Practical Parallel Block Tree Construction

Robert Clausecker
Zuse Institute Berlin, Germany

Florian Kurpicz
University of Münster, Germany

Etienne Palanga
TU Dortmund University, Germany

Abstract

The block tree [Belazzougui et al., J. Comput. Syst. Sci. '21] is a compressed representation of a length-n text that supports access, rank, and select queries while requiring only $O(z \log \frac{n}{z})$ words of space, where z is the number of Lempel-Ziv factors of the text. In other words, its space requirements are asymptotically comparable to those of the compressed text itself. In practice, block trees offer query performance comparable to that of state-of-the-art compressed rank and select indices. However, their construction is significantly slower, and the fastest known construction algorithms additionally require a significant amount of working memory. To address these limitations, we propose fast and lightweight parallel algorithms for the efficient construction of block trees.

Our algorithm achieves similar construction speed than the currently fastest block tree construction algorithm on a single core and is up to eight times faster using 64 cores, while requiring an order of magnitude less memory. Overall, we achieve a speedup of up to 15.5 on 64 cores, which is in line with the parallel construction of the Lempel-Ziv compression.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms; Theory of computation → Data compression; Computer systems organization → Single instruction, multiple data

Keywords and phrases block tree, shared memory, compression, SIMD, Karp-Rabin fingerprints

Digital Object Identifier 10.4230/LIPIcs.SEA.2026.13

Related Version *Previous Version*: `https://arxiv.org/abs/2512.23314`

Supplementary Material *Software (Source code)*: `https://github.com/pasta-toolbox/block_tree` archived at `swh:1:dir:240ee9e80c0d6c9d25f66f161d25848530591709`

1 Introduction

In today's information age, textual data in the form of DNA and protein sequences, source code, and textual content such as news articles and Wikipedia pages is generated at a much higher rate than hardware capabilities are advancing. As a result, we can no longer cope with the growing volume of data simply by investing in more powerful processing infrastructure. Instead, we require efficient and scalable data structures. Here, "scalable" refers to two dimensions: (1) efficient parallel processing of data and (2) compression of the data structure itself. Both dimensions help to better utilize modern hardware by enabling the processing of larger amounts of data within the same time and space constraints.

With respect to compression, we can exploit the fact that many real-world inputs are highly repetitive. For example, when considering DNA sequences, humans share approximately 99.9 % of their DNA with each other, making such data highly repetitive. A similar level of repetitiveness arises in versioned data, such as code repositories, as well as in versioned textual content like Wikipedia articles.

Some of the most fundamental queries we can answer on a text T of length n over an alphabet of size σ are access, rank, and select queries:

- $access(i) = T[i]$ returns the character at the i-th position of the text,
- $rank_\alpha(i) = |\{j \leq i \colon T[j] = \alpha\}|$ returns the number of occurrences of α in the prefix $T[1..i]$, and
- $select_\alpha(i) = \arg\min_j(rank_\alpha(j) = i)$ returns the position of the i-th occurrence of α in T.

Such queries are fundamental building blocks in many applications, including compressed full-text indices for pattern matching [25, 55, 29, 15, 14], lossless data compression [24, 32, 41], and computational geometry [12].

Currently, *wavelet trees* [32] are the "de facto standard" [4] for answering these queries in compressed space. However, the compression of wavelet trees is limited to statistical compression. As a consequence, their compression ratio on highly repetitive inputs is limited, since they require at least one bit per character. *Block trees* [6] address this limitation by exploiting dictionary-based compression.

Structure of this Paper. First, in Section 2, we introduce concepts used throughout the paper. Then, in Sections 3 and 4, we give an overview and comparison of the underlying compression techniques and block trees, including a detailed discussion of their construction, resp. Our new parallel construction algorithms are presented and analyzed in Section 5. We then evaluate our implementation experimentally in Section 6. Finally, Section 7 summarizes our results and discusses directions for future work and open problems.

Our Contributions. Block tree construction has so far not been studied in a parallel setting. To the best of our knowledge, all previous work [6, 40] considers either sequential or external-memory construction. Although the current state-of-the-art sequential construction algorithm [40] supports some degree of parallelism, this capability is merely a byproduct of the underlying data structures, which can themselves be constructed in parallel. Since the construction algorithm as a whole is not parallelized, its scalability is severely limited, see our experimental evaluation in Section 6. Furthermore, the algorithm's substantial memory requirements, both in theory and in practice, limit its applicability to large inputs.

We present the *first* dedicated parallel block tree construction algorithm that explicitly trades scalability for reduced memory requirements (see Section 5). Our approach is based on domain decomposition: we initially partition the input, execute the construction algorithm (semi-)independently on each partition in parallel, and merge the partial results. By adjusting the number of partitions, we can control the trade-off between parallel scalability and memory usage. Our experimental evaluation (see Section 6) shows that the peak space overhead during construction is up to an order of magnitude lower than that of the currently fastest block tree construction algorithms. Furthermore, the COST [51], i.e., the number of threads required for our algorithm to outperform the sequential stat- of-the-art, is only two. As a result of independent interest, we also present a SIMD-based algorithm for computing Karp–Rabin fingerprints [36] in Section 5.3 that are essential for the block tree's construction.

2 Preliminaries

A text $T \in [1..\sigma]^n$ is a sequence of length n over an alphabet of size σ. For $1 \leq i \leq j \leq n$, we denote by $T[i, j]$ the substring $T[i]T[i + 1] \dots T[j]$. We also use $|T[i, j]| = j - i + 1$ to denote the length of a substring.

Any two substrings can be compared in constant time with high probability using a rolling hash function for strings known as the *Karp-Rabin fingerprint* [36]. Let $T \in [1..\sigma]^n$, let $q \in \Theta(n^c)$ for some constant $c > 1$, and let $r < q$ be a positive integer such that $r \nmid q$. The Karp-Rabin fingerprint of a substring $T[s..e]$ is defined as $\phi(s, e) = \left(\sum_{i=s}^{e} T[i] \cdot r^{\,e-i}\right) \bmod q$.

For any two substrings $T[i..i + \ell]$ and $T[j..j + \ell]$, we have $\phi(i, i + \ell) = \phi(j, j + \ell)$ if $T[i..i + \ell] = T[j..j + \ell]$. Otherwise, if $T[i..i + \ell] \neq T[j..j + \ell]$, the probability that their fingerprints collide is $\text{Prob}[\phi(i, i + \ell) = \phi(j, j + \ell)] \in O\left(\frac{\ell \log \sigma}{n^c}\right)$. Karp-Rabin fingerprints are *rolling* hash functions, which allows, among other things, the computation of all length-ℓ fingerprints of a text $T \in [1..\sigma]^n$ in $O(n)$ time.

2.1 Measures of Compressibility

There exists a wide variety of different measures of compressibility. We refer to the excellent survey by Navarro [54] for a comprehensive overview. The *empirical entropy* [42] is based on the distribution of the characters (or substrings) of T. The 0-th order entropy is defined as $H_0(T) = \sum_{\alpha \in \Sigma} \frac{n_\alpha}{n} \log \frac{n}{n_\alpha}$, where $n_\alpha = \mathit{rank}_\alpha(n + 1)$. For $k > 0$, the k-th order entropy is given by $H_k(T) = \sum_{s \in \Sigma^k} \frac{|T_s|}{n} H_0(T_s)$, where T_s denotes the subsequence of characters occurring immediately after the substring $s \in [1, \sigma]^k$ in T. For example, for $T = \mathit{abcabd}$, we have $T_{ab} = \mathit{cd}$. Overall, it holds that $\log \sigma \geq H_0(T) \geq H_1(T) \geq \cdots \geq H_n(T)$. A major drawback of such statistical compression measures is that they fail to capture long-range repetitions, as they consider the text on a character-by-character basis.

