The All Nearest Smaller Values Problem Revisited in Practice, Parallel and External Memory

Nodari Sitchinava
University of Hawaii at Mānoa
Honolulu, HI, USA
nodari@hawaii.edu

Rolf Svenning
Aarhus University
Aarhus, Denmark
rolfsvenning@cs.au.dk

ABSTRACT

We present a thorough investigation of the *All Nearest Smaller Values (ANSV)* problem from a practical perspective. The ANSV problem is defined as follows: given an array A consisting of n values, for each entry A_i compute the largest index $l < i$ and the smallest index $r > i$ such that $A_i > A_l$ and $A_i > A_r$, i.e., the indices of the nearest smaller values to the left and to the right of A_i. The ANSV problem was solved by Berkman, Schieber, and Vishkin [J. Algorithms, 1993] in the PRAM model. Their solution in the CREW PRAM model, which we will refer to as the *BSV* algorithm, achieves optimal $O(n)$ work and $O(\log n)$ span. Until now, the BSV algorithm has been perceived as too complicated for practical use, and we are not aware of any publicly available implementations. Instead, the best existing practical solution to the ANSV problem is the implementation by Shun and Zhao presented at DCC'13. They implemented a simpler $O(n \log n)$-work algorithm with an additional heuristic first proposed by Blelloch and Shun at ALENEX'11. We refer to this implementation as the *BSZ* algorithm. In this paper, we implement the original BSV algorithm and demonstrate its practical efficiency. Despite its perceived complexity, our results show that its performance is comparable to the BSZ algorithm. We also present the first theoretical analysis of the heuristic implemented in the BSZ algorithm and show that it provides a tunable trade-off between optimal work and optimal span. In particular, we show that it achieves $O\left(n\left(1+\frac{\log n}{k}\right)\right)$ work and $O\left(k(1+\log\frac{n}{k})\right)$ span, for any integer parameter $1 \leq k \leq n$. Thus, for $k = \Theta(\log n)$, the BSZ algorithm can be made to be work-optimal, albeit at the expense of increased span compared to BSV. Our discussion includes a detailed examination of different input types, particularly highlighting that for random inputs, the low expected distance between values and their nearest smaller values renders simple algorithms efficient. Finally, we analyze the input/output (I/O) complexities of the BSV algorithm.

CCS CONCEPTS

• **Theory of computation → Shared memory algorithms**.

Work supported by Independent Research Fund Denmark grant 9131-00113B and National Science Foundation grant CCF-1911245.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs International 4.0 License.

SPAA '24, June 17–21, 2024, Nantes, France
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0416-1/24/06.
https://doi.org/10.1145/3626183.3659979

KEYWORDS

Algorithm analysis, parallel algorithms, external memory, PRAM, algorithm engineering, all nearest smaller values problem, ANSV

ACM Reference Format:
Nodari Sitchinava and Rolf Svenning. 2024. The All Nearest Smaller Values Problem Revisited in Practice, Parallel and External Memory. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3626183.3659979

1 INTRODUCTION

1.1 The All Nearest Smaller Values problem

In the All Nearest Smaller Values (ANSV) problem, we are given an array A consisting of n totally ordered elements $A_1, A_2, A_3, \ldots, A_n$, referred to as the *values*. The objective is to compute for each value the indices of the nearest smaller values to its left and right. Specifically, for a given value A_i, find the index l of the nearest smaller value A_l on its left. This index is $l = \max\{j \mid A_j < A_i \land j < i\}$, where A_l is termed the *left match* of A_i. Similarly, the nearest smaller value on the right, or the *right match*, should also be computed. The output of the problem is two arrays, L and R, each of size n. These arrays store the indices of each element's left and right matches in A, respectively. For simplicity, we extend A such that $A_0 = A_{n+1} = -\infty$, and let the index 0 indicate that a value has no left match, and the index $n+1$ that a value has no right match. All entries of L are initially 0, and all entries of R are initially $n+1$.

As is standard, we assume without loss of generality that all values in A are distinct. The case with equal values can be handled by simple modifications of the input and running the algorithm twice, once for left matches and once for right matches. We adopt the same notation as in [6] such that $l(A_i)$ and $r(A_i)$ denote the indices of the left and right match of A_i, respectively.

ANSV is a fundamental problem since many problems directly reduce to an ANSV computation or ANSV can be used as an important subroutine. A non-exhaustive list of such problems is: finding the min/max among n elements, merging sorted lists, constructing Cartesian trees [23], monotone polygon triangulation, range minimum queries, parenthesis matching, binary tree reconstruction. See [4–6] for more details.

To highlight one example, consider merging two sorted lists, $a = a_1, a_2, a_3, \ldots, a_n$ and $b = b_1, b_2, b_3, \ldots, b_m$. This task directly reduces to computing the ANSV on $a \circ \text{rev}(b)$, where $\text{rev}(b)$ denotes the reverse of list b, and $\circ$ represents concatenation. Specifically, the position of each a_i and b_j in the merged list can be determined as $n + m - r(a_i) + i + 1$ and $l(b_j) + j + 1$, respectively, where $r(a_i)$ and $l(b_j)$ are the indices of the right and left matches for a_i and b_j in $a \circ \text{rev}(b)$.

1.2 Models of computation

For analyzing algorithms, we focus on the CREW PRAM [14, 15] and the external memory (EM) [1] models of computation and the only allowed operations on values are comparisons. The CREW PRAM model is characterized by multiple processors operating synchronously on shared memory with concurrent reads (CR) and exclusive writes (EW) capabilities. Algorithms in this model are analyzed in terms of *work* – the total number of operations performed by all processors, and *span* – the depth of the longest computation path using an infinite number of processors. We adopt the notation T_P to denote the time it takes to execute an algorithm on a P-processor CREW PRAM. Then work and span correspond to T_1 and T_∞, respectively. Analyzing just work and span is sufficient because Brent's scheduling principle [9] can then be used to obtain the runtime T_P for an arbitrary number of processors $P \geq 1$ as $T_P = O\left(\frac{T_1}{P} + T_\infty\right)$.

The EM model by Aggarwal and Vitter [1] (also known as the *ideal-cache model* [11][1]) is characterized by a processor with an internal memory of size M and an infinite external memory. Initially, the input of size n is placed in $\lceil \frac{n}{B} \rceil$ consecutive blocks of B consecutive elements in the external memory. Each processor can perform *input/output* (I/O) operations to move a block of elements between the external and internal memories and data must be in the internal memory to perform any computation on it. The cost in this model is the number of I/Os performed. If an algorithm does not use the parameters M and B in its description, it is referred to as being *cache-oblivious* [11].

Arge et al. [2] extended the EM model to the parallel setting. The *Parallel External Memory (PEM)* [2] model consists of P processors, each containing internal memory of size M and sharing the external memory. The I/Os are performed in parallel, with a *parallel I/O* consisting of up to P processors transferring one block in each time step. Since no equivalent to Brent's scheduling principle exists in the PEM model, analysis of parallel I/Os must be performed for a specific value of P.

1.3 Previous results

Sequentially, the ANSV problem can be solved in linear time and $O\left(\frac{n}{B}\right)$ I/Os cache-obliviously using a stack to push elements from left to right. Before each element A_i is pushed on the stack, pop all elements larger than A_i from the stack. The remaining element at the top of the stack is smaller than A_i and is the left match of A_i. Right matches can be found similarly. This algorithm behaves exactly like the stack-based algorithm in [12] for constructing Cartesian trees. Instead of a stack, a simple implementation using arrays can also be employed [3].

