden either for reading or for writing.

Theorem 1.1. *The longest increasing subsequence problem can be solved in the EREW PRAM model by:*

- *a deterministic algorithm in $O(n \log^2 n \log \log n)$ total work and $O(\log^4 n)$ span, or*
- *a randomized algorithm in $O(n \log^2 n)$ work and $O(\log^4 n)$ span with high probability.*

1.1 LIS via Implicit Subunit-Monge Matrix Multiplication

Our algorithm is a divide and conquer based algorithm, following the *implicit subunit-Monge matrix multiplication* framework from Krusche and Tiskin [21]. Let the input $A = (a_1, \ldots, a_n)$ be a permutation of $\{1, \ldots, n\}$. Consider the *filtered sequence* $A_{i,j}$: the subsequence of A whose values are all within the range $[i : j]$. Let $LIS(A_{i,j})$ be the length of LIS of $A_{i,j}$. A beautiful theorem from

This work was supported by NSF CCF-2008422.

SPAA '23, June 17–19, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9545-8/23/06.
https://doi.org/10.1145/3558481.3591078

[1]Different dynamic programs have different realizations of dependency chains. The dependency chain length of LIS is simply the length of LIS of the input sequence.

Table 1: Parallel LIS algorithms. Here, n refers to the length of the input array, k refers to the length of the LIS solution and p refers to the number of processors.

Reference	Total Work	Span	Notes
Galil and Park [13]	$O(n^2)$	$O(\sqrt{n}\log n)$	General 1D Dynamic Programming.
Garcia, Myoupo, and Semè [14]	$O(n^2)$	$O(n^2/p)$	Only $O(p)$ communications.
Semè [30]	$O(np\log(n/p))$	$O(n\log(n/p))$	
Nakashima and Fujiwara [24]	$O(n\log n)$	$O((n\log n)/p)$ or $O(k^2\log n)$	Requires $p < n/k^2$.
Krusche and Tiskin [20]	$O(n^{1.5})$	$O(n/\sqrt{p})$	
Krusche and Tiskin [21]	$O(n\log^2 n)$	$O((n\log p)/p)$ or $\tilde{O}(n^{2/3})$	Requires $p < \sqrt[3]{n}$.
Shen, Wan, Gu, and Sun [31]	$O(n\log^3 n)$	$O(k\log^2 n)$	With high probability.
Gu, Men, Shen, Sun, and Wan [15]	$O(n\log k)$	$O(k\log n)$	
This paper.	$O(n\log^2 n\log\log n)$	$O(\log^4 n)$	
	$O(n\log^2 n)$	$O(\log^4 n)$	With high probability; AC^{0}operations.

Tiskin [34–36]^2 states that if we define an $(n+1)\times(n+1)$ matrix M_A^Σ, where

$$M_A^\Sigma(i,j) = \begin{cases} j - i - LIS(A_{i+1,j}) & \text{if } 0 \le i < j \le n, \\ 0 & \text{otherwise,} \end{cases}$$

Then there exists a *sub-permutation matrix* M_A (a binary matrix where each row and column sums up to at most 1) whose *distribution matrix* is exactly the same as M_A^Σ (see Section 2 for definitions). The distribution matrix of a sub-permutation matrix is called a *simple subunit-Monge matrix*.

Furthermore, Tiskin [34] proved that the entries of M_A^Σ can be obtained by the matrix distance multiplications (or so-called $(\min,+)$-multiplication), leading to a divide and conquer approach that computes LIS. That is, if one splits the sequence A into two halves $L = (a_1,\ldots,a_{n/2})$ and $R = (a_{n/2+1},\ldots,a_n)$, then

$$M_A^\Sigma(i,k) = \min_{i\le j\le k}\{M_L^\Sigma(i,j) + M_R^\Sigma(j,k)\}.$$

Note that the matrices M_L^Σ and M_R^Σ are still $(n+1)\times(n+1)$ matrices, since the range of values in both L and R are still $[1,n]$. To enable recursion, one simply renames the elements in L and R to the range $[1,n/2]$ (see Section 4 for details). Following the above idea, Krusche and Tiskin [20] successfully reduced the LIS problem to polylogarithmic *implicit subunit-Monge matrix multiplication problems*:

Theorem 1.2. *Assume there is an EREW PRAM algorithm solving the implicit $n\times n$ subunit-Monge matrix multiplication problem in $O(W(n))$ work and $O(S(n))$ span. Then, there is an EREW PRAM algorithm that computes an LIS from a sequence of n integers in $O(W(n)\log n + n\log n)$ work and $O(S(n)\log n + \log^2 n)$ span.*

Since the entire matrix M_A^Σ can be encoded implicitly as a sub-permutation using $O(n)$ machine words, the only task that remains is to design an efficient (parallel) algorithm that multiplies two subunit-Monge matrices, given the implicit $O(n)$-size sub-permutations.

Few sequential algorithms for fast implicit subunit-Monge matrix multiplication were known. In particular, Tiskin [36] gives an $O(n\log n)$ sequential algorithm, which is much faster compared with the most general matrix distance multiplication in $O(n^3\frac{\log\log n}{\log^2 n})$ time [17] (see also [6, 23]), and Monge-matrix distance multiplications in $O(n^2)$ time [33]. For parallel algorithms, the best nearly work-efficient parallel algorithm for fast implicit subunit-Monge matrix multiplication has $O(n\log n)$ total work and $\tilde{O}(n^{2/3})$ span, by Krusche and Tiskin [21]. This is also a divide and conquer based algorithm. The high-level idea is described below. Suppose that one wants to multiply M_L^Σ and M_R^Σ. The divide and conquer framework somehow reduces the task into computing an entry-wise minimum between two matrices M'_{lo} and M'_{hi}. A crucial observation is that the entry-wise difference $\delta := M'_{lo} - M'_{hi}$ has a nice monotone property: The values are non-decreasing within each row and each column. Once the algorithm obtains the *boundary* of the negative (resp. positive) region in δ, the correct M_A^Σ can be implicitly derived.

Notice that all the matrices are provided implicitly. The core task is then to obtain the boundaries from a δ matrix when M'_{lo} and M'_{hi} are given implicitly. Due to the monotone property, the boundary can be described in $O(n)$ size, and there is a simple walk-and-check sequential subroutine that obtains the desired boundaries. Krusche and Tiskin [21] tried to parallelize this walk-and-check subroutine and ended up with a $\tilde{O}(n^{2/3})$ span EREW PRAM algorithm.

1.2 Our Contribution

We parallelize the sequential walk-and-check subroutine with a third-layer divide and conquer algorithm. After obtaining the δ matrix described above, our algorithm recursively finds the index of the boundary in the middle row and divides the entire walk into the upper half and the lower half.

The main challenge in this approach is to efficiently obtain the middle row entry values in δ, from the implicit representations of M'_{lo} and M'_{hi}. Krusche and Tiskin [21] observe that every entry in δ can be expressed as a difference of two quantities, where each quantity corresponds to the number of nonzero entries in the sub-permutation matrix associated with M'_{lo} and M'_{hi}. Thus, these quantities can be computed efficiently via 2D range query data structures. Applying an off-the-shelf parallel 2D range tree (e.g., [32]) leads to an $O(\log^6 n)$ span parallel algorithm for LIS.

^{2}The theorem statements from the literature were described in a more general longest common subsequence scenario, here for clarity we exhibit their results in LIS.

Nevertheless, because of the nature of our approach, if we aim to compute *all* δ values within a row, we do not even need a 2D range tree – a simple parallel prefix sum subroutine suffices.

As a consequence, our algorithm achieves $O(\log^4 n)$ span while the total work stays near work-optimal. If one implements this third layer divide and conquer algorithm directly, it imposes an additional $O(\log n)$ factor to the total work. With a cute trick of *mini-batching*, our divide and conquer algorithm brings only $O(\log\log n)$ additional factor to the total work: In our algorithm solving the core task, we do not compute every entry in the middle row of δ. Instead, we compute every other L entry along the row, where $L = \Omega(\log n \log\log n)$. The trick essentially partitions the δ matrix into $L \times L$ small squares, forming a coarse-grained walk. Once the algorithm obtains the walk, an additional walk-and-check subroutine can be applied in parallel to all the small squares that contain the actual walk. To enable the trick, our algorithm splits the sub-permutation into length L chunks and sorts each chunk such that each range query can be done in $O(\log L) = O(\log\log n)$ time via a binary search. We remark that this mini-batching trick is not invented abruptly. This trick is inspired by the classical "Four Russians" paradigm and occurs in many sequential sequence alignment algorithms [7, 34] and range query data structures [19].

Furthermore, if the algorithm is further allowed to use randomness and integer sorting (such as signature sort [3]), the algorithm only brings $O(1)$ overhead to the total work with high probability. We summarize the result below:

Theorem 1.3. *There exists a randomized algorithm in the EREW PRAM model with* AC^0 *operations*[3] *that computes implicit simple sub-unit Monge matrix multiplication in* $O(\log^3 n)$ *span and* $O(n \log n)$ *total work with high probability.*

Combining Theorem 1.3 and Theorem 1.2 gives us Theorem 1.1.

Application: Semi-Local LCS Algorithms. The longest common subsequence (LCS) problem can be thought of as a generalization of LIS. The input consists of two sequences x and y, where x has length $m = |x|$ and y has length $n = |y|$. The output is the LCS between x and y. In PRAM models, there are already optimal parallel algorithms for computing LCS with $\tilde{O}(mn)$ total work and $\tilde{O}(1)$ span [22], and in the decision tree model, there is a matching lower bound $\Omega(mn)$ given by Aho, Hirschberg, and Ullman [1].

The semi-local LCS problem extends the LCS problem: the algorithm must compute the LCS between x and *every subsequence* of y. Tiskin's algorithm [34] solves the semi-local LCS problem in $O(mn)$ total work and $\tilde{O}(n^{2/3})$ span, via the implicit subunit-Monge matrix multiplication. Thus, our implicit subunit-Monge matrix multiplication algorithm also implies a faster algorithm for solving the semi-local LCS problem.

Corollary 1.4. *Given two sequences of lengths m and n, there exists a deterministic semi-local LCS algorithm with* $O(mn)$ *total work and* $O(\log^4(m+n))$ *span.*

[3]An AC^0machine allows any machine word operations that could be implemented by a constant depth circuit. This includes common bitwise operations, addition and subtractions of two machine words.