Another class of measures is based on the size of a *Lempel-Ziv* factorization of a text. Such a factorization consists of z factors, each of which can be stored as a reference to a previous occurrence in the text. Among the most prominent techniques in this class are Lempel-Ziv-based compression algorithms. The Lempel-Ziv 77 (LZ77) factorization [68] parses a text T into factors $f_1, \ldots, f_z \in \Sigma^+$ such that $T = f_1 \ldots f_z$ and, for each factor f_i with $i \in [1, z]$, there exists a position $j \in [1, |f_1 \ldots f_{i-1}|]$ in the already parsed prefix of the text such that $f_i = T[j \ldots j + |f_i| - 1]$.

Strings can also be represented using context-free *grammars* [38]. Determining the size of the smallest grammar g that generates only the given text is NP-complete. However, grammars of size $O\left(z \log \frac{n}{z}\right)$ can be computed in linear time [63]. The currently strongest known measure is the *string complexity* δ [39], defined as $\delta = \max_{k \in [1,n]} \frac{d_k}{k}$, where $d_k = |\{T[i..i + k - 1] | i \in [0, n - k]\}|$. Interesting properties of δ include that it can be computed in linear time, that $z \in O(\delta \log \frac{n}{\delta})$, and that there exist families of length-n strings that require $\Omega(\delta \log \frac{n}{\delta})$ words to be encoded for every n and every $\delta \in [2, n]$ [39]. Recently, it has been shown that, at least in theory, the most important string queries can be answered using $O(\delta \frac{n \log \sigma}{\delta \log n})$ words of space [37].

2.2 Model of Computation

We analyze our algorithms in the PRAM model. In this model, multiple processors (PEs, processing elements) share a common memory. The PRAM model is synchronous, i.e., in each time step all PEs execute exactly one instruction. Several variants of the model exist, which differ in whether memory locations can be accessed exclusively or concurrently during a single time step. We further differentiate between read and write access.

The weakest variant is the EREW PRAM, which allows only exclusive read and exclusive write access to memory cells. A slightly stronger variant is the CREW PRAM model, where multiple PEs may concurrently read from the same memory cell, but only a single PE is allowed to write to a given memory cell during a time step. The strongest model considered

here allows both concurrent read and concurrent write access (CRCW). We consider two variants of the CRCW model. In the Common-CRCW model, multiple PEs may write to the same memory cell simultaneously only if they all write the same value. In the Arbitrary-CRCW model, multiple PEs may write to the same memory cell without any restriction on the written values. In this case, it is unspecified which of the written values is stored.

When analyzing PRAM algorithms, we are primarily interested in their *time* and *work*. The time is the number of synchronous time steps executed by the algorithm, while the work is the total number of operations performed, including arithmetic on local data as well as memory reads and writes. The work thus corresponds to the running time of the algorithm when executed on a single PE. The differences between weaker (EREW) and stronger (CRCW) models become apparent when considering the time complexity of basic parallel primitives.

▶ **Lemma 1** (All-Prefix Operation [30]). *Given n integers $a_1, \ldots, a_n$ and a binary associative operator $\otimes$ that can be evaluated in $O(1)$ time, the sequence $(s_1, \ldots, s_n)$ with $s_i = \bigotimes_{j=1}^{i} a_j$ can be computed in the EREW PRAM model in $O(\log n)$ time, $O(n)$ work, and $O(n)$ space.*

▶ **Lemma 2** ([18]). *In the Common-CRCW PRAM model, the All-Prefix operation from Lemma 1 with $\otimes = +$ (addition) can be computed in $O(\log n / \log\log n)$ time, $O(n)$ work, and $O(n)$ space.*

In addition to the All-Prefix operation, we frequently require parallel sorting as a basic subroutine.

▶ **Lemma 3** ([17]). *In the EREW and CREW PRAM models, sorting n integers requires $O(\log n)$ time, $O(n \log n)$ work, and $O(n)$ words of space.*

▶ **Lemma 4** ([17]). *In the CRCW PRAM model, sorting n integers requires $O(\frac{\log n}{\log\log\log n})$ time using $n \log^c n$ PEs for some constant $c \geq 1$.*

3 Related Work

There exists a plethora of compressed data structures that support access, rank, and select queries. However, with the notable exception of wavelet trees, very little work has addressed their parallel construction.[1] We discuss related work on block trees in more detail in Section 4.

In the simplest setting, namely, for a binary alphabet of size two – access, rank, and select queries can be answered in constant time using only sublinear space [13, 35]. A wide range of data structures has been proposed that employ different techniques to achieve succinct [59, 56, 65, 67, 48, 31, 43] and compressed [61, 57, 60, 29, 10] bit vectors with rank and select support. To the best of our knowledge, no work has been published on the parallel construction of bit vectors with rank and select support beyond word-level parallelism.

Wavelet trees [32] generalize rank and select support to larger alphabets. Informally, for a text $T \in [1, \sigma]^n$, a wavelet tree consists of $\lceil \log \sigma \rceil$ bit vectors of length n, each equipped with binary rank and select support. These bit vectors store information about subsequences of the original text and are used to answer queries. For a detailed description of wavelet trees and their numerous applications, we refer to several comprehensive surveys [53, 47, 33, 24, 19].

Wavelet trees can be statistically compressed, resulting in a data structure that occupies $n\lceil H_0(T) \rceil (1 + o(1))$ bits of space and supports rank and select queries in $O(\log \sigma)$ time [32]. Using multi-ary wavelet trees, the query time can be improved to $O(1 + \frac{\log \sigma}{\log\log n})$ [26]. Owing to

[1] For full-text indices with pattern-matching support, there is a large body of work on parallel construction. Since this topic is beyond the scope of this paper, we refer to a recent survey [9].

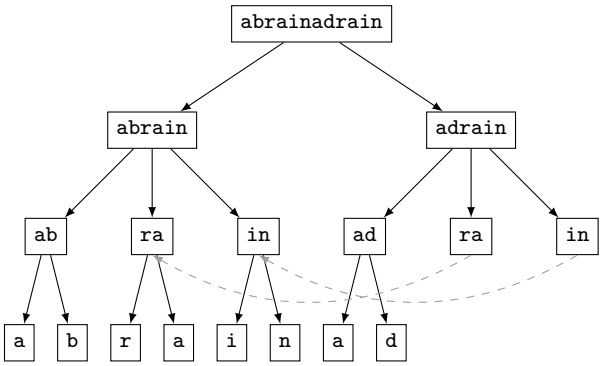

Figure 1 Example of a block tree for $T = \texttt{abrainadrain}$ with $s = 2$ and $\tau = 3$. On the third level, only two children are created, as the block size is not divisible by three. This case is included for the sake of a small example. Pointers to marked blocks are shown as dashed gray arrows.

their importance, wavelet tree construction has been extensively studied both in theory [5, 52] and in practice [66, 21, 23, 46, 28, 27, 20]. There is also a work focused on improving their query performance in practice [16, 11, 34].

Beyond wavelet trees, there exist statistically compressed data structures that require $nH_k(T)+o(n\log\sigma)$ bits of space and support rank queries in $O(\log\log_w\sigma)$ time, select queries in $\omega(1)$ time, and access queries in $O(1)$ time [8]. To overcome the limitations of statistical compression on highly repetitive inputs, grammar compression has also been explored. Using $O(g\sigma)$ words of space, grammar-based indices can support rank and select queries in $O(\log n)$ time [7, 58]. The parallel construction of grammars, albeit without query support, has also been investigated [49, 50].

4 Block Trees

A major drawback of statistically compressed data structures is that they require at least one bit per character of the input. For highly repetitive inputs, this results in a significant waste of space. To address this limitation, we consider block trees. Block trees support access, rank, and select queries, making them an ideal drop-in replacement for wavelet trees.

A block tree [6] is a tree in which the root has out-degree s and all other internal nodes have out-degree τ. For simplicity, we assume that the text for which we build the block tree is $T \in [1,\sigma]^n$ with $n = s\cdot\tau^h$ for some integer h. Here, h also corresponds to the height of the block tree.