We call the stack/array-based algorithm *SEQ*. These approaches are based on the following basic observation.

[1]The main difference between the EM and the ideal-cache models is that in the ideal-cache model the transfers between the internal and external memories are delegated to a separate omniscient paging algorithm. Therefore, the paging algorithm can optimize the I/Os based on its knowledge of future accesses and performs no worse than any explicitly stated block transfer algorithm in the EM model. The well-known resource-augmentation result [11] states that any reasonable automated paging algorithm, e.g., the one that evicts only the least-recently-used (LRU) block from the internal memory, with internal memory size of $2M$ performs asymptotically similarly to the omniscient paging algorithm with internal memory of size of M.

Observation 1. *The matches in the ANSV problem are non-overlapping. That is, for any value A_i with a right match r_i, there is no other value A_j for $j < i$ with a right match r_j, such that $i < r_j < r_i$ (likewise for the left match).*

In the parallel setting, Berkman, Schieber, and Vishkin (BSV) [6] presented an optimal $O(n)$ work and $O(\log \log n)$ span algorithm in CRCW PRAM and an optimal $O(n)$ work and $O(\log n)$ span algorithm in CREW PRAM. The latter is our focus and we call this the *BSV* algorithm. In the literature, their work-efficient algorithm is perceived as being *"very complicated"* [8, 19]. They also presented a much simpler $O(n \log n)$ work and $O(\log n)$ span algorithm which we call the *work-inefficient BSV* algorithm. The algorithm proceeds in two stages, first, it constructs in $O(n)$ work and $O(\log n)$ span (see [6] section 3.2) a balanced binary tree with the values A stored in its leaves in the original order. Each internal node then takes the value of the minimum of its children. We call this a *min binary tree*. Second, to find the left match of A_i it follows the path towards the root until a left child has a value smaller than A_i. From that child, it follows a path towards the leaves always choosing the right child if its value is smaller than A_i. It is straightforward to see that this finds the left match $l(A_i)$ in $O(\log n)$ time and symmetrically the same for right matches. This algorithm was implemented for parallel Lempel-Ziv factorization by Shun and Zhao [20] and Cartesian tree construction by Blelloch and Shun [8, 19] where ANSV was used as a subroutine. Interestingly, Blelloch and Shun added a surprisingly effective heuristic, and in [8] they write:

> *"... we note that the ANSV only takes about 10% of the total time even though it is an $O(n \log n)$ algorithm. This is likely due to the optimization discussed above."*

In their paper, the role of the heuristic was not emphasized as a critical element, and its operational details were somewhat unclear to us.

However, this heuristic is implemented in the publicly available code [21], and we provide an in-depth explanation of it in Section 1.4. In this paper, we analyze the heuristic and show that it results in a provably better work than $O(n \log n)$ and provides a tunable trade-off between work and span.

Generally, the BSV algorithm generalizes well to parallel models other than PRAM and has been adapted in various other models. For example, it has been used to solve ANSV in the bulk synchronous parallel model [13] and a formal derivation using Coq [18]. It has been implemented in the Distributed Memory models, both in theory [16] and in practice using MPI [10]. In the hypercube model, the ANSV was solved in optimal $O(\log n)$ time with n processors [17].

1.4 The BSZ algorithm

In this section, we describe the BSZ algorithm and the heuristic in detail. We begin with the heuristic which is based on an integer parameter $1 \leq k \leq n$ and modifies the simple $O(n \log n)$ work-inefficient algorithm [6] in three ways. :

- **H1** Partition the input into $\lceil \frac{n}{k} \rceil$ blocks of size k (except the last block which may not be full). For each block run the sequential ANSV algorithm to find all matches within the block, we call these *local* matches.
- **H2** When using the min-binary tree to search for a match, perform an exponential search in the direction of the match,

to find it in $O(\log d)$ time, where d is the distance in the input to the match. To do this, in addition to following parent pointers also move horizontally in the tree. In the code, they facilitate this by implementing the binary tree as an array of arrays, where the secondary arrays store the nodes at a given height.

H3 For each block, the elements without a right match form an increasing sequence (Observations 1a and 1b in [6]). Instead of finding the remaining right matches in parallel, do so sequentially from right to left. When performing an exponential search for the right match, start from where the previous search ended. Symmetrically for the left matches.

In short and focusing on the right matches, computing the local matches in each block leaves an increasing sequence of $1 \leq d \leq k$ unmatched values at indices $u_1, u_2, u_3, ..., u_d$. Thus, proceeding from right to left, for each pair of adjacent unmatched values $A_{u_{i-1}}$ and A_{u_i}, the search in the min binary tree for the match of $A_{u_{i-1}}$ can start from where A_{u_i} found its match.

The BSZ algorithm works on a global array A of n values and stores the right matches in an array R of n matches, all initialized to $n+1$ (we omit the left matches for simplicity). It is supplied with the hyperparameter k and has access to a min binary tree T on A, which can be built in $O(n)$ work and $O(\log n)$ span. We adopt the Python notation for subarrays where $A[x : y]$ denotes $[A_i \mid x \leq i < y]$ and the notation $A^i = A[(i-1)k + 1 : \min(ik\,,\ n) + 1]$ for the ith block of A. The BSZ algorithm is described in pseudocode as Algorithm 1.

Algorithm 1: The BSZ algorithm for ANSV

```
Output: Computes the right matches of A and stores them
        in R
1 for i = 1 to ⌈n/k⌉ in parallel do
2   Compute local matches in A^i using the SEQ algorithm
      and store them in R^i
3   start ⟵ min(ik, n) + 1
4   for j = min(ik, n) down to (i − 1)k + 1 do
5     if R_j == n + 1 then
6       R_j ⟵ matchRight(start, j) // traverses T
          from index start for the match of A_j
7       start ⟵ R_j
```

1.5 The BSV algorithm

This section presents an overview of the BSV algorithm as described in [6]. The algorithm is described in pseudocode as Algorithm 2. Like the BSZ algorithm, the BSV algorithm operates on global arrays A and R, each of size n, representing values and their right matches, respectively. For simplicity, we omit left matches. The algorithm uses a min binary tree T based on A and is supplied with a hyperparameter k. Initially, the algorithm computes the local matches within each block A^i. It also determines the index i_m of the smallest value A_{i_m} in each block, along with its left and right matches $l(A_{i_m})$ and $r(A_{i_m})$, respectively. This latter is done by traversing the min binary tree T. The results are stored in array M. While using M is not mandatory, it prevents redundant searches in the tree. Then, for each block, the algorithm identifies the boundaries of a merging problem in constant time based on the contents of M (refer to Lemmas 3.3 and 3.4 in [6] for details). For $1 \leq a < b < c < d \leq n$ the two subsequences $A[a : b]$ and $A[c : d]$ that are merged may be far apart. Through this merging process, the algorithm identifies all the right matches for $A[a : b]$ and the left matches for $A[c : d]$. Note that each block defines at most two merging problems, leading to a total of $O\left(\frac{n}{k}\right)$ merging problems. By setting $k = \Theta(\log n)$, traversing the tree a constant number of times for each block results in $\Theta(n)$ work. Similarly, computing local matches within a group or performing a merge can be accomplished in $\Theta(k) = \Theta(\log n)$ time.