1.3 Related Works

Sequential Word RAM model with AC^0 *operations.* In the word RAM model, we assume that each machine word has $\Omega(\log n)$ bits. By allowing AC^0operations, it is possible to perform bitwise operations, which enables *in-word parallelism*: packing several small integers into a single machine word and then performing operations to these integers at once. Under this model, the lower bound based on decision trees does not apply anymore. For example, sorting a sequence of integers can now be done in $O(n \log\log n)$ time with $O(n)$ space by Han [16]. With Han's integer sorting algorithm, LIS can then be found in $O(n \log\log n)$ time using a more efficient integer dictionary that supports insertion, deletion, and successor operations (e.g., using a van Emde Boas tree [37, 38]). Crochemore and Porat [8] further improve the runtime to $O(n \log\log k)$ in the sequential model, where k refers to the length of the LIS.

Massively Parallel Computation model. Another well-investigated parallel model is the massively parallel computation model (MPC). In the MPC model, a parameter δ $(0 < \delta < 1)$ is given. On input of size n, the algorithm is allowed to use n^δ machines, and on each machine, the memory size is constrained by $\tilde{O}(n^{1-\delta})$. The computation proceeds in *rounds*. In each round, an arbitrary computation with limited memory can be performed locally on each machine. After local computations in a round are done, messages can be sent between machines as long as the total amount of messages received from a single machine does not exceed its memory limit. The ultimate goal in designing MPC algorithms is to minimize the number of rounds.

Regarding the solution of LIS, the work of Krusche and Tiskin [20, 21] implies an $O(\log n)$ round algorithm in the MPC model. Im, Moseley, and Sun [18] give an $O(1)$ round MPC algorithm for $(1-\epsilon)$-approximating LIS. Whether or not the exact LIS can be solved using $O(1)$ rounds is still open.

Paper Outline

In Section 3, we will present our main result — an $\tilde{O}(1)$ span EREW PRAM algorithm that multiplies two subunit-Monge matrices implicitly. In Section 4, we finish the proof of Theorem 1.2 with the details of computing an LIS from implicit subunit-Monge matrix multiplications.

2 PRELIMINARIES

Notations. For integers i and j, we use $[i : j]$ to denote set $\{i, i+1, ..., j\}$ and $[i : j)$ to denote set $\{i, i+1, ..., j-1\}$. We denote the half integer set $\{i+\frac{1}{2}, i+\frac{3}{2}, ..., j-\frac{1}{2}\}$ by $\langle i : j\rangle$. To distinguish the half-integer variable and the integer variable, we mark the half-integer variables by $\hat{}$. For example, i, j are integer variables, and $\hat{\imath}, \hat{\jmath}$ are half-integer variables. We heavily use half-integers as matrix indices. For a matrix M and half-integer indexes $\hat{\imath}, \hat{\jmath}$, we define $M(\hat{\imath}, \hat{\jmath}) = M(\hat{\imath} + \frac{1}{2}, \hat{\jmath} + \frac{1}{2})$. We denote the Cartesian product by $[i_1 : i_2] \times [j_1 : j_2]$.

Given a matrix M of size $m \times n$, its distribution matrix M^Σ is defined as

$$M^\Sigma(i, j) = \sum_{\hat{\imath} \in \langle i:m\rangle, \hat{\jmath} \in \langle 0:j\rangle} M(\hat{\imath}, \hat{\jmath})$$

for all $i \in [0 : m], j \in [0 : n]$. In other words, $M^{\Sigma}(i, j)$ is the sum of the submatrix M with index $(\hat{\imath}, \hat{\jmath}) \in \langle i : m\rangle \times \langle 0 : j\rangle$.

In this work, we will focus on the sub-permutation matrix P. A matrix P is a ***sub-permutation matrix*** if and only if

- each element in P is either 0 or 1,
- there is at most one element equal to 1 in each row and each column of P.

We can maintain a vector to support querying row(column) non-zero index in $O(1)$ time.

The distribution matrix P^{Σ} of a sub-permutation matrix P is called a ***subunit-Monge matrix***. Given two sub-permutation matrix P_A, P_B, the ***implicit subunit-Monge matrix multiplication problem*** is to compute $P_C = P_A \boxdot P_B$ such that $P_C^{\Sigma}(i,k) = min_j(P_A^{\Sigma}(i,j) + P_B^{\Sigma}(j,k))$, that is to compute the $(min, +)$ product of subunit-Monge matrix P_A^{Σ} and P_B^{Σ}. We use $\boxdot$ to represent the implicit subunit-Monge matrix multiplication operation. The sub-permutations matrices are closed under the $\boxdot$ operation. More formally,

Lemma 2.1 ([33]). *Given two sub-permutation matrix P_A, P_B, let $P_C = P_A \boxdot P_B$, then P_C is also a sub-permutation matrix.*

Given two permutation matrices, the indices of their non-zero entries can be stored in $O(n)$ space. Based on Lemma 2.1, it is possible to store the output of $\boxdot$ in $O(n)$ space. The question is whether we can design an algorithm that supports the $\boxdot$ operation fast.

3 ALGORITHM

In this section, we present an algorithm to solve the implicit subunit-Monge matrix multiplication problem of two $n\times n$ sized sub-permutation matrices using $O(n \log n)$ work and $O(\log^3 n)$ span.

Our algorithm follows the blueprint of Krusche and Tiskin's algorithm. We first provide an overview of their algorithm and identify its bottleneck in Section 3.1. In Section 3.2, we partition the matrix into $\frac{n}{L} \times \frac{n}{L}$ grids of size $L \times L$ to reduce the size of the problem. We then show that solving the bottleneck on these grids efficiently will allow us to solve the bottleneck on the original problem efficiently. Finally, in Section 3.3, we present a divide and conquer algorithm to solve the bottleneck on the grids efficiently.

3.1 Overview of Krusche and Tiskin's algorithm

Given two sub-permutation matrices P_A, P_B of size $n \times n$, the goal is to compute $P_C = P_A \boxdot P_B$. Krusche and Tiskin's approach utilizes a divide-and-conquer method. First, P_A is split **vertically** into two matrices of size $n \times \frac{n}{2}$. The resulting matrices, denoted $P_{A,lo}$ and $P_{A,hi}$, respectively, contain the column indices of P_A in the ranges $\langle 0 : \frac{n}{2}\rangle$ and $\langle \frac{n}{2} : n\rangle$. Similarly, P_B is split **horizontally** into two matrices of size $\frac{n}{2} \times n$, denoted $P_{B,lo}$ and $P_{B,hi}$, which contain the row indices of P_B in the ranges $\langle 0 : \frac{n}{2}\rangle$ and $\langle \frac{n}{2} : n\rangle$.

In the divide step the algorithm obtains $P_{C,lo}$ and $P_{C,hi}$ such that

$$P_{C,lo} = P_{A,lo} \boxdot P_{B,lo} \text{ and } P_{C,hi} = P_{A,hi} \boxdot P_{B,hi}.$$

Although $P_{A,lo}$ has size $n \times \frac{n}{2}$, the key observation is that $P_{A,lo}$ contains at most $\frac{n}{2}$ non-zero rows. It can be shown that if the j-th row of $P_{A,lo}$ contains only zero, then the j-th row of $P_{C,lo}$ also contains only zero. Therefore, we can safely remove $\frac{n}{2}$ rows of $P_{A,lo}$ and $\frac{n}{2}$ columns of $P_{B,lo}$ which contains only 0. Then we apply $\boxdot$ operation on the sub permutation matrices of size $\frac{n}{2} \times \frac{n}{2}$ and plug in the corresponding missing rows and columns back to $P_{C,hi}$ and $P_{C,lo}$. In the divide step, we need to compute two subproblems of size $\frac{n}{2}$.

It is highly nontrivial to obtain P_C from $P_{C,lo}$ and $P_{C,hi}$. Indeed, $\boxdot$ operates on the distribution matrices. $P_{A,hi}^{\Sigma}$ (or $P_{B,lo}^{\Sigma}$) are different from the distribution matrix of the corresponding part of P_A (or P_B). Although $P_{C,lo} + P_{C,hi}$ is still a sub-permutation matrix, in the most cases $P_C \neq P_{C,lo} + P_{C,hi}$.

Fortunately, the good news is that one can compute $P_C^{\Sigma}(i,k)$ using the definition of the distribution matrix, replace the corresponding term with $P_{C,lo}^{\Sigma}, P_{C,hi}^{\Sigma}$, and conclude that

$$P_C^{\Sigma}(i,k) = min\left\{P_{C,lo}^{\Sigma}(i,k) + P_{C,hi}^{\Sigma}(0,k), P_{C,lo}^{\Sigma}(i,n) + P_{C,hi}^{\Sigma}(i,k)\right\}.$$

The final P_C depends on the difference between the above two parameters in the min function. Let δ be the difference matrix of size $(n+1) \times (n+1)$, where

$$\begin{aligned}\delta(i,k) &= \left(P_{C,lo}^{\Sigma}(i,k) + P_{C,hi}^{\Sigma}(0,k)\right) - \left(P_{C,lo}^{\Sigma}(i,n) + P_{C,hi}^{\Sigma}(i,k)\right) \\ &= \sum_{\hat{\imath}\in\langle 0:i\rangle,\hat{k}\in\langle 0:k\rangle} P_{C,hi}(\hat{\imath},\hat{k}) - \sum_{\hat{\imath}\in\langle i:n\rangle,\hat{k}\in\langle k:n\rangle} P_{C,lo}(\hat{\imath},\hat{k}).\end{aligned}$$

See Figure 1a for how to compute $\delta(i,k)$. By a careful analysis of the relation between δ and P_C [33], we have the following condition for locating all non-zero terms in P_C.

Lemma 3.1 ([33]). *Let P_A, P_B be two sub-permutation matrices and $P_C = P_A \boxdot P_B$. Let $P_{C,lo}, P_{C,hi}, \delta$ be the matrix defined above, then for any $\hat{\imath}, \hat{k} \in \langle 0 : n\rangle$, $P_C(\hat{\imath},\hat{k}) = 1$ if and only if one of the following three conditions holds:*

(3.1a) $\delta(\hat{\imath}+\frac{1}{2},\hat{k}+\frac{1}{2}) \leq 0$ and $P_{C,lo}(\hat{\imath},\hat{k}) = 1$,
(3.1b) $\delta(\hat{\imath}+\frac{1}{2},\hat{k}+\frac{1}{2}) > 0$ and $P_{C,hi}(\hat{\imath},\hat{k}) = 1$,
(3.1c) $\delta(\hat{\imath}+\frac{1}{2},\hat{k}+\frac{1}{2}) > 0$, $\delta(\hat{\imath}+\frac{1}{2},\hat{k}-\frac{1}{2}) = 0$ and $\delta(\hat{\imath}-\frac{1}{2},\hat{k}-\frac{1}{2}) < 0$.