Each node i in the block tree represents a substring of T, called the *block* B_i. The root represents the entire text. Each child of the root represents a consecutive block of length n/s. At any level of the block tree, we call two blocks *consecutive* if their substrings are adjacent in T. Let $\alpha = B_i\cdot B_{i+1}$ denote the concatenation of two consecutive blocks. We mark the blocks B_i and B_{i+1} if this is the leftmost occurrence of α on that level. All marked blocks are internal nodes with τ children, which represent consecutive blocks on the next level. An unmarked block B_u, on the other hand, is a leaf that stores a pointer to the pair of consecutive (marked) blocks $B_i\cdot B_{i+1}$ containing the leftmost occurrence of B_u, together with its offset within these two blocks. See Figure 1 for an example.

Interestingly, the number of blocks per level is bounded, which will later help us with analyzing the running time of our algorithms.

▶ **Lemma 5** ([6]). *Any level of a block tree (except the first) contains at most* $3z\tau$ *blocks.*

We do not have to continue the marking of blocks until block have size one. Instead, we stop as soon as explicitly storing the subtrees requires less space than storing the pointers and offsets for unmarked blocks B, i.e., when $|B| \in \Theta(\log_\sigma n)$. Overall, the block tree requires $O(s + z\tau \log_\tau \frac{n \log \sigma}{s \log n})$ words of space. When choosing $s = z$, the block tree requires $O(z\tau \log \frac{n \log \sigma}{z \log n})$ words of space. Note that there are different space-time trade-offs depending on the choice of s and τ [6]. When choosing $s = \log_\sigma n$, we also get $h = \log_\tau \frac{n \log \sigma}{s \log n}$.

4.1 Sequential Block Tree Construction

We first describe the original block tree construction algorithm [6]. Note that there also exists a highly engineered construction algorithm based on fundamental toolbox data structures for compression [40]. However, the underlying algorithmic idea is the same and, at a high level, it simulates the procedure described below.

The block tree is constructed in two phases, followed by an optional pruning phase. In the first phase, we mark all leftmost occurrences of consecutive blocks $B_i \cdot B_{i+1}$. In the second phase, we compute the pointers (and offsets) from unmarked blocks to their leftmost occurrences on the same level. During the pruning phase, we might reduce the number of nodes in the tree by adding additional pointers (and offsets). This phase, however, does not improve the asymptotic size of the block tree.

First Phase: In the first phase, we use a hash table to identify the leftmost occurrence of each pair of consecutive blocks. If a pair is already present in the hash table, it cannot correspond to the leftmost occurrence. Since we are not only interested in matches at block boundaries, we additionally scan the text to identify positions where such a pair may occur across block boundaries. To efficiently compare blocks, we use Karp-Rabin fingerprints [36]. Although matching fingerprints must still be verified to ensure that the corresponding substrings are equal, each substring needs to be verified only once. This results in a running time linear in the size of the level.

Second Phase: In the second phase, we again use hash tables and Karp-Rabin fingerprints, this time focusing on unmarked blocks. For each such block, we compute the pointer (and offset) to its leftmost occurrence, which is contained within a pair of consecutive marked blocks. To this end, we store the fingerprints of all unmarked blocks in a hash table. We then scan the current level to locate the leftmost occurrences of these blocks using their fingerprints. As before, this can be done in time linear in the size of the level, since each block is considered only once.

Optional Pruning Phase: Finally, there is an optional pruning step that can further reduce the size of the block tree in practice. However, this step does not yield any asymptotic improvement. Recall that, in the first phase, we mark all leftmost occurrences of consecutive pairs of blocks. This ensures that the leftmost occurrence of any unmarked block exists on the same level, possibly spanning one or two marked blocks. However, not all marked blocks are the target of a pointer. During the pruning step, these marked but untargeted blocks are removed. We do not consider this step in our parallel algorithm described below, as it does not improve the asymptotic space bounds. Nevertheless, it is included in our implementation; see Section 5.4.

4.2 Running Time

The running time depends on the sizes of the levels of the block tree. Let b_k denote the block size on the k-th level and let c_k denote the number of blocks on that level. Thus, $b_k \cdot c_k$ characters must be processed on level k. According to Lemma 5, there are at most $3z\tau$ blocks per level.

For levels starting from $\ell \geq 1 + \log_\tau \frac{3z}{s}$, we can bound the total number of characters processed across all levels from ℓ to the last level h as $\sum_{m=\ell}^{h} b_m \cdot c_m \leq \sum_{m=\ell}^{h} \frac{n}{\tau^{m-\ell}} = n \sum_{m=\ell}^{h} \frac{1}{\tau^{m-\ell}} \leq \frac{n}{1-\frac{1}{\tau}} = O(n)$. All preceding levels also contain at most $O(n)$ characters each. Since each character can be processed in constant expected time with high probability, we obtain the following running time bound.

▶ **Lemma 6** ([6]). *The block tree of a text $T \in [1, \sigma]^n$ can be constructed in $O(n(1 + \log_\tau \frac{z}{s}))$ expected time with high probability and $O(s + z\tau \log_\tau \frac{n \log \sigma}{s \log n})$ working space.*

5 Parallel Block Tree Construction

We now present the main result of this paper: the first dedicated parallel algorithms for block tree construction. In Section 5.1, we describe a simple parallelization based on sorting that requires $O(n)$ words of working memory. In Section 5.2, we then introduce a more space-efficient construction algorithm that provides a trade-off between scalability and working memory requirements. Next, in Section 5.3, we show how to compute Karp-Rabin fingerprints using SIMD. Finally, in Section 5.4, we give some implementation details.

5.1 Parallel Sorting using $O(n)$ Words of Memory

Our first parallel block tree construction algorithm replaces hash tables with sorting. Whenever fingerprints are required, we compute them in parallel.

▶ **Lemma 7** ([22]). *For any substring $T[i..i + \ell - 1]$ of length ℓ, the Karp-Rabin fingerprint $\phi(i, i + \ell - 1)$ can be computed in the EREW model with $O(\log \ell)$ time, $O(\ell)$ work, and $O(\ell)$ words of space.*

Since we require the fingerprints of all pairs of blocks in the first phase and of all substrings of length b_ℓ in the second phase on each level ℓ, we apply Lemma 7.

▶ **Lemma 8.** *Computing the Karp-Rabin fingerprint of every substring of length ℓ in a text of length n can be done in the EREW model with $O(\log n)$ time, $O(n)$ work, and $O(n)$ words of space.*

The space required to store these fingerprints constitutes the main memory bottleneck of the construction algorithm, as this storage is needed on every level (not at the same time).

We can now efficiently compute the fingerprints, but we still need to validate all comparisons. This verification can be performed in $O(\log n)$ time in the EREW and CREW models and in $O(1)$ time in the CRCW models. In both cases, the verification requires $O(n)$ work and no additional space.

Next, we sort tuples consisting of a fingerprint and the corresponding block position. In the sorted order, equal fingerprints appear consecutively. Using Lemmas 3 and 4, this sorting step can be performed in $O(\log n)$ time in the EREW model and $O\left(\frac{\log n}{\log \log \log n}\right)$ time in the CRCW model. Once the fingerprints are sorted (first by fingerprint and then by position), the leftmost occurrences can be identified in $O(1)$ time.

The second phase is slightly more involved. Here, we again use sorting to identify, for each unmarked block, the block and position to which it must point. To this end, we combine the fingerprints obtained during the scan with the fingerprints of the unmarked blocks. We sort these fingerprints as tuples consisting of the fingerprint value and the corresponding text position. For unmarked blocks, the text position additionally encodes that the entry corresponds to an unmarked block. After sorting, equal fingerprints are grouped together and ordered by text position, with the unmarked blocks placed at the end of each group.

We then construct an array *occ* that helps us compute the pointers and offsets. It has an entry for every fingerprint we just sorted. Then, for each fingerprint, we write a 0 to *occ* if it is equal to the fingerprint immediately to its left, and otherwise write its text position. For the rightmost occurrence of a fingerprint, we instead write the negated text position of the first occurrence to *occ*.[2] After computing the prefix sum over *occ*, the array contains the correct text positions for all but the rightmost occurrence of each fingerprint group (for which the correct value is found to its left). The prefix sum can be computed in $O(\log n)$ time in the EREW model and in $O\left(\frac{\log n}{\log\log n}\right)$ time in the Common-CRCW model.

