Dynamic Range Minimum Queries on the Ultra-Wide Word RAM*

Philip Bille[1[0000−0002−1120−5154]], Inge Li Gørtz[1[0000−0002−8322−4952]], Máximo Pérez López[1[0009−0001−8507−4820]], and Tord Stordalen[1[0000−0002−1525−0104]]

Technical University of Denmark, DTU Compute, Kgs. Lyngby, Denmark
{phbi,inge,mpelo,tjost}@dtu.dk

Abstract. We consider the dynamic range minimum problem on the ultra-wide word RAM model of computation. This model extends the classic w-bit word RAM model with special ultrawords of length w^2 bits that support standard arithmetic and boolean operation and scattered memory access operations that can access w (non-contiguous) locations in memory. The ultra-wide word RAM model captures (and idealizes) modern vector processor architectures.
Our main result is a linear space data structure that supports range minimum queries and updates in $O(\log\log\log n)$ time. This exponentially improves the time of existing techniques. Our result is based on a simple reduction to prefix minimum computations on sequences $O(\log n)$ words combined with a new parallel, recursive implementation of these.

Keywords: Ultra-wide word RAM model · Range minimum queries · Prefix minimum.

1 Introduction

Supporting *range minimum queries* (RMQ) on arrays is a well-studied, classic data structure problem, see e.g., [1, 2, 3, 4, 9, 10, 11, 12, 14, 16, 17, 19, 23, 25, 27, 28]. This paper considers the *dymamic RMQ problem* defined as follows: maintain an array $A[0, \ldots, n-1]$ of w-bit integers and subject to the following operations.

- $\mathsf{rmq}(i, j)$: return a smallest integer in the subarray $A[i..j]$.
- $\mathsf{update}(i, \alpha)$: set $A[i] \leftarrow \alpha$.

On most models of computation, the complexity of the dynamic RMQ problem is well-understood [2, 9, 11, 23, 25]. For instance, on the word RAM, a tight $\Theta(\log n/\log\log n)$ time bound on the operations is known [11]. Hence, a natural question is whether practical models of computation capturing modern hardware advances will allow us to improve this bound significantly. One such model is the *ultra-wide word RAM model* (UWRAM) introduced by Farzan et al. [15]. The UWRAM extends the word RAM model with special *ultrawords* of w^2 bits. The model supports standard boolean and arithmetic operations on

* Partially supported by the Danish Research Council via grant DFF-8021-002498.

ultrawords and *scattered* memory operations that access w words in parallel. The UWRAM model captures (and idealizes) modern vector processing architectures [13,22,24,26]. We present the details of the UWRAM model of computation in Section 2. By extending recent techniques for the UWRAM model [5], we can immediately solve the dynamic RMQ problem using $O(\log\log n)$ time per operation.

1.1 Results and Techniques

Our main result is an exponential improvement of the $O(\log\log n)$ time bound. More precisely, we show the following bound:

Theorem 1. *Given an array A of n w-bit integers, we can construct an $O(n)$ space data structure in $O(n)$ time on the UWRAM that supports* rmq *and* update *in $O(\log\log\log n)$ time.*

Technically, our solution is based on a simple linear space and logarithmic time folklore solution, which we call the *range minimum tree* (see Section 3 for a detailed description). The range minimum tree is a balanced binary tree over the input array and supports operations in $O(\log n)$ time by sequentially traversing the tree. On the UWRAM, we show how to efficiently compute the access patterns of the operations on the range minimum tree in parallel using scattered memory access operations and prefix minimum computation on ultrawords. More precisely, we show that given any algorithm for a prefix minimum computation on a sequence of words of length $\ell = O(w)$ stored in a constant number of ultrawords that uses time $t(\ell)$ implies a linear space solution for dynamic RMQ that supports both operations in $O(t(\log n))$ time. If we implement a standard parallel prefix computation algorithm [20] (see also the survey by Blelloch [8]) using the UWRAM techniques in Bille et al. [5] we immediately obtain an algorithm that uses $O(\log \ell)$ time. Our main technical contribution is a new, exponentially faster prefix minimum algorithm that achieves $O(\log\log \ell)$ time. The key idea is a constant time prefix minimum algorithm for "short" sequences of $O(\sqrt{w})$ words that takes advantage of parallel computations on multiple copies of the sequence packed into a constant number of ultrawords (note that a constant number of ultrawords can store $O(\sqrt{w})$ copies of a sequence of $O(\sqrt{w})$ words). We implement the idea recursively and in parallel on each recursion level to obtain a fast solution for general sequences of words of length $O(w)$. Each recursion step uses constant time, and the depth is $O(\log\log \ell)$, leading to the $O(\log\log \ell)$ time bound.

1.2 Outline

The paper is organized as follows. In Section 2 and 3 we review the UWRAM model of computation and the range minimum tree. In Section 4.1, we present the UWRAM implementation of the range minimum tree that leads to the reduction to prefix minimum computation on word sequences. Finally, in Section 5, we present our fast prefix minimum algorithm.

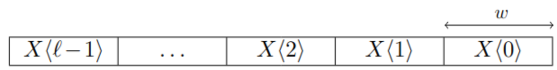

Fig. 1. The layout of a word sequence X.