Algorithm 2: The BSV algorithm for ANSV

```
Output: Computes the right matches of A and stores them
        in R
1 M ⟵ Array of size ⌈n/k⌉ // For storing information
    to identify merging problems
2 for i = 1 to ⌈n/k⌉ in parallel do
3   Compute local matches in A^i using the SEQ algorithm
      and store them in R^i
4   i_m ⟵ index of min value in A^i
5   M[i] ⟵ {i_m, l(A_{i_m}), r(A_{i_m})} // Uses T
6 for i = 1 to ⌈n/k⌉ in parallel do
7   (a, b, c, d) ⟵ mergeBoundaries(i) // Computes
      boundaries of a merging problem. Uses M
      instead of searching in the tree T
8   merge(a, b, c, d) // Computes nonlocal matches by
      merging A[a : b] and A[c : d]
```

2 OUR RESULTS

We give the first theoretical analysis of the heuristic used in the BSZ algorithm which improves on the simple $O(n \log n)$ work and $O(\log n)$ span algorithm in [6]. We show that it provides a tunable trade-off between optimal work and optimal span for the hyperparameter $1 \leq k \leq n$. In particular, we show that it achieves $O\left(n\left(1 + \frac{\log n}{k}\right)\right)$ work and $O\left(k(1 + \log \frac{n}{k})\right)$ span, for any integer $1 \leq k \leq n$. Note that setting $k = 1$ corresponds to the work-inefficient BSV algorithm; setting $k = \Theta(\log n)$ achieves linear work, matching the work of the BSV algorithm, but resulting in the $O\left(\log n \log \frac{n}{k}\right)$ span; and setting $k = n$ corresponds to the SEQ algorithm.

Second, we present the first implementation of the BSV algorithm for shared memory machines, to our knowledge. Although the BSV algorithm has been perceived as being theoretically complicated, our implementation is a simple ~175 line C++ implementation, with ~90 lines reused directly from the publicly available ~120 line implementation of the BSZ algorithm from [20]. Our implementation is comparable with the current state-of-the-art BSZ implementation and achieves parallel speedup of up to 13.7 on 24 cores (48 threads with hyper-threading). We also verify experimentally that the heuristic introduced in the BSZ algorithm significantly speeds up the algorithm and reduces the work to $O(n)$ for large enough k.

Third, we show that when each value is drawn i.i.d. from a discrete distribution on a totally ordered set of size m the expected distance from a value to its match is at most H_m – the mth Harmonic number. Thus, these random inputs are not hard instances for this problem, as even the trivial $O(n^2)$ solution for ANSV is expected to achieve $O(n \log n)$ work and $O(\log n)$ span if $m = O(poly(n))$. Similarly, the work-inefficient BSV algorithm achieves $O(n \log \log n)$ work and $O(\log \log n)$ span.

Finally, we present the first I/O complexity analysis of the BSV algorithm in the (P)EM model. As with many other parallel models, the BSV algorithm generalizes well in the PEM model too and we show that simple modifications to the BSV algorithm yield $O(\frac{n}{PB} + \log_B n)$ parallel I/Os for any positive integer $P \geq 1$ of processors. Finally, we show that the array-based version like the stack-based version of the SEQ algorithm uses $O(n/B)$ I/Os cache-obliviously.

3 ANALYZING THE BSZ ALGORITHM

In this section, we analyze the heuristic introduced in the BSZ algorithm and show that it provides a tunable trade-off between work and span as described by the following Theorem.

THEOREM 1. *For an input of size n and any integer hyperparameter $1 \leq k \leq n$, the BSZ algorithm achieves $O\left(n\left(1 + \frac{\log n}{k}\right)\right)$ work and $O\left(k(1 + \log \frac{n}{k})\right)$ span.*

We use the terminology that when a value finds its match in the same block as itself we call it a *local* match. Likewise, when a value finds its match in a different block than itself we call it a *nonlocal* match. We begin by analyzing the span.

LEMMA 1. *For an input of size n and any integer hyperparameter $1 \leq k \leq n$, the span of the BSZ algorithm is $O\left(k(1 + \log \frac{n}{k})\right)$.*

PROOF OF LEMMA 1. The algorithm has three parts. First, constructing the min binary tree takes $O(\log n)$ time. Second, finding the local matches in a block takes $O(k)$ time. Third, using the heuristic to sequentially find the nonlocal left matches in a block takes $O\left(\sum_{i=1}^{k}(1 + \log n_i)\right)$ time where n_i denotes the distance between the $(i-1)$th and ith match and the plus one accounts for any case $n_i \leq 1$.[2] The sum is then upper bounded as follows: $\sum_{i=1}^{k}(1 + \log n_i) \leq k + \sum_{i=1}^{k} \log \frac{n-k}{k} \leq k\left(1 + \log \frac{n}{k}\right)$. Adding all parts together gives $O\left(k\left(1 + \log \frac{n}{k}\right) + \log n\right) = O\left(k(1 + \log \frac{n}{k})\right)$ span. □

Next, we will prove the following lemma, which bounds the work W_{BSZ} of the BSZ algorithm:

LEMMA 2. *Let W_{BSZ} be the work of the BSZ algorithm on an input of size n using the hyperparameter k. Then $W_{BSZ} = O\left(n\left(1 + \frac{\log n}{k}\right)\right)$.*

To prove Lemma 2, we will focus on a slightly different recursive algorithm, which we call *REC*, for the ANSV problem. We stress that this algorithm is only used for the analysis of the work of the BSZ algorithm. The idea is that this algorithm is simpler to analyze and uses about the same work as the BSZ algorithm. Like the BSZ algorithm, the REC algorithm operates on global arrays A and R of size n, with R being initialized to $n + 1$, storing the values and right matches, respectively (left matches are omitted for simplicity). It also uses a min binary tree T on A, which can be built using $O(n)$ work, and is supplied a hyperparameter k. The REC algorithm is described in pseudocode as Algorithm 3, and its behavior on a specific input is exemplified in Figure 1.

Algorithm 3: The REC(x, y) algorithm for ANSV

Input: Indices $x \leq y$
Output: Computes the right matches of values $A[x : y]$ and stores them in $R[x : y]$

```
1  if x == y then return ;
2  x_k ⟵ min(x + k , y)
3  Compute local matches in A[x : x_k] using the SEQ
     algorithm and store them in R[x : x_k]
4  start ⟵ x_k
5  for i = x_k − 1 down to x do
6      if R_i == n + 1 then
7          R_i ⟵ matchRight(start, i) // traverses T from
             index start for the match R_i of A_i
8          REC(start, R_i)
9          start ⟵ R_i
10 REC(start, y)
```

To solve the ANSV problem the initial call is REC$(1, n + 1)$. The REC algorithm, guided by the heuristics, identifies disjoint parts of A that can be solved independently. To do so, for the first block of k values, it runs the SEQ algorithm. Among these, some will find their match locally within the block. It uses the min binary tree T from right to left for the remaining matches using the same heuristics (**H2** and **H3**) as the BSZ algorithm. The indices of these nonlocal matches partition the remaining $n - k$ values into at most $k + 1$ disjoint subproblems which can be solved independently. The following Lemma makes that precise.

LEMMA 3. *For any call to REC(x, y), the right matches of all values in $A[x : y]$ are in $A[x : y + 1]$.*

PROOF. Follows directly from Observation 1 that all matches are non-overlapping. □

To bound the work of the BSZ algorithm using the REC algorithm, we will first prove Lemma 4, which established that they use about the same work.

LEMMA 4. *Let W_{BSZ} and W_{REC} be the work of the BSZ and REC algorithms for a particular input of size n using the same value for the hyperparameter k. Then $W_{BSZ} = O\left(W_{REC} + n\left(1 + \frac{\log n}{k}\right)\right)$.*

The converse (swapping W_{BSZ} and W_{REC}) also holds, but this direction is not required for our analysis.