This procedure allows us to compute every level of the block tree using only sorting and prefix sums. In the description above, we implicitly assume that we operate on $O(n)$ objects throughout. However, on each level of the block tree, some blocks may be replaced by references.

▶ **Theorem 9.** *The block tree of a text $T \in [1,\sigma]^n$ can be constructed in $O(\log n \log_\tau \frac{n\log\sigma}{s\log n})$ expected time with high probability and $O(n\log n(1+\log_\tau \frac{n\log\sigma}{s\log n}))$ work using $O(n)$ words of space in the EREW PRAM model.*

Proof. By Lemma 5, there are at most $3z\tau$ blocks on every level of the block tree except the first. Hence, there exists a level $\ell > 1+\log_\tau \frac{3z}{s}$ such that the total block length over all levels from ℓ to the last level h is linear in the text length. For all preceding levels, that is, levels $< \ell$, we conservatively assume that each level contains $O(n)$ characters.

Each level must be processed individually, as the computation depends on the marking information from the previous level. The total length of the remaining levels is bounded by $\sum_{m=\ell}^{h} b_m \cdot c_m \le \sum_{m=\ell}^{h} \frac{n}{\tau^{m-\ell}}$.

On each level, all required operations can be performed in logarithmic time and linear work in the size of that level. Thus, the total time over these levels is bounded by $\sum_{m=\ell}^{h}(\log n - \log\tau^{m-\ell}) \le \sum_{m=\ell}^{h}\log n$.

Since the height of the block tree is $h = \log_\tau \frac{n\log\sigma}{s\log n}$, the total running time for these levels is $O(\log n \cdot \log_\tau \frac{n\log\sigma}{s\log n})$. Including the initial levels yields the claimed bound. ◀

When considering the Common-CRCW model, we can further improve the time bound by using faster sorting and prefix-sum algorithms.

▶ **Lemma 10.** *In the Common-CRCW model, the block tree of a text $T \in [1,\sigma]^n$ can be constructed in $O(\frac{\log n}{\log\log n}\log_\tau \frac{n\log\sigma}{s\log n})$ expected time with high probability and $O(n\log n(1+\log_\tau \frac{n\log\sigma}{s\log n}))$ work using $O(n)$ words of space.*

5.2 Domain Decomposition using $O(s + K(z\tau))$ Words of Memory

A major drawback of the algorithm described above is its high working-memory requirement. We address this issue by employing *domain decomposition* to introduce a trade-off between scalability and memory usage. The idea of domain decomposition is to partition the input and process each partition as independently as possible.

For block trees, however, each level depends heavily on the previous one. Therefore, we have to partition the input on each level, after combining the results of the previous level. The algorithm from Lemma 7 remains largely unchanged, with the key difference that we now apply a local pre-filtering step.

[2] Negated, i.e., $-pos$ for a position *pos* and not bit-wise negated, where we flip each bit individually.

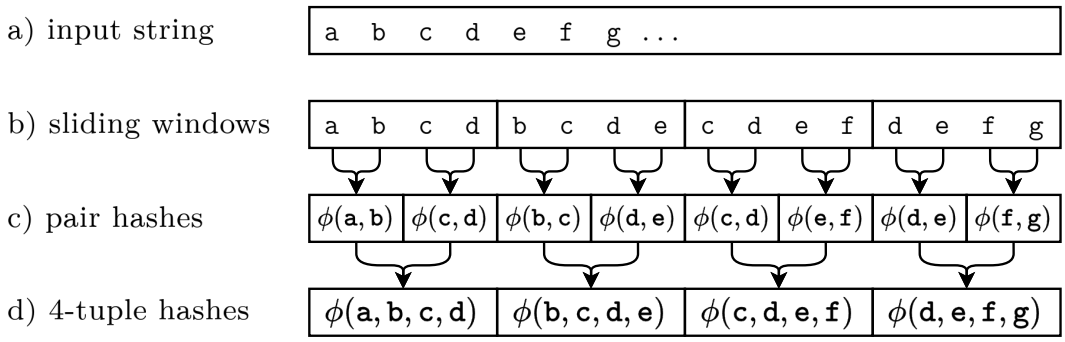

Figure 2 Computing the hashes of 4-tuples using 1 permutation and 2 pairwise multiply-adds.

Our algorithm uses K partitions. For each partition, we maintain a hash table that acts as a local filter. Only the locally leftmost occurrences of block pairs (and, in the second phase, substrings encountered during the scan) are forwarded to the sorting step. This significantly reduces the memory overhead, as we now operate on only a K-approximation of the blocks required for the construction.

▶ **Theorem 11.** *For any integer $K > 0$, the block tree of a text $T \in [1, \sigma]^n$ can be constructed in $O(\frac{n}{K} \log n \log_\tau \frac{n \log \sigma}{s \log n})$ expected time with high probability and $O(n \log n(1 + \log_\tau \frac{n \log \sigma}{s \log n}))$ work using $O(s + K(z\tau))$ words of space in the EREW PRAM model.*

Proof. The running-time argument is similar to that of the previous approach. The main difference is that, on each level, we now perform an additional local filtering step, which introduces a new computational bottleneck.

This filtering, however, gives us control over the amount of working memory required during construction. Since we can now control how often a fingerprint is forwarded beyond its local partition while still identifying leftmost occurrences, the required space per partition matches that of the sequential construction, namely $O(s + z\tau)$ [6]. In the worst case, a fingerprint may appear in every partition once, leading to the stated bound. ◀

▶ **Lemma 12.** *For any integer $K > 0$, in the Common-CRCW model, the block tree of a text $T \in [1, \sigma]^n$ can be constructed in $O(\frac{n}{K} \cdot \frac{\log n}{\log \log n} \log_\tau \frac{n \log \sigma}{s \log n})$ expected time with high probability and $O(n \log n(1 + \log_\tau \frac{n \log \sigma}{s \log n}))$ work using $O(s + K(z\tau))$ words of space.*

5.3 Data-Parallel Karp–Rabin Fingerprints

We accelerate the computation of Karp-Rabin fingerprints using SIMD techniques. Our implementation was initially developed for the Intel AVX2 instruction set extension and later ported to AArch64 ASIMD. We use the parameters $q = 2^{32}$, $r = 33$, and $\sigma = 256$, similar to the well-known `djb2` hash function.

Since AVX2 vectors are 32 bytes wide, they can store eight 32-bit hash values. In each iteration of our function, 16 new hashes $\phi(s..e), \phi(s+1, e+1), \ldots, \phi(s+15, e+15)$, stored in two vectors, are computed from the previous 16 hashes according to the scheme

$$\phi(s\,..\,e) = \phi(s-16\,..\,e-16) \cdot r^{16} - \phi(s-16\,..\,s-1) \cdot r^{e-s+1} + \phi(e-15\,..\,e). \tag{1}$$

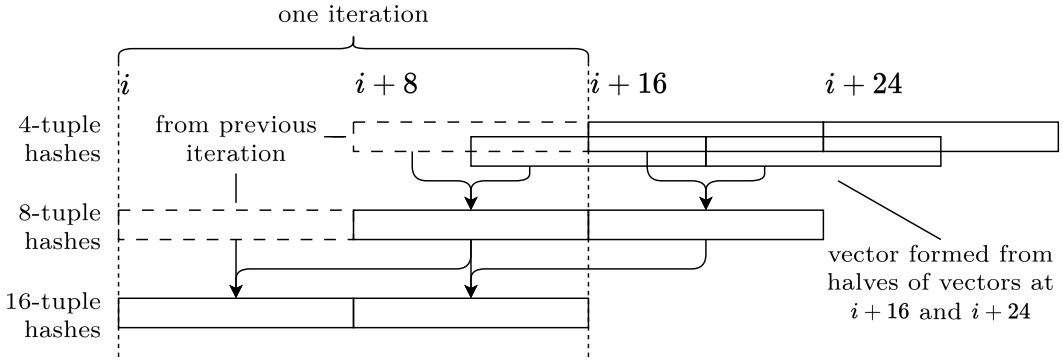

Figure 3 Computing 16 hashes of 16-tuples in vectors of 8 hashes each.