2 The Ultra-Wide Word RAM Model

The *word RAM* model of computation [18] consists of an unbounded memory of w-bit words and a standard instruction set including arithmetic, boolean, and bitwise operations (denoted '&,' '|,' and '∼' for *and*, *or* and *not*) and shifts (denoted '≫' and '≪') such as those available in standard programming languages (e.g., C). We assume that we can store a pointer to the input in a single word and hence $w \geq \log n$, where n is the input size. The time complexity of a word RAM algorithm is the number of instructions, and the space is the number of words the algorithm stores.

The *ultra-wide word RAM* (UWRAM) model of computation [15] extends the word RAM model with special *ultrawords* of w^2 bits. As in [15], we distinguish between the *restricted UWRAM* that supports a minimal set of instructions on ultrawords consisting of addition, subtraction, shifts, and bitwise boolean operations, and the *multiplication UWRAM* that additionally supports multiplications. We extend the notation for bitwise operations and shifts to ultrawords. The UWRAM (restricted and multiplication) also supports contiguous and scattered memory access operations, as described below. The time complexity is the number of instructions (on standard words or ultrawords), and the space complexity is the number of words used by the algorithms, where each ultraword is counted as w words. The UWRAM model captures (and idealizes) modern vector processing architectures [13,22,24,26]. See Farzan et al. [15] for a detailed discussion of the applicability of the UWRAM model.

2.1 Instructions and Componentwise Operations

Recall that ultrawords consist of w^2 bits. We often use ultrawords to store and manipulate small sequences of $O(w)$ words. A *word sequence* X of length ℓ is a sequence of ℓ words (also called the components of X). We number the words from right to left starting from 0 and use the notation $X\langle i\rangle$ to denote the ith word in X (see Figure 1).

We define common operations on word sequences that we will use later. Let X and Y be word sequences of length ℓ. The *componentwise addition* of X and Y is the word sequence Z such that $Z\langle i\rangle = X\langle i\rangle + Y\langle i\rangle$. The *componentwise comparison* of X and Y is the word sequence Z such that $Z\langle i\rangle = 1$ if $X\langle i\rangle < Y\langle i\rangle$ and 0 otherwise. Given another word sequence I of length ℓ, where each word is either 0 and 1 (we will call this a *binary word sequence*), the *componentwise extract* of X wrt. I is the word sequence Z such that $Z\langle i\rangle = X\langle i\rangle$ if $I\langle i\rangle = 1$ and $Z\langle i\rangle = 0$ otherwise. We can also *concatenate* X and Y, denoted $X \cdot Y$ producing the length 2ℓ word sequence $X\langle \ell-1\rangle \cdots X\langle 0\rangle Y\langle \ell-1\rangle \cdots Y\langle 0\rangle$ or *split* X at any point k into word sequences $X\langle \ell-1\rangle \cdots X\langle k+1\rangle$ and $X\langle k\rangle \cdots X\langle 0\rangle$. Note that using two split operations, we can compute any contiguous subsequence $X\langle i\rangle \cdots X\langle j\rangle$ of X. All these operations can be implemented in constant time for word sequences of length $O(w)$ on the restricted UWRAM using standard-word level parallelism techniques [5, 6, 18]. Note that we can manipulate word sequences with length $\leq cw$, for some constant $c > 1$, by storing them in c ultrawords and simulating operations on them in constant time.

The UWRAM also supports a *compress* operation that takes a binary word sequence I of length ℓ and constructs the bitstring of the ℓ bits of I. The inverse *spread* operation takes a bitstring of length ℓ and constructs the corresponding binary word sequence of length ℓ. This is the UWRAM model that we will use throughout the rest of the paper. Note that these operations are widely supported directly in modern vector processing architectures.

Given a word sequence of X of length ℓ, we define the *prefix minimum* of X, denoted $\mathsf{pmin}(X)$, to be the word sequence P of length ℓ such that $P\langle i\rangle = \mathsf{min}(X\langle i\rangle, \ldots, X\langle 0\rangle)$. We also define the *minimum* of X, denoted $\mathsf{min}(X)$, as the smallest entry among all entries in X. Note that we can use a prefix minimum algorithm to compute the minimum. The prefix minimum operation is central in our solutions, and as discussed, we will show how to implement it in $O(\log\log \ell)$ time.

Some of our operations require precomputed constant word sequences, which we assume are available (e.g., computed at "compile-time"). If not, we can compute those needed for Theorem 1 in $\log^{O(1)} n$ time, which is negligible.

