Constructing the Wavelet Tree and Wavelet Matrix in Distributed Memory *

Patrick Dinklage † Johannes Fischer† Florian Kurpicz†

Abstract

The wavelet tree (Grossi et al. [SODA, 2003]) is a compact index for texts that provides rank, select, and access operations. This leads to many applications in text indexing, computational geometry, and compression. We present the first distributed memory wavelet tree construction algorithms, which allow us to process inputs that are orders of magnitude larger than what current shared memory construction algorithms can work with. In addition, our algorithms can easily be adapted to compute the wavelet matrix (Claude et al. [Inf. Syst., 47:15–32, 2015]), an alternative representation of the wavelet tree. In practice, one of our distributed memory wavelet matrix construction algorithms is the first parallel algorithm that can compute the wavelet matrix for alphabets of arbitrary size.

1 Introduction

The wavelet tree [14] is a space-efficient data structure with numerous applications in text indexing [14], the construction of the Burrows-Wheeler Transform [3] and answering longest common extension queries [18], data compression [15], computational geometry [21] (as an alternative to fractional cascading), and other areas [8, 25]. The wavelet matrix [5] is an alternative representation of the wavelet tree for large alphabets.

Related Work. In recent years, a lot of work has been conducted on the parallel construction of wavelet trees in shared memory [6, 11, 13, 20, 29, 30] as well as external memory [6]. Extensive practical evaluations [6, 11] show that the most promising approach is the so called *domain decomposition* [11, 13, 20], which achieves the best overall running times. On the other hand, the parallel *split* algorithm [20] scales the best.

However, the proposed algorithms are bound to work on a single machine and hence, the problem sizes that can be handled are limited by its hardware. While the amount of data being processed is typically becoming larger (*big data*), Moore's Law is nearing its physical bounds [31] and so even though single machines can be equipped with terabytes of RAM, the number of CPU cores—and therefore the computational power—is essentially limited.

Our Contributions. We alleviate this issue by presenting the first parallel distributed memory wavelet tree construction algorithms, allowing us to use multiple nodes communicating over a network. The evaluation of our algorithms on inputs of size up to 96 GiB using up to 1,920 cores on 96 nodes shows that we (1) can compute wavelet trees for inputs no other wavelet tree construction algorithm can handle, (2) can achieve nearly linear speed-up, (3) have algorithms that are communication efficient, with a communication volume never exceeding the input size, and (4) have algorithms that outperform the best known sequential algorithm using two nodes (a COST [23] of 2) and the best known shared memory parallel algorithm using five nodes. In addition, one of our wavelet matrix construction algorithms is the only parallel algorithm that can compute wavelet matrices for inputs over large alphabets.

Since distributed algorithms already exist for suffix sorting [2, 9, 10, 12], our new algorithms are the next logical step in distributed full-text indexing, e.g., towards a distributed FM-index [14].

2 Preliminaries

Let T be a text of length n over an alphabet Σ. We use zero-based indexing: $T[0]$ is the first symbol of T and $T[n-1]$ is the last. For integers i and j with $i < j < n$, we

*This work was supported by the German Research Foundation (DFG), priority programme "Algorithms for Big Data" (SPP 1736).

†TU Dortmund University, Department of Computer Science, patrick.dinklage@tu-dortmund.de, johannes.fischer@cs.tu-dortmund.de, florian.kurpicz@tu-dortmund.de,

m	0	1	2	3	4	5	6	7
$(m)_b$	000	001	010	011	100	101	110	111
$(m)_b^r$	000	100	010	110	001	101	011	111
$\rho_3(m)$	0	4	2	6	1	5	3	7

Figure 1: Bit-reversal permutation for $k = 3$.

denote by $T[i..j]$ the substring of T starting at position i and ending at position j, also including i and j. A *bit vector* is a text over the binary alphabet.

For an integer m, let $B = (m)_b$ with $|B| = k$ be the binary representation of m using k bits and let B^r be the reversal of B. We call the integer represented by $B^r = (m)_b^r$ the *k-bit reversal* of m and write $\rho_k(m)$ for short. For a fixed k, the *bit-reversal permutation* maps each integer $m < 2^k$ to $\rho_k(m)$. See Fig. 1 for an example.

The *histogram* of T maps each symbol $c \in \Sigma$ to its number $\mathrm{occ}_T(c)$ of occurrences in T. The σ symbols with $\mathrm{occ}_T(c) > 0$ are mapped to the *effective alphabet* $\Sigma' = [0, \sigma - 1]$ of T, such that the lexicographic order of Σ is preserved. Let $\mathrm{eff}_T(c) \in \Sigma'$ be the rank of c in the effective alphabet. Then, we call T' with $T'[i] \coloneqq \mathrm{eff}_T(T[i])$ for each $i < n$ the *effective transformation* of T. See Fig. 2 for an example.

2.1 The Wavelet Tree. The *wavelet tree* [14] for a text T is a binary tree of height $\lceil \lg \sigma \rceil$. Each node v represents an interval $[a, b] \subseteq \Sigma$ and is labeled by a bit vector B_v. B_v contains one bit for each text position i in text order where $T[i] \in [a, b]$: a 0-bit if $T[i] \leq \lfloor \frac{a+b}{2} \rfloor$ and a 1-bit otherwise (if $T[i] > \lfloor \frac{a+b}{2} \rfloor$).

The root node represents the entire alphabet $[a, b] = \Sigma$ and thus its bit vector has length n. A node v has two children iff $|[a, b]| > 2$. We apply the described structure recursively for the left child to represent the interval $[a, \lfloor \frac{a+b}{2} \rfloor]$ and the right child to represent $[\lfloor \frac{a+b}{2} \rfloor + 1, b]$. Following that, a leaf represents an interval of size one or two. Fig. 2 shows an example of a wavelet tree.

There are two different representations of the wavelet tree. In the *node-based* representation, we store a bit vector for each node (as in Fig. 2) and use pointers for navigation from a node to its children. Here, we can interpret a bit in the wavelet tree as the direction one needs to take in order to navigate to the leaf representing the corresponding symbol: when encountering a 0, we move to the left child and otherwise, we move to the right child. This leads to the following observation.

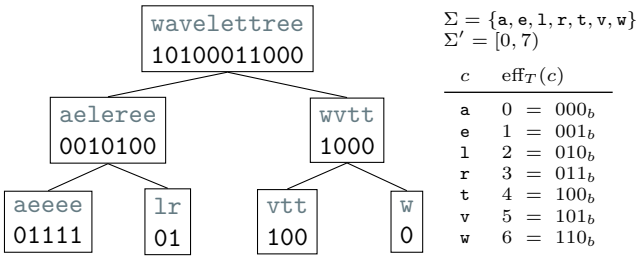

Figure 2: The wavelet tree (left), alphabet, effective alphabet and binary representations of symbols (right) for $T =$ `wavelettree`. The effective transformation of T is $T' = 6\,0\,5\,1\,2\,1\,4\,4\,3\,1\,1$. The texts above the node bit vectors are shown only for comprehensibility; they are not a part of the node labels and are not stored.

Observation 1. ([13]) *For $\ell \geq 0$ and $x \in \Sigma$, the ℓ-bit prefix of $(x)_b$ encodes the path in the wavelet tree from the root to the node on level ℓ whose interval contains x.*

In the *levelwise* representation, we concatenate the bit vectors on each level so that we have precisely n bits per level. For navigation, we use constant-time rank/select queries [26], which is asymptotically as fast as using pointers [25], but reduces the overall memory requirements (rank/select structures are needed anyway).

The size of any node's bit vector can be precomputed independently of the chosen representation. For every $c \in \Sigma$, let the array C contain the sum of all occurrences of symbols in T that are lexicographically smaller than c, i.e., $C[c] \coloneqq \sum_{x=0}^{c-1} \mathrm{occ}_T(x)$. We define $C[\sigma] \coloneqq n$.

Observation 2. *Let $[a, b] \subseteq \Sigma$ be the alphabet interval represented by a wavelet tree node v. The length of the bit vector B_v that labels v is $|B_v| = C[b + 1] - C[a]$.*

The C array, typically used in the context of the FM-index [14] to support the backward search algorithm [7], can be computed from the histogram in time $\mathcal{O}(\sigma)$ and requires $\sigma \lceil \lg n \rceil$ bits of space. Because there are at most $2\sigma - 1$ nodes in the wavelet tree, we can precompute the sizes (i.e., the number of bits) of all nodes also in time $\mathcal{O}(\sigma)$ and store them in less than $2\sigma \lceil \lg n \rceil$ bits.