The 16-tuple hashes $\phi(e-15\,..\,e)$ to $\phi(e\,..\,e+15)$ are computed in two phases. First, hashes of 4-tuples are computed as shown in Figure 2. A chunk of input (a) is loaded into a SIMD register and permuted to form sliding windows (b) of four characters each. Using a data-parallel pairwise multiply–add instruction, pairs of characters $(T[i], T[i+1])$ are multiplied with $(r, 1)$ to produce pair hashes (c) according to $\phi(i, i+1) = T[i] \cdot r + T[i+1]$. This phase is then repeated by multiply–adding pairs of pair hashes with $(r^2, 1)$, yielding the 4-tuple hashes (d). Since $r < 256$, the pair hashes fit into 16 bits, preserving the vector width throughout steps (a) to (d).

These 4-tuple hashes are then combined into 8-tuple hashes and finally into 16-tuple hashes as depicted in Figure 3, using element-wise operations. Some intermediate values are carried over from the previous iteration to reduce the number of required operations. Vectors of 4-tuple hashes offset by 4, which are needed to compute the 8-tuple hashes, are formed by combining the rear half of the previous vector with the front half of the next vector.

The 16-character hashes are cached in a ring buffer so that they can be subtracted once the start of the sliding window reaches the current end. For windows of 17–32 characters, a variant of the above algorithm is used in which the ring buffer remains in vector registers to avoid costly store-reload latency. For hashes of up to 16 characters, specialized routines directly compute each hash from the corresponding characters.

The total computational effort per iteration consists of 4 pairwise multiply-add operations, 8 multiplications, 8 additions, and 6 permutations. However, due to the high degree of data parallelism and the latency of 32-bit SIMD multiplications on current Intel microarchitectures, throughput is nevertheless latency-bound at approximately 11 cycles per 16 characters (10 cycles to compute the product $\phi(s-16\,..\,e-16) \cdot r^{16}$ and 1 cycle to add the remaining terms) [1]. This compares favorably with the ~ 2.5 cycles per character required to compute Karp–Rabin fingerprints using the conventional recurrence [36]

$$\phi(s\,..\,e) = \phi(s-1\,..\,e-1) \cdot r - T[s-1] \cdot r^{e-s+1} + T[e],$$

and yields an overall performance improvement of approximately 5 % across all tested configurations of our algorithm.

5.4 Further Implementation Details

We implemented our block tree in `C++`, following the same interface as previous block tree implementations [6, 40]. A practical block tree implementation requires only one bit vector per level (to mark inner nodes), along with pointers and offsets to earlier occurrences for

inner nodes. We implemented the memory-efficient algorithm based on domain decomposition presented in Section 5.2, since the number of available PEs is determined by the hardware and is typically much smaller than the input size. Preliminary experiments showed that using a number of partitions different from the number of PEs negatively impacts performance.

In our implementation, we do not perform sorting. Although sorting simplifies the theoretical analysis, it is unnecessary in practice. Instead, each PE is responsible for a local partition of the current level. Blocks are initially marked only with respect to occurrences within the local partition; for this purpose, we use a hash table (`ankerl::unordered_dense::map`) as a filter to identify locally leftmost occurrences.

These marked blocks are then communicated to the responsible PE. To this end, we use a wait-free multiple-producer single-consumer (MPSC) queue [2]. Since fingerprints are uniformly distributed, they are used to assign blocks to PEs via a modulo operation. In practice, this results in a well-balanced workload without noticeable skew. Once all blocks have been marked globally, we proceed with the second phase in the same manner.

Another observation is that, beyond a certain threshold, the size of the MPSC queue does not influence performance. In our experimental evaluation (Section 6), we use a queue capacity of 512 fingerprints. Smaller queues led to load imbalances between PEs, while larger queues did not yield any measurable performance improvement.

6 Experimental Evaluation

We conducted our experiments on a server equipped with an AMD EPYC 7713 CPU (64 physical cores with hyperthreading support, running at a base frequency of 2.0 GHz with a turbo boost up to 3.66 GHz, 64 KB L1 and 512 KB L2 cache per core, and 256 MB of shared L3 cache) and 1024 GB of DDR4 RAM. The server runs Ubuntu 20.04 (kernel version 5.15.0). All code was compiled using the GNU Compiler Collection (GCC) version 11.4.0 with the provided build scripts.

We compare our new algorithm (`Par-BT`, see Section 5) with the current state-of-the-art block tree construction algorithm (`LPF-BT`) [40].[3] It uses standard LZ-compression tools (LPF-array) to compute the block tree. Since there are no other direct competitors, we additionally include a state-of-the-art parallel LZ77 compression algorithm (`Par-LZ`) [64],[4] the parallel LZMA compressor `pxz` [62], and the parallel `gzip` implementation `pigz` [3]. We will make our implementation publicly available once the paper has been accepted. As the implementation is part of a larger code base, releasing it would de-anonymize the authors.

Note that the latter three tools perform only compression of the input, without supporting random access or rank/select queries. Nevertheless, they provide insight into a closely related problem, as block trees can be seen as an LZ77-style approximation with additional query support. Thus, on the one hand, block trees relax the compression problem, while on the other hand, we have to compute additional information to enable queries.

We do not include the original block tree construction implementation [6], as it is purely sequential and up to an order of magnitude slower than `LPF-BT` [40]. We used the fastest configuration of `LPF-BT` and selected the same parameters (arity) for `Par-BT`, ensuring that both algorithms construct identical block trees. While block trees can be used as drop-in replacement for wavelet trees, we do not include them in the evaluation. Wavelet trees can be constructed up to an order of magnitude faster than block trees and their construction also scales well [19].

[3] `https://github.com/pasta-toolbox/block_tree`; archived at Zenodo [44], last accessed 2026-01-01.
[4] `https://github.com/zfy0701/Parallel-LZ77`, last accessed 2026-01-01.

Table 1 Name, size n, alphabet size σ, number of LZ77 factors z, and average LZ77 factor length $\lceil n/z \rceil$ of the inputs used in the experimental evaluation.

Name	n	σ	z	$\lceil n/z \rceil$
cere	461286644	5	1700630	272
coreutils	205281778	236	1446468	142
einstein.de	92758441	117	34572	2684
einstein.en	467626544	139	89467	5227
escherichia	112689515	15	2078512	55
influenza	154808555	15	769286	202
kernel	257961616	160	793915	325
para	429265758	5	2332657	185
world_leaders	46968181	89	175740	268

As input data, we use the real-world repetitive texts from the Pizza&Chili corpus[5], which is commonly used in text indexing research. The same or very similar (but smaller) inputs have been used in prior evaluations of the compared algorithms [40, 64]. Details on the input data are given in Table 1.

6.1 Scalability

We first evaluate the scalability of all implementations. To this end, we conducted a strong-scaling experiment in which each PE is mapped to a single physical CPU core. The throughput, measured as processed input in MiB per second, is shown in Figure 4.

Overall, all tested algorithms exhibit similar behavior across inputs, largely independent of input size, alphabet size, and average Lempel-Ziv factor length. We include `Par-LZ` to illustrate the inherent difficulty of efficiently computing the LZ77 factorization in parallel. Consequently, `Par-LZ` is the slowest of all evaluated algorithms. Interestingly, it scales similarly to `Par-BT`, despite `Par-BT` computing an LZ77-style approximation together with additional information required for query support rather than the LZ77 factorization itself.

Among the general-purpose compression tools, `pigz` is the fastest and best-scaling algorithm overall. However, this performance comes at the cost of a weaker compression ratio compared to `pxz`, the parallel LZMA compressor included in our benchmark. Overall, `pxz` achieves performance comparable to `Par-BT`(on some inputs it is faster, on others similar, and on some slower). Notably, it does not scale as well as `pigz`. We emphasize again that these compressors cannot be used for block tree construction, nor does the data they produce support access, rank, or select queries.