Next, we will bound the work of the REC algorithm. For clarity of exposition, let $W_{REC} = O(T(n))$, i.e., denote an upper bound on the work of the recursive algorithm on an input of size n, excluding

[2] We adopt the convention that $\log 0 = 0$.

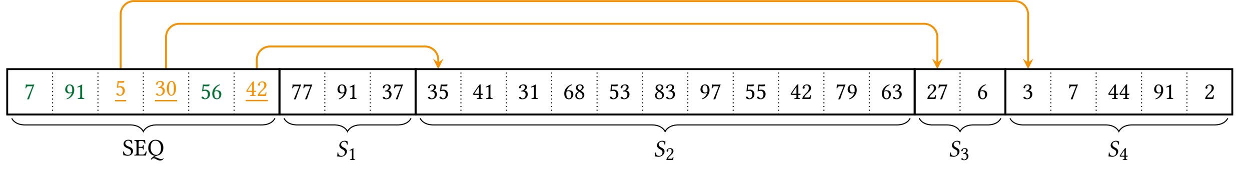


Figure 1: The REC algorithm begins by running the SEQ algorithm on the first $k = 6$ values (line 3), finding local matches for $7, 91$, and 56. The other three values, 5, 30, and 42, have nonlocal matches, indicated by arrows, which are found using the min binary tree T. These values partition the remaining input into four independent subproblems: S_1, S_2, S_3, and S_4. For instance, the recursive call for S_1 is REC$(7, 10)$ (line 8). For the last subproblem S_4, the recursive call is REC$(23, 28)$ (line 10).

the construction of the min binary tree. Then, $T(n)$ is defined by the following recursion:

$$T(n) = \begin{cases} k + \max\left(\sum_{i=1}^{k+1} T(n_i) + \sum_{i=1}^{k} \log(n_i)\right) & n > k \\ n & n \leq k, \end{cases}$$

where the maximum is defined over all possible partitions of the input into $k+1$ subproblems, each of size n_i, such that $\sum_{i=1}^{k+1} n_i = n - k$ (because the k values that define the partitions are already matched), and the sum of logarithmic terms comes from performing the non-local searches in the min tree using heuristic **H2**. Lemma 5 bounds $T(n)$ and demonstrates that $T(n)$ decreases as the block size k increases:

Lemma 5. $T(n) = O\left(n\left(1 + \frac{\log n}{k}\right)\right)$.

Together, Lemmas 4 and 5 will imply the bound on the work of the BSZ algorithm as stated in Lemma 2.

We first prove Lemma 4, which shows that the work of the BSZ and REC algorithms is about the same. We fix an input A of size n and focus solely on the right matches, as the behavior of the left matches is symmetric. The analysis begins by splitting the work of the BSZ and REC algorithms into two parts. The first part is constructing the binary tree and computing local matches in each block. The second part is finding the nonlocal match of each element. Thus, $W_{BSZ} = \Theta(n + \sum_{i=1}^{n} z_i)$ and $W_{REC} = \Theta(n + \sum_{i=1}^{n} r_i)$, where z_i and r_i denote the number of nodes visited in the min binary tree when finding the ith nonlocal right match of value A_i by the BSZ and REC algorithms, respectively. The count is 0 if the match is found locally or the search starts from the index of the match, and it is at least 1 otherwise. At a high level, our strategy will be to bound the difference between $\sum_{i=1}^{n} z_i$ and $\sum_{i=1}^{n} r_i$ for each block of size k. More precisely, for the BSZ algorithm, consider blocks of indices $Z = \left[Z^1, Z^2, Z^3, ..., Z^{\lceil n/k \rceil}\right]$, each corresponding to indices where local matches are computed on line 2. That is, $Z^i = [j \mid (i-1)k + 1 \leq j < \min(ik\,,\, n) + 1]$, noting that all blocks are of size k, except possibly the last one. For the REC algorithm, consider $m < n$ blocks of indices $R = \left[R^1, R^2, R^3, ..., R^m\right]$, where each R^i corresponds to indices where local matches are computed on line 3. Specifically, if a recursive call REC(x, y) computes local matches in $A[x : x_k]$ on line 3, it yields a block of indices $[j \mid x \leq j < x_k]$. Each block R^i has a maximum size of k.

Proof of Lemma 4. We begin by showing that $\left|\sum_{j \in Z^i} z_j - r_j\right| = O(k + \log n)$ for any $1 \leq i \leq \lceil n/k \rceil$. Consider running the BSZ algorithm on a fixed input resulting in Z^i. Also consider running the REC algorithm on the same input and focus on the subset of $c \leq k$ blocks $\left[R^{i_1}, R^{i_2}, R^{i_3}, ..., R^{i_c}\right] = \left[R^\ell \mid Z^i \cap R^\ell \neq \emptyset \land 1 \leq \ell \leq m\right]$ that overlap with Z^i.

Case 1 ($c = 1$).

Since a block from the REC algorithm perfectly overlaps with Z^i, exactly the same nodes in the min binary tree (with repetition) are visited, resulting in $\sum_{j \in Z^i} z_j - r_j = 0$.

Case 2. *[$c = 2$]*

Here the blocks R^{i_1} and R^{i_2} split Z^i into two parts. Consequently, they split the $1 \leq d \leq k$ unmatched values at indices $u_1, u_2, u_3, ..., u_d$ for the BSV algorithm in Z^i. Let $\bar{s}$ be the last index of R^{i_1} where the split occurs. Consider the most general case when the split is strictly between two unmatched values at indices $u_s < \bar{s} < u_{s+1}$ for $1 \leq s \leq d - 1$. See Figure 2 for an example. The cases where the split occurs before $s < u_1$, after $u_d < s$, or overlaps with some u_* are simpler.

In the part of Z^i to the left of the split, i.e., in $Z^i \cap R^{i_1}$, the nonlocal matches for the BSZ algorithm are also nonlocal for the REC algorithm. Now, there are only two places where the running time between the two algorithms may differ. First, since r_{u_s} does not start its search in the min binary tree from where $r_{u_{s+1}}$ found its match, it follows that $0 \leq r_{u_s} - z_{u_s} = O(\log n)$. Second, all values in $[u_s + 1, \bar{s}]$ are matched locally for the BSZ algorithm but some of them may be unmatched for the REC algorithm. We denote these unmatched values by R_+ and establish that $0 \leq \sum_{j \in R_+} r_j - z_j = \sum_{j \in R_+} r_j = O\left(|R_+| \log \frac{k}{|R_+|}\right) = O(k)$, using the concavity of the logarithm and that $\sum_{j \in R_+} r_j$ actually corresponds to $\Theta(\sum_i \log n_i)$, where $\sum_i n_i \leq k$.

Next, we consider the part of Z^i after the split, i.e., $Z^i \cap R^{i_2}$. If r_{u_d} is not the last nonlocal match in R^{i_2}, then it is the only place where a different number of nodes of the min binary tree are visited, and the difference is $0 \leq z_{u_d} - r_{u_d} = O(\log n)$. If r_{u_d} is the last nonlocal match in R^{i_2}, then exactly the same nodes are visited. Finally, there may be multiple unmatched values for the BSZ algorithm that are matched locally for the REC algorithm starting with r_{u_d}. We denote these local matches by R_-. As previously, we establish that $0 \leq \sum_{j \in R_-} z_j - r_j = \sum_{j \in R_-} z_j = O\left(|R_-| \log \frac{k}{|R_-|}\right) = O(k)$. For the last nonlocal match at d_r, if it exists, the difference is $0 \leq r_{d_r} - z_{d_r} = O(\log n)$. Combining all the contributions results in $\left|\sum_{j \in Z^i} z_j - r_j\right| = O(k + \log n)$, concluding this case.