2.2 Memory Access

The UWRAM supports standard memory access operations that read or write a single word or a sequence of w contiguous words. More interestingly, the UWRAM also supports *scattered* access operations that access w memory locations (not necessarily contiguous) in parallel. Given a word sequence A containing w memory addresses, a *scattered read* loads the contents of the addresses into a word sequence X, such that $X\langle i\rangle$ contains the contents of memory location $A\langle i\rangle$. Given word sequences X and A of length $O(w)$ a *scattered write* sets the contents of memory location $A\langle i\rangle$ to be $X\langle i\rangle$. We can implement the following *shuffle* operations in constant time using scattered read and write. Given two word sequences A and X of length ℓ, a *shuffled read* computes the word sequence Y, such that $Y\langle i\rangle = X\langle A\langle i\rangle\rangle$. Given word sequences X and A of length ℓ, a *shuffled write* computes the word sequence Y, such that $Y\langle A\langle i\rangle\rangle = X\langle i\rangle$. Scattered memory accesses capture the memory model used in IBM's *Cell* architecture [13]. They also appear (e.g., `vpgatherdd`) in Intel's AVX vector extension [24]. Scattered memory access operations were also proposed by Larsen and Pagh [21] in the context of the I/O model of computation. Note that while the addresses for scattered writes must be distinct, we can read simultaneously from the same address.

3 Range Minimum Tree

Let A be an array of n w-bit integers and assume for simplicity that n is a power of two. The *range minimum tree* T is the perfectly balanced rooted binary tree over A such that the ith leaf corresponds to the ith entry in A. We associate each node v in T with a *weight*, denoted $\mathsf{weight}(v)$. If v is a leaf, the weight is the value represented by the corresponding entry of A, and if v is an internal node, the weight is the minimum of the weights of the descendant leaves. Note that T has height $h = O(\log n)$ and uses $O(n)$ space. See Figure 2.

For nodes $i, j \in T$, let $\mathsf{lca}(i, j)$ denote the lowest common ancestor of i and j. To answer an $\mathsf{rmq}(i, j)$ query, we traverse the path from i to $\mathsf{lca}(i, j)$ and from j to $\mathsf{lca}(i, j)$ and return the minimum weight of i, j, and of the nodes hanging off to the right and left of these paths (except for the children of $\mathsf{lca}(i, j)$), respectively (see Figure 2). To perform an $\mathsf{update}(i, \alpha)$, we traverse the path from i to the root. At leaf i, we set the weight to be α, and at each internal node v, we set the weight to be the minimum of the weights of the two children of v. If we do not modify the weight of a node at some node in the traversal, we may stop since no weights need to be updated on the remaining path. See Figure 3. Both operations traverse paths of $O(\log n)$ length and use constant time at each node. Hence, both operations use $O(\log n)$ time.

We introduce the following *node sequences* to implement the range minimum tree on the UWRAM efficiently. Let i be an index in A corresponding to the ith leaf in T, and let p be the path of nodes from i to the root in T. The *path sequence* is the sequence of nodes on the path p. Define the *left sequence* to be the sequence of nodes that are hanging off to the left of p, i.e., a node v is in the left sequence if it is the left child of a node on p and is not on p itself. Similarly, define the *right sequence* and the *off-path sequence* to be the sequence of nodes to the right of p and the sequence of nodes to the left or right of p, respectively. All node sequences are ordered from i to the root in order of increasing height. See Figures 2 and 3.

We can use the node sequences to describe the traversed nodes during the rmq and *update* operations on the range minimum tree. Consider an $\mathsf{rmq}(i, j)$ with $u = \mathsf{lca}(i, j)$ of depth d. Then $\mathsf{rmq}(i, j)$ is the minimum of the weights of the leaves i and j and of the nodes of depth $> d + 1$ on the right sequence of i and the left sequence of j. Next, consider an $\mathsf{update}(i, \alpha)$ and let $i^p_0, \ldots, i^p_{h-1}$ and $i^o_0, \ldots, i^o_{h-2}$ be the path and off-path sequence, respectively, for i. Let $\mathsf{weight}(\cdot)$ and $\mathsf{weight}'(\cdot)$ denote the weight of nodes in T before and after the update. Recall that only the nodes on the path sequence may change. We have that $\mathsf{weight}(i^p_0) = \alpha$ and $\mathsf{weight}'(i^p_j) = \min(\mathsf{weight}'(i^p_{j-1}, \mathsf{weight}(i^o_{j-1})))$ for $0 < j < h$. If we unfold the recursion, it follows that

$$\mathsf{weight}'(i^p_j) = \min(\mathsf{weight}(i^o_{j-1}), \ldots, \mathsf{weight}(i^o_0), \alpha) \qquad \text{for } 0 \leq j < h. \tag{1}$$

In other words, the new weights of the nodes on the path sequence are the prefix minimums of the sequence $\mathsf{weight}(i^o_{h-2}), \ldots, \mathsf{weight}(i^o_0), \alpha$.

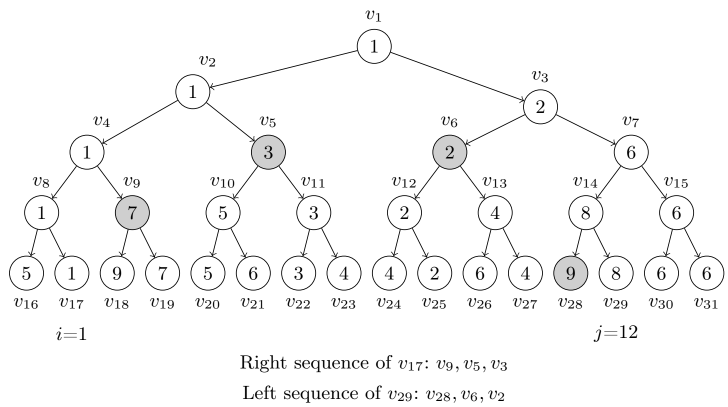

Fig. 2. An example array A, with its range minimum tree. For a query $\mathsf{rmq}(1, 12)$, we illustrate with grey circles the right sequence vertices of $i = 1$ and the left sequence vertices of $j = 12$ of depth greater than $d + 1 = 1$.