Currently, `LPF-BT` is the state-of-the-art block tree construction algorithm in terms of construction speed. Its authors also provide a naive parallelization, which, to the best of our knowledge, is the only other parallel block tree construction approach. This parallelization relies on parallel algorithms for toolbox compression data structures. Since only the construction of these toolbox data structures is parallelized, rather than the entire algorithm, `LPF-BT` exhibits limited scalability, with little to no speedup beyond 8 PEs.

In contrast, our algorithm `Par-BT` achieves speedups of up to 15.5 (on `einstein.de`) and an average speedup of 11.1 when using 64 PEs. While this is far from linear speedup, it is consistent with expectations given the difficulty of the underlying problem, the LZ77-like processing with its data dependencies and the scalability behavior observed for `Par-LZ`.

[5] `https://pizzachili.dcc.uchile.cl/repcorpus.html`, last accessed 2026-01-01.

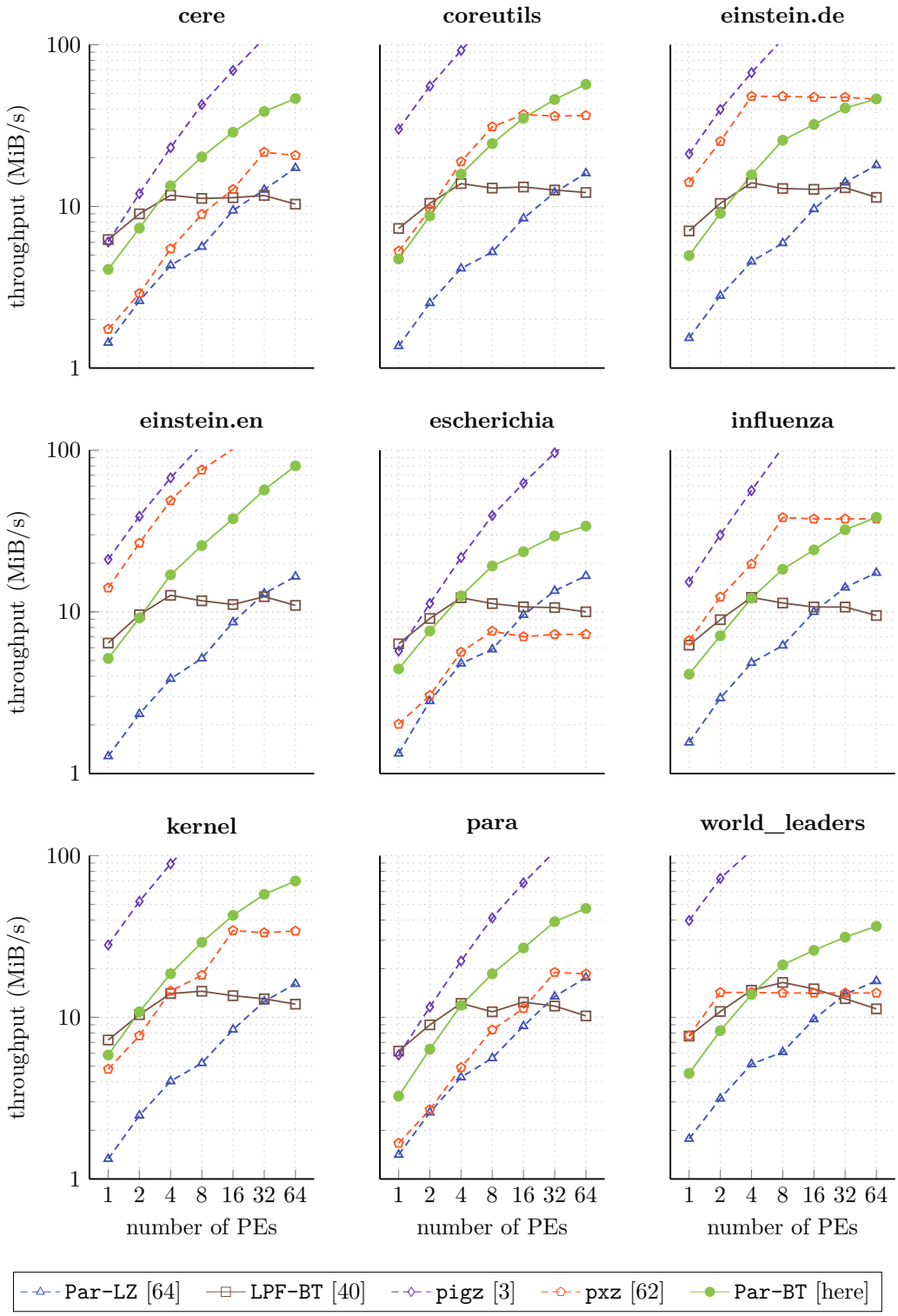

Figure 4 Throughput (MiB/s) of the parallel block tree construction algorithms and the parallel general purpose compression algorithms (dashed lines) in a strong scaling experiment.

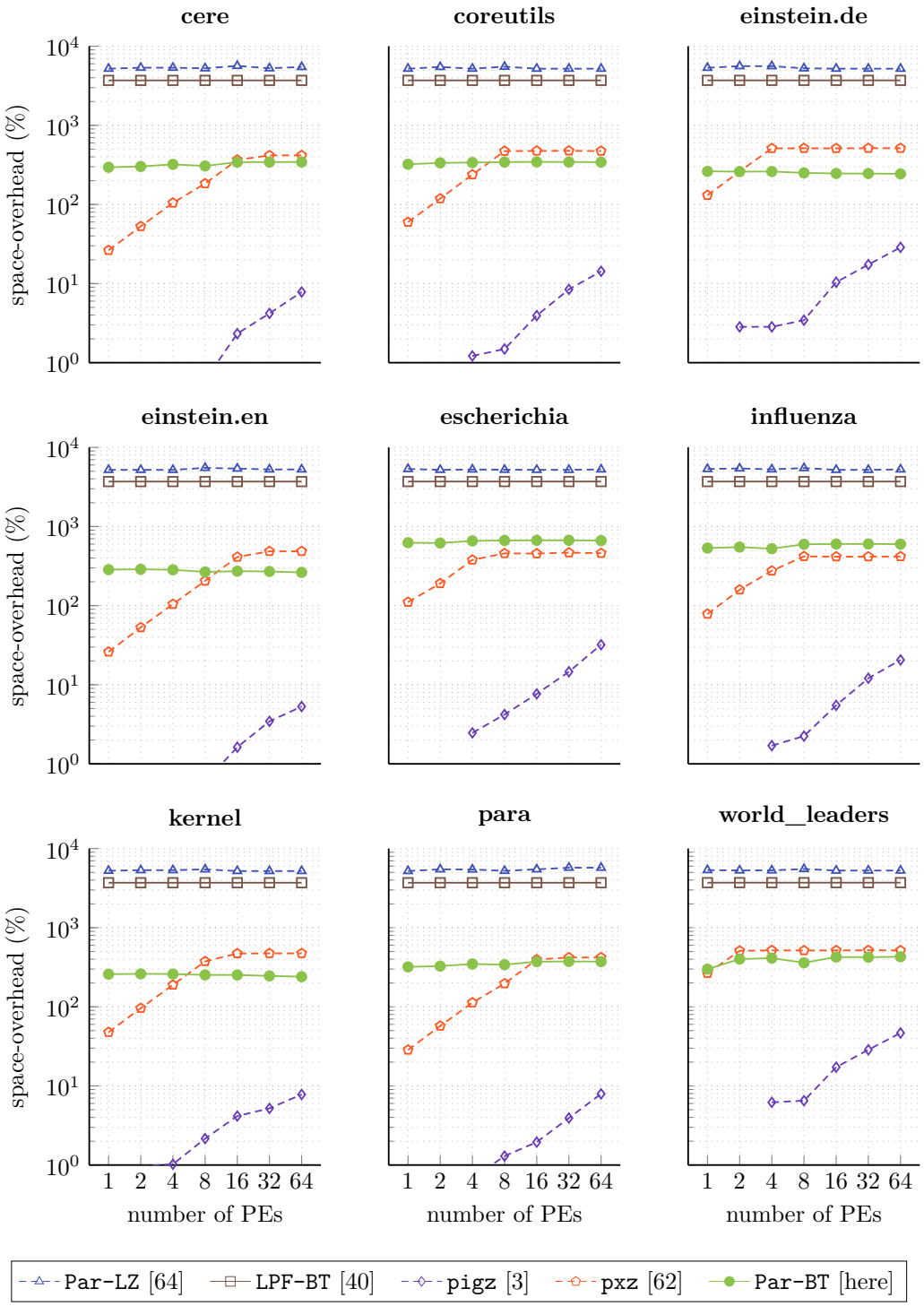

Figure 5 Space-overhead (%) of the parallel block tree construction algorithms and the parallel general purpose compression algorithms (dashed lines) in a strong scaling experiment. Missing data points refer to no heap allocation.

