technische universität
dortmund

Master's Thesis

Distributed Wavelet Tree Construction

Patrick Dinklage
March 28, 2019

Reviewers:
Prof. Dr. Johannes Fischer
M. Sc. Florian Kurpicz

TU Dortmund University
Department of Computer Science
Chair 11: Algorithm Engineering
http://ls11-www.cs.tu-dortmund.de

Contents

1 Introduction

The *wavelet tree* is a flexible data structure with numerous applications in text indexing, data compression and other areas [9, 23]. Ever since being introduced as a full-text index by Grossi et al. [13], there have been myriad approaches on improving the time and space requirements for its construction [6, 7, 14, 22, 34] as well as transforming it for faster practical querying (e.g., the wavelet matrix [5]).

Related work and motivation. In recent work, quite some focus has been laid on the parallel construction of wavelet trees in multi-core scenarios with shared memory [10, 19, 29, 30, 31, 32]. However, all existing construction algorithms share a common limitation: albeit being able to use multiple cores, they are bound to work on a single machine and hence are limited by that machine's hardware. While the amount of data being processed is typically becoming larger (*big data*), Moore's Law is nearing its physical boundaries [33] and the computational power that a single machine can achieve is limited.

This development leads to the idea of distributing work across multiple, physically separate machines that communicate over a network. For this scenario, a different view on algorithms is required that is commonly referred to as *distributed computing*. Thus far, no distributed algorithm for the construction of wavelet trees has been presented yet.

Goals and challenges. The main goal of this thesis is to develop, analyze, implement and practically evaluate the first distributed algorithms for the construction of wavelet trees. The practical evaluation shall be conducted on large inputs in the order of tens to hundreds of gigabytes to demonstrate the practical use of distributed computing.

The key challenge is the fact that, unlike in sequential or shared memory scenarios, randomly accessing parts of the input or the data structure during its construction is not feasible. Typically, in distributed computing, each processing element has a view on only a slice of the input and merely constructs a local part of the global data structure. In order to exchange information with other processing elements, e.g., in order to merge partial results into a final result, this information needs to be communicated. Communication, however, may be expensive — due to limited bandwidth of the network for example, but also because the mere transmission and reception of messages causes computational overhead in itself. Therefore, the amount of communicated data becomes a key measure for the overall performance of a distributed algorithm.

Thrill and MPI. The distributed wavelet tree construction algorithms developed in this thesis were originally to be implemented using the Thrill framework [3] only. We extended this goal by implementations against the message-passing interface (MPI) [21] for two reasons: on one hand, the implementations in Thrill were surprisingly easy to realize thanks to its high-level interface design. On the other hand, *because* of the high abstraction level, some ideas could not be implemented using Thrill.

Structure. In chapter 2, following some basic definitions required throughout the thesis, we take a close look at *wavelet trees* and their construction in sequential as well as parallel, shared memory computing. We then introduce the basic concepts and caveats of *distributed computing* and a computation model for the analysis of distributed algorithms and take a look at the features that MPI and Thrill provide to implement these algorithms.

In the main part of the thesis, we develop and analyze algorithms for the distributed construction of wavelet trees (chapter 3), document their implementation (chapter 4) and perform a practical evaluation (chapter 5).

Wavelet tree vs. wavelet matrix construction. In chapter 6, we additionally deal with a theoretical problem concerning wavelet trees and wavelet matrices that is not directly related to distributed computing. In said chapter, we continue the research of Fischer et al. [10] on whether construction algorithms for the wavelet tree can be used to construct instead the wavelet matrix and vice versa without worsening the asymptotic time and space boundaries. This was an optional goal of this thesis that has been reached, albeit with limitations.

2 Preliminaries

In this chapter, we define the basic terms, data structures and algorithms on which this thesis is based. Furthermore, we describe the Thrill framework and MPI, which we use for the implementations presented in chapter 4.