4 From Range Minimum Queries to Prefix Minimum on the UWRAM

In this section, we show that any UWRAM data structure that supports prefix minimum computations on word sequences of length $O(\log n)$ implies a UWRAM solution for the range minimum query problem.

Theorem 2. *Let A be an array of n w-bit integers, and let $t(\ell)$ be the time to compute* pmin *on a word sequence of length at most $\ell = O(w)$. Then, we can construct an $O(n)$ space data structure in $O(n)$ time on the UWRAM that supports* rmq *and* update *in $O(1 + t(\log n))$ time.*

In the next section, we will show how to compute pmin on a word sequence of length $\ell = O(w)$ in $O(\log \log \ell)$ time, implying the main result of Theorem 1.

4.1 Range Minimum Tree on the UWRAM

We first show a simple direct implementation of the range minimum tree that achieves the rmq and update time bounds of Theorem 2 but with $O(n \log n)$ space and preprocessing time.

Data Structure Our data structure consists of the input array A, the range minimum tree T over A, and a data structure that supports lowest common ancestor queries on T. This data structure can be implemented in linear space and preprocessing time to support lca queries in constant time [1,3,19]. Furthermore, for each index i in A, we store the path sequence, the left path sequence, the right path sequence, and the off-path sequence. The sequences are stored as sequences

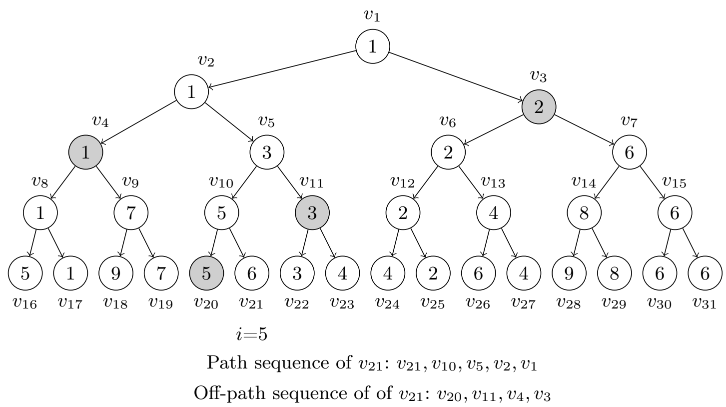

Fig. 3. Illustration of an $\mathsf{update}(5, \alpha)$ query. We draw with grey circles the off-path sequence vertices. Note how the value of v_{10} should be $\min(v_{20}, \alpha)$, the value of v_5 should be $\min(v_{10}, v_{11}) = \min(v_{11}, v_{20}, \alpha)$, and the value of v_2 should be $\min(v_4, v_5) = \min(v_4, v_{11}, v_{20}, \alpha)$, and so on. These values are the prefix minimum of the grey vertices and α.

of pointers to the nodes, and together with the left and right sequences, we also store the sequence of depths of the nodes in the sequence.

The array, the range minimum tree, and the lowest common ancestor data structure use $O(n)$ space. Each of the $O(n)$ sequences uses $O(\log n)$ space. In total, we use $O(n \log n)$ space and preprocessing time.

Range Minimum Queries To answer a $\mathsf{rmq}(i, j)$ query, compute $u = \mathsf{lca}(i, j)$ and the depth d of u. Read the left and right path sequences of i and j into word sequences denoted I^r and J^l, and their corresponding depth sequences into word sequences denoted DI^r and DJ^l. Then, construct masks MI^r and MJ^l containing 1s in the positions in DI^r and DJ^l that are greater than $d+1$ and use these to extract the prefixes $\hat{I^r}$ and $\hat{J^l}$ of I^r and J^l, respectively, that contains the nodes that have depth greater than $d + 1$. Finally, compute the weights $W\hat{I^r}$ and $W\hat{J^l}$ of the nodes in $\hat{I^r}$ and $\hat{J^l}$ using two scattered reads, and return $\mathsf{min}(W\hat{I^r} \cdot W\hat{J^l})$. The masks MI^r and MJ^l can be computed using a scattered read and a comparison operation. Thus, all operations take constant time, except min, which uses $O(t(\log n))$ time. In total, we use $O(1 + t(\log n))$ time.

Updates To implement $\mathsf{update}(i, \alpha)$, we first set $\mathsf{weight}(i) = \alpha$. We read the off-path sequence into a word sequence, denoted I^o, and then compute the corresponding sequence of weights WI^o using a scattered read. We then compute $P = \mathsf{pmin}(WI^o \cdot \alpha)$ and perform a scattered write with I^p and P. All operations use constant time, except pmin, which uses $O(t(\log n))$ time. In total, we use $O(1 + t(\log n))$.

In summary, the UWRAM range minimum tree uses $O(n \log n)$ space and preprocessing time and supports rmq and update in $O(1 + t(\log n))$ time.