2.2 The Wavelet Matrix. The *wavelet matrix* [5] is an alternative representation of the levelwise wavelet tree. Wavelet trees for texts over large alphabets have two disadvantages: either storing the pointers of the (node-based) wavelet tree dominates the memory requirements, or we have to use additional binary rank and select queries

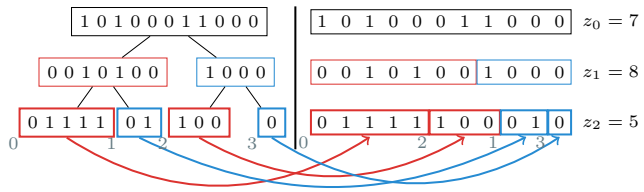

Figure 3: Node ordering in the wavelet tree (left) and the wavelet matrix (right). On the first two levels, the ordering is equal due to the nature of the bit-reversal permutation. On the third level, we observe how nodes 0 and 2 (left children of their respective parents) go to the left part of the corresponding wavelet matrix bit vector and nodes 1 and 3 (right children) go to the right. The z_ℓ values display the number of 0-bits for each level ℓ.

to navigate when querying the (levelwise) wavelet tree, which makes them slow in practice. The wavelet matrix solves this problem: it no longer needs pointers, but can still be navigated using the same number of binary rank and select queries as the node-based wavelet tree.

In the wavelet tree, the bit vectors of the single nodes on level ℓ are concatenated from *left to right*. In the wavelet matrix, they are concatenated in a different order: all left children of their respective parents are moved to the left and all right children are moved to the right. Like in the levelwise wavelet tree, we then concatenate the bit vectors of all nodes on every level. Fig. 3 shows an example. The re-ordering of nodes corresponds to the bit-reversal permutation of the node ranks on the respective level [11]. In addition, for each level ℓ, we store the value z_ℓ, which is the number of 0-bits in bit vector B_ℓ. It can be used to accelerate queries [5] while requiring only negligible $\lceil \lg \sigma \rceil \lceil \lg n \rceil$ bits of additional memory.

2.3 Distributed Parallel Computing. In distributed computing, we have p *processing elements* (PEs) that can communicate to execute an algorithm in parallel. For $i < p$, let *PE* i denote the i-th PE.

We analyze distributed algorithms in the *bulk-synchronous parallel* (BSP) model [28, 33], where the execution of an algorithm is divided into *supersteps* and each superstep consists of three phases: (1) local computation, (2) communication, during which messages are passed but no local computation occurs on received data, and finally a (3) barrier synchronization, which marks the point in time where all PEs have received their messages and are ready to proceed with local computation, initiating the next superstep (if any).

Let G be the time required to communicate one machine word over the network and L the time for one barrier synchronization. Then, the time required for one superstep is $w + Gh + L$, with w the maximum time required for local computation and h the maximum number of words communicated during the superstep on any PE. The *BSP costs* of an algorithm are the time required for all supersteps. We are also interested in the *communication volume*, which is the total number of words communicated over all supersteps by all PEs. Regarding computations on single PEs, we use the word RAM model, where we can perform arithmetic operations on words of width $\mathcal{O}(\lg n)$ bits in time $\mathcal{O}(1)$.

2.4 Prefix and All-to-All Sums. A common problem in distributed computing is finding the *prefix sum* of a vector $(x_0, x_1, \ldots, x_{p-1})$, which is distributed so that PE i knows only x_i. The task of prefix summing is to compute the vector of sums $X_i = \sum_{j=0}^{i-1} x_j$ so that after the operation, PE i knows the sum X_i of values held by PEs with lower rank. Similar to a PRAM-optimal text book algorithm [16, section 2.1.1], in the distributed setting, we can use a binary merge tree to compute the prefix sum in time $\mathcal{O}(\lg p)$ and $\mathcal{O}(\lg p)$ supersteps with any PE sending $\mathcal{O}(1)$ words in each superstep, resulting in a communication volume of $\mathcal{O}(p)$ words. Assuming that each x_i comes from a universe of n words, the local memory required is $\mathcal{O}(\lg n)$ bits.

A related problem is computing of the sum $\sum_{j=0}^{p-1} x_j$ of all local values and broadcasting it back to all PEs. The general case is known as an *all-to-all reduction* (or *AllReduce*) and supports any associative reduction operation. In our case, where the reduction of two elements is their sum, we henceforth use the term *all-to-all sum*, which can also be computed using a binary merge tree [27] and thus has the same asymptotic BSP costs as prefix summing.

LEMMA 1. *Using p PEs, we can compute the prefix sums and the all-to-all sum of a distributed vector of p words from a universe of n words with BSP costs of $\mathcal{O}(\lg p + G \lg p + L \lg p)$, a communication volume of $\mathcal{O}(p)$ words and using $\mathcal{O}(\lg n)$ bits of local memory.*

Algorithm	BSP costs in $\mathcal{O}(\cdot)$ Local Computation	Communication	Synchronization	Comm. Volume in $\mathcal{O}(\cdot)$ words
Domain Decomposition (§3.1)	$W_{localWT}(n/p) + \sigma \lg p$	$G(\sigma \lg p + (n/p) \lg \sigma / \lg n)$	$L(\lg p + \lg \sigma)$	$n \lg \sigma / \lg n + \sigma p$
Bucket Sort (WT) (§3.2)	$(n/p) \lg \sigma + \sigma \lg p$	$G(\sigma \lg p + (n/p) \lg \sigma)$	$L \lg \sigma \lg p$	$n \lg \sigma + \sigma p$
Bucket Sort (WM) (§3.2)	$(n/p) \lg \sigma + \lg \sigma \lg p$	$G(\lg \sigma \lg p + (n/p) \lg \sigma)$	$L \lg \sigma \lg p$	$n \lg \sigma + \lg \sigma p$
Split*(§3.3)	$(n/p) \lg \sigma + \sigma \lg p$	$G(\sigma \lg p + (n/p) \lg \sigma)$	$L(\lg \sigma + \lg p + \lg(\sigma) \lg(p))$	$n \lg \sigma + \sigma p$

* Under the condition that the input is uniform, σ is a power of two and p is a multiple of σ; for details see §3.3.

Table 1: Overview of the BSP costs and communication volumes of the algorithms described in §3, listing the wavelet tree (WT) and matrix (WM) versions of the bucket sort algorithm separately. The asymptotic amount of required local memory is $\mathcal{O}((n/p) \lg \sigma + \sigma \lg n)$ for all algorithms except for Bucket Sort (WM), which does not require the $\mathcal{O}(\sigma)$ factor.

2.5 Histogram and Effective Alphabet. We describe how to compute the histogram and effective alphabet of an input text T in distributed memory. Assuming $n \geq p$, we partition T such that PE i initially has part $T_i = T[i\lceil n/p \rceil \,..\, (i+1)\lceil n/p \rceil - 1]$ of length $\lceil n/p \rceil$, i.e., $T = T_0 T_1 \cdots T_{p-1}$. In the case that n is not a multiple of p, PE $p-1$ may have a shorter part, which we consider an implementation detail.

Each PE first computes a *local* histogram by scanning its local text part once. This takes $\mathcal{O}(n/p + |\Sigma|)$ time (accounting also for initialization) and requires $|\Sigma| \lceil \lg n \rceil$ bits of local memory. In a second step, we compute the all-to-all sum of the $|\Sigma|$ words containing the local occurrence counts for each symbol in Σ. In $\mathcal{O}(|\Sigma|)$ subsequent time, we can reduce Σ to the effective alphabet Σ' and store the effective alphabet ranks of all symbols using $|\Sigma| \lg |\Sigma'|$ bits on all PEs. We can then override T by its effective transformation T' in one more scan taking $\mathcal{O}(n/p)$ time.

LEMMA 2. *Using p PEs, we can compute the histogram, effective alphabet Σ', effective transformation and the node sizes of the wavelet tree for a text $T \in \Sigma^n$ with BSP costs of $\mathcal{O}(n/p + |\Sigma| + \lg p + G|\Sigma| \lg p + L \lg p)$, a communication volume of $\mathcal{O}(|\Sigma| p)$ words and using $|\Sigma|(\lceil \lg n \rceil + \lg |\Sigma'|)$ bits of local memory.*

We note that for $|\Sigma| > |\Sigma'|$, this preprocessing may asymptotically dominate the wavelet tree and matrix construction algorithms that we describe in the following sections. However, this step is optional in that a larger alphabet would merely result in more levels being constructed. In the following, we assume that the input is given already in its effective transformation.