2.1 Basic Definitions

Let $T \in \Sigma^n$ be a text over an alphabet Σ and let $i, j \in [0, n)$. We denote the i-th symbol of T by $T[i]$, and by $T[i \dots j]$ we denote the substring of T starting at position i and ending after position j. For a symbol $c \in \Sigma$, we denote by $\mathrm{occ}_T(c)$ the number of occurrences of c in T. The *histogram* $H : c \mapsto \mathrm{occ}_T(c)$ maps each symbol to the number of its occurrences. A *bit vector* is a text over the *binary alphabet* $\mathbb{B} = \{0, 1\}$.

Notation. Array indices are zero-based, i.e., the first item of an array A of length n is denoted by $A[0]$ and the last by $A[n-1]$. In algorithm listings, we use common bitwise operators as listed in Table 1.

Symbol	Operator	Example
$\ll$	Bitwise shift to the left	$001_b \ll 2 = 100_b$
$\gg$	Bitwise shift to the right	$100_b \gg 2 = 001_b$
OR	Bitwise OR	100_b OR $001_b = 101_b$
AND	Bitwise AND	100_b OR $101_b = 100_b$

Table 1: Bitwise operators used in algorithm listings.

In the context of binary trees, we refer to nodes using their *BFS rank*, i.e., their zero-based rank in a breadth-first traversal of the tree. To that end, the root node has BFS rank 0 and given a node v, the left child has BFS rank $2(v+1)-1$ and the right child has BFS rank $2(v+1)$.

2.1.1 Effective Transformation

The *effective alphabet* of T is the set of those symbols $c \in \Sigma$ with $\mathrm{occ}_T(c) > 0$. We denote its size by σ and represent it as the interval $[0, \sigma)$, where the lexicographically smallest symbol is represented by 0, the second smallest by 1 and so forth, until the lexicographically largest symbol, which is represented by $\sigma - 1$.

We define the *effective transformation* T_{eff} of T as follows: for an $i \in [0, n)$, let k be the lexicographic rank of the symbol at $T[i]$, i.e., it is the lexicographically k-smallest symbol in the effective alphabet. Then $T_{\text{eff}}[i] \coloneqq k$.

To give an example, let $T = \texttt{effective}$. The underlying alphabet is $\Sigma = \{\texttt{c}, \texttt{e}, \texttt{f}, \texttt{i}, \texttt{t}, \texttt{v}\}$ and consists of $\sigma = 6$ distinct symbol, i.e., we can represent it as the effective alphabet $[0, 6)$. Mapping each symbol of Σ to its lexicographic rank ($\texttt{c} \mapsto 0$, $\texttt{e} \mapsto 1$, etc.), we receive the effective transformation $T_{\text{eff}} = 1\,2\,2\,1\,0\,4\,3\,5\,1$.

2.1.2 Operations on Sequences

A tuple $(a_0, \ldots, a_{n-1}) \in A^n$ over a base set A can be viewed as a *sequence* of n elements from A ordered by their indices $i \in [0, n)$. In Table 2, we define several operations on sequences that are used throughout this thesis.

Operation / Signature	Description
Concat $A^n \times A^m \to A^{n+m}$	Concatenates two input sequences to form a new sequence, retaining the order of the items.
Filter $A^n \times (A \to \mathbb{B}) \to A^m$	Filters those m elements from the input sequence on which the given predicate function $p : A \to \mathbb{B}$ yields $p(a) = 1$ and places them (in input order) into the output sequence.
Map $A^n \times (A \to B) \to B^n$	Applies the mapping function $m : A \to B$ on each item from the input sequence in order and returns the sequence $(m(a_0), \ldots, m(a_{n-1}))$.
PrefixSum $A^n \times \mathbb{N} \times (A \to \mathbb{N}) \to \mathbb{N}$	Given a key mapping $k : A \to \mathbb{N}$, computes the sum $\sum_{i=0}^{m-1} k(a_i)$ of the keys of the first $m \leq n$ items (prefix) of the sequence.
Sort $A^n \times (A \times A \to \mathbb{B}) \to A^n$	Sorts the input sequence according to the given comparison function, which, for any two $a_1, a_2 \in A$, returns 1 if a_1 is strictly less than a_2 in the corresponding order and 0 otherwise.
SortStable $A^n \times (A \times A \to \mathbb{B}) \to A^n$	Sorts the input sequence, retaining the relative order of elements that are equal in terms of the comparison function. Thus, the sorting is stable.
Zip $A^n \times B^n \times (A \times B \to C) \to C^n$	Combines two sequences into a new sequence by applying the zipping function $z : A \times B \to C$ pairwise on two elements from the respective input sequences in their order, i.e., it computes the sequence $(z(a_0, b_0), \ldots, z(a_{n-1}, b_{n-1}))$. This requires the input sequences to be of the same size.