4.2 Reducing Space

We now improve the space and preprocessing time to $O(n)$ by using a single level of indirection.

Data Structure We partition A into *blocks* $B_0, \ldots, B_{n/\log n - 1}$ each of $\log n$ consecutive entries. We store the minimum of each block in a *block array* B of length $n/\log n$ and construct the UWRAM range minimum data structure from Section 4.1 on B. This uses $O(n/\log(n/\log n)) = O(n)$ space and preprocessing time.

Range Minimum Queries To answer an $\mathsf{rmq}(i, j)$ query there are two cases:

Case 1: i and j are within the same block. Let B_k, where $k = \lfloor i/\log n \rfloor$ be the block containing i and j and compute the corresponding local indices $i' = i \mod \log n$ and $j' = j \mod \log n$ in B_k. We read B_k and return $\mathsf{min}(B_k\langle i' \rangle, \cdots B_k\langle j' \rangle)$.

Case 2: i and j are in different blocks. Let $B_l, B_{l+1}, \ldots, B_r$ be the blocks covering the range from i to j and let i' and j' be the local indices in B_l and B_r. We decompose the range into three parts. We compute the minimum in the leftmost block as $l_{\min} = \mathsf{min}(B_l\langle i' \rangle \cdots, B_l\langle \log n - 1 \rangle)$ and the rightmost block as $r_{\min} = \mathsf{min}(B_r\langle \log n - 1 \rangle \cdots B_r\langle j' \rangle)$. We then compute the minimum $m_{\min}$ of the middle blocks using the range minimum tree. Finally, we return $\mathsf{min}(l_{\min}, c_{\min}, r_{\min})$.

Updates Consider an $\mathsf{update}(i, \alpha)$ operation. Let B_k, where $k = \lfloor i/\log n \rfloor$ be the block containing i, and let $i' = i \mod \log n$ be the local index in B_k. We set $B_k\langle i' \rangle = \alpha$. We then read B_k and compute $b_{\min} = \mathsf{min}(B_k)$. If $b_{\min}$ differs, we update T with the new value.

Both operations use constant time except for (prefix) minimum computations and operations on the range minimum tree that take $O(1 + t(\log(n/\log n))) = O(t(\log n))$ time. Hence, the total time is $O(1 + t(\log n))$. In summary, we have shown Theorem 2.

5 Computing Prefix Minimum on Word Sequences

We now show how to efficiently compute the prefix minimum on word sequences of length $\ell = O(w)$ in $O(\log \log \ell)$ time. We first show how to do so in constant time for word sequences of length $O(\sqrt{w})$. We then show how to implement this algorithm in parallel and then recursively leading to the result.

Our algorithm often partitions a word sequence into multiple equal-length sequences to work on them in parallel. We define a b-way word sequence to be a

$$
\begin{array}{rl}
X = & \langle \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 9\ 2\ 5\ 3\ \rangle \\
\widehat{X} = & \langle\ 9\ 2\ 5\ 3 \quad 9\ 2\ 5\ 3 \quad 9\ 2\ 5\ 3 \quad 9\ 2\ 5\ 3\ \rangle \\
\widetilde{X} = & \langle\ 9\ 9\ 9\ 9 \quad 2\ 2\ 2\ 2 \quad 5\ 5\ 5\ 5 \quad 3\ 3\ 3\ 3\ \rangle \\
C = & \langle\ 1\ 0\ 0\ 0 \quad 1\ 1\ 1\ 1 \quad 1\ 0\ 1\ 0 \quad 1\ 0\ 1\ 1\ \rangle \\
D = & \langle\ 0\ 0\ 0\ 0 \quad 1\ 1\ 1\ 1 \quad 0\ 0\ 0\ 0 \quad 0\ 0\ 1\ 1\ \rangle \\
M = & \langle\ 1\ 0\ 0\ 0 \quad 1\ 1\ 0\ 0 \quad 1\ 1\ 1\ 0 \quad 1\ 1\ 1\ 1\ \rangle \\
E = & \langle\ 0\ 0\ 0\ 0 \quad 1\ 1\ 0\ 0 \quad 0\ 0\ 0\ 0 \quad 0\ 0\ 1\ 1\ \rangle \\
E' = & \langle\ 0\ 0\ 0\ 0 \quad 3\ 2\ 0\ 0 \quad 0\ 0\ 0\ 0 \quad 0\ 0\ 1\ 0\ \rangle \\
E'' = & \langle\ 19\ 18\ 17\ 16 \quad 0\ 0\ 13\ 12 \quad 11\ 10\ 9\ 8 \quad 7\ 6\ 0\ 0\ \rangle \\
P = & \langle\ 19\ 18\ 17\ 16 \quad 3\ 2\ 13\ 12 \quad 11\ 10\ 9\ 8 \quad 7\ 6\ 1\ 0\ \rangle \\
Y = & \langle \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 2\ 2\ 3\ 3\ \rangle
\end{array}
$$

Fig. 4. Our prefix minimum algorithm.

word sequence $X = X_{s-1} \cdots X_0$ where each subsequence X_i is a *block* of length b. Thus, the total length of X is sb. We use $X\langle i, j\rangle$ to denote entry j in block i, that is, $X\langle i, j\rangle = X_i\langle j\rangle$.