3 Wavelet Tree and Matrix Construction in Distributed Memory

We present three distributed wavelet tree construction algorithms that compute the wavelet tree of T such that PE i holds the i-th part of length $\lceil n/p \rceil$ bits of each level's bit vector. For these algorithms, we assume that the input T is partitioned as described in §2.5. Furthermore, we show how to alter each algorithm to compute the wavelet matrix. Tab. 1 gives an overview of the BSP costs and communication volumes of our algorithms.

3.1 Domain Decomposition. Domain decomposition [11, 13, 20] is a straightforward technique to distribute work. First, each PE i constructs the entire wavelet tree for its input part T_i using a sequential algorithm. We use the notation $W_{localWT}(n/p)$ for time and $M_{localWT}(n/p)$ for bits of memory required for the local sequential construction to abstract from the different available algorithms (e.g., most algorithms require time $\mathcal{O}(n \lg \sigma)$, but more sophisticated algorithms require less time [17] or less space [32][4]). Furthermore, we assume that the wavelet tree is produced in its node-based representation so that after local construction on PE i, we have access to a bit vector $B_{v,i}$ for every wavelet tree node v. This scenario is depicted in Fig. 4. Empty nodes are possible as the alphabet of the entire input is used for construction of the local wavelet tree and some symbols may never occur in the local part of the input. We now *merge* the p partial wavelet trees into the *global* wavelet tree for T. Let $v_0, \ldots, v_{2^\ell - 1}$ be the 2^ℓ nodes on level $\ell < \lceil \lg \sigma \rceil$. The bit vector for level ℓ is the concatenation

$$B_\ell \coloneqq (B_{v_0,0} \cdots B_{v_0,p-1}) \cdots (B_{v_{2^\ell-1},0} \cdots B_{v_{2^\ell-1},p-1}).$$

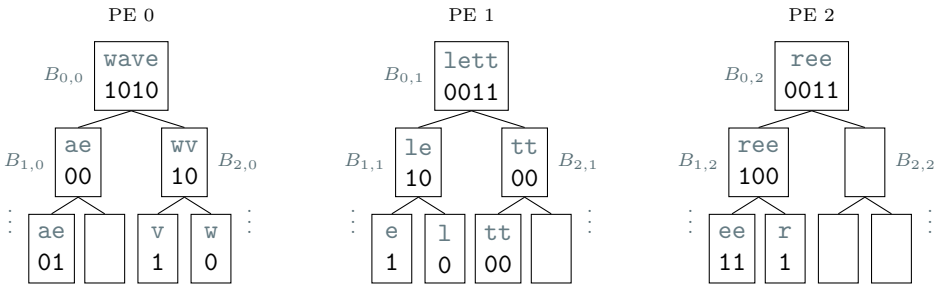

Figure 4: Domain decomposition of the wavelet tree for $T = \texttt{wavelettree}$ using three PEs with $T_0 = \texttt{wave}$ (left), $T_1 = \texttt{lett}$ (middle), and $T_2 = \texttt{ree}$ (right). Nodes are numbered in breadth-first search ordering.

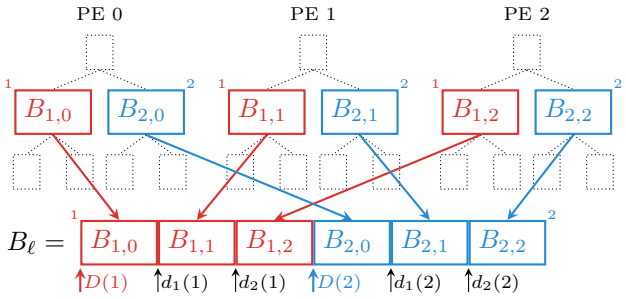

Figure 5: Balanced merge operation for three PEs, merging the bit vectors $B_{1,i}$ and $B_{2,i}$ of the second level's nodes ($\ell = 1$) into level bit vector B_ℓ. The top part shows the local wavelet trees for each PE, the bottom part shows the merged level bit vector. In B_ℓ, we mark the node offsets $D(1)$ and $D(2)$ as well as the inner offsets $d_i(1)$ and $d_i(2)$ for both nodes and each PE $i < p$.

In other words, we first concatenate the p bit vectors for each node v_k on level ℓ (with $k < 2^\ell$) in PE order to get the global node bit vectors B_{v_k}. We then concatenate these 2^ℓ node bit vectors in node order to retrieve the level bit vector B_ℓ. The fact that B_ℓ should be balanced across the p PEs poses an additional challenge. For PE i to find out where to send a certain part of $B_{v_k,i}$ for some node v_k, it needs to know

(1) the *node offset* $D(v_k) := \sum_{j=0}^{k-1} |B_{v_j}|$, i.e., the sum of the (global) bit vector lengths of the nodes that come before v_k on level ℓ, and
(2) the *inner offset* $d_i(v_k) := \sum_{j=0}^{i-1} |B_{v_k,j}|$, i.e., the number of bits for v_k held by PEs before i.

The position $D(v_k) + d_i(v_k)$ is then the position of the first bit of local node bit vector $B_{v_k,i}$ in global level bit vector B_ℓ. Since every PE will receive $\lceil n/p \rceil$ bits, the recipient for each local bit can be determined by dividing its position in B_ℓ by $\lceil n/p \rceil$. Fig. 5 shows an example.

We can compute the node offset $D(v_k)$ in time $\mathcal{O}(2^\ell) = \mathcal{O}(\sigma)$ using precomputed node sizes according to Obs. 2 without any additional communication. The computation of the inner offset $d_i(v_k)$, however, requires a prefix sum. After the local construction phase, we compute d_i for all $\mathcal{O}(\sigma)$ nodes in one single prefix sum computation, i.e., for a vector of $\mathcal{O}(\sigma)$ words. This requires $\mathcal{O}(\sigma)$ time and sending of $\mathcal{O}(\sigma)$ words in each superstep in a total of $\mathcal{O}(\lg p)$ supersteps.

Because the local wavelet tree was constructed for a part of the input of length $\lceil n/p \rceil$, it holds that $|B_{v_k,i}| \leq \lceil n/p \rceil$ for any node v_k. As a consequence, there can be at most two different recipients for the bits of $B_{v_k,i}$. Therefore, with $D(v_k)$ and $d_i(v_k)$ known, we can determine the recipients for all node bit vectors in $\mathcal{O}(\sigma)$ total time. On any PE, the total number of words sent is bounded by the number $\mathcal{O}((n/p)\lg\sigma)$ of bits in the local wavelet tree. Since we can pack $\mathcal{O}(\lg n)$ bits into one word, we send $\mathcal{O}((n\lg\sigma)/(p\lg n))$ words. However, in case $n/(p\lg n)$ is not integer, each PE may send an additional incomplete word for every node, resulting in a total of $\mathcal{O}(\sigma)$ incomplete words in the worst case. We perform the merge level by level, so we need a barrier synchronization on each level, amounting to $\mathcal{O}(\lg\sigma)$ total BSP barriers for the merge. Regarding local memory consumption, we require $\lceil n/p \rceil \lceil \lg\sigma \rceil$ bits to store the wavelet tree and less than $2\sigma\lceil \lg n \rceil$ bits for the precomputed node sizes. Finally, we require a send/receive buffer of $\lceil n/p \rceil$ bits.

LEMMA 3. *Using domain decomposition with p PEs, we can construct the wavelet tree for $T \in \Sigma^n$ with BSP costs of $W_{localWT}(n/p) + \mathcal{O}(\sigma\lg p + G(\sigma\lg p + (n\lg\sigma)/(p\lg n)) + L(\lg p + \lg\sigma))$, a communication volume of $\mathcal{O}((n\lg\sigma)/\lg n + \sigma p)$ words and using at most $\max\{M_{localWT}(n/p), \lceil n/p \rceil(\lceil \lg\sigma \rceil + 1) + 2\sigma\lceil \lg n \rceil\}$ bits of local memory.*

Adaptation to the Wavelet Matrix. As described in §2.2, the bit vector layout of the wavelet matrix only differs from that of the wavelet tree in that the nodes are re-ordered on each level according to the bit-reversal permutation. Therefore, we can easily adapt the domain decomposition for wavelet tree construction to construct instead the wavelet matrix. Locally, on each PE, we first construct the wavelet *tree* for the local part of the input. We now only modify the merge operation to take into account the bit-reversal permutation when concatenating node bit vectors on each level. We can compute the bit-reversal of a word of width $\mathcal{O}(\lg n)$ bits in constant time using a universal lookup table of size $o(n)$, though in practice, we use techniques using a constant number of bitwise operations [19, p. 144]. Therefore, Lemma 3 applies to the distributed construction of the wavelet matrix. It remains to compute the z_ℓ values, i.e., the number of 0-bits on each level ℓ. Taking Obs. 2 into account, we can use the histogram to accumulate the sizes of left children in the wavelet tree on level $\ell + 1$, which equals the number of 0-bits on level ℓ. This requires $\mathcal{O}(\sigma)$ time and a subsequent all-to-all sum, not worsening the asymptotic BSP costs.