Table 2: Operations on sequences used in this thesis.

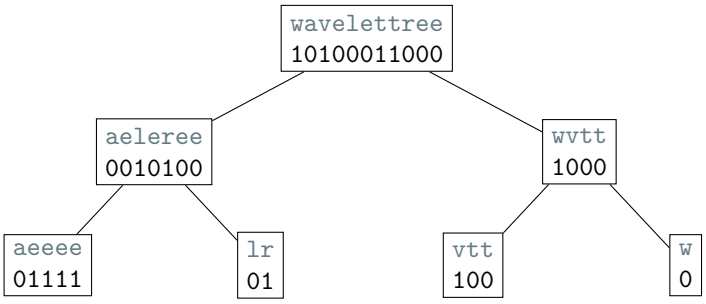


Figure 1: The wavelet tree for the input text $T =$ `wavelettree`. The texts above the bit vectors are shown only for comprehensibility; they are not a part of the node labels and are not stored. The last level of the wavelet tree would consist solely of leaves and is therefore omitted. Even though the node for `w` is also a leaf, we keep it so the number of bits on each level equals the input length n, a property that we are going to use at various points in the thesis.

2.1.3 Rank and Select on Bit Vectors

Let $B \in \mathbb{B}^n$ be a bit vector of length n. For an $i \in [0, n)$, the operation $\text{rank}_1(B, i)$ computes the amount of 1-bits contained in the substring $B[0, i]$. Correspondingly, for any $k \in [1, \text{occ}(T, 1)]$, the operation $\text{select}_1(B, k)$ computes the position i of the k-th occurrence of a 1-bit in B. We define $\text{rank}_0(B, i)$ and $\text{select}_0(B, k)$ analogously.

If B is static, we can construct a data structure that answers rank and select queries on B in constant time and requires $o(n)$ bits of additional space [25].

2.2 Wavelet Trees

For a text $T \in \Sigma^n$, the *wavelet tree* is a binary tree of height $\lceil \log \sigma \rceil$, where σ is the size of the effective alphabet of T. Each node v represents an interval $[a, b] \subseteq [0, \sigma)$ of the effective alphabet and contains a bit vector $B_v \in \mathbb{B}^*$, which has one bit for those text positions i, in text order, with $T[i] \in [a, b]$: a 0-bit if $T_{\text{eff}}[i] \leq \lfloor \frac{a+b}{2} \rfloor$, i.e., if the symbol $T[i]$ lies in the *left half* of the alphabet interval, and analogously a 1-bit otherwise.

The root node represents the entire effective alphabet $[0, \sigma)$. Therefore, its bit vector has length n, because every symbol in T in contained in the effective alphabet. A node v has two children iff $a < b$. We apply the described structure recursively for the left child to represent $[a, \lfloor \frac{a+b}{2} \rfloor]$ (the *left half*) and the right child to represent $[\lfloor \frac{a+b}{2} \rfloor + 1, b]$ (the *right half*). Following that, the tree's leaves are those nodes that represent an interval of size one, i.e., precisely one symbol from the input alphabet ($a = b$). Since a leaf's bit vector contains only zero-bits, it needs not be stored. Figure 1 shows an example for a wavelet tree.

Bounds. The number of nodes in the wavelet tree is bounded by σ: if σ is a power of two, the wavelet tree is a full binary tree of height $\log \sigma$, i.e., then it has precisely $2^{\log \sigma} - 1 = \sigma - 1$ nodes. If σ is not a power of two, the number of nodes is less than $\sigma - 1$.

The concatenation of all bit vectors on any level of the wavelet tree amounts to at most n bits. Therefore, because the tree has height $\lceil \log \sigma \rceil$, the bit vectors consume at most $n \lceil \log \sigma \rceil$ bits in total.