5.1 Prefix Minimum on Small Word Sequences

We now show how to compute the prefix minimum on a word sequence X of length $\ell = O(\sqrt{w})$. For simplicity, we first assume all entries in X are distinct. Our algorithm proceeds as follows. See also the example in Figure 4.

Step 1: Compare all pairs of words in X We construct a b-way word sequence C containing the results of all pairwise comparisons of words in X. To do so, we first construct the word sequences:

$$\widehat{X} = \underbrace{X \cdot X \cdots X}_{\ell}$$

$$\widetilde{X} = \underbrace{X\langle \ell-1\rangle \cdots X\langle \ell-1\rangle}_{\ell} \cdot \underbrace{X\langle \ell-2\rangle \cdots X\langle \ell-2\rangle}_{\ell} \cdots \underbrace{X\langle 0\rangle \cdots X\langle 0\rangle}_{\ell}$$

We compute these using shuffled read operations on X with the constant word sequences

$$\widehat{A} = \underbrace{\langle \ell-1, \cdots 0\rangle \cdots \langle \ell-1, \cdots 0\rangle}_{\ell}$$

$$\widetilde{A} = \underbrace{\langle \ell-1, \ldots \ell-1\rangle}_{\ell} \cdot \underbrace{\langle \ell-2, \ldots \ell-2\rangle}_{\ell} \cdots \underbrace{\langle 0, \ldots 0\rangle}_{\ell}.$$

Note that both $\widehat{X}$ and $\widetilde{X}$ have length $\ell^2 = O(w)$. We then do a componentwise comparison of $\widehat{X}$ and $\widetilde{X}$. This produces a word sequence C of length ℓ^2, which viewed as an ℓ-way word sequence is defined by:

$$C\langle i, j\rangle = \begin{cases} 1 & \text{if } X\langle i\rangle \leq X\langle j\rangle \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Thus, the ith block in C stores the comparison of $X\langle i\rangle$ with all other words in X.

Step 2: Compute Prefix Minima We construct an ℓ-way word sequence E that contains the positions of the prefix minima. First, compute the ℓ-way word sequence D such that

$$D\langle i,j\rangle = \begin{cases} 1 & \text{if } C\langle i,j\rangle = C\langle i,j-1\rangle = \cdots C\langle i,0\rangle = 1 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Thus, the block D_i in D takes the rightmost 0 in C_i and "smears" it to the left. In the full version, we show how to compute this operation, which we call *left clear*, in constant time [7]. We then mask out all entries in D where $i > j$ using an ℓ-way mask $M = M_{\ell-1}\cdots M_0$, where $M\langle i,j\rangle = 1$ if $i \leq j$ and 0 otherwise. By (2) we have that $D\langle i,j\rangle = 1$ iff $X\langle i\rangle \leq X\langle j\rangle, X\langle j-1\rangle, \ldots, X\langle 0\rangle$. Hence, $E\langle i,j\rangle = 1$ iff $X\langle i\rangle \leq X\langle j\rangle, X\langle j-1\rangle, \ldots, X\langle 0\rangle$ and $i \leq j$, i.e., $X\langle i\rangle$ is the prefix minimum of $X\langle j-1\rangle, \ldots, X\langle 0\rangle$.

Step 3: Extract Prefix Minimum We now extract the prefix minima entries indicated by D and then compact them into a word sequence of length ℓ. The key observation is that $E\langle i,j\rangle = 1$ implies that $X\langle i\rangle$ should appear in position j in the final prefix minimum. We use the following constant word sequences:

$$P' = \underbrace{\langle \ell-1,\cdots 0\rangle \cdots \langle \ell-1,\cdots 0\rangle}_{\ell}$$
$$P'' = \langle \ell^2+\ell-1, \ell^2+\ell-2, \ldots \ell\rangle$$

We do a component-wise extraction of the positions in P' wrt. E and of P'' wrt. $\overline{E}$, where $\overline{E}$ is the negation of E, and then | the resulting sequences E' and E'' together to get a word sequence P. We then do a shuffled write on $\widetilde{X}$ with P and clear out all but the ℓ rightmost entries. The resulting word sequence Y of length ℓ contains a $X\langle i\rangle$ in position j iff $E\langle i,j\rangle = 1$ as desired. Since we assumed all input words are distinct and since P'' only contains numbers greater than $\ell-1$, there is no write conflict.

All of the above word sequences have length at most $\ell^2 = O(w)$, and thus, each step takes constant time. Recall that we assumed all entries in X were distinct. If not, we may have a write conflict at the end of step 3. To fix this, we can always double the length of the input word sequence X and represent each entry using two words consisting of the entry and the position in the sequence, thus breaking ties. We can then simulate the above algorithms with constant factor slowdown. In summary, we have shown the following result.