We note that this modification does not embrace the motivation for which the wavelet matrix was invented, which is the ability to handle large alphabets. Typically, one wants to lose the linear dependency of the alphabet size σ in the algorithm's costs. The following approach allows us to achieve this.

3.2 Stable Sorting. The stable sorting approach [11, 29] constructs the wavelet tree level by level by making use of Obs. 1. Let $c \in \Sigma$ be a symbol from the effective input alphabet with its binary representation $(c)_b$. For $\ell < \lceil \lg \sigma \rceil$, let $(c)_b[\ell]$ be the ℓ-th most significant bit of $(c)_b$. We then call $(c)_b[0..\ell]$ the *ℓ-bit prefix* of $(c)_b$.

We construct level ℓ of the wavelet tree as follows: from the *current text* T_ℓ (with initially $T_0 := T$), we first compute the bit vector B_ℓ with $B_\ell[i] := (T_0[i])_b[\ell]$ for every text position i. If ℓ is not yet the last level, we compute $T_{\ell+1}$ by *stably sorting* the symbols of T_ℓ in ascending order by their $(\ell + 1)$-bit prefixes and proceed with $T_{\ell+1}$. An example is shown in Fig. 6.

The correctness of B_ℓ follows from the fact that the symbols of T_ℓ are in the same order as their corresponding bits in B_ℓ. This is easy to see for $T_0 = T$. Sorting by the bit prefix moves all symbols whose next bit is 0 to the left and those whose next bit is 1 to the right. This corresponds to the path to their representing leaves in the wavelet tree according to Obs. 1. Because the sorting is stable, the relative order of the symbols is retained.

$T_0 =$	w	a	v	e	l	e	t	t	r	e	e
$B_0 =$	1	0	1	0	0	0	1	1	0	0	0
	110	**0**00	**1**01	**0**01	**0**10	**0**01	**1**00	**1**00	**0**11	**0**01	**0**01
	↓ stable sort by first bit										
$T_1 =$	a	e	l	e	r	e	e	w	v	t	t
$B_1 =$	0	0	1	0	1	0	0	1	0	0	0
	000	**00**1	**01**0	**00**1	**01**1	**00**1	**00**1	**11**0	**10**1	**10**0	**10**0
	↓ stable sort by two-bit prefix										
$T_2 =$	a	e	e	e	e	l	r	v	t	t	w
$B_2 =$	0	1	1	1	1	0	1	1	0	0	0

Figure 6: The stable sorting algorithm for $T =$ `wavelettree`. Below the symbols, we show the corresponding bit in the level's bit vector and the symbols' binary representations. The bit prefix used as the sort key is highlighted in bold.

Since this approach reduces most of the wavelet tree construction to stable integer sorting, it can be applied to distributed computing with relative ease using a stable distributed sorting algorithm, which ideally also balances the sorted sequence across the available PEs.

Using the idea of a stable *bucket sort*, we can employ the same techniques already used in the domain decomposition. In the sort operation on level ℓ, there are $2^{\ell+1}$ distinct sort keys: the $(\ell + 1)$-bit prefixes of the symbols. This equals the number of nodes on level $\ell + 1$. Thus, we allocate $2^{\ell+1}$ buckets on each PE and append each symbol to the corresponding local bucket in a distributed left-to-right scan of the text. In order to allocate the buckets, we previously scan the text once and use counters to determine their sizes, which temporarily requires at most $\sigma\lceil \lg(n/p) \rceil$ bits of space. We concatenate the buckets in the same fashion as the nodes of one level in the domain decomposition's merge, which produces the stably sorted representation of $T_{\ell+1}$.

Constructing the level bit vectors and filling the buckets requires $\mathcal{O}((n/p) \lg \sigma)$ time. Furthermore, we need to compute prefix sums of the bucket sizes on each level. Since there are $\mathcal{O}(\sigma)$ buckets in total during the wavelet tree construction, this requires $\mathcal{O}(\sigma \lg p)$ words to be sent and takes $\mathcal{O}(\sigma \lg p)$ time. Furthermore, the

prefix summing requires $\mathcal{O}(\lg p)$ BSP barriers per level, i.e., $\mathcal{O}(\lg p \lg \sigma)$ barriers for all levels. Additionally to prefix sum communication, during the concatenations, we send $\mathcal{O}((n/p)\lg\sigma)$ words (one word per symbol), corresponding to the number of bits in the wavelet tree. We require at most $\lceil n/p \rceil \lceil \lg \sigma \rceil + 2\sigma \lceil \lg n \rceil$ bits of local memory to store the wavelet tree and precomputed node sizes and an additional send/receive buffer of $\lceil n/p \rceil \lceil \lg \sigma \rceil$ bits for the sort buckets. As opposed to domain decomposition, we directly construct the wavelet tree in its distributed levelwise representation and no subsequent merge operation is necessary.

LEMMA 4. *Using distributed stable bucket sorting with p PEs, we can construct the wavelet tree for $T \in \Sigma^n$ with BSP costs of $\mathcal{O}((n/p)\lg\sigma + \sigma \lg p + G(\sigma \lg p + (n/p)\lg\sigma) + L \lg\sigma \lg p)$, a communication volume of $\mathcal{O}(n \lg \sigma + p\sigma)$ words and using at most $2(\lceil n/p \rceil \lceil \lg \sigma \rceil + \sigma \lceil \lg n \rceil) + \sigma \lceil \lg(n/p) \rceil$ bits of local memory.*

Adaptation to the Wavelet Matrix. As mentioned in §3.1, the main idea behind the wavelet matrix is to be able to handle large input alphabets. To this end, we can simplify the bucket sorting approach to construct the wavelet matrix using only a constant number of buckets.

Like in §2.2, we can describe the reordering of nodes in the wavelet matrix in a more intuitive way: on level $\ell + 1$, all left children of the wavelet tree nodes on level ℓ are moved to the left and all right children are moved to the right. Following this idea, in order to bring the symbols of the text in the correct order to compute the following level in the wavelet matrix, on level ℓ, we stably sort them according to their ℓ-th bit only as opposed to the $(\ell+1)$-bit prefix for the wavelet tree. Then, we have only two distinct sort keys (0 and 1) and we can use the bucket sort approach of with merely two sort buckets. These buckets can be filled on the fly as we scan the text to compute the bit vector of level ℓ, which contains precisely the ℓ-th bits of each symbol. The fact that we only have to deal with two buckets allows us to allocate a single buffer of length $\lceil n/p \rceil$ (where both buckets fit in precisely) and insert the symbols with sort key 0 from left to right and those with sort key 1 from right to left in reverse while keeping track of the number of entries in the 0-bucket, which marks the bucket boundaries. Subsequently, we reverse the contents of the 1-bucket in-place, requiring only a computation time linear in the size of that bucket. This way, we do not need a preliminary scan in order to find the bucket sizes. We can furthermore use the tracked number of 0-bits to compute z_ℓ in an all-to-all sum operation.

Using this approach, we lose the linear dependency of σ in our costs: we no longer need to precompute the node sizes in advance and store them, saving us $\mathcal{O}(\sigma)$ time and up to $2\sigma\lceil \lg n \rceil$ bits of local memory, and we only require sizes and prefix sums for two instead of $\mathcal{O}(\sigma)$ buckets on each level. However, we now need to account for $\mathcal{O}(\lg p)$ time and $\mathcal{O}(\lg p)$ words sent for the prefix sums *per level*, i.e., $\mathcal{O}(\lg\sigma \lg p)$ time and $\mathcal{O}(\lg\sigma \lg p)$ words in total. The all-to-all summing operation on each level for computing z_ℓ has the same asymptotic costs as the prefix sum computation.