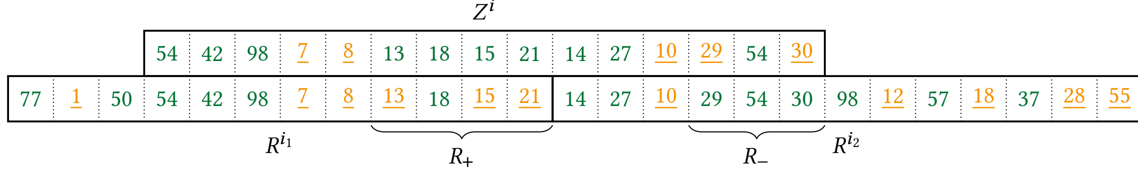


Figure 2: Case 2 in the proof of Lemma 4, where exactly two blocks, R^{i_1} and R^{i_2}, overlap with the block Z^i. The numbers that are underlined are unmatched in their respective blocks. For example, the $d = 5$ unmatched values in Z^i are $7, 8, 10, 29,$ and 30, and the split $\bar{s}$ occurs strictly between the unmatched values 8 and 10 at indices $u_2 = u_s$ and $u_3 = u_{s+1}$, respectively. The values that are unmatched in R^{i_1} but matched in Z^i, are $13, 15,$ and 21, corresponding to R_+. The values that are matched in R^{i_2} but unmatched in Z^i, are 29 and 30, corresponding to R_-. The braces indicate the range where values in R_+ or R_- are located.

Case 3 ($3 \le c$).

Consider any block R^{i_t} for $2 \le t \le c-1$ and the corresponding recursive call REC(x, y) with $y \le n$, which computed local matches at R^{i_t}. Since $R^{i_t} \subsetneq Z^i$ then $\left|R^{i_t}\right| = y - x < k$, and all nonlocal matches in R^{i_t} match A_y, by Lemma 3. For the REC algorithm, since the search starts from index y (line 4), no nodes are visited in the min binary tree. For the BSZ algorithm, since $y \in Z^i$, then all the values at R^{i_t} are matched locally, and no nodes of min binary tree are visited. Thus, the running times may differ only at R^{i_1} and R^{i_c}, which is similar to the case for $c = 2$.

We have now established that $\left|\sum_{j \in Z^i} z_j - r_j\right| = O(k + \log n)$ for any $1 \le i \le \lceil n/k \rceil$. Summing over each block Z^i concludes the proof:

$$
\begin{aligned}
W_{BSZ} &= \Theta\left(n + \sum_{i=1}^{n} z_i\right) \\
&= \Theta(n) + O\left(\sum_{i=1}^{\lceil n/k \rceil} \sum_{j \in Z^i} r_j + k + \log n\right) \\
&= \Theta(n) + O\left(\left(\sum_{i=1}^{n} r_i\right) + \lceil n/k \rceil \,(k + \log n)\right) \\
&= O\left(W_{REC} + n\left(1 + \frac{\log n}{k}\right)\right)
\end{aligned}
$$

□

Next, we prove Lemma 5, which shows that the work to find nonlocal matches decreases as k increases.

Proof of Lemma 5. We will prove that $T(n) \le 2n + \left(\frac{n}{k} - 1\right) \log n$ by induction. In the base case, when $n \le k$, $T(n) = n \le 2n + \left(\frac{n}{k} - 1\right) \log n$. For the inductive case:

$$
\begin{aligned}
T(n) &= k + \max\left(\sum_{i=1}^{k+1} T(n_i) + \sum_{i=1}^{k} \log n_i\right) \\
&\le k + \max\left(\sum_{i=1}^{k+1} \left(2n_i + \left(\frac{n_i}{k} - 1\right) \log n_i\right) + \sum_{i=1}^{k} \log n_i\right) \\
&\le k + 2(n-k) + \frac{1}{k} \cdot \max\left(\sum_{i=1}^{k+1} n_i \log n_i\right) \\
&\le 2n - k + \frac{1}{k}\left(\sum_{i=1}^{k+1} n_i\right) \log \sum_{i=1}^{k+1} n_i \\
&< 2n + \left(\frac{n}{k} - 1\right) \log n
\end{aligned}
$$

□

Using Lemmas 4 and 5, it is now straightforward to bound the work of the BSZ algorithm, which concludes the proofs of Lemma 2 and Theorem 3.

4 RANDOM INPUTS

Consider a random input where each input value is drawn independently and identically distributed from a discrete distribution over a totally ordered set of size m. Then, the expected distance between a value and its match (here the first smaller or equal value) is strictly less than $\sum_{i=1}^{m} p_i \frac{1}{\sum_{j=1}^{i} p_j}$. The strictness follows since the array is bounded. For example, with a uniform distribution, the expected distance is strictly less than H_m. Thus, for $m = O(poly(n))$ the expected distance is $\Theta(\log n)$. The arguably simplest ANSV algorithm is a double for-loop that scans left and right for the match of each value in parallel. For the uniform distribution with m as discussed earlier, this algorithm achieves expected $O(n \log n)$ work. Similarly, the work-inefficient algorithm with heuristic **H2** spends $O(\log d)$ time on finding the match of a value that is at a distance of d from its match. Thus, using Jensen's Inequality, it achieves expected $O(n \log \log n)$ work. For this reason, we consider random inputs to be easy.

5 EXTERNAL MEMORY

In this section, we prove Theorems 2 and 3, which give the I/O complexity of the BSV (Algorithm 2) and array-based SEQ algorithms in the PEM model.

THEOREM 2. *For any $P \geq 1$, the ANSV problem can be solved on the P-processor PEM model on an input of size n in $\Theta\left(\frac{n}{PB} + \log_B n\right)$ parallel I/Os.*

PROOF. Set group sizes to $k = \Theta(B \log_B n)$ and replace the binary tree with a B-tree. Then run the same BSV algorithm in $\lceil \frac{n}{kP} \rceil$ rounds, each round processing a contiguous segment of kP elements.

The straightforward bottom-up parallel construction of the B-tree takes $O\left(\frac{n}{PB} + \log_B n\right)$ parallel I/Os. In each round, merging is done sequentially by each processor, with each processor spending $O\left(\frac{k}{B}\right) = O(\log_B n)$ I/Os per round. Finally, in each round, each processor traverses the B-tree once, resulting in $\Theta(\log_B n)$ parallel I/Os per round. Combining the I/Os over the $\lceil \frac{n}{kP} \rceil$ rounds results in overall $O\left(\frac{n}{PB} + \log_B n\right)$ parallel I/O complexity. □

THEOREM 3. *The array-based SEQ algorithm is cache-oblivious, and for an input of size n, it uses $O\left(\frac{n}{B}\right)$ I/Os.*

PROOF. The SEQ algorithm sequentially finds the left (similarly right) matches by looping over A from left to right. It maintains the invariant that in the ith iteration, all left matches of values $A[1:i]$ have been found and are stored in the array $L[1:i]$. To find the left match of A_i, the algorithm simply follows the matches previously found in L, starting with L_{i-1}, until it finds a value $A_j < A_i$, and then sets $L[i] = j$. Given that the matches are non-overlapping (as noted in Observation 1), this result is not too surprising.