We can use Lemma 3.1 to compute P_C. First, we notice that $\delta(i,k)$ is non-decreasing with respect to i (and k). Specifically:

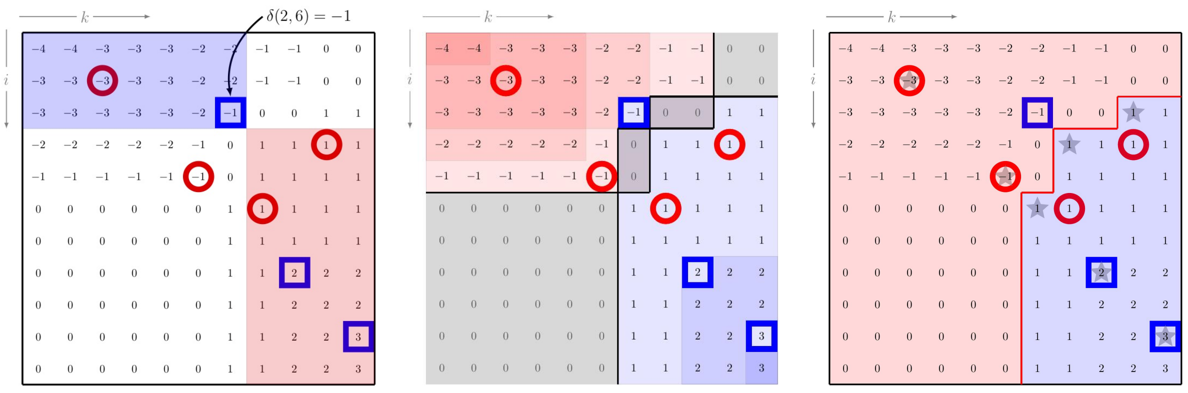
Lemma 3.2. *For any $i, k \in [1 : n]$, $\delta(i,k) - \delta(i-1,k) \in \{0,1\}$ and $\delta(i,k) - \delta(i,k-1) \in \{0,1\}$.*

Proof. We prove one case and the other one is similar,

$$\delta(i,k) - \delta(i-1,k) = \sum_{\hat{k}\in\langle 0:k\rangle} P_{C,hi}(i-\tfrac{1}{2},\hat{k}) + \sum_{\hat{k}\in\langle k:n\rangle} P_{C,lo}(i-\tfrac{1}{2},\hat{k})$$

Since $P_{C,lo} + P_{C,hi}$ is a sub-permutation matrix [33], at most one term is 1 in the right hand side. □

Since δ is non-decreasing with respect to i and k, there will be a line splitting the positive and non-positive terms of δ. See Figure 1c for the line. If we already know the line, we can figure out the non-zero terms of P_C in (3.1a) and (3.1b) with $O(n)$ work and $O(\log n)$ span. For (3.1c), we can decide the corresponding $\delta(\hat{\imath}+\frac{1}{2},\hat{k}+\frac{1}{2}) = 1$ for each row $\hat{\imath}+\frac{1}{2}$ using the line. However, we still need to check $\delta(\hat{\imath}+\frac{1}{2},\hat{k}-\frac{1}{2}), \delta(\hat{\imath}-\frac{1}{2},\hat{k}-\frac{1}{2})$ value. Fortunately, if we already know $\delta(\hat{\imath}+\frac{1}{2},\hat{k}+\frac{1}{2})$, one can query the neighboring value in $O(1)$ time based on the following lemma.



(a) An illustration of computing $\delta(2, 6)$. **(b) An example of δ-table.** **(c) An example of computing P_C.**

Figure 1: The red circles represent the non-zero terms of sub-permutation $P_{C,lo}$ and the blue squares represent the non-zero terms of sub-permutation $P_{C,hi}$. In Figure 1a to compute $\delta(2, 6)$, we first count the blue squares in the blue area, then subtract the number of red circles in the red area. Figure 1b is an example δ-table that represents the very last recursion step when computing the LIS of the following sequence: (6, 2, 1, 8, 10, 7, 3, 9, 5, 4). In Figure 1c, the red line splits positive and non-positive terms in δ-table. the star symbols represent the non-zero terms of sub-permutation P_C. The red line splits the positive and non-positive terms of δ. There are 3 cases for non-zero terms of P_C: the red circles in the red area of (3.1a), the blue squares in the blue area of (3.1b), and the turning points of (3.1c).

Lemma 3.3. *Let δ be the difference matrix defined above. If we are given $\delta(i, j)$ for fixed i and j, and the locations of the nonzeros in $P_{C,hi}$ and $P_{C,lo}$, then we can compute the four neighbor values $\delta(i \pm 1, j), \delta(i, j \pm 1)$ in $O(1)$ time.*

Proof. We show how to compute $\delta(i+1, j)$. Based on definition of δ, we have

$$\delta(i+1, j) - \delta(i, j) = \sum_{\hat{j} \in \langle 0:j \rangle} P_{C,hi}(i + \tfrac{1}{2}, \hat{j}) + \sum_{\hat{j} \in \langle j:n \rangle} P_{C,lo}(i + \tfrac{1}{2}, \hat{j})$$

Since $P_{C,hi}, P_{C,lo}$ are both sub-permutation matrices, there will be only 1 non-zero item in each row. We can access each non-zero index in $O(1)$ time and compute $\sum_{\hat{j} \in \langle 0:j \rangle} P_{C,hi}(i+\frac{1}{2}, \hat{j}) + \sum_{\hat{j} \in \langle j:n \rangle} P_{C,lo}(i+\frac{1}{2}, \hat{j})$. We already know the value of $\delta(i, j)$, so the value of $\delta(i+1, j)$ can be computed in $O(1)$ time. □

Combining all three cases, to determine the non-zero term of P_C, the core problem is to find the index of the first positive term of the matrix δ per row given two sub-permutation matrices $P_{C,lo}, P_{C,hi}$.

Core problem

More formally, we are given two sub permutation matrices $P_{C,lo}$, $P_{C,hi}$. Let δ be $(n + 1) \times (n + 1)$ size matrix and

$$\delta(i, k) = \sum_{\hat{i} \in \langle 0:i \rangle, \hat{k} \in \langle 0:k \rangle} P_{C,hi}(\hat{i}, \hat{k}) - \sum_{\hat{i} \in \langle i:n \rangle, \hat{k} \in \langle k:n \rangle} P_{C,lo}(\hat{i}, \hat{k})$$

The target is to compute the first positive term index $f(i)$ for each row i, i.e., for each $i \in [0 : n]$, $f(i)$ is the smallest integer value k' such that $\delta(i, k') > 0$. We assume that such $f(i) \in [0 : n]$ always exists. Otherwise, $\delta(i, n) \le 0$ holds and one can use the prefix sum to compute $\delta(i, n)$ and check it.

In the following subsections, we will show how to compute $f(i)$ for $i \in [0 : n]$ with $O(n)$ work and $O(\log^2 n)$ span using randomization, or $O(n \log\log n)$ work and $O(\log^2 n)$ deterministically. Our main theorem for implicit subunit-Monge matrix multiplication is given as follows.

Theorem 3.4. *Consider two $n \times n$ sub-permutation matrices.*

- *There is a deterministic EREW PRAM algorithm for implicit subunit-Monge matrix multiplication with $O(n \log n \log\log n)$ work and $O(\log^3 n)$ span.*
- *There exists a randomized EREW PRAM algorithm with AC^0 operations on $\Theta(\log n)$-bit machine words that computes implicit subunit-Monge matrix multiplication with $O(n \log n)$ work and $O(\log^3 n)$ span.*

Bottleneck of the core problem. Note that for each row of δ, its value is non-decreasing, if we can query each $\delta(i, k)$ efficiently, then we can compute the value f efficiently by a binary search of each row. The bottleneck is to query each value, we have to construct a 2D-range query data structure. There is indeed such parallel 2D-range query data structure [32]. However, their construction requires $\Theta(n \log n)$ total work for preprocessing and $O(\log^2 n)$ work for each query, which imposes at least extra $\Omega(\log n)$ overhead on the total work.

3.2 Computing f

Instead of considering all points of $\delta(i, k)$ for $i, k \in [0, n]$, we will consider points $\delta(iL, kL)$, where $i, k \in [0, \frac{n}{L}]$ and L is a parameter to be determined. Note that we view an $(n+1) \times (n+1)$ sized matrix as a $O(\frac{n}{L}) \times O(\frac{n}{L})$ grid. Most grid cells are sub-matrices of size $L \times L$ except the cells at the boundary. We assume that L divides n.

Otherwise, we can always add at most $L-1$ extra zero rows and/or columns to make $\frac{n}{L}$ an integer.

Our new target is to compute the *approximate index* $\tilde{f}$ for each grid row. For each $i \in [0, \frac{n}{L}]$, we define $\tilde{f}(i)$ such that

- $\tilde{f}(i)$ is the smallest integer k' such that $\delta(iL, k'L) > 0$; or
- if $\delta(iL, kL) \leq 0$ for all $k \in [0, \frac{n}{L}]$, then $\tilde{f}(i) = \frac{n}{L}$.

In the next subsection, we will describe an algorithm that outputs $(iL, \tilde{f}(i)L)$ and $\delta(iL, \tilde{f}(i)L)$ for $i \in [0, \frac{n}{L}]$.

Lemma 3.5. *Consider two $n \times n$ sized sub-permutation matrices $P_{C,hi}$ and $P_{C,lo}$. Let $\delta, f, \tilde{f}$ be the matrix and value defined above, and $L = \Omega(\log n \log\log n)$. Then, to compute all approximate indices $\tilde{f}(i)$ and the associated $\delta(iL, \tilde{f}(i)L)$ values for all $i \in [0 : \frac{n}{L}]$,*

- *there exists a deterministic EREW PRAM algorithm with an $O(n \log\log n)$ total work and $O(\log^2 n)$ span; and*
- *there exists a randomized EREW PRAM algorithm with AC^0 operations on word size $Z = \Theta(\log n)$ in $O(n)$ work and $O(\log^2 n)$ span with high probability.*

From approximate index $\tilde{f}$ to f. The remaining part of this section aims to show how to compute f using $\tilde{f}$. Before we establish the algorithm, we first describe some useful properties to f and $\tilde{f}$.