LEMMA 5. *Using distributed stable bucket sorting with p PEs, we can construct the wavelet matrix for $T \in \Sigma^n$ with BSP costs of $\mathcal{O}((n/p + \lg p)\lg\sigma + G(\lg\sigma \lg p + (n/p)\lg\sigma) + L\lg\sigma\lg p)$, a communication volume of $\mathcal{O}(n\lg\sigma + p\lg\sigma)$ words and using at most $2\lceil n/p \rceil \lceil \lg\sigma \rceil$ bits of local memory.*

3.3 Split. The splitting algorithm follows an idea similar to the recursive parallel split algorithm [20, *recursiveWT*] for shared memory scenarios: given text T of length n over alphabet $[a, b] \subseteq \Sigma$ for wavelet tree node v, we first compute the node's bit vector B_v in parallel using the p available PEs and count the number z of 0-bits in the process. Then, we split up T into T^0 and T^1 according to the bits of B_v: if $B_v[k] = 0$ (for some $k < n$), the symbol $T[k]$ is appended to T^0 and analogously, if $B_v[k] = 1$, $T[k]$ is appended to T^1. We recurse on T^0 for the left child of v using $p\frac{z}{n}$ PEs (but at least one) and on T^1 for the right child using $p\frac{n-z}{n}$ PEs (also at least one). By tying the number of used PEs to the relative number of 0- and 1-bits, we achieve load balancing in that we assign more PEs to larger nodes. Fig. 7 visualizes the splitting operation. In case only one PE remains to process a node, we use a sequential construction algorithm to construct the remaining wavelet subtree.

Although the distribution of node bit vectors is different, the scenario after the construction using splitting is equivalent to that after the local construction phase in domain decomposition, since the bit vectors remain in PE rank order. Therefore, we can use the same merge operation to retrieve the levelwise representation, for

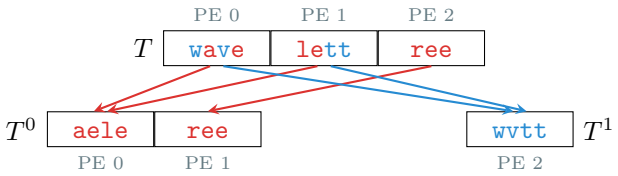

Figure 7: Split of the first level of our running example $T = \texttt{wavelettree}$ using three PEs. Initially, each PE i has part T_i and decides which symbols go to T^0 and which go to T^1. The symbols are communicated accordingly.

which we can reuse the precomputed node sizes as well as the memory allocated for send/receive buffers during the splitting phase.

The BSP cost analysis for this algorithm is complex since whenever only one PE remains to compute a wavelet subtree of height greater than one, we switch to sequential construction. From here on, the PE no longer communicates and the local computation depends on the chosen sequential algorithm. Whether and how often this case occurs during construction depends on the distribution of the symbols in the input and thus, so do the BSP costs. For this reason, we refrain from a detailed analysis and consider only a favorable case.

We call a text *uniform* if all the symbols in it occur equally often. We now assume that the input T is uniform, that σ is a power of two and p a multiple of σ. Then, T^0 and T^1 are always of the same length $n/2$ in every recursion and we recurse on both using $p/2$ PEs each.

The lengths of T^0 and T^1 equal the sizes of the left and right child of v, respectively. Therefore, the communication of the symbols of T^0 and T^1 can be done similarly to the bits in the domain decomposition's merge operation with all nodes on the same level being processed in parallel. For this, we need to communicate $O(\sigma \lg p)$ words for prefix sums and $O((n/p)\lg\sigma)$ words for the split texts themselves. It again holds that each text part needs to be sent to at most two different recipients. Since this applies to both T^0 and T^1, we have up to four recipients, which, however, has no effects asymptotically. The splits thus take $\mathcal{O}((n/p)\lg\sigma + \sigma\lg p)$ time. To store the partial wavelet tree and precomputed node sizes, we require at most $\lceil n/p\rceil\lceil\lg\sigma\rceil + 2\sigma\lceil\lg n\rceil$ bits of memory. The two prefix sums (for T^0 and T^1) and z can be stored in negligible $\lceil\lg n\rceil$ additional bits and finally, we need to store T^0 and T^1 as well as send/receive buffers in additional $2\lceil n/p\rceil\lceil\lg\sigma\rceil$ bits of local memory. As mentioned previously, we merge the partial wavelet trees using the same operation as for the domain decomposition.

PROPOSITION 1. *For a uniform text T of length n over an alphabet of size $\sigma = 2^k$ for some integer $k > 0$, we can construct the wavelet tree using splitting with $p = c\sigma$ PEs for some integer $c > 0$ with BSP costs of $\mathcal{O}((n/p)\lg\sigma + \sigma\lg p + G((n/p)\lg\sigma + \sigma\lg p) + L(\lg\sigma + \lg p + \lg\sigma\lg p))$, a communication volume of $\mathcal{O}(n\lg\sigma + \sigma p)$ words and using at most $3\lceil n/p\rceil\lceil\lg\sigma\rceil + 2\sigma\lceil\lg n\rceil$ bits of local memory.*

Adaptation to the Wavelet Matrix. Since the scenario after the construction using splitting is the same as that after local construction in the domain decomposition, we can apply the same idea as in §3.1 to construct the wavelet matrix by simply altering the communication pattern of the merge operation with respect to the bit-reversal permutation.

4 Experiments

We implemented our distributed wavelet tree and matrix construction algorithms and conduct two types of scaling experiments in a cluster: a *weak scaling* experiment, where we increase the size of the input as we increase the number of PEs, and a *breakdown* experiment where we increase the size of the input while keeping the number of PEs fixed. To address the issue that the speedups achieved by distributed memory computations may not be justified in regard of the extra resources required for these computations (as made aware by the COST model [23]), we furthermore identify the number of nodes required to outperform the best known sequential algorithm on a single PE, as well as that required to outperform the best known shared memory algorithm on a single node.

4.1 Experimental Setup.

We conduct our evaluation on a cluster of nodes, each equipped with two Intel Xeon E5-2640v4 processors (20 PEs in per node) running at 2.4 GHz with 25 MB of L3 cache and 64 GB of RAM. The cluster's nodes have hyperthreading disabled and it cannot be activated by users. They are connected via InfiniBand QDR (40 Gbit /s) with a blocking ratio of 1:3. We construct wavelet trees and matrices for five texts over different alphabets:

1. CC – a collection of texts from the Common Crawl (`http://commoncrawl.org`) with $\sigma = 242$,

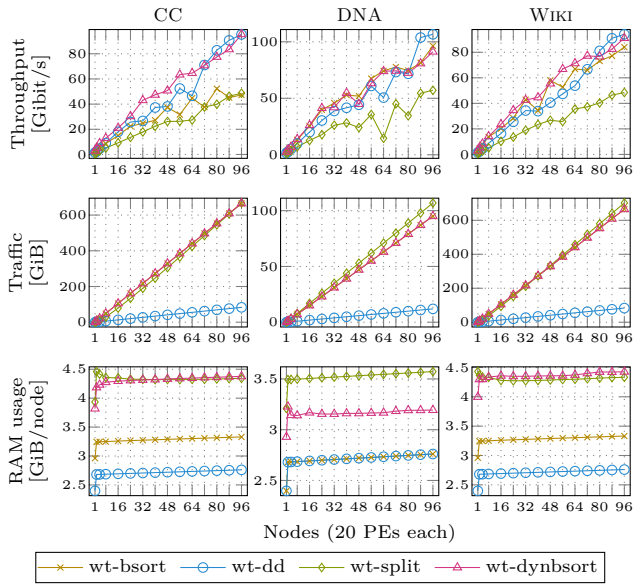

Figure 8: Weak scaling results for wavelet tree construction with 1 GiB of input per node.

2. DNA – a sequence from the 1000 Genomes project (`http://www.internationalgenome.org`) with FASTQ information removed such that $\sigma = 4$,
3. Wiki – a Wikimedia dump in XML format of English, Finnish, French, German, Italian, Polish and Spanish Wikipedia articles (`https://dumps.wikimedia.org`) with $\sigma = 213$,
4. RuWB – Russian websites in Common Crawl (`http://web-language-models.s3-website-us-east-1.amazonaws.com/wmt16/deduped/ru.xz`) over a word-based alphabet with $\sigma = 152{,}547{,}239$, and
5. SA – the suffix array [22] of the 128 GiB prefix of CC with $\sigma = 137{,}438{,}953{,}472$ (the suffix array is a permutation, so each symbol is unique).

The texts over small alphabets (CC, DNA and Wiki) are stored in plain ASCII format with one byte per input symbol. For these texts, we first compute the effective transformation—still encoding it using one byte per symbol. The time required for this is not measured in our evaluation. The texts over large alphabets, RuWB and SA, are already stored in their effective transformation with four or five bytes per symbol, respectively.

Implementation. We provide the implementations *wt-dd* (domain decomposition of §3.1) and *wm-dd* (the wavelet matrix variant), *wt-split* and *wm-split* (split of §3.3), and finally *wt-bsort* and *wm-concat* (bucket sorting of §3.2). In the variant *wt-dynbsort* of *wt-bsort*, we grow buckets dynamically while constructing the level's bit vector instead of precomputing their sizes in a preliminary scan. This causes more memory allocations, but the skipped scan results in notably faster running times.