Finally, even when using a single PE, the achievable memory throughput is significantly higher than the block tree construction throughput. Specifically, memory throughput exceeds 1 GiB/s on a single thread, which is several orders of magnitude higher than the construction throughput. This indicates that additional memory bandwidth from multiple threads provides little benefit and that the algorithms are CPU-bound.

COST. The configuration that outperforms a single thread (COST) [51] denotes the number of PEs required for a parallel algorithm to outperform the fastest sequential algorithm. Without considering the COST metric, one could present impressive speedups simply by parallelizing a very slow algorithm. In our case, the COST of `Par-BT` is 2, that is, two PEs suffice for the parallel algorithm to be faster than the fastest sequential block tree construction algorithm (`LPF-BT` using one PE). This is essentially optimal, since our algorithm is not the fastest sequential approach.

6.2 Memory Requirements

In addition to construction speed, we are interested in the amount of memory required during construction. Excessive working-memory demands can severely limit the applicability of these algorithms in memory-constrained environments, such as HPC clusters or shared hardware. We report the memory allocated on the heap as a percentage of the input size in Figure 5. To the best of our knowledge, none of the evaluated implementations allocate significant memory on the stack, so considering heap usage alone is sufficient.

Although very fast, `LPF-BT` requires a significant amount memory, as it relies on several auxiliary data structures that need $O(n \log n)$ bits for an input of length n. A similar observation holds for `Par-LZ`. Interestingly, the working-memory requirement of `LPF-BT` is independent of the number of PEs used. As expected, `Par-BT` requires significantly less space than `LPF-BT`; its space overhead is roughly an order of magnitude smaller. Among all evaluated tools, `Par-LZ` exhibits the highest memory consumption during construction.

Both `pigz` and `pxz` are comparatively memory-efficient, but their memory usage increases proportionally with the number of PEs. Notably, for small PE counts, `pigz` does not require any heap allocations. However, as discussed earlier, this also results in weaker compression ratios compared to `pxz`. We observe slight variations in the space overhead of `Par-BT` depending on the number of PEs. This behavior can be explained by the local hash tables, whose sizes depend on the distribution of fingerprints. Since the used hash function is only pseudo-random, perfect load balancing across partitions cannot be guaranteed. Still, the observed distribution is sufficiently balanced to keep hash table sizes comparable across PEs.

7 Conclusion and Future Work

In this paper, we presented the first dedicated parallel algorithms for block tree construction. Our algorithms provide an effective trade-off between scalability and working-memory requirements during construction. Our `C++` implementation exhibits the best scalability among existing block tree construction algorithms. Moreover, when using only two threads, our implementation already outperforms the current state-of-the-art sequential algorithm, while requiring roughly an order of magnitude less working memory during construction.

Despite these improvements, block trees are not yet a universal drop-in replacement for wavelet trees. Although they compress very well on highly repetitive inputs, their size is at best comparable to that of wavelet trees on non-repetitive inputs. We aim to address this limitation by integrating more effective compression techniques and more space-efficient rank

and select data structures for bit vectors, e.g., [45]. Furthermore, there remains a considerable space-overhead due to auxiliary data structures required to support rank and select queries, which we have not considered in this paper. In future work, we plan to address these challenges and move closer to making block trees strictly superior to wavelet trees in practice.

References

1 Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*, ASPLOS '19, pages 673–686, New York, NY, USA, 2019. ACM. doi:10.1145/3297858.3304062.

2 Dolev Adas and Roy Friedman. Brief announcement: Jiffy: A fast, memory efficient, wait-free multi-producers single-consumer queue. In *DISC*, volume 179 of *LIPIcs*, pages 50:1–50:3. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.50.

3 Mark Adler. pigz - parallel implementation of gzip. https://github.com/madler/pigz, 2024. Accessed: 2025-04-22. URL: https://github.com/madler/pigz.

4 Jarno N. Alanko, Elena Biagi, Simon J. Puglisi, and Jaakko Vuohtoniemi. Subset wavelet trees. In *SEA*, volume 265 of *LIPIcs*, pages 4:1–4:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SEA.2023.4.

5 Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In *SODA*, pages 572–591. SIAM, 2015. doi:10.1137/1.9781611973730.39.

6 Djamal Belazzougui, Manuel Cáceres, Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Gonzalo Navarro, Alberto Ordóñez Pereira, Simon J. Puglisi, and Yasuo Tabei. Block trees. *J. Comput. Syst. Sci.*, 117:1–22, 2021. doi:10.1016/j.jcss.2020.11.002.

7 Djamal Belazzougui, Patrick Hagge Cording, Simon J. Puglisi, and Yasuo Tabei. Access, rank, and select in grammar-compressed strings. In *ESA*, volume 9294 of *Lecture Notes in Computer Science*, pages 142–154. Springer, 2015. doi:10.1007/978-3-662-48350-3_13.

8 Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Trans. Algorithms*, 11(4):31:1–31:21, 2015. doi:10.1145/2629339.

9 Timo Bingmann, Patrick Dinklage, Johannes Fischer, Florian Kurpicz, Enno Ohlebusch, and Peter Sanders. Scalable text index construction. In *Algorithms for Big Data*, volume 13201 of *Lecture Notes in Computer Science*, pages 252–284. Springer, 2022. doi:10.1007/978-3-031-21534-6_14.

10 Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. A "learned" approach to quicken and compress rank/select dictionaries. In *ALENEX*, pages 46–59. SIAM, 2021. doi:10.1137/1.9781611976472.4.

11 Matteo Ceregini, Florian Kurpicz, and Rossano Venturini. Faster wavelet tree queries. *Softw. Pract. Exp.*, 55(1):1931–1946, 2025. doi:10.1002/spe.70013.

12 Yu-Feng Chien, Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Geometric BWT: compressed text indexing via sparse suffixes and range searching. *Algorithmica*, 71(2):258–278, 2015. doi:10.1007/S00453-013-9792-1.

13 David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage (extended abstract). In *SODA*, pages 383–391. ACM/SIAM, 1996. URL: http://dl.acm.org/citation.cfm?id=313852.314087.

14 Francisco Claude, Antonio Fariña, Miguel A. Martínez-Prieto, and Gonzalo Navarro. Universal indexes for highly repetitive document collections. *Inf. Syst.*, 61:1–23, 2016. doi:10.1016/J.IS.2016.04.002.

15 Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In *SPIRE*, volume 7608 of *Lecture Notes in Computer Science*, pages 180–192. Springer, 2012. doi:10.1007/978-3-642-34109-0_19.

16 Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Syst.*, 47:15–32, 2015. doi:10.1016/j.is.2014.06.002.

17 Richard Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988. doi:10.1137/0217049.

18 Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Inf. Comput.*, 81(3):334–352, 1989. doi:10.1016/0890-5401(89)90036-9.

19 Patrick Dinklage, Jonas Ellert, Johannes Fischer, Florian Kurpicz, and Marvin Löbel. Practical wavelet tree construction. *ACM J. Exp. Algorithmics*, 26:1.8:1–1.8:67, 2021. doi:10.1145/3457197.