Lemma 1. *Given a word sequence X of length $\ell = O(\sqrt{w})$, we can compute the prefix minimum of X in $O(1)$ time.*

As a corollary, we implement the above operation to compute prefix minima of blocks in a b-way word sequence X. More precisely, let $X = X_{s-1} \cdots X_0$ be a b-way word sequence and define the *b-way prefix minimum* of X to be

$$\mathsf{pmin}^b(X) = \mathsf{pmin}(X_{s-1}) \cdot \mathsf{pmin}(X_{s-2}) \cdots \mathsf{pmin}(X_0) \ .$$

To compute $\mathsf{pmin}^b(X)$, where X has length $\ell = O(w/b)$, consider each block as a word sequence of size b to which we apply the above algorithm in parallel as follows. In Step 1, we create two word sequences $\widehat{X}'$ and $\widetilde{X}'$ that are the concatenation of all the word sequences $\widehat{X}$ and $\widetilde{X}$ of size b^2 corresponding to each block, and we apply all the operations outlined above in parallel for each subsequence of b^2 words. Since the length of X is $\ell = O(w/b)$, the number of blocks is $O(w/b^2)$ and the length of the word sequences $\widehat{X}$ and $\widetilde{X}$ is $O(w)$. We proceed similarly for the other steps.

Since the length of the word sequences we work with is $O(w)$ we can do all the operations in constant time. Note that, if $b > \sqrt{w}$, the condition $\ell = O(w/b)$ implies that $\ell = O(\sqrt{w})$, and also that we have a constant number of blocks to process, which we can do with Lemma 1 instead. We have the following result:

Corollary 1. *Given a b-way word sequence X of length $O(w/b)$ we can compute the b-way prefix minimum of X in constant time.*

5.2 Prefix Minima on General Word Sequences

We now show how to recursively apply Corollary 1 to compute the prefix minima on a word sequence X of length $\ell = O(w)$. Given the b-way prefix minimum of X, we show how to compute the b^2-way prefix minimum of X in constant time. We then show how to apply this recursively to obtain our $O(\log\log \ell)$ algorithm. Let $X^b = X^b_{s-1} \cdots X^b_0$ be the b-way prefix minimum of X. Our algorithm proceeds as follows (see Figure 5).

Step 1: Compute Prefix Minima of Block Minima We compute the word sequence

$$B^b = \mathsf{pmin}^b(X^b_{s-1}\langle b-1\rangle \cdots X^b_0\langle b-1\rangle)$$

containing the prefix minimum of the minimum (leftmost) entries of the blocks in X^b. To do so, we first shuffle the leftmost entries of the blocks into a word sequence Y and then compute $B^b = \mathsf{pmin}^b(Y)$ using Corollary 1 (note that Y has length $O(w/b)$ as required).

Step 2: Propagate Prefix Minima of Block Minima. We construct a word sequence E that, for each block of X^b, has the minimum of the previous blocks in each sequence of b blocks. First, we compute the word sequences

$$\widehat{B}^b = \underbrace{B^b\langle s-1\rangle \cdots B^b\langle s-1\rangle}_{b} \cdots \underbrace{B\langle 0\rangle \cdots B\langle 0\rangle}_{b},$$

$$M = \underbrace{0 \cdots 0}_{b^2-b} \cdot \underbrace{1^w \cdots 1^w}_{b} \cdots \underbrace{0 \cdots 0}_{b^2-b} \cdot \underbrace{1^w \cdots 1^w}_{b},$$

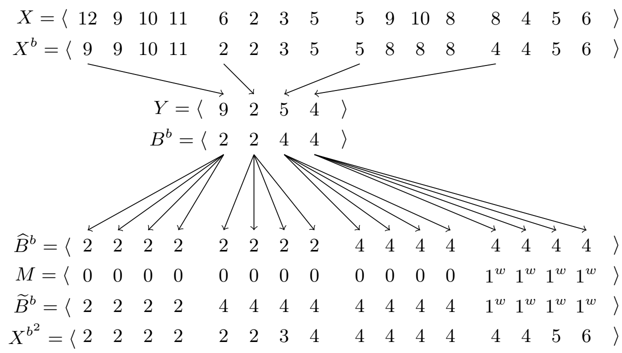

Fig. 5. Illustration of Step 2 of section 5.2: computing the 16-way prefix minimum of a word sequence X, given the 4-way prefix minimum of X in X^b.

where $\widehat{B}^b$ contains b copies of every entry in B^b, and M contains a repeated pattern of b words of 1s followed by $b^2 - b$ 0s. Both word sequences have length ℓ, and we compute them using shuffled read operations. After this, we shift left $\widehat{B}^b$ by b words, and we calculate the logical '$|$' of $\widehat{B}^b$ and M in a word sequence $\widetilde{B}^b$. Finally, we compute the componentwise minimum of $\widetilde{B}^b$ and X^b to produce X^{b^2}.

To see why this is correct, consider a sequence of b blocks of the b^2-way prefix minimum of X, denoted $X^{b^2}_{ib}, X^{b^2}_{ib+1}, \ldots, X^{b^2}_{ib+b-1}$ for any $0 \leq i < \lceil \ell/b^2 \rceil$. We can obtain the value of any $X^{b^2}_{ib+j}$ by the componentwise minimum of X^b_{ib+j}, and a block that contains b copies of the minimum of $X_{ib+j-1}, \ldots X_{ib}$. This minimum is computed in the $(j-1)$th word of B^b_i, so we copy that word b times in $\widetilde{B}^b$, and we shift it left by b words to compute the componentwise minimum with X^b_{ib+j}. Note that for $j = 0$, the value of $X^{b^2}_{ib}$ is the same as the value of X^b_{ib}, therefore for these blocks, we instead compare to the maximum word available (1^w) in order not to change their value. See Fig 5 for a visualization.