Lemma 3.6. *The following property holds for f and $\tilde{f}$:*

(3.6a) for any $i < j$, we have $f(i) \geq f(j)$ and $\tilde{f}(i) \geq \tilde{f}(j)$;
(3.6b) for any $j \in [iL : (i+1)L)$, we have $f(j) \in (\tilde{f}(i+1)L-L, \tilde{f}(i)L]$.

Proof. (3.6a) We only show $f(i) \geq f(j)$, $\tilde{f}(i) \geq \tilde{f}(j)$ holds for the same reason. Based on Lemma 3.2, $\delta(j, f(i)) \geq \delta(i, f(i)) > 0$ and $f(j) \leq f(i)$ based on the definition of $f(j)$.

(3.6b) Based on the definition of $f(j)$, we have $\delta(j, f(j)) > 0$. Based on Lemma 3.2, $\delta((i+1)L, f(j)) \geq \delta(j, f(j)) > 0$. Since $\tilde{f}(i+1)$ is the smallest integer such that $\delta((i+1)L, \tilde{f}(i+1)L) > 0$, we have $f(j) > \tilde{f}(i+1)L - L$.

On the other hand, if $f(j) = 0$, then $f(j) \leq \tilde{f}(i)L$ because $\tilde{f}(i) \geq 0$. Otherwise, we have $\delta(j, f(j)-1) \leq 0$. Based on Lemma 3.2, $\delta(iL, f(j)-1) \leq \delta(j, f(j)-1) \leq 0$. $\tilde{f}(i)$ is the smallest integer such that $\delta(iL, \tilde{f}(i)L) > 0$, so $\tilde{f}(i)L > f(j) - 1$ and $f(j) \leq \tilde{f}(i)L$. □

Lemma 3.6 provides useful properties for computing f using $(iL, \tilde{f}(i)L)$ and $\delta(iL, \tilde{f}(i)L)$. According to Lemma (3.6b), for any row $j \in [iL, (i+1)L)$, the index $(j, f(j))$ lies within the rectangle $[iL : (i+1)L) \times (\tilde{f}(i+1)L - L, \tilde{f}(i)L]$. Therefore, for each $i \in [0, \frac{n}{L})$, it suffices to search through $\tilde{f}(i) - (\tilde{f}(i+1) - 1)$ grid cells. In total, there are $\sum_i (\tilde{f}(i) - \tilde{f}(i+1) + 1) = O(\frac{n}{L})$ grid cells where $(j, f(j))$ is located. See Figure 2 for an example. Below, we give an algorithm to scan through a grid cell.

Identifying f Inside a Grid Cell

Consider a grid cell whose upper-right corner has the matrix index $(iL, (k+1)L)$ and the lower-left corner is indexed by $((i+1)L-1, kL+1)$. Whenever we have $\delta(iL, (k+1)L)$, then Algorithm 1 identifies all $f(j)$ such that $(j, f(j)) \in [iL : (i+1)L) \times (kL : (k+1)L]$.

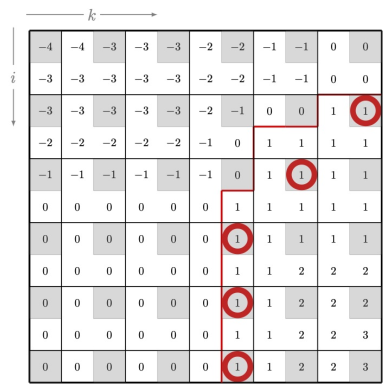


Figure 2: An example of $\tilde{f}$. $L = 2$. We only consider points in the gray shadow. The red line is for f. The red circles represent the node $(iL, \tilde{f}(i)L)$ for $i \in [0 : 5]$.

Algorithm 1 Description. Inside a grid, Algorithm 1 searches starting from the top right node. The algorithm uses the current and neighboring δ to decide where to traverse next. If i_c or k_c touches the boundary, the algorithm terminates. See Figure 3 for example.

Correctness of Algorithm 1. For a fixed row i, there are 3 cases:

(1) The row i contains only non-positive δ values, i.e., $\delta(i, k_c) \leq 0$ for all $k_c \in (kL : (k+1)L]$,
(2) $f(i) \in (kL : (k+1)L]$, in this case, we need to show that Algorithm 1 traverse point $(i, f(i))$,
(3) $f(i) \leq kL$. In other words, $\delta(j, kL) > 0$.

It is important to note that these cases appear in order of increasing row index. That is, if row i_1 is the first case and row i_2 is the second case, then we have $i_1 < i_2$. That is because $\delta(i, k)$ is non-decreasing.

Another observation is that if row i and $i+1$ are both in case (2) and we have already traversed $(i, f(i))$, then Algorithm 1 will traverse $(i+1, f(i))$. Based on Lemma (3.6a), we have $f(i+1) \leq f(i)$. For all points (i, k_c) such that $k_c \in (f(i+1), f(i)]$, we have $\delta(i, k_c) > 0$ and $\delta(i, k_c - 1) > 0$ and the algorithm chooses to decrease k_c by 1. Finally, the algorithm traverses the point $(i+1, f(i+1))$.

We can show that when the algorithm goes to the first row satisfying case (2), $k_c = (k+1)L$. That is because, before reaching case (2) rows, the algorithm must only have case (1) rows. For each case (1) row, we always have $\delta(i_c, k_c) \leq 0$, and the algorithm chooses to increase i_c by 1 without changing k_c. When the algorithm transverses to the first row satisfying case (2), it starts from $(i_c, (k+1)L)$ from right to left until it arrives at $(i_c, f(i_c))$. Then, based on our second observation, Algorithm 1 can traverse all case (2) rows, and thereby the following lemma holds.

Lemma 3.7. *For any $f(j)$ such that $j \in [iL : (i+1)L)$ and $f(j) \in (kL : (k+1)L]$, Algorithm 1 computes $f(j)$ correctly.* □

Running time of Algorithm 1. Algorithm 1 only increases i_c or decreases k_c by one in each iteration. In total, there will be at most $2L$ iteration. The only difficulty is that we need to compute $\delta(i_c, k_c - 1)$ and the new $\delta(i_c, k_c)$. Lemma 3.3 shows that querying the neighbor value of $\delta(i_c, k_c)$ takes $O(1)$ time. Since we know $\delta(i_c, k_c)$ at

Algorithm 1 Identifying f inside a grid.
Input: the grid's top-right index $(iL, (k+1)L)$ and $\delta(iL, (k+1)L)$.
Output: $f(j)$ for all $j \in [iL : (i+1)L)$ with $f(j) \in (kL : (k+1)L]$.

```
1: function IdentifyGrid($(iL, (k+1)L), \delta(iL, (k+1)L)$)
2:     Set $i_c \leftarrow iL$ and $k_c \leftarrow (k+1)L$.
3:     while $i_c < (i+1)L$ and $k_c > kL$
4:         if $\delta(i_c, k_c) > 0$ and $\delta(i_c, k_c - 1) \le 0$ then
5:             $f(i_c) = k_c$, $i_c \leftarrow i_c + 1$
6:         else if $\delta(i_c, k_c) > 0$ then $k_c \leftarrow k_c - 1$
7:         else   $i_c \leftarrow i_c + 1$
8:         compute $\delta(i_c, k_c)$
```

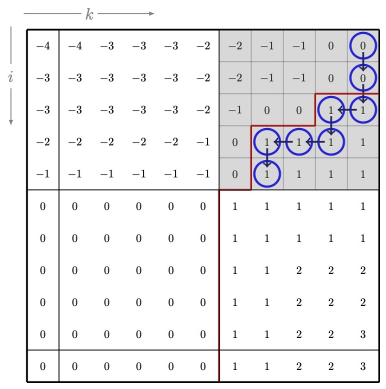


Figure 3: An example of Algorithm 1, $L = 5$. The algorithm runs on the gray points. The red line corresponds to f. The blue circles represent the traversal of Algorithm 1.

the beginning, each δ query only takes $O(1)$ time. Combining all together, Algorithm 1 takes $O(L)$ time.

The input to Algorithm 1. Using Algorithm 1, we can identify f inside a grid. However, we need to clarify one last point, the input to Algorithm 1 is $(iL, (k+1)L)$ and $\delta(iL, (k+1)L)$. We already have $\delta(iL, \tilde{f}(i)L)$ for fixed i, but we do not know $\delta(iL, (k+1)L)$ for all $k+1 \in [\tilde{f}(i+1) : \tilde{f}(i)]$. We still need to figure out $\delta(iL, (k+1)L)$ for $k+1 \in [\tilde{f}(i+1) : \tilde{f}(i))$. Note that for a fixed i and for $j \in [\tilde{f}(i+1)L : \tilde{f}(i)L)$, we have

$$\delta(iL, j) - \delta(iL, j-1) = \sum_{\hat{\imath} \in \langle 0:iL \rangle} P_{C,hi}(\hat{\imath}, j - \tfrac{1}{2}) + \sum_{\hat{\imath} \in \langle iL:n \rangle} P_{C,lo}(\hat{\imath}, j - \tfrac{1}{2}).$$

The right hand side can be computed in $O(1)$ time. We can set a vector of length $(\tilde{f}(i) - \tilde{f}(i+1))L$ and, for $j \in [\tilde{f}(i+1)L : \tilde{f}(i)L)$, we fill $-(\sum_{\hat{\imath} \in \langle 0:iL \rangle} P_{C,hi}(\hat{\imath}, j - \frac{1}{2}) + \sum_{\hat{\imath} \in \langle iL:n \rangle} P_{C,lo}(\hat{\imath}, j - \frac{1}{2}))$ inside. Then we computes a prefix-sum to obtain $\delta(iL, j) - \delta(iL, \tilde{f}(i)L)$ for $j \in [\tilde{f}(i+1)L : \tilde{f}(i)L)$. We can finally obtain $\delta(iL, j)$ by adding $\delta(iL, \tilde{f}(i)L)$ back. Lines 2–8 of Algorithm 2 is used for computing the input of Algorithm 1.