The amortized analysis of the I/O complexity is the same as for the number of comparisons in the RAM model [12], but with a potential of one I/O for each block instead of the individual elements. In particular, in the ith iteration, define the potential to be the number of blocks currently hit by the path generated by following the pointers starting from $L[i-1]$. Without going through all the cases, when an insertion (that is not the first in a block and itself extends the path) causes $3 \leq k$ blocks to be visited on the path, then the new path will hit $k-2$ fewer blocks, and the potential can pay for the visited blocks. □

6 EXPERIMENTS

In this section, we investigate the performance of the BSZ and BSV algorithms in practice. We show that even though the BSV algorithm has been perceived as theoretically complicated, the code is simple, and it achieves comparable performance to the current state-of-the-art implementation. Our code is available online [22]. We also confirm experimentally that the heuristic introduced for the BSZ algorithm is effective, and it significantly speeds up the algorithm and reduces the work to be linear for a large enough k.

6.1 Experimental setup

The experiments were run on two Intel Xeon Silver 4214 2.20GHz 12-core CPUs distributed across 2 sockets with hyper-threading enabled, totaling 48 threads and 126GB of shared RAM. The cache configuration included a 32K L1 cache, a 1024K L2 cache, and a 16896K L3 cache. Our implementation is in C++ 17 and compiled using GCC 7.5.0 with the `-O3` optimization. All inputs are of type `long` (8 bytes). For parallelization, we used the ParlayLib library [7], which supports parallel loops with `parlay::parallel_for` and `parlay::blocked_for`. For consistency with the BSZ implementation [8, 20], we use basic arrays instead of `parlay::sequences` and switched their parallel loops from using Cilk to ParlayLib. Our simple C++ implementation of the BSV algorithm uses ~175 lines of code, of which ~90 are reused directly from the BSZ implementation, which totals ~120 lines. Both algorithms use a min binary tree and find local matches using the SEQ algorithm. They differ in their approach to finding nonlocal matches. The BSV algorithm uses merging, while the BSZ algorithm searches within the tree. We simplified the merging in the BSV algorithm by ignoring already matched elements. This contrasts with the original description in [6] steps 6.1 and 6.2, which uses prefix and suffix minimas to identify the unmatched values. For the SEQ algorithm, we decided to use the array-based implementation since we found it to be about 30% faster than the stack-based implementation.

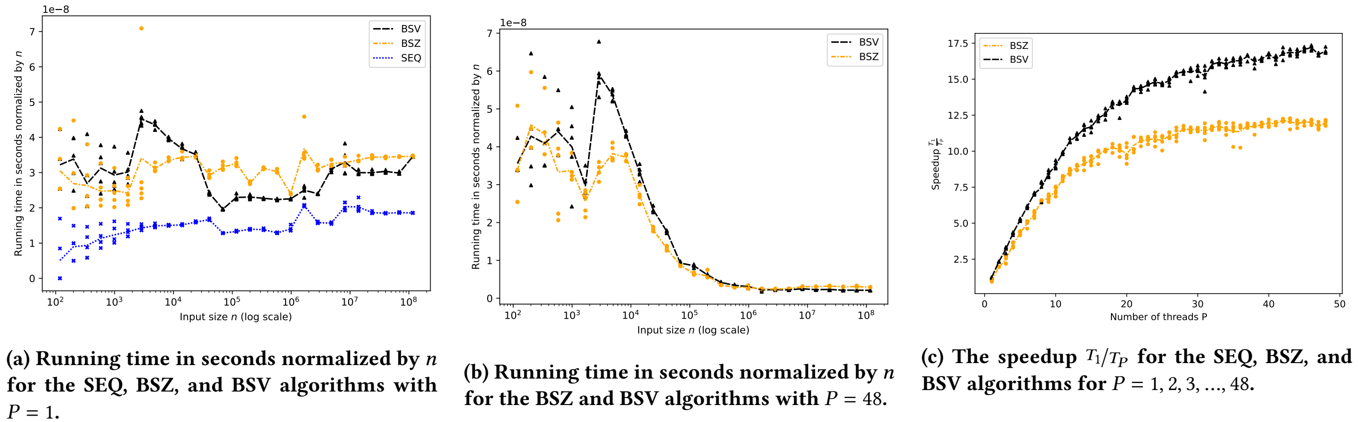
	P=1			P=48	
	SEQ	BSZ	BSV	BSZ	BSV
Sorted	1.03	1.95	2.26	0.27	0.17
Random	3.25	4.86	7.39	0.27	0.31
Merge	1.27	2.51	2.33	0.34	0.24
RandomMerge	2.16	4.03	4.02	0.34	0.24

Table 1: Running times in seconds for the SEQ, BSZ and BSV algorithms on the Sorted, Random, Merge and RandomMerge inputs for $P = 1$ and $P = 48$. We report the average of 5 runs, each with $n = \lfloor 1.7^{35} \rfloor = 116335496$ and block size $k = 256\lfloor \log_2 n \rfloor = 6656$.

We considered 4 different types of inputs. First, Sorted: the of numbers $1, 2, 3, \ldots, n$. Second, Random: a random permutation of $1, 2, 3, \ldots, n$. Third, Merge: the numbers $0, 2, 4, \ldots, \lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor - 1, \lfloor n/2 \rfloor - 3, \ldots, 1$, corresponding to the reduction from merging sorted lists to ANSV where the two sorted lists must be perfectly interleaved (for example, $0, 2, 4$ and $1, 3, 5$ forming input $0, 2, 4, 5, 3, 1$). Fourth, RandomMerge: similar to the Merge input, except the two values in each consecutive pair are swapped with probability 0.5.

6.2 Performance

In Table 1, we list the average running time in seconds for the three different algorithms across the four types of inputs, both for $P = 1$ and $P = 48$ threads, with $n \approx 10^8$ and block size $k = 256\lfloor \log_2 n \rfloor = 6656$. The block size is chosen to achieve $\Theta(n)$ work, and the constant 256 was determined through initial experiments. In section 6.3, we explore the impact of the block size k in more depth. The Sorted input is a trivial ANSV instance and was primarily used as a baseline to gauge how the algorithms should perform on an easy input. In practice, it also turned out to be the fastest. The Random input was the slowest overall for $P = 1$, whereas for $P = 48$, there was no decidedly slowest input type. We suspected the slowdown was due to additional branch mispredictions, which we investigated using the *perf* command. The results in Table 2



(a) Running time in seconds normalized by n for the SEQ, BSZ, and BSV algorithms with $P = 1$.

(b) Running time in seconds normalized by n for the BSZ and BSV algorithms with $P = 48$.

(c) The speedup T_1/T_P for the SEQ, BSZ, and BSV algorithms for $P = 1, 2, 3, \ldots, 48$.

Figure 3: For all three plots, each dot corresponds to the running time in seconds of an algorithm on the RandomMerge input. For each input size n, we repeat the experiment 5 times and draw a line through the average of these 5 runs. In each run we set the block size $k = 256 \log n$ to ensure $\Theta(n)$ work. For plots 3a and 3b we use a log scale and test inputs of size $n = 1.7^p$ for $p = 1, 2, 3, \ldots$ and $n \le 2^{27} = 134,217,728$. The running time is in seconds normalized by n (see the 1e-8 on the axis).

	P=1		P=48	
	BSZ	BSV	BSZ	BSV
Sorted	1.6	1.6	5.0	4.8
Random	258.4	425.5	267.0	437.2
Merge	1.4	1.4	4.1	4.1
RandomMerge	138.5	129.4	144.8	137.4

Table 2: Branch mispredictions in millions for the BSZ and BSV algorithms for the same parameters as in Table 1.