20 Patrick Dinklage, Johannes Fischer, and Florian Kurpicz. Constructing the wavelet tree and wavelet matrix in distributed memory. In *ALENEX*, pages 214–228. SIAM, 2020. doi:10.1137/1.9781611976007.17.

21 Patrick Dinklage, Johannes Fischer, Florian Kurpicz, and Jan-Philipp Tarnowski. Bit-parallel (compressed) wavelet tree construction. In *DCC*, pages 81–90. IEEE, 2023. doi:10.1109/DCC55655.2023.00016.

22 Jonas Ellert, Johannes Fischer, and Nodari Sitchinava. Lcp-aware parallel string sorting. In *Euro-Par*, volume 12247 of *Lecture Notes in Computer Science*, pages 329–342. Springer, 2020. doi:10.1007/978-3-030-57675-2_21.

23 Jonas Ellert and Florian Kurpicz. Parallel external memory wavelet tree and wavelet matrix construction. In *SPIRE*, volume 11811 of *Lecture Notes in Computer Science*, pages 392–406. Springer, 2019. doi:10.1007/978-3-030-32686-9_28.

24 Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of wavelet trees. *Inf. Comput.*, 207(8):849–866, 2009. doi:10.1016/j.ic.2008.12.010.

25 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly fm-index. In *SPIRE*, volume 3246 of *Lecture Notes in Computer Science*, pages 150–160. Springer, 2004. doi:10.1007/978-3-540-30213-1_23.

26 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):20, 2007. doi:10.1145/1240233.1240243.

27 Johannes Fischer, Florian Kurpicz, and Marvin Löbel. Simple, fast and lightweight parallel wavelet tree construction. In *ALENEX*, pages 9–20. SIAM, 2018. doi:10.1137/1.9781611975055.2.

28 José Fuentes-Sepúlveda, Erick Elejalde, Leo Ferres, and Diego Seco. Parallel construction of wavelet trees on multicore architectures. *Knowl. Inf. Syst.*, 51(3):1043–1066, 2017. doi:10.1007/s10115-016-1000-6.

29 Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Softw. Pract. Exp.*, 44(11):1287–1314, 2014. doi:10.1002/SPE.2198.

30 Tal Goldberg and Uri Zwick. Optimal deterministic approximate parallel prefix sums and their applications. In *ISTCS*, pages 220–228. IEEE Computer Society, 1995. doi:10.1109/ISTCS.1995.377028.

31 Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *WEA*, pages 27–38, 2005.

32 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850. ACM/SIAM, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

33 Roberto Grossi, Jeffrey Scott Vitter, and Bojian Xu. Wavelet trees: From theory to practice. In *CCP*, pages 210–221. IEEE Computer Society, 2011. doi:10.1109/CCP.2011.16.

34 Aaron Hong, Christina Boucher, Travis Gagie, Yansong Li, and Norbert Zeh. Another virtue of wavelet forests. In *SPIRE*, volume 14899 of *Lecture Notes in Computer Science*, pages 184–191. Springer, 2024. doi:10.1007/978-3-031-72200-4_14.

35 Guy Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.

36 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.

37 Dominik Kempa and Tomasz Kociumaka. Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. In *FOCS*, pages 1877–1886. IEEE, 2023. doi:10.1109/FOCS57990.2023.00114.

38 John C. Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Inf. Theory*, 46(3):737–754, 2000. doi:10.1109/18.841160.

39 Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Towards a definitive measure of repetitiveness. In *LATIN*, volume 12118 of *Lecture Notes in Computer Science*, pages 207–219. Springer, 2020. doi:10.1007/978-3-030-61792-9_17.

40 Dominik Köppl, Florian Kurpicz, and Daniel Meyer. Faster block tree construction. In *ESA*, volume 274 of *LIPIcs*, pages 74:1–74:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ESA.2023.74.

41 Dominik Köppl, Gonzalo Navarro, and Nicola Prezza. HOLZ: high-order entropy encoding of lempel-ziv factor distances. In *DCC*, pages 83–92. IEEE, 2022. doi:10.1109/DCC52660.2022.00016.

42 S. Rao Kosaraju and Giovanni Manzini. Compression of low entropy strings with lempel-ziv algorithms. *SIAM J. Comput.*, 29(3):893–911, 1999. doi:10.1137/S0097539797331105.

43 Florian Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. In *SPIRE*, volume 13617 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2022. doi:10.1007/978-3-031-20643-6_19.

44 Florian Kurpicz. pasta-toolbox/block_tree: v0.1.0, July 2023. doi:10.5281/zenodo.8114255.

45 Florian Kurpicz, Niccolò Rigi-Luperti, and Peter Sanders. Theory meets practice for bit vectors supporting rank and select. *CoRR*, abs/2509.17819, 2025. doi:10.48550/arXiv.2509.17819.

46 Julian Labeit, Julian Shun, and Guy E. Blelloch. Parallel lightweight wavelet tree, suffix array and fm-index construction. *J. Discrete Algorithms*, 43:2–17, 2017. doi:10.1016/j.jda.2017.04.001.

47 Christos Makris. Wavelet trees: A survey. *Comput. Sci. Inf. Syst.*, 9(2):585–625, 2012. doi:10.2298/CSIS110606004M.

48 Stefano Marchini and Sebastiano Vigna. Compact fenwick trees for dynamic ranking and selection. *Softw. Pract. Exp.*, 50(7):1184–1202, 2020. doi:10.1002/spe.2791.

49 Masaki Matsushita and Yasushi Inoguchi. Parallel processing of grammar compression. In *DCC*, page 358. IEEE, 2021. doi:10.1109/DCC50243.2021.00068.

50 Masaki Matsushita and Yasushi Inoguchi. Applying practical parallel grammar compression to large-scale data. In *DCC*, page 473. IEEE, 2022. doi:10.1109/DCC52660.2022.00084.

51 Frank McSherry, Michael Isard, and Derek Gordon Murray. Scalability! but at what cost? In *HotOS*. USENIX Association, 2015. URL: https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry.

52 J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. Fast construction of wavelet trees. *Theor. Comput. Sci.*, 638:91–97, 2016. doi:10.1016/j.tcs.2015.11.011.

53 Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014. doi:10.1016/j.jda.2013.07.004.

54 Gonzalo Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2022. doi:10.1145/3434399.

55 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):2–es, 2007. doi:10.1145/1216370.1216372.

56 Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *SEA*, volume 7276 of *Lecture Notes in Computer Science*, pages 295–306. Springer, 2012. doi:10.1007/978-3-642-30850-5_26.

57 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *ALENEX*. SIAM, 2007. doi:10.1137/1.9781611972870.6.

58 Alberto Ordóñez Pereira, Gonzalo Navarro, and Nieves R. Brisaboa. Grammar compressed sequences with rank/select support. *J. Discrete Algorithms*, 43:54–71, 2017. doi:10.1016/J.JDA.2016.10.001.

59 Giulio Ermanno Pibiri and Shunsuke Kanda. Rank/select queries over mutable bitmaps. *Inf. Syst.*, 99:101756, 2021. doi:10.1016/j.is.2021.101756.

60 Mihai Pătraşku. Succincter. In *FOCS*, pages 305–313. IEEE Computer Society, 2008. doi:10.1109/FOCS.2008.83.

61 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

62 Jindřich Nový. pxz – Parallel XZ compressor, 2024. Accessed: 2025-04-22. URL: https://github.com/jnovy/pxz.

63 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.

64 Julian Shun and Fuyao Zhao. Practical parallel lempel-ziv factorization. In *DCC*, pages 123–132. IEEE, 2013. doi:10.1109/DCC.2013.20.

65 Sebastiano Vigna. Broadword implementation of rank/select queries. In *WEA*, volume 5038 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2008. doi:10.1007/978-3-540-68552-4_12.

66 Kaneta Y. Fast wavelet tree construction in practice. In *SPIRE*, volume 11147 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2018. doi:10.1007/978-3-030-00479-8_18.

67 Dong Zhou, David G. Andersen, and Michael Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *SEA*, volume 7933 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 2013. doi:10.1007/978-3-642-38527-8_15.

68 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.