For each $i \in [0 : \frac{n}{L}]$, it takes $O((\tilde{f}(i) - \tilde{f}(i+1))L)$ work and $O(\log n)$ span. In total, computing $\delta(iL, (k+1)L)$ for all $i \in [0 : \frac{n}{L}]$ and $k+1 \in [\tilde{f}(i+1) : \tilde{f}(i))$ takes $O(n)$ work and $O(\log n)$ span. This gives us the following lemma:

Algorithm 2 Identifying f using $\tilde{f}$.
Input: $(i, \tilde{f}(i))$ and $\delta(iL, \tilde{f}(i)L)$ for $i \in [0 : \frac{n}{L}]$.
Output: $f(j)$ for $j \in [0, n]$.

```
1: function computeFirstIndex()
2:     for $i \in [0, \frac{n}{L})$
3:         for $j \in [\tilde{f}(i+1)L : \tilde{f}(i)L)$
4:             $\Delta(j) = \sum_{\hat{\imath} \in \langle 0:iL \rangle} P_{C,hi}(\hat{\imath}, j) + \sum_{\hat{\imath} \in \langle iL:n \rangle} P_{C,lo}(\hat{\imath}, j)$
5:         $\Delta(\tilde{f}(i)L) = 0$
6:         for $j$ from $\tilde{f}(i)L - 1$ downto $\tilde{f}(i+1)L$
7:             $\Delta(j) \leftarrow \Delta(j+1) + \Delta(j)$
8:             $\delta(iL, j) = \delta(iL, \tilde{f}(i)L) - \Delta(j)$
9:         for $k \in [\tilde{f}(i+1) - 1 : \tilde{f}(i))$
10:            IdentifyGrid($(iL, (k+1)L), \delta(iL, (k+1)L)$)
11:    IdentifyGrid($(n, \tilde{f}(\frac{n}{L})L), \delta(n, \tilde{f}(\frac{n}{L})L)$)  ▷ For the last row.
```

Lemma 3.8. *Given two $n \times n$ sized sub-permutation matrices $P_{C,hi}$ and $P_{C,lo}$, assume that we have already obtained all approximate indices $\tilde{f}(i)$ and $\delta(iL, \tilde{f}(i)L)$ for all $i \in [0 : \frac{n}{L}]$, then there exists a deterministic algorithm that computes $\delta(iL, (k+1)L)$ for $i \in [0, \frac{n}{L}]$, $k+1 \in [\tilde{f}(i+1) : \tilde{f}(i)]$ in $O(n)$ work and $O(\log n + L)$ span.* □

Parallel Implementation of Main Algorithm

Proof of Theorem 3.4. Given vector $\tilde{f}(i)$ and $(\delta(iL, \tilde{f}(i)L)$ for $i \in [0, \frac{n}{L}]$, we now discuss each of the main steps of solving the implicit subunit-Monge matrix multiplication in the EREW PRAM model. First, we use Algorithm 2 to compute f. For each $i \in [0 : \frac{n}{L})$, we initialize a vector of length $\tilde{f}(i)L - \tilde{f}(i+1)L + 1$ for Δ. Initialization takes $O(n)$ work and $O(1)$ span in total. For each $i \in [0 : \frac{n}{L})$ and $j \in [\tilde{f}(i+1)L : \tilde{f}(i)L)$, we can query $\Delta(j)$ independently. Note that j starts from $\tilde{f}(i+1)L$, but we can shift the index by $\tilde{f}(i+1)L$ and then the starting index of Δ will be 0. Next, we can use the prefix sum to compute $\Delta(j) \leftarrow \Delta(j+1) + \Delta(j)$. The prefix sum over $O(n)$ terms takes $O(n)$ work and $O(\log n)$ span.

Once we compute all possible grid cells $(iL, (k+1)L)$ and $\delta(iL, (k+1)L)$, we call Algorithm 1 to compute f. Since each $(j, f(j))$ will be located in exactly one grid cell, there will be no concurrent writes when we write the $f(j)$ values. To compute the neighboring δ, Algorithm 2 might query the same $P_{C,hi}(\hat{\imath}, \hat{\jmath})$ for several different grid cells. However, in the EREW PRAM model, different processors are not allowed to read the same entry simultaneously. To avoid such potential concurrent reads, we can attach the necessary $P_{C,hi}$ and $P_{C,lo}$ terms to the grid when we call Algorithm 1. We have at most $O(\frac{n}{L})$ possible grids for Algorithm 1, so we at most attach $O(n)$ rows and columns of $P_{C,hi}$ and $P_{C,lo}$ and this can be done with $O(n)$ work and $O(\log n)$ span. In total, Algorithm 2 takes $O(n)$ work and $O(L + \log n)$ span.

Using $f(j)$ for all $j \in [0, n]$, the algorithm can set up the non-zero terms of P_C by itself for (3.1a) and (3.1b) . For (3.1c) and fixed $\hat{\imath}$, there is one $\hat{k}$ such that $\delta(\hat{\imath} + \frac{1}{2}, \hat{k} + \frac{1}{2}) > 0$ and $\delta(\hat{\imath} + \frac{1}{2}, \hat{k} - \frac{1}{2}) = 0$. The last point is that based on the Lemma 2.1, P_C is also a subunit-Monge matrix, so we do not need to worry about collision when we read or write. Combining Lemma 3.5 and setting $L = O(\log^2 n)$ gives Theorem 3.4. □

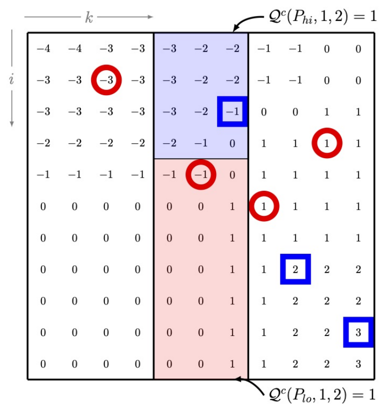


Figure 4: An example of Q^c. $L = 3$. The red circles are non-zero terms of $P_{C,lo}$. The blue squares are non-zero terms of $P_{C,hi}$. $Q^c(P_{hi}, 1, 2)$ counts blue squares in the blue region and $Q^c(P_{lo}, 1, 2)$ counts red circles in the red region.

3.3 Computing approximate index $\tilde{f}$

Mini-Batch Query Oracle. We need an oracle to support counting queries for some type of rectangle. More Formally, Given sub-permutation matrices P_{hi} and P_{lo}, we need to support the following queries:

- For sub-permutation matrix P_{hi}, $Q^r(P_{hi}, i, j)$ returns $\sum_{\hat{\imath}\in\langle(i-1)L:iL\rangle,\hat{\jmath}\in\langle 0:jL\rangle} P_{hi}(\hat{\imath},\hat{\jmath})$, i.e, the number of non-zero elements of $P_{hi}(\hat{\imath},\hat{\jmath})$, where $(\hat{\imath},\hat{\jmath}) \in \langle(i-1)L : iL\rangle \times \langle 0 : jL\rangle$. The r in the Q^r means to query grid rows.
- For sub-permutation matrix P_{lo}, $Q^r(P_{lo}, i, j)$ returns $\sum_{\hat{\imath}\in\langle(i-1)L:iL\rangle,\hat{\jmath}\in\langle jL:n\rangle} P_{lo}(\hat{\imath},\hat{\jmath})$,
- For sub-permutation matrix P_{hi}, $Q^c(P_{hi}, i, j)$ returns $\sum_{\hat{\imath}\in\langle 0:iL\rangle,\hat{\jmath}\in\langle(j-1)L:jL\rangle} P(\hat{\imath},\hat{\jmath})$, i.e, the number of non-zero elements of P_{hi} with index in the rectangle $\langle 0 : iL\rangle \times \langle (j-1)L : jL\rangle$.
- For sub-permutation matrix P_{lo}, $Q^c(P_{lo}, i, j)$ returns $\sum_{\hat{\imath}\in\langle iL:n\rangle,\hat{\jmath}\in\langle(j-1)L:jL\rangle} P_{lo}(\hat{\imath},\hat{\jmath})$.

See Figure 4 for Q^c query example. All oracles can be constructed and queried in a similar way. We use $Q^c(P_{hi}, i, j)$ for example. Note that P_{hi} is a sub-permutation matrix, and there will be at most L non-zero terms for P_{hi} in the rectangle $\langle 0 : n\rangle \times \langle (j-1)L : jL\rangle$. We can sort those elements based on their row index, which takes $O(L \log L)$ work and $O(\log L)$ span to sort for a fixed j. Since there are $\frac{n}{L}$ different j values, sorting for all j takes $O(n \log L)$ work and $O(\log L)$ span. To query, we can use binary search in the sorted array, which takes $O(\log L)$ work and $O(\log L)$ span to answer one query. Later, we will show a faster way to construct such a data structure.

Now we show that using Q, we can compute $\tilde{f}$.