For local computations in the domain decomposition (for local wavelet tree construction) and split (in case only one PE remains to construct a wavelet subtree), we pick *wt_pc* [11], because it is the fastest practical sequential wavelet tree construction algorithm known to us.

We use MPI [24] for communication between PEs and specifically their point-to-point send (non-blocking, e.g., `MPI_Isend`) and probe/receive (blocking, e.g., `MPI_Probe`/`MPI_Recv`) operations. Where applicable, we make use of MPI's implementations of collective operations like synchronization (`MPI_Barrier`), exclusive prefix summing (`MPI_ExScan`) and all-to-all reduction (`MPI_Allreduce`). Furthermore, we use MPI's concept of communicators to group nodes constructing the same wavelet subtree in *wt-split*.

The source code is written in C++ and publicly available at `https://github.com/pdinklag/distwt` (the repository also contains implementations using the Thrill framework [1], which could not achieve competitive throughputs and are not covered in this work). We compile using the GNU `g++` compiler version 7.3.0 and link against the Intel MPI Library version 2018.3.

Performance Measurements. Throughout the experiments, we measure three major performance figures: (1) *running time*, (2) *network traffic* and (3) *memory usage*. Time and memory usage are local figures that we measure using wall clock times or counting memory allocations, respectively. For measuring traffic, we build a façade for all MPI operations that we use and estimate their inter-node traffic, i.e., we do not count communication between PEs located on the same node. The estimation is straightforward for the primitve send and receive operations, where we can simply count the size of each passed message. For the collectives (prefix summing and all-to-all sums), since we do not know details about the MPI implementations, we assume a binary merge tree communication pattern as mentioned in §2.4. We believe that this estimation is somewhat pessimistic in the worst case and probably exceeds the actual traffic caused by these operations. However, since they only make up for

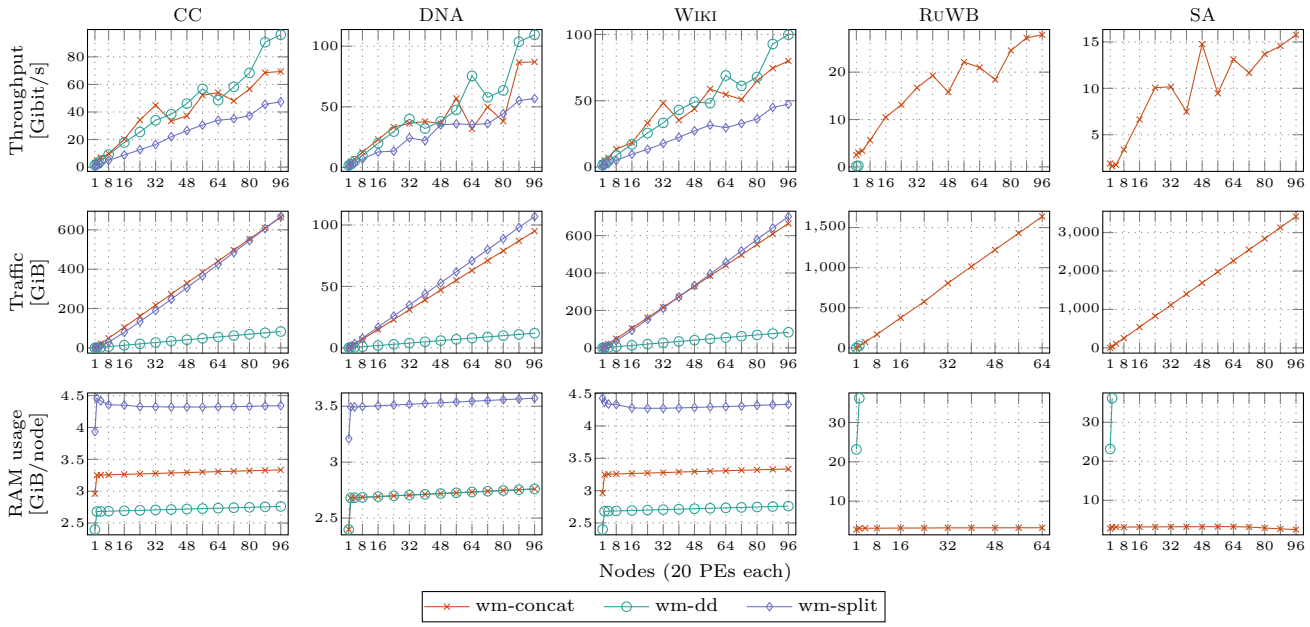

Figure 9: Weak scaling results for wavelet matrix construction with 1 GiB of input per node.

a small percentage of the total traffic (compared to the larger payloads such as (bit-)strings), we consider this potential excess negligible.

4.2 Weak Scaling Results. We conduct weak scaling experiments where for N nodes, i.e., $p = 20N$, we process a prefix of size $n := N \cdot \text{GiB}$ (one gibibyte per node) of each input file T, so that PE i initially has input part $T[i\lceil n/p\rceil \,..\, (i+1)\lceil n/p\rceil - 1]$. It is crucial to note that, for the large alphabet inputs RuWB and SA, taking a longer prefix also results in the input alphabet becoming larger. We show

(1) the median *throughput* (the number of output bits divided by the running time) over five iterations of the histogram computation and construction of the wavelet tree,
(2) the total *network traffic* caused by all nodes (excl. traffic between PEs on the same node) and
(3) the average *memory usage* (RAM) per node.

We start timing as soon as the input is loaded into memory on all PEs. The results for wavelet trees are given in Fig. 8 and those for wavelet matrices in Fig. 9. In the following, we first consider the results for the small alphabet inputs (CC, DNA and Wiki) and then look at the large alphabet inputs (RuWB and SA) afterwards.

Throughput. We see that all implementations scale well with an increasing number of nodes. To exemplify this, on CC, *wt-dd* achieves a median throughput of 1.42 Gibit /s using one node on a 1 GiB prefix and 95.32 Gibit /s using 96 nodes on a 96 GiB prefix (speedup of about 67).

The highest throughputs are achieved by *wt-dynbsort* (up to 95.47 Gibit /s on CC) and *wt-dd* (up to 95.32 Gibit /s on CC), where *wt-dynbsort* seems to be slightly faster overall (up to 61.5 % on CC at 32 nodes). Their similarity in throughputs is notable and reflected by the fact that *wt-dynbsort* can be seen as just the merge operation of *wt-dd* with some slight differences: (1) *wt–dynbsort* requires no preliminary local construction of the wavelet tree and (2) communication is more local in *wt-dynbsort*: when sorting by an $(\ell+1)$-bit prefix, the order only changes within blocks of symbols whose ℓ-bit prefix was the same, i.e., buckets get refined in each sort operation and the distance between PEs that any symbol is sent over becomes smaller on each level. This results in increasing fast shared memory communication of PEs on the same node, which is an advantage over *wt-dd*. However, (3) in *wt-dd*'s merge, we communicate bit vectors and pack eight bits into one byte, while we need to communicate substrings to concatenate buckets in *wt-dynbsort*. These occupy one byte per symbol (for

the small alphabets) and thus increase the amount of handled data by a factor of eight (or a multiple of eight for larger alphabets).

The analysis is also valid for *wt-bsort*. Here, however, the additional scan to determine the bucket sizes in advance causes the throughput to plummet (half as low as *wt-dynbsort* on CC at 96 nodes). On DNA, on the other hand, we only do a single concatenation of two buckets after constructing the first level and the throughputs of *wt-dynbsort* and *wt-bsort* become nearly equal.

The fact that *wt-split* fares worst (up to 60 % lower throughput than wt-dynbsort on CC at 56 nodes) is not surprising, as it combines the two expensive operations of *wt-dynbsort* and *wt-dd*: the split texts T^0 and T^1, for a whole level, directly correspond to sorted buckets and after construction, we need to perform a merge operation.

We observe similar throughputs for our the wavelet matrix constructors. However, it is notable that even though *wm-concat* is closely related to the wavelet tree constructor *wt-dynbsort*, it achieves only lower throughputs in comparison. The reason for this lies in the fact that we only concatenate two buckets for each level of the wavelet matrix. Other than in the wavelet tree scenario, these buckets are arbitrarily scattered over the PEs, so that no locality advantage comes into play at lower levels. This issue is again less severe for DNA, where we only construct two levels.