	T_1/T_{12}	T_1/T_{24}	T_1/T_{48}
BSZ	8.59	10.86	11.98
BSV	8.64	11.99	13.71

Table 3: Speedup of algorithms BSZ and BSV for 12, 24, and 48 processors. Speedup is calculated as T_1/T_p, where T_1 is the running time for 1 processor and T_p is the running time for p processors.

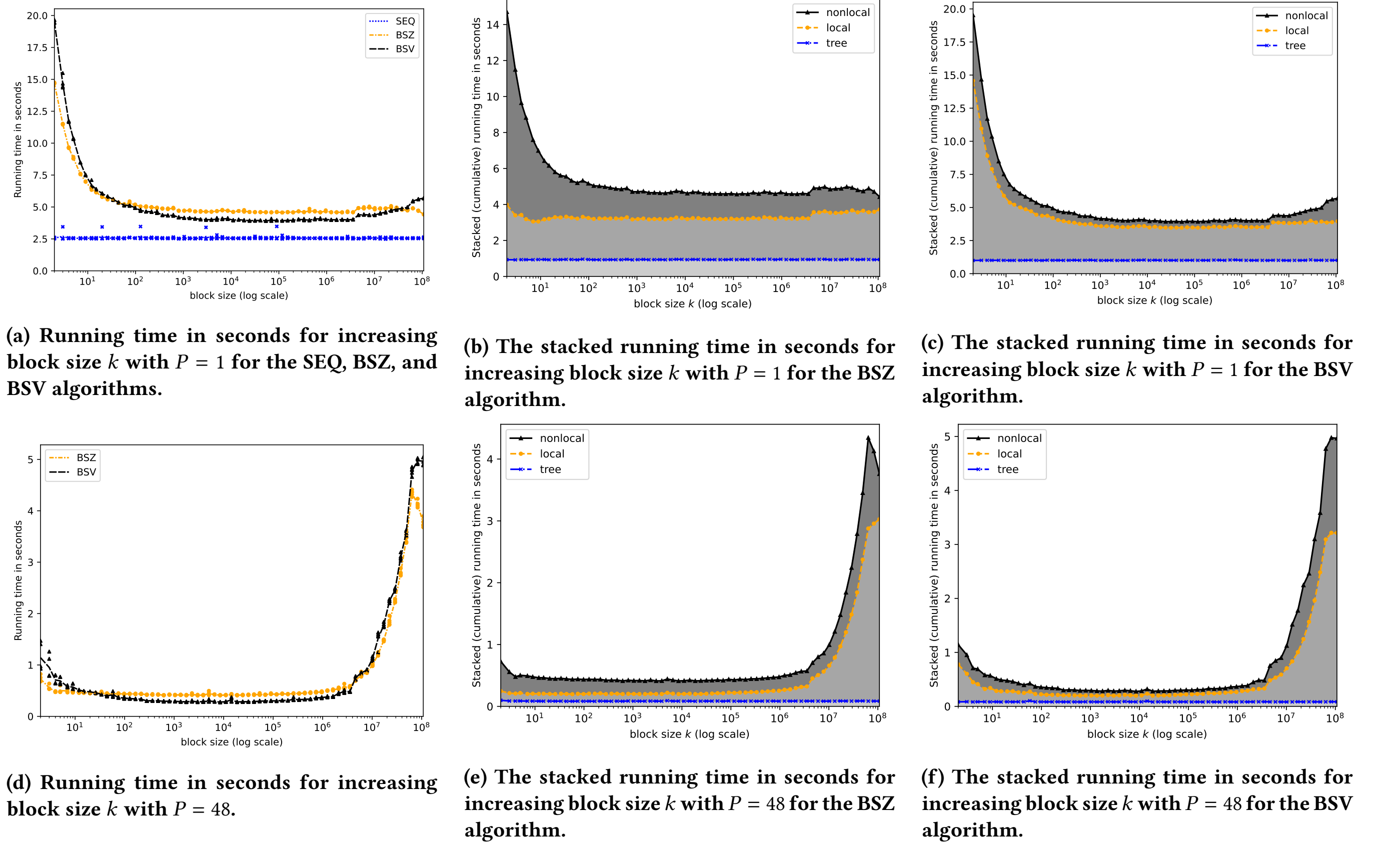
provide evidence of this for $P = 1$. Based on the discussion in section 4, we decided not to focus on the random inputs in further experiments. The behavior of the Merge input is comparable to that of RandomMerge, experiencing only a 20-30% slowdown. We believe the latter is the most well-motivated for three reasons. First, it naturally occurs in the reduction from merging sorted lists to ANSV. Second, without the heuristic, the BSZ algorithm performs $\Theta(n \log n)$ work. Third, there are many far-away matches which are the hard ones to compute. Taking inspiration from the Random input, we added some randomness, giving us the RandomMerge input, which, as expected for $P = 1$, is $60 - 80\%$ slower. Across the four inputs, for $P = 48$, the BSV algorithm is comparable to or slightly faster than the BSZ algorithm.

In Figure 3, we plot increasing n against running time in seconds normalized by n for each algorithm on the Merge input. In plot 3a where $P = 1$, we observe a mostly flat trend as expected, since all algorithms perform $\Theta(n)$ work. Even though there is a slight trend upward, the variance and performance of the BSZ and BSV algorithms mimic the SEQ algorithm, which serves as a simple baseline for what $\Theta(n)$ work should look like. In plot 3b where $P = 48$, both the BSZ and BSV algorithms converge nicely at around $0.26 \cdot 10^{-8}$ seconds, with a running time of ~ 0.305 seconds normalized by $n = 116335496$.

The speedup of a parallel algorithm is the ratio between its sequential running time T_1 with $P = 1$ processors and its running time T_P with P processors. In Figure 3 plot 3c, we plot the speedup of the BSZ and BSV algorithms as we increase the number of threads $P = 1, 2, 3, \ldots, 48$. The final speedup T_1/T_{48} is 11.98 and 13.71 for the BSZ and BSV algorithms, respectively. Until around $P = 12$ we observe a strong linear speedup, likely because one of the two CPUs with 12 cores is active and no hyper-threading is activated yet. From 12 to 24 processors, the speedup continues to increase steadily for both algorithms. From about 24 processors onwards, the speedup tapers off, but still increases more for the BSV algorithm than for the BSZ algorithm. See Table 3 for the exact speedups for $P = 12$ and $P = 24$.

6.3 Block size

Both the BSZ and BSV algorithms use a hyperparameter k for the block size, where in each block local matches are found sequentially. The primary distinction between the algorithms lies in their approach for handling the remaining nonlocal matches. This section explores the impact of the block size k on the different components of the algorithms. We categorize the running time into three parts. First, the *tree* part denotes the time to construct the min binary tree for both algorithms. Second, the *local* part represents the time to compute local matches in both algorithms, and for the BSV algorithm, it also includes the time to set up the merging problems. Third, the *nonlocal* part denotes the time spent traversing the min binary tree to find nonlocal matches in the BSZ algorithm. For the BSV algorithm it denotes the time spent on finding nonlocal



(a) Running time in seconds for increasing block size k with $P = 1$ for the SEQ, BSZ, and BSV algorithms.

(b) The stacked running time in seconds for increasing block size k with $P = 1$ for the BSZ algorithm.

(c) The stacked running time in seconds for increasing block size k with $P = 1$ for the BSV algorithm.