Algorithm Description. computeAppIndex($[i_{lo} : i_{hi}], [k_{lo} : k_{hi}]$) computes $\tilde{f}(i)$ and $\delta(iL, \tilde{f}(i)L)$ for $i \in [i_{lo}, i_{hi}]$. Before we call computeAppIndex($[i_{lo} : i_{hi}], [k_{lo} : k_{hi}]$), we require that

- $\delta(iL, k_{lo}L)$ for $i \in (i_{lo} : i_{hi}]$
- $\delta(i_{lo}L, kL)$ for $k \in (k_{lo} : k_{hi}]$

Algorithm 3 Computing $\tilde{f}(i)$ for $i \in [0, \frac{n}{L}]$
Input: A rectangle $[i_{lo}L : i_{hi}L] \times [k_{hi}L : k_{lo}L]$; $\delta(iL, k_{lo}L)$ for $i \in (i_{lo} : i_{hi}]$, $\delta(i_{lo}L, kL)$ for $k \in (k_{lo} : k_{hi}]$. For the space reason, we ignore the δ in Line 1 below.
Output: $\tilde{f}(j)$ and $\delta(jL, \tilde{f}(j)L)$ for $j \in [i_{lo} : i_{hi}]$.

```
1:  function computeAppIndex($[i_{lo} : i_{hi}], [k_{lo} : k_{hi}]$)
2:      if $i_{lo} > i_{hi}$ then return
3:      $i_{mid} \leftarrow \lfloor (i_{lo} + i_{hi})/2 \rfloor$
4:      ▷ Compute $\delta(i_{mid}L, kL)$ for $k \in (k_{lo} : k_{hi}]$
5:      for $k \in (k_{lo} : k_{hi}]$
6:          $\Delta(k) \leftarrow Q^c(P_{hi}, i_{mid}, k) + Q^c(P_{lo}, i_{mid}, k)$
7:      $\Delta(k_{lo}) \leftarrow 0$
8:      for $k \in (k_{lo} : k_{hi}]$
9:          $\Delta(k) \leftarrow \Delta(k-1) + \Delta(k)$
10:         $\delta(i_{mid}L, kL) \leftarrow \delta(i_{mid}L, k_{lo}L) + \Delta(k)$
11:     ▷ Compute $\tilde{f}(i_{mid})$
12:     for $k$ from $k_{lo}$ upto $k_{hi}$
13:         if $\delta(i_{mid}L, kL) > 0$ then $\tilde{f}(i_{mid}) = k$, break
14:         if $k = \frac{n}{L}$ then $\tilde{f}(i_{mid}) = k$
15:     ▷ Compute $\delta(iL, \tilde{f}(i_{mid})L)$ for $i \in [i_{lo} : i_{mid}]$
16:     for $i \in (i_{lo} : i_{mid}]$
17:         $\Delta(k) \leftarrow Q^r(P_{hi}, i, \tilde{f}(i_{mid})) + Q^r(P_{lo}, i, \tilde{f}(i_{mid}))$
18:     $\Delta(i_{lo}) \leftarrow 0$
19:     for $i \in (i_{lo} : i_{mid}]$
20:         $\Delta(i) \leftarrow \Delta(i-1) + \Delta(i)$
21:         $\delta(iL, \tilde{f}(i_{mid})L) \leftarrow \delta(i_{lo}L, \tilde{f}(i_{mid})L) + \Delta(i)$
22:     computeAppIndex($[i_{lo} : i_{mid} - 1], [\tilde{f}(i_{mid}) : k_{hi}]$)
23:     computeAppIndex($[i_{mid} + 1 : i_{hi}], [k_{lo} : \tilde{f}(i_{mid})]$)
```

have been computed. We first compute the middle row index i_{mid}, then based on $\delta(iL, k_{lo}L)$, Lines 5–10 is to compute $\delta(i_{mid}L, kL)$ for $k \in (k_{lo}, k_{hi}]$. We will set $\tilde{f}(i_{mid})$ to be the smallest integer k such that $\delta(i_{mid}L, kL) > 0$. If no such integer exists, we set $\tilde{f}(i_{mid})$ to $\frac{n}{L}$. Based on Lemma (3.6a), for $j < i_{mid}$, we have $\tilde{f}(j) \geq \tilde{f}(i_{mid})$ and for $j > i_{mid}$, we have $\tilde{f}(j) \leq \tilde{f}(i_{mid})$. We can reduce the search area to two rectangles $[i_{lo} : i_{mid} - 1] \times [\tilde{f}(i_{mid}) : k_{hi}]$ and $[i_{mid} + 1 : i_{hi}] \times [k_{lo} : \tilde{f}(i_{mid})]$. Since we require the top and left δ value for the rectangle $[i_{lo} : i_{mid} - 1] \times [\tilde{f}(i_{mid}) : k_{hi}]$, lines 12–21 are used to compute the $\delta(iL, \tilde{f}(i_{mid})L)$, where $i \in [i_{lo} : i_{mid} - 1]$. See Figure 5 for an illustration.

We will call computeAppIndex($[0 : \frac{n}{L}], [0 : \frac{n}{L}]$) to compute $\tilde{f}(j)$, where $j \in [0 : \frac{n}{L}]$. To show the correctness of the algorithm, we first give the following invariant.

Lemma 3.9. *When we call* computeAppIndex($[i_{lo} : i_{hi}], [k_{lo} : k_{hi}]$)*, we have already computed*

- *$\delta(iL, k_{lo}L)$ for $i \in [i_{lo} : i_{hi}]$, and*
- *$\delta(i_{lo}L, kL)$ for $k \in [k_{lo} : k_{hi}]$.*

In addition, $\tilde{f}(i) \in [k_{lo}L, k_{hi}L]$ for all $i \in [i_{lo}, i_{hi}]$.

Proof. Assuming that we already computed the corresponding δ when we call computeAppIndex($[i_{lo} : i_{hi}], [k_{lo} : k_{hi}]$), we need to show that when we call the subproblem, the corresponding δ is

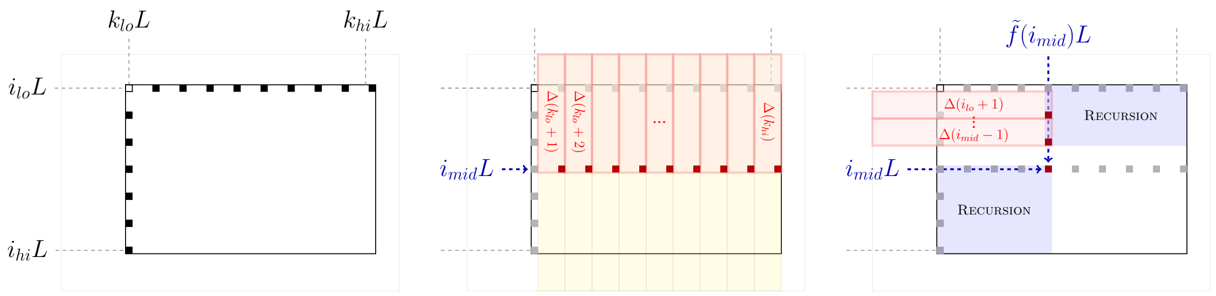


Figure 5: An illustration of Algorithm 3. In the left subfigure, we call COMPUTEAPPINDEX on $[i_{lo}:i_{hi}]$, $[k_{lo}:k_{hi}]$. Then in the middle subfigure, we first compute $\delta(i_{mid}L, kL)$ using the oracle Q^c, where $k \in [k_{lo}:k_{hi}]$. Then we can use those δ to derive $\tilde{f}(i_{mid})$. In the right subfigure, we recurse on the subproblems.

correctly computed. First, we show that $\delta(i_{mid}L, kL)$ is correctly computed. Note that for $k \in (k_{lo}, k_{hi}]$,

$$\begin{aligned}
&\delta(i_{mid}L, kL) - \delta(i_{mid}L, k_{lo}L)\\
&= \sum_{\substack{\hat{\imath}\in\langle 0:i_{mid}L\rangle,\\ \hat{\jmath}\in\langle k_{lo}L:kL\rangle}} P_{C,hi}(\hat{\imath},\hat{\jmath}) + \sum_{\substack{\hat{\imath}\in\langle i_{mid}L:n\rangle,\\ \hat{\jmath}\in\langle k_{lo}L:kL\rangle}} P_{C,lo}(\hat{\imath},\hat{\jmath})\\
&= \sum_{k'\in(k_{lo},k]} \left(\sum_{\substack{\hat{\imath}\in\langle 0:i_{mid}L\rangle,\\ \hat{\jmath}\in\langle (k'-1)L:k'L\rangle}} P_{C,hi}(\hat{\imath},\hat{\jmath}) + \sum_{\substack{\hat{\imath}\in\langle i_{mid}L:n\rangle,\\ \hat{\jmath}\in\langle (k'-1)L:k'L\rangle}} P_{C,lo}(\hat{\imath},\hat{\jmath}) \right)\\
&= \sum_{k'\in(k_{lo},k]} \left(Q^c(P_{hi}, i_{mid}, k) + Q^c(P_{lo}, i_{mid}, k)\right).
\end{aligned}$$

In line 9, we set $\Delta(k)$ to be the above quantity via the prefix sum. Based on the assumption, we have $\delta(i_{mid}L, k_{lo}L)$ and $\delta(i_{mid}L, kL)$ correctly computed. Since we also assume that $\tilde{f}(i_{mid}) \in [i_{lo}, i_{hi}]$, the first positive term index or $\frac{n}{L}$ must be $\tilde{f}(i_{mid})$. From Lemma (3.6a), we know that $\tilde{f}(j) \in [\tilde{f}(i_{mid}), i_{hi}]$ for $j \in [i_{lo}, i_{mid}-1]$ and $\tilde{f}(j) \in [i_{lo}, \tilde{f}(i_{mid})]$ for $j \in [i_{mid}+1, i_{hi}]$.

To enable recursion, several new δ entries should be computed before invoking the recursive calls. For COMPUTEAPPINDEX($[i_{mid}+1 : i_{hi}]$, $[k_{lo} : \tilde{f}(i_{mid})]$), we already computed the left and top line δ value. Lines 12–21 compute $\delta(iL, \tilde{f}(i_{mid})L)$ for COMPUTEAPPINDEX($[i_{lo} : i_{mid}-1]$, $[\tilde{f}(i_{mid}) : k_{hi}]$). The correctness follows from the correctness of lines 5–10 so we omit the repeated details here. □

The correctness of Algorithm 3 follows from Lemma 3.9. In each step, we can compute $\tilde{f}(i_{mid})$ and we will transverse all $i_{mid} \in [0, \frac{n}{L}]$. The running time not only depends on Algorithm 3, but also depends on the oracle Q. In next section, we will show we can construct Q in $O(n)$ work.

Mini-Batch Query Oracle

Throughout the execution of the algorithm, an important step is to count the number of elements within a thin strip of width $L = \Omega(\log n \log\log n)$, i.e, construct the oracle Q. The simplest way to achieve this is to partition the input permutation into size L mini-batches, and sort the numbers within each mini-batch in increasing order. To answer the range query, the algorithm performs two binary searches in $O(\log L)$ time. To preprocess each mini-batch, we may use the signature sorting algorithm from Andersson et al. [3] to obtain a randomized linear work sorting algorithm.

THEOREM 3.10 (A SIMPLE CASE FROM [3]). *Let A be an array of L integers. There exists a Monte Carlo algorithm on an EREW PRAM model supporting* AC^0 *operations with word size $Z = \Omega(\log^3 L)$, such that with probability at least $1 - 1/L$, the algorithm sorts A in $O(L)$ work and $O(\log L \log Z)$ time*[4].