Traffic. Comparing the traffic of *wt-dynbsort* and *wt-dd*, the factor of eight in the amount of communicated data (symbols instead of bits) becomes very visible: on CC at 96 nodes, we have a traffic of approximately 83 GiB for *wt-dd* and 665 GiB for *wt-dynbsort* (factor 8.01). We can also make out the equivalence of the split texts communicated in *wt-split* to the buckets of *wt-dynbsort*, causing very similar traffic footprints. Slight differences occur as *wt-split* requires a merge operation after construction and *wt-dynbsort* does not. However, it should be noted that *wt-split* has even better data locality on deeper levels than *wt-dynbsort*, because T^0 and T^1 are guaranteed to stay within the same node range. This ultimately results in more shared memory traffic, which we do not measure. The same observations apply to the wavelet matrix constructors, however, with the extra issues concerning *wm-concat* described above. The traffic footprints of *wt-dynbsort* and *wt-bsort* are exactly

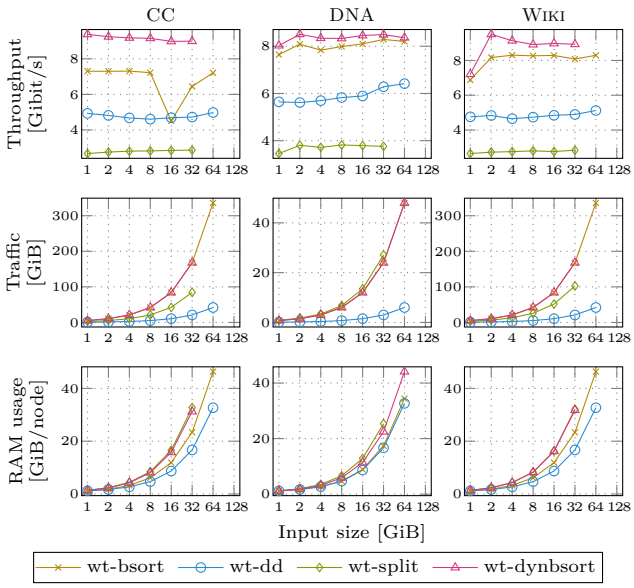

Figure 10: Breakdown test results for wavelet tree construction with four nodes.

equal: they only differ in local memory consumption; the communicated data is the same.

RAM usage. Due to the low memory requirements of *wt_pc*, *wt-dd* requires the lowest amount of RAM overall. For the merge operation after construction, we only need *one* buffer for sending and receiving wavelet tree levels (bit vectors) as we merge level by level. We see that the difference in memory usage of *wt-bsort* and *wt-split* is precisely the memory required by the subsequent merge operation in *wt-split*, i.e., that of *wt-dd*, matching our theoretical analysis in Sect. 3.3. The difference between *wt-dynbsort* and *wt-bsort* is the excess memory allocated by *wt-dynbsort* when doubling bucket capacities as they are filled (up to 33 % excess memory on CC at two nodes).

Similar observations apply for the wavelet matrix constructors. Since *wm-concat* only needs two sort buckets, which we manage in the same buffer (filling one from the left and the other from the right), no dynamic re-allocation is needed and thus the memory footprint is better than that of *wm-split* (about 25 % less memory on CC at two nodes). Still, the fact that *wm-dd* only requires one buffer for merging bit vectors causes it to achieve the lowest RAM usage for small alphabets.

Large Alphabet Inputs. We note that we only have roughly 69 GiB of RuWB available and therefore did not

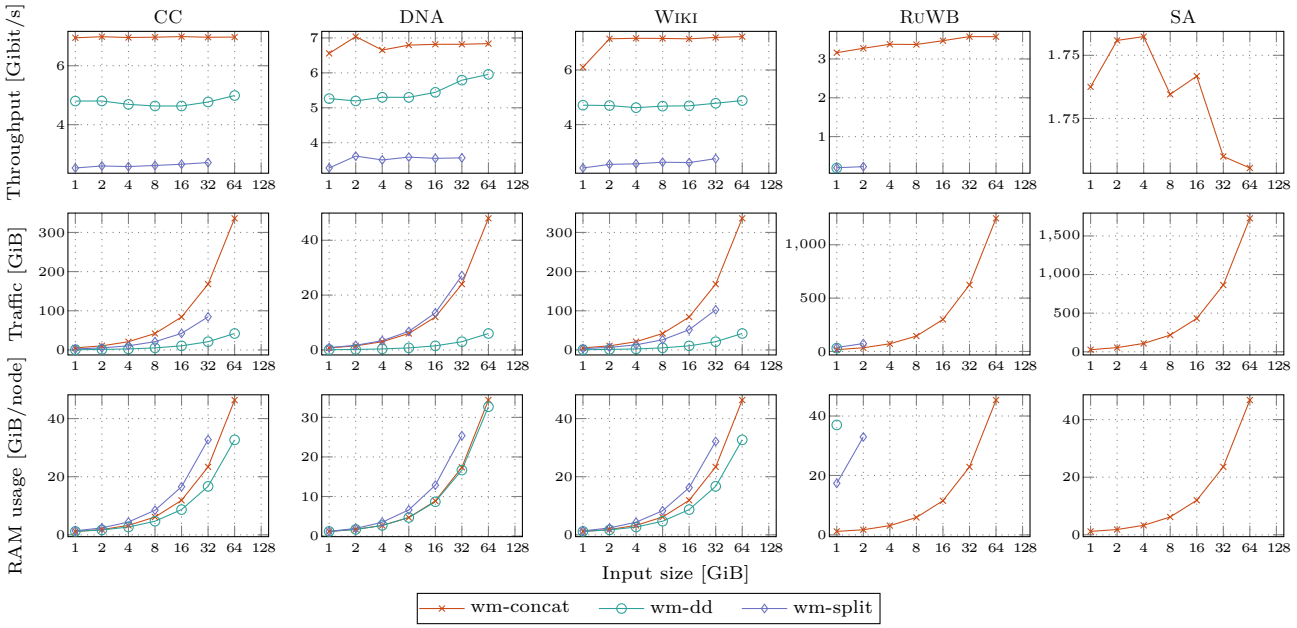

Figure 11: Breakdown test results for wavelet matrix construction with four nodes.

do any experiments for this input beyond 64 nodes.

The algorithms with a linear dependency on the input alphabet size, i.e., all except *wm-concat*, fail to process most prefixes of RuWB and SA due to RAM limitations. The only algorithm able to process all instances is *wm-concat* with a near-constant memory footprint (2.49 GiB on average per node on SA at 96 nodes, i.e., 2.49 times the local input size) thanks to the constant number of required buckets. No wavelet tree constructor processed any prefix of SA successfully and only *wt-split* could process small instances of RuWB (omitted in Fig. 8).

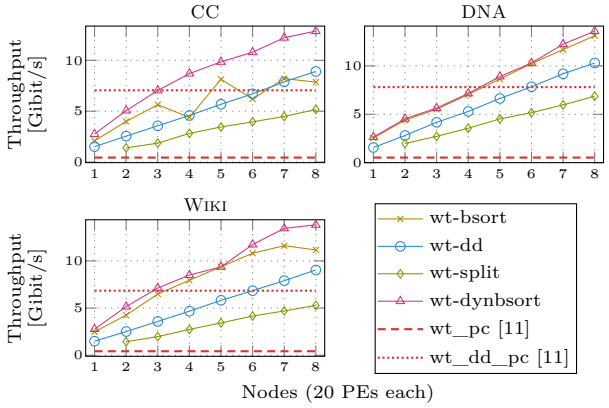

Figure 12: COST [23] of wavelet tree construction for 16 GiB inputs.

4.3 Breakdown Test Results. In our breakdown test, we fix the number of nodes to four ($p = 80$) and double the length n of the processed prefix in each iteration, starting with 1 GiB and stopping where none of our implementations succeeds any longer. We measure the same values as in the weak scaling experiments and give the results in Fig. 10 and Fig. 11.

The measured traffic and RAM are as expected considering the weak scaling results. Correspondingly, *wt-dd* and *wt-bsort* for the wavelet tree and *wm-dd* for the matrix have the lowest memory footprints and can therefore handle the largest inputs up to $2^{36} = 64$ GiB for the small alphabets, i.e., up to 16 GiB per node. Regarding the large alphabets, *wm-concat* is again the only implementation that can successfully handle similarly sized instances. The throughputs remain roughly constant throughout all breakdown experiments (decreasing by at most 15 % for *wt-bsort* on CC), so the overhead caused by additional distribution is small.

4.4 Outperforming Single Threads and Nodes. We conclude the experiments by finding the *configuration that outperforms a single thread*, also known as COST [23], which is the number of nodes required to achieve a higher throughput than the best known sequential algorithm. We extend this notion and also look for the configuration

that outperforms the best known parallel shared memory algorithm using PEs on a single node. To this end, we compare to *wt_pc* as the single-threaded implementation and to *wt_dd_pc* (the domain decomposition variant of *wt_pc*) as the parallel shared memory implementation, the two that achieved the highest throughputs [11]. We fix the input prefix length to 16 GiB, which is the largest that *wt_pc* and *wt_dd_pc* can successfully process in our setup. For our distributed implementations, we increase the number of nodes until we outperform both *wt_pc* and *wt_dd_pc*, see Fig. 12. For wavelet matrix construction, we do the same until we outperform *wm_pc* and *wm_dd_pc*, see Fig. 13.