Each operation uses constant time, and hence we have the following result.

Lemma 2. *Let X be a word sequence of length $\ell = O(w/b)$. Given a b-way prefix minimum of X, we can compute a b^2-way prefix minimum of X in constant time.*

It now follows that we can compute the prefix minimum of a word of length ℓ by applying Lemma 2 for double exponentially increasing values of b over $O(\log \log \ell)$ rounds. Hence, we have the following result.

Theorem 3. *Given a word sequence X of length $\ell = O(w)$, we can compute the prefix minimum of X in $O(\log \log \log \ell)$ time.*

Plugging in Theorem 3 into our reduction of Theorem 2, we have shown the main result of Theorem 1.

References

1. Alstrup, S., Gavoille, C., Kaplan, H., Rauhe, T.: Nearest common ancestors: A survey and a new algorithm for a distributed environment. Theory Comput. Syst. **37**, 441–456 (2004)
2. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked ancestor problems. In: Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science. pp. 534–543 (1998)
3. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Proceedings of the 4th Latin American Symposium on Theoretical Informatics. pp. 88–94 (2000)
4. Berkman, O., Breslauer, D., Galil, Z., Schieber, B., Vishkin, U.: Highly parallelizable problems. In: Proc. 21st STOC. pp. 309–319 (1989)
5. Bille, P., Gørtz, I.L., Skjoldjensen, F.R.: Partial sums on the ultra-wide word RAM. Theor. Comput. Sci. **905**, 99–105 (2022)
6. Bille, P., Gørtz, I.L., Stordalen, T.: Predecessor on the ultra-wide word RAM. Algorithmica **86**(5), 1578–1599 (2024)
7. Bille, P., Gørtz, I.L., Stordalen, T., López, M.P.: Dynamic range minimum queries on the ultra-wide word RAM. arXiv 2411.16281 (2024)
8. Blelloch, G.E.: Prefix sums and their applications. In: Synthesis of Parallel Algorithms (1990)
9. Brodal, G.S., Chaudhuri, S., Radhakrishnan, J.: The randomized complexity of maintaining the minimum. In: Proc. 5th SWAT. pp. 4–15 (1996)
10. Brodal, G.S., Davoodi, P., Lewenstein, M., Raman, R., Satti, S.R.: Two dimensional range minimum queries and fibonacci lattices. Theoret. Comput. Sci. **638**, 33–43 (2016)
11. Brodal, G.S., Davoodi, P., Srinivasa Rao, S.: Path minima queries in dynamic weighted trees. In: Proc. 12th WADS. pp. 290–301 (2011)
12. Chazelle, B., Rosenberg, B.: Computing partial sums in multidimensional arrays. In: Proc. 5th SOCG. pp. 131–139 (1989)
13. Chen, T., Raghavan, R., Dale, J.N., Iwata, E.: Cell broadband engine architecture and its first implementation—a performance view. IBM J. Res. Dev. **51**(5), 559–572 (2007)
14. Demaine, E.D., Landau, G.M., Weimann, O.: On cartesian trees and range minimum queries. Algorithmica **68**, 610–625 (2014)
15. Farzan, A., López-Ortiz, A., Nicholson, P.K., Salinger, A.: Algorithms in the ultra-wide word model. In: Proc. 12th TAMC. pp. 335–346 (2015)
16. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. **40**(2), 465–492 (2011)
17. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proc. 16th STOC. pp. 135–143 (1984)
18. Hagerup, T.: Sorting and searching on the word ram. In: Proc. 15th STACS. pp. 366–398 (1998)
19. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. **13**(2), 338–355 (1984)
20. Ladner, R.E., Fischer, M.J.: Parallel prefix computation. J. ACM **27**(4), 831–838 (1980)
21. Larsen, K.G., Pagh, R.: I/o-efficient data structures for colored range and prefix reporting. In: Proc. 23rd SODA. pp. 583–592 (2012)
22. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A unified graphics and computing architecture. IEEE micro **28**(2), 39–55 (2008)

23. Pătraşcu, M., Demaine, E.D.: Logarithmic lower bounds in the cell-probe model. SIAM J. Comput. **35**(4), 932–963 (2006), announced at SODA 2004
24. Reinders, J.: AVX-512 instructions. Intel Corporation (2013)
25. Sleator, D., Tarjan, R.E.: A data structure for dynamic trees. J. Comput. Syst. Sci. **26**(3), 362–391 (1983)
26. Stephens, N., et al.: The ARM scalable vector extension. IEEE Micro **37**(2), 26–39 (2017)
27. Yao, A.C.: On the complexity of maintaining partial sums. SIAM J. Comput. **14**(2), 277–288 (1985)
28. Yuan, H., Atallah, M.J.: Data structures for range minimum queries in multidimensional arrays. In: Proc. 21st SODA. pp. 150–160 (2010)