LEMMA 3.11. *Let $L = \Omega(\log n \log\log n)$. Consider $\frac{n}{L}$ groups of integers where each group has L elements. Then there exists an EREW PRAM algorithm with* AC^0 *operations on word size $Z = \Theta(\log n)$ such that, with probability $1 - n^{-10}$, the algorithm sorts all groups of integers in $O(n)$ work and $O((\log\log n)^2)$ time.*

PROOF. It suffices to perform a two-phase algorithm. In the first phase, we apply signature sort (Theorem 3.10) to each group. The algorithm then checks whether or not each group is sorted in $O(1)$ time and $O(n)$ work. In the second phase, the algorithm applies any standard deterministic sorting to the groups where they are not yet sorted. For example, we can use the AKS sorting network [2] which sorts each group deterministically in $O(L \log L)$ work and $O(\log L)$ time. By Theorem 3.10 and a standard Chernoff bound, with probability $1 - n^{-10}$, there are at most $O(n \log n / L^2)$ groups that are not sorted. Therefore, the second phase takes only $O(\frac{n \log n}{L^2} \cdot (L \log L)) = O(n)$ work and $O(\log L) = O(\log\log n)$ time. □

Finally, we are able to analyze the work and span of Algorithm 3 and then prove Lemma 3.5.

Work and Span Analysis

The total work to the step satisfies the following recurrence relation:

$$W(h,d) = W(h/2, x+1) + W(h/2, d-x) + O((d+h)\cdot \log L)$$

where $h = i_{hi} - i_{lo}$ and $d = k_{hi} - k_{lo}$ and $x \in [0, d)$. The $O((d+h)\log L)$ term comes from the fact that we need to query Q $O(d+h)$ times and each query takes $O(\log L)$ time. The above expression solves to $W(h,d) = O((h+d)\log h \log L)$. By applying $h = d = \frac{n}{L}$ we obtain a total work of $O(\frac{n}{L}\log n \log L)$ for Algorithm 3. Notice that we still need to construct the oracle Q. Based on Lemma 3.11, the construction of Q takes $O(n \log\log n)$ work for deterministic

[4]If we allow constant time multiplications, then the signature sort runs in only $O(\log L)$ time.

algorithm and $O(n)$ work for randomized algorithm when $L = \Omega(\log n \log\log n)$.

As for the span, constructing Q has $O((\log\log n)^2)$ span by Lemma 3.11 so it is not a bottleneck. The span of Algorithm 3 satisfies the following recurrence relation: $S(h,d) \le S(h/2,d) + O(\log h + \log d + \log L)$, which solves to $S(h,d) = O(\log h(\log h + \log d + \log L))$. Whenever $h = d = \frac{n}{L}$ this is $O(\log^2 n)$.

Proof of Lemma 3.5. Algorithm 3 is naturally parallelized, and exclusive reads are achievable. Whenever the algorithm makes queries to Q, it will never query the same columns or rows, so concurrent read operations are not required. Furthermore, $\tilde{f}(i_{mid})$ is updated once, so there will be no concurrent write operations. Setting $L = \Omega(\log n \log\log n)$ gives us Lemma 3.5. □

4 COMPUTING LIS WITH SUBUNIT-MONGE MATRIX MULTIPLICATION

In this section, we will explain the connection between the LIS problem and implicit subunit-Monge matrix multiplication $\boxdot$. We will give more details of Theorem 1.2.

Let $A = (a_1, a_2, \dots, a_m)$ be any *partial permutation* of $[n]$. That is, for any i we have $a_i \in [n]$ and for any $i \ne j$ we have $a_i \ne a_j$. Recall that the filtered sequence $A_{i,j}$ is the subsequence of A whose values are all within the range $[i:j]$ and $LIS(A_{i,j})$ is the length of LIS of $A_{i,j}$. Let $M_A^{\Sigma}(i,j) = j - i - LIS(A_{i+1,j})$ whenever $0 \le i < j \le n$ and 0 otherwise. Let $L = (a_1, a_2, \dots, a_{\lfloor m/2\rfloor})$ and $R = (a_{\lfloor m/2\rfloor+1}, \dots, a_m)$. The following lemma explains that the matrix M_A^{Σ} can be obtained from the $(\min,+)$-matrix multiplication of M_L^{Σ} and M_R^{Σ}.

Lemma 4.1. *For any $0 \le i < j \le n$, we have*

$$M_A^{\Sigma}(i,j) = \min_{0\le k\le n}\{M_L^{\Sigma}(i,k) + M_R^{\Sigma}(k,j)\}.$$

Proof. We view $M_A^{\Sigma}(i,j)$ as the minimum number of elements that have to be kicked out in order to obtain an increasing sequence from the *affixed filtered subsequence* $A'_{i+1,j}$, which is defined to be the filtered subsequence $A_{i+1,j}$ attached with $j - i - |A_{i+1,j}|$ must-be-deleted dummy elements. Then the lemma follows by a standard shortest path argument. □

A celebrated result from Tiskin [33] is that M_A^{Σ} is actually a distribution matrix of an $n \times n$ sub-permutation matrix P_A.

Lemma 4.2 ([33]). *Fix an integer n. For any partial permutation A of $[n]$, there is a sub-permutation matrix P_A such that $M_A^{\Sigma} = P_A^{\Sigma}$.* □

By applying Lemma 4.2 to Lemma 4.1, we know that whenever $A = L \circ R$ then $P_A = P_L \boxdot P_R$. To enable efficient divide and conquer algorithm, the algorithm must be able to reduce the size of the sub-permutation matrices P_L and P_R. The following lemma shows that whenever $m < n$ we can inductively relabel the elements in A.

Lemma 4.3 ([33]). *Let A be any partial permutation of $[n]$. Then for any half integer $\hat{\imath} \in \langle 0 : n\rangle$, $P_A(\hat{\imath},\hat{\imath}) = 1$ if and only if $i = \hat{\imath} + \frac{1}{2}$ does not appear in A. Suppose in addition that i is not in A. Let A' be a partial permutation of $[n-1]$, by taking a copy of A and then decreasing all elements greater than i by one. Then, the corresponding sub-permutation matrix $P_{A'}$ is obtained by removing the $\hat{\imath}$-th row and the $\hat{\imath}$-th column from P_A.* □

Proof of Theorem 1.2. Consider a permutation A of $[n]$. Without loss of generality we assume that n is a power of 2. The algorithm obtains P_A through the following divide and conquer method.

Divide. The algorithm first splits A into two halves $A = A_{lo} \circ A_{hi}$. After relabeling A_{lo} and A_{hi} we obtain permutations A'_{lo} and A'_{hi} of $[n/2]$. This can be done in $O(n)$ work and $O(\log n)$ span.

Conquer. Then, the algorithm invokes recursive calls to both A'_{lo} and A'_{hi} and obtains two $(n/2) \times (n/2)$ sub-permutation matrices $P_{A'_{lo}}$ and $P_{A'_{hi}}$.

Combine. Using Lemma 4.3, the algorithm recovers $P_{A_{lo}}$ and $P_{A_{hi}}$, and then apply the $\boxdot$ operation which takes a total of $O(W(n) + n)$ work and $O(S(n) + \log n)$ span.

Base Case. The base case contains a single element, where we can set up $P(\frac{1}{2},\frac{1}{2}) = 1$ directly.

Reporting the Length of LIS and the Analysis. In the end, the length of LIS can be obtained by inspecting the cell $M_A^{\Sigma}(0,n)$, which can be obtained by counting non-zero elements of P_A. It is straightforward to check that the algorithm takes $O(W(n)\log n + n\log n)$ work and $O(S(n)\log n + \log^2 n)$ span. □

The above divide and conquer algorithm only returns the length of LIS. Now we give another recursive algorithm that returns an LIS using the intermediate results.

Reporting LIS

To illustrate the algorithm, we first label each subproblem during the computation of the LIS length. In particular, the r-th subproblem of recursion depth d can be labelled by a pair of integers (d,r), where $d \in [0 : \log n]$ and $r \in [0 : 2^d)$. Let the consecutive subsequence used for this subproblem to be $X(d,r) = (a_{start}, a_{start+1}, ..., a_{end})$, where $start = r \cdot (n/2^d)$, $end = (r+1)\cdot(n/2^d) - 1$. We note that the algorithm does not work directly with $X(d,r)$. In the divide step, the algorithm relabeled the values before recursion, which results in the corresponding permutation $X'(d,r)$ of $\{1,2,\dots,n/2^d\}$. At the subproblem (d,r), the algorithm returns the sub-permutation matrix $P_{X'(d,r)}$. These sub-permutation matrices are used for reconstructing $M_{X'(d,r)}^{\Sigma}$ implicitly, which are useful for us.

The algorithm starts with $(i,j,d,r) := (0,n,0,0)$. That is, we are looking for what constitutes the value $M_A^{\Sigma}(0,n)$ of the input permutation $A = X(0,0)$. Using Lemma 4.1, the algorithm is able to identify the argmin index k such that $M_A^{\Sigma}(i,j) = M_{A_{lo}}^{\Sigma}(i,k) + M_{A_{hi}}^{\Sigma}(k,j)$, where $A = A_{lo} \circ A_{hi}$ is split into two halves. Notice that now $A'_{lo} = X'(d+1,2r)$ and $A'_{hi} = X'(d+1,2r+1)$. Before invoking the recursive call, the algorithm calculates the corresponding new indices (i',k') from (i,k), and (k'',j') from (j,k). Then, the algorithm recurses on $(i',k',d+1,2r)$ and $(k'',j',d+1,2r+1)$, seeking for an optimal sequence in A'_{lo} and A'_{hi} that can be recovered (via Lemma 4.3) and then combined into an LIS in A.

In each recursion of parameters (i,j,d,r), the algorithm requires only the i-th row of $M_{X(d+1,2r)}^{\Sigma}$ and the j-th column of $M_{X(d+1,2r+1)}^{\Sigma}$. Using the prefix sum subroutine, the algorithm can obtain the argmin in $O(|X(d,r)|)$ total work and $O(\log|X(d,r)|) = O(\log n)$ span. Therefore, this additional algorithm that reports an LIS takes $O(n\log n)$ total work and $O(\log^2 n)$ span.