We note that *wt-split* failed for all inputs on a single node due to RAM limitations. Apart from this, all of our other implementations outperform the single-threaded *wt_pc* on only one node, where further experiments have shown that two PEs on that node already suffice. To that regard, the COST [23] of our distributed algorithms is very low. Outperforming the single-node parallel shared memory implementations requires between three (wavelet tree for WIKI) and five nodes (tree and matrix for DNA).

Fig. 14 shows the per-node memory usages for this experiment, which are the same for *wt_pc* and *wt_dd_pc* for all inputs. As one would expect, *wt-dd* requires the least memory and has the same footprint as *wt_pc*, as it uses the same algorithm for local construction. However, in the case of DNA, *wt-dd* uses excess memory to store each input symbol in a byte each rather than compressing it to two bits. The bucket sorters, *wt-bsort* and *wt-dynbsort*, require additional memory for the sort buckets, which have a similar memory footprint to the split texts in *wt-split*. Our implementations require less per-node memory than the sequential and shared memory implementations starting with two nodes (CC and WIKI) to three nodes (DNA), which does not impact our low COST given above. Due to high similarity of the results (also studied in [11]), we omit the per-node memory usage analysis for the wavelet matrix constructors.

5 Conclusions

We implemented and evaluated the first distributed wavelet tree and matrix construction algorithms. The approach that works best for small alphabets is the *domain decomposition*, however, we also gave an algorithm based on bucket sorting that is able to handle large alphabets.

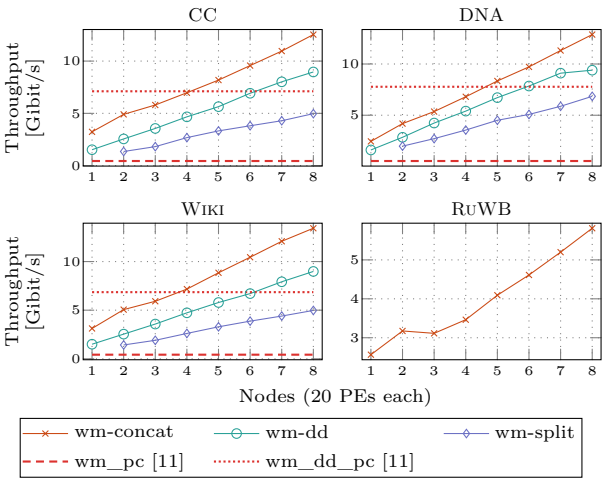

Figure 13: COST [23] of wavelet matrix construction for 16 GiB inputs.

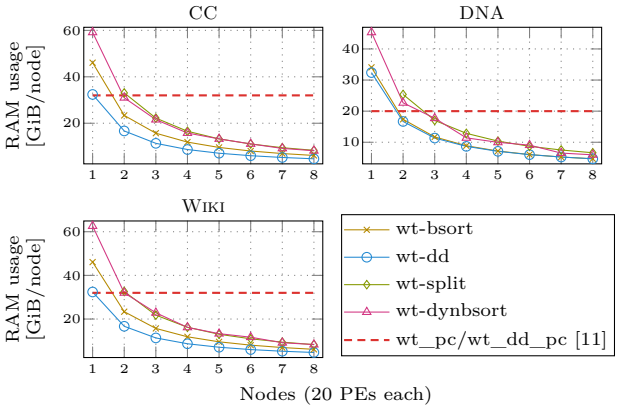

Figure 14: RAM usage in COST [23] experiments for 16 GiB inputs.

As an anonymous reviewer pointed out, an interesting question arising from this work is how hybrids of shared and distributed memory approaches fare in comparison to those presented in this and previous works. For instance, one could use domain decomposition in a way that each *node* uses a shared memory algorithm to compute a local wavelet tree, as opposed to each PE using a sequential algorithm. We are planning to investigate this in the future. Furthermore, in future work, we are going to look into the distributed computation of the Huffman-shaped wavelet tree and matrix [34] where the fact that it is no longer balanced poses additional challenges. Finally, it remains open how to efficiently implement queries using a distributed wavelet tree or matrix.

References

[1] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. Thrill: High-performance algorithmic distributed batch data processing with C++. In *IEEE International Conference on Big Data*, pages 172–183. IEEE, 2016.

[2] Timo Bingmann, Simon Gog, and Florian Kurpicz. Scalable construction of text indexes with thrill. In *2018 IEEE International Conference on Big Data*, pages 634–643. IEEE, 2018.

[3] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[4] Francisco Claude, Patrick K. Nicholson, and Diego Seco. Space efficient wavelet tree construction. In *18th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 185–196. Springer, 2011.

[5] Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Syst.*, 47: 15–32, 2015.

[6] Jonas Ellert and Florian Kurpicz. Parallel external memory wavelet tree and wavelet matrix construction. In *26th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 392–406. Springer, 2019.

[7] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Symposium on Foundations of Computer Science (FOCS)*, pages 390–398. IEEE, 2000.

[8] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of wavelet trees. *Inform. and Comput.*, 207(8):849–866, 2009.

[9] Johannes Fischer and Florian Kurpicz. Lightweight distributed suffix array construction. In *21st Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 27–38. SIAM, 2019.

[10] Johannes Fischer, Florian Kurpicz, and Peter Sanders. Engineering a distributed full-text index. In *19th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 120–134. SIAM, 2017.

[11] Johannes Fischer, Florian Kurpicz, and Marvin Löbel. Simple, fast and lightweight parallel wavelet tree construction. In *20th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 9–20. SIAM, 2018.

[12] Patrick Flick and Srinivas Aluru. Parallel distributed memory construction of suffix and longest common prefix arrays. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 16:1–16:10. ACM, 2015.

[13] 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.

[14] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850. SIAM, 2003.

[15] Roberto Grossi, Jeffrey Scott Vitter, and Bojian Xu. Wavelet trees: From theory to practice. In *First International Conference on Data Compression, Communications and Processing (CCP)*, pages 210–221. IEEE, 2011.

[16] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992. ISBN 0-201-54856-9.

[17] Yusaku Kaneta. Fast wavelet tree construction in practice. In *25th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 218–232. Springer, 2018.

[18] Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 756–767. ACM, 2019.

[19] Donald E. Knuth. *The Art of Computer Programming: Combinatorial Algorithms, Part 1.* Addison-Wesley Professional, 2011. ISBN 0-201-03804-8.

[20] 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.

[21] Veli Mäkinen and Gonzalo Navarro. Position-restricted substring searching. In *7th Latin American Theoretical Informatics Symposium (LATIN)*, volume 3887 of *Lecture Notes in Computer Science*, pages 703–714. Springer, 2006.

[22] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[23] Frank McSherry, Michael Isard, and Derek Gordon Murray. Scalability! But at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS)*. USENIX Association, 2015.

[24] Message Passing Interface Forum. MPI: A message-passing interface standard, 1994. `https://www.mpi-forum.org`.

[25] Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014.

[26] Gonzalo Navarro. *Compact Data Structures - A Practical Approach.* Cambridge University Press, 2016. ISBN 978-1-10-715238-0.

[27] Rolf Rabenseifner. Optimization of collective reduction operations. In *4th International Conference on Computational Science (ICCS)*, pages 1–9. Springer, 2004.

[28] Sherif Sakr and Albert Y. Zomaya, editors. *Encyclopedia of Big Data Technologies*, pages 381–381. Springer, 2019. ISBN 978-3-319-63962-8.

[29] Julian Shun. Parallel wavelet tree construction. In *2015 Data Compression Conference (DCC)*, pages 63–72. IEEE, 2015.

[30] Julian Shun. Improved parallel construction of wavelet trees and rank/select structures. In *2017 Data Compression Conference (DCC)*, pages 92–101. IEEE, 2017.

[31] Thomas N. Theis and H.-S. Philip Wong. The end of moore's law: A new beginning for information technology. *Comput. Sci. Eng.*, 19(2):41–50, 2017.

[32] German Tischler. On wavelet tree construction. In *22nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 208–218. Springer, 2011.

[33] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[34] Yi Zhang, Zhili Pei, Jinhui Yang, and Yanchun Liang. Canonical Huffman code based full-text index. *Progr. Natur. Sci.*, 18(3):325–330, 2008.