(d) Running time in seconds for increasing block size k with $P = 48$.

(e) The stacked running time in seconds for increasing block size k with $P = 48$ for the BSZ algorithm.

(f) The stacked running time in seconds for increasing block size k with $P = 48$ for the BSV algorithm.

Figure 4: For all six plots, we measured the average running time for increasing block size k (log scale) on the RandomMerge input of fixed size $n = 2^{27} = 134,217,728$, repeated 5 times, and drew a line through those averages. For the three top plots 4a, 4b and 4c we have $P = 1$, and for the three bottom plots 4d, 4e and 4f we have $P = 48$. In plots 4c, 4c, 4e and 4f, we show the stacked running time in seconds for the three parts of the BSZ and BSV algorithms. The *tree* part denotes the time to construct the min binary tree for both algorithms. The *local* part denotes the time to compute local matches for both algorithms, and for the BSV algorithm also the time to set up the merging problems. The *nonlocal* part denotes for the BSZ algorithm time spent traversing the min binary tree for nonlocal matches. For the BSV algorithm, it denotes the time spent on finding nonlocal matches by merging.

matches by merging. All parts take $\Theta(n)$ work, except for the nonlocal part of the BSZ algorithm, which takes $O\left(n\left(1+\frac{\log n}{k}\right)\right)$ time, as given by Lemma 2. Similarly, for the BSV algorithm, the local part takes $O\left(n\left(1+\frac{\log n}{k}\right)\right)$ time, due to the necessity of traversing the min binary tree twice for each block to set up the merging problems. For the RandomMerge input, the parts depending on k are $O\left(n\left(1+\frac{\log n}{k}\right)\right)$ as expected. Figure 4 plot 4a clearly shows this behavior, with the work decreasing rapidly as k increases. Plots 4b and 4c further confirm that it is nonlocal and local parts for the BSZ and BSV algorithms, respectively, that decrease, and that the others parts are independent of k. For $P = 48$, we still see an improvement in running time for small k in Figure 4 plot 4d. Not surprisingly, for large k, the running time increases dramatically, as both algorithms solve each block sequentially.

7 ACKNOWLEDGMENTS

The authors express gratitude to the Data-Intensive Systems group at Aarhus University for their provision of computing resources, and to the anonymous reviewers whose constructive feedback contributed to the improved presentation of the paper.

REFERENCES

[1] Alok Aggarwal and S. Vitter, Jeffrey. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (1988), 1116–1127. https://doi.org/10.1145/48529.48535
[2] Lars Arge, Michael T. Goodrich, Michael J. Nelson, and Nodari Sitchinava. 2008. Fundamental Parallel Algorithms for Private-Cache Chip Multiprocessors. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '08)*. 197–206. https://doi.org/10.1145/1378533.1378573
[3] Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. 2012. LRM-Trees: Compressed Indices, Adaptive Sorting, and Compressed Permutations. *Theoretical Computer Science* 459 (2012), 26–41. https://doi.org/10.1016/j.tcs.2012.08.010
[4] O. Berkman, Dany Breslauer, Zvi Galil, Baruch Schieber, and Uzi Vishkin. 1989. Highly Parallelizable Problems. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing* (Seattle, Washington, USA)

(STOC '89). Association for Computing Machinery, New York, NY, USA, 309–319. https://doi.org/10.1145/73007.73036

[5] O Berkman, B Schieber, and U Vishkin. 1988. Some Doubly Logarithmic Parallel Algorithms Based on Finding All Nearest Smaller Values. In *Technical Report UMIACS-TR-88-79*. Univ. of Maryland Inst. for Advanced Computer Studies New York/Berlin.

[6] O. Berkman, B. Schieber, and U. Vishkin. 1993. Optimal Doubly Logarithmic Parallel Algorithms Based on Finding All Nearest Smaller Values. *Journal of Algorithms* 14, 3 (1993), 344–370. https://doi.org/10.1006/jagm.1993.1018

[7] Guy E Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib-A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20)*. 507–509. https://doi.org/10.1145/3350755.3400254

[8] Guy E. Blelloch and Julian Shun. 2011. A Simple Parallel Cartesian Tree Algorithm and Its Application to Suffix Tree Construction. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. 48–58. https://doi.org/10.1137/1.9781611972917.5

[9] Richard P. Brent. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM* 21, 2 (1974), 201–206. https://doi.org/10.1145/321812.321815

[10] Patrick Flick and Srinivas Aluru. 2017. Parallel Construction of Suffix Trees and the All-Nearest-Smaller-Values Problem. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 12–21. https://doi.org/10.1109/IPDPS.2017.62

[11] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. 1999. Cache-Oblivious Algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. 285–297. https://doi.org/10.1109/SFFCS.1999.814600

[12] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. 1984. Scaling and Related Techniques for Geometry Problems. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing (STOC '84)*. Association for Computing Machinery, New York, NY, USA, 135–143. https://doi.org/10.1145/800057.808675

[13] Xin He and Chun-Hsi Huang. 2001. Communication Efficient BSP Algorithm for All Nearest Smaller Values Problem. *J. Parallel and Distrib. Comput.* 61, 10 (2001), 1425–1438. https://doi.org/10.1006/jpdc.2001.1741

[14] Joseph JáJá. 1992. *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., USA.

[15] Richard M. Karp and Vijaya Ramachandran. 1991. *Parallel Algorithms for Shared-Memory Machines*. MIT Press, Cambridge, MA, USA, 869–941.

[16] Jyrki Katajainen. 1996. Finding All Nearest Smaller Values on a Distributed Memory Machine. In *Proceedings of the Computing: The 2nd Australasian Theory Symposium (Australian Computer Science Communications)*, Vol. 18. Computer Science Association (Australia), 100–107.

[17] D. Kravets and C.G. Plaxton. 1994. An Optimal Hypercube Algorithm for the All Nearest Smaller Values Problem. In *Proceedings of 1994 6th IEEE Symposium on Parallel and Distributed Processing*. 505–512. https://doi.org/10.1109/SPDP.1994.346129

[18] Frédéric Loulergue, Simon Robillard, Julien Tesson, Joeffrey Legaux, and Zhenjiang Hu. 2014. Formal Derivation and Extraction of a Parallel Program for the All Nearest Smaller Values Problem. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing* (Gyeongju, Republic of Korea) *(SAC '14)*. Association for Computing Machinery, New York, NY, USA, 1577–1584. https://doi.org/10.1145/2554850.2554912

[19] Julian Shun and Guy E. Blelloch. 2014. A Simple Parallel Cartesian Tree Algorithm and Its Application to Parallel Suffix Tree Construction. *ACM Trans. Parallel Comput.* 1, 1, Article 8 (10 2014), 20 pages. https://doi.org/10.1145/2661653

[20] Julian Shun and Fuyao Zhao. 2013. Practical Parallel Lempel-Ziv Factorization. In *2013 Data Compression Conference*. IEEE, 123–132. https://doi.org/10.1109/DCC.2013.20

[21] Julian Shun and Fuyao Zhao. 2013. Practical Parallel Lempel-Ziv Factorization. https://github.com/zfy0701/Parallel-LZ77/blob/release. Accessed: 2023-10-01.

[22] Nodari Sitchinava and Rolf Svenning. 2024. A Parallel Implementation of an Optimal ANSV Algorithm. https://github.com/algoparc/ANSV/.

[23] Jean Vuillemin. 1980. A Unifying Look at Data Structures. *Commun. ACM* 23, 4 (1980), 229–239. https://doi.org/10.1145/358841.358852