REFERENCES

[1] Alfred V. Aho, Daniel S. Hirschberg, and Jeffrey D. Ullman. 1976. Bounds on the Complexity of the Longest Common Subsequence Problem. *J. ACM* 23, 1 (1976), 1–12. https://doi.org/10.1145/321921.321922

[2] Miklós Ajtai, János Komlós, and Endre Szemerédi. 1983. An O(n log n) Sorting Network. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas (Eds.). ACM, 1–9. https://doi.org/10.1145/800061.808726

[3] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. 1995. Sorting in linear time?. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, Frank Thomson Leighton and Allan Borodin (Eds.). ACM, 427–436. https://doi.org/10.1145/225058.225173

[4] Jinho Baik, Percy Deift, and Kurt Johansson. 1999. On the distribution of the length of the longest increasing subsequence of random permutations. *Journal of the American Mathematical Society* 12, 4 (1999), 1119–1178.

[5] Laura Baxter, Aleksey Jironkin, Richard Hickman, Jonathan Moore, Christopher Barrington, Peter Krusche, Nigel Dyer, Vicky Buchanan-Wollaston, Alexander Tiskin, Jim Beynon, Katherine Denby, and Sascha Ott. 2012. Conserved Noncoding Sequences Highlight Shared Components of Regulatory Networks in Dicotyledonous Plants. *The Plant cell* 24 (10 2012). https://doi.org/10.1105/tpc.112.103010

[6] Timothy M Chan. 2007. More algorithms for all-pairs shortest paths in weighted graphs. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. 590–598.

[7] Maxime Crochemore, Gad M. Landau, and Michal Ziv-Ukelson. 2003. A Subquadratic Sequence Alignment Algorithm for Unrestricted Scoring Matrices. *SIAM J. Comput.* 32, 6 (2003), 1654–1673. https://doi.org/10.1137/S0097539702402007

[8] Maxime Crochemore and Ely Porat. 2010. Fast computation of a longest increasing subsequence and application. *Inf. Comput.* 208, 9 (2010), 1054–1059. https://doi.org/10.1016/j.ic.2010.04.003

[9] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg. 1999. Alignment of whole genomes. *Nucleic Acids Res.* 27, 11 (1999), 2369–2376. https://doi.org/10.1093/nar/27.11.2369

[10] Arthur L Delcher, Steven L Salzberg, and Adam M Phillippy. 2003. Using MUMmer to identify similar regions in large sequence sets. *Current protocols in bioinformatics* 1 (2003), 10–3.

[11] Michael L. Fredman. 1975. On computing the length of longest increasing subsequences. *Discret. Math.* 11, 1 (1975), 29–35. https://doi.org/10.1016/0012-365X(75)90103-X

[12] James C Fu and Yu-Fei Hsieh. 2015. On the distribution of the length of the longest increasing subsequence in a random permutation. *Methodology and Computing in Applied Probability* 17, 2 (2015), 489–496.

[13] Z. Galil and K. Park. 1994. Parallel Algorithms for Dynamic Programming Recurrences with More Than O(1) Dependency. *J. Parallel and Distrib. Comput.* 21, 2 (1994), 213–222. https://doi.org/10.1006/jpdc.1994.1053

[14] Thierry Garcia, Jean Frédéric Myoupo, and David Semé. 2001. A work-optimal CGM algorithm for the LIS problem. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2001, Heraklion, Crete Island, Greece, July 4-6, 2001*, Arnold L. Rosenberg (Ed.). ACM, 330–331. https://doi.org/10.1145/378580.378756

[15] Yan Gu, Ziyang Men, Zheqi Shen, Yihan Sun, and Zijin Wan. 2023. Parallel Longest Increasing Subsequence and van Emde Boas Trees. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23), June 17–19, 2023, Orlando, FL, USA*. https://doi.org/10.48550/arXiv.2208.09809

[16] Yijie Han. 2004. Deterministic sorting in $O(n \log\log n)$ time and linear space. *J. Algorithms* 50, 1 (2004), 96–105. https://doi.org/10.1016/j.jalgor.2003.09.001

[17] Yijie Han and Tadao Takaoka. 2016. An $O(n^3 \log\log n/\log^2 n)$ time algorithm for all pairs shortest paths. *J. Discrete Algorithms* 38-41 (2016), 9–19. https://doi.org/10.1016/j.jda.2016.09.001

[18] Sungjin Im, Benjamin Moseley, and Xiaorui Sun. 2017. Efficient massively parallel methods for dynamic programming. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, Hamed Hatami, Pierre McKenzie, and Valerie King (Eds.). ACM, 798–811. https://doi.org/10.1145/3055399.3055460

[19] Joseph F. JáJá, Christian Worm Mortensen, and Qingmin Shi. 2004. Space-Efficient and Fast Algorithms for Multidimensional Dominance Reporting and Counting. In *Algorithms and Computation, 15th International Symposium, ISAAC 2004, Hong Kong, China, December 20-22, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3341)*, Rudolf Fleischer and Gerhard Trippen (Eds.). Springer, 558–568. https://doi.org/10.1007/978-3-540-30551-4_49

[20] Peter Krusche and Alexander Tiskin. 2009. Parallel Longest Increasing Subsequences in Scalable Time and Memory. In *Parallel Processing and Applied Mathematics, 8th International Conference, PPAM 2009, Wroclaw, Poland, September 13-16, 2009. Revised Selected Papers, Part I (Lecture Notes in Computer Science, Vol. 6067)*, Roman Wyrzykowski, Jack J. Dongarra, Konrad Karczewski, and Jerzy Wasniewski (Eds.). Springer, 176–185. https://doi.org/10.1007/978-3-642-14390-8_19

[21] Peter Krusche and Alexander Tiskin. 2010. New algorithms for efficient parallel string comparison. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, Friedhelm Meyer auf der Heide and Cynthia A. Phillips (Eds.). ACM, 209–216. https://doi.org/10.1145/1810479.1810521

[22] Mi Lu and Hua Lin. 1994. Parallel Algorithms for the Longest Common Subsequence Problem. *IEEE Trans. Parallel Distributed Syst.* 5, 8 (1994), 835–848. https://doi.org/10.1109/71.298210

[23] Amgad Madkour, Walid G. Aref, Faizan Ur Rehman, Mohamed Abdur Rahman, and Saleh M. Basalamah. 2017. A Survey of Shortest-Path Algorithms. *CoRR* abs/1705.02044 (2017). arXiv:1705.02044 http://arxiv.org/abs/1705.02044

[24] Takaaki Nakashima and Akihiro Fujiwara. 2006. A Cost Optimal Parallel Algorithm for Patience Sorting. *Parallel Process. Lett.* 16, 1 (2006), 39–52. https://doi.org/10.1142/S0129626406002459

[25] Ryan O'Donnell and John Wright. 2017. Guest Column: A Primer on the Statistics of Longest Increasing Subsequences and Quantum States (Shortened Version). *SIGACT News* 48, 3 (2017), 37–59. https://doi.org/10.1145/3138860.3138869

[26] Emma Picot, Peter Krusche, Alexander Tiskin, Isabelle Carré, and Sascha Ott. 2010. Evolutionary analysis of regulatory sequences (EARS) in plants. *The Plant Journal* 64, 1 (2010), 165–176.

[27] Prakash Ramanan. 1997. Tight $\Omega(n \lg n)$ lower bound for finding a longest increasing subsequence. *Int. J. Comput. Math.* 65, 3-4 (1997), 161–164. https://doi.org/10.1080/00207169708804607

[28] Yoshifumi Sakai and Shunsuke Inenaga. 2022. A Faster Reduction of the Dynamic Time Warping Distance to the Longest Increasing Subsequence Length. *Algorithmica* 84, 9 (2022), 2581–2596. https://doi.org/10.1007/s00453-022-00968-2

[29] David Sankoff. 1983. Time warps, string edits, and macromolecules. *The Theory and Practice of Sequence Comparison, Reading* (1983).

[30] David Semé. 2006. A CGM Algorithm Solving the Longest Increasing Subsequence Problem. In *Computational Science and Its Applications - ICCSA 2006, International Conference, Glasgow, UK, May 8-11, 2006, Proceedings, Part V (Lecture Notes in Computer Science, Vol. 3984)*, Marina L. Gavrilova, Osvaldo Gervasi, Vipin Kumar, Chih Jeng Kenneth Tan, David Taniar, Antonio Laganà, Youngsong Mun, and Hyunseung Choo (Eds.). Springer, 10–21. https://doi.org/10.1007/11751649_2

[31] Zheqi Shen, Zijin Wan, Yan Gu, and Yihan Sun. 2022. Many Sequential Iterative Algorithms Can Be Parallel and (Nearly) Work-efficient. In *SPAA '22: 34th ACM Symposium on Parallelism in Algorithms and Architectures, Philadelphia, PA, USA, July 11 - 14, 2022*, Kunal Agrawal and I-Ting Angelina Lee (Eds.). ACM, 273–286. https://doi.org/10.1145/3490148.3538574

[32] Yihan Sun and Guy E. Blelloch. 2019. Parallel Range, Segment and Rectangle Queries with Augmented Maps. In *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*, Stephen G. Kobourov and Henning Meyerhenke (Eds.). SIAM, 159–173. https://doi.org/10.1137/1.9781611975499.13

[33] Alexander Tiskin. 2007. Semi-local string comparison: algorithmic techniques and applications. (2007). https://doi.org/10.48550/ARXIV.0707.3619

[34] Alexander Tiskin. 2008. Semi-local longest common subsequences in subquadratic time. *J. Discrete Algorithms* 6, 4 (2008), 570–581. https://doi.org/10.1016/j.jda.2008.07.001

[35] Alexander Tiskin. 2010. Fast Distance Multiplication of Unit-Monge Matrices. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, Moses Charikar (Ed.). SIAM, 1287–1296. https://doi.org/10.1137/1.9781611973075.103

[36] Alexander Tiskin. 2015. Fast Distance Multiplication of Unit-Monge Matrices. *Algorithmica* 71, 4 (2015), 859–888. https://doi.org/10.1007/s00453-013-9830-z

[37] Peter van Emde Boas. 1977. Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space. *Inf. Process. Lett.* 6, 3 (1977), 80–82. https://doi.org/10.1016/0020-0190(77)90031-X

[38] Peter van Emde Boas, R. Kaas, and E. Zijlstra. 1977. Design and Implementation of an Efficient Priority Queue. *Math. Syst. Theory* 10 (1977), 99–127. https://doi.org/10.1007/BF01683268