Queries. The wavelet tree provides the means to answer *rank* and *select* queries on T for any symbol $c \in [0, \sigma)$ as well as *access* queries on T in time $\mathcal{O}(\log \sigma)$ [24, Ch. 6.2]. In this way, it can be viewn as an alternative representation of T. In order to achieve these query times, each bit vector is prepared for constant-time rank and select queries, which requires $o(n)$ bits of additional space for each level (see section 2.1.3). However, because this thesis focuses on the *construction* of the wavelet tree, we do not go into any further detail concerning these queries.

Path encoding. An important property that we use for construction is that the bits in the $\lceil \log \sigma \rceil$-bit binary representation of a symbol $c \in [0, \sigma)$ encode the path in the wavelet tree that leads from the root to the leaf that represents c. This means that for a 0-bit, we go on to the left child of the current node and for a 1-bit, we go to the right child. More generally, the ℓ-bit prefix of said representation of c leads to the corresponding node on level ℓ [29].

2.2.1 Wavelet Trees and the C Array

Let the array C contain, for each symbol $c \in [0, \sigma)$, the accumulated number of occurrences of the symbols in T_{eff} that are lexicographically smaller than c, that is: $C[c] \coloneqq \sum_{x=0}^{c-1} \text{occ}_{T_{\text{eff}}}(x)$. Table 3 shows an example. By definition, we have $C[0] = 0$. Furthermore, we define $C[\sigma] \coloneqq n$ (the length of the text). C can easily be computed in time $\mathcal{O}(\sigma)$ using the text's histogram and occupies $\sigma \lceil \log n \rceil$ bits of space.

Computing node sizes. Using the C array, we can predict the size $|B_v|$ of any node v in the wavelet tree, i.e., the length of its bit vector. Let $[a, b] \subseteq [0, \sigma)$ be the alphabet interval represented by v. Then, $|B_v| = C[b+1] - C[a]$.

	a	e	l	r	t	v	w	
c	0	1	2	3	4	5	6	
$\text{occ}_T(c)$	1	4	1	1	2	1	1	
$C[c]$	0	1	5	6	7	9	10	11

Table 3: The histogram and the C array for $T =$ `wavelettree`.

2.2.2 Storage

There two common ways to store the wavelet tree: the *levelwise* and the *pointer-based* representation.

Levelwise. In the levelwise representation, the bit vectors of all nodes on the same level are concatenated to bit vectors of length n and prepared for constant-time rank and select queries. Note that in order to prevent gaps, we also store bit vectors of leaves in case there are any inner nodes on the same level (as the leaf for `w` in Figure 1).

The amount of 0-bits in a node's bit vector on level ℓ equals precisely the offset at which the first bit of the right child's label on level $\ell + 1$ is located. Due to this fact, navigation in the wavelet tree can be simulated using a combination of rank and select queries (see [24, Ch. 6.2] for details). Since the constant-time rank and select data structures occupy $o(n)$ bits per level, the levelwise wavelet tree can be stored using $n\lceil \log \sigma \rceil + o(n) \cdot \lceil \log \sigma \rceil$ bits.

Pointer-based. In the more explicit pointer-based representation, each node stores pointers to its two children. This allows for faster navigation in the tree, because it saves rank and select queries to determine node boundaries within the bit vectors. However, this comes at the cost of $\mathcal{O}(\log n)$ additional bits required per pointer. Since the wavelet tree has at most $2^{\lceil \log \sigma \rceil} - 1 = \mathcal{O}(\sigma)$ nodes, this amounts to $\mathcal{O}(\sigma \log n) + n\lceil \log \sigma \rceil + o(n) \cdot \lceil \log \sigma \rceil$ bits for the node-based wavelet tree. Note that constant-time rank and select support is still required on each bit vector in order to answer queries such as rank and select using the wavelet tree.

2.2.3 Sequential Construction

Navarro [24, section 6.2.3] presents a very straightforward algorithm for the sequential construction of the wavelet tree in its node-based representation, which is listed as Algorithm 1 (*simpleWT*). Given text T of length n over the alphabet interval $[a, b] \subseteq [0, \sigma)$, we compute the wavelet tree with root node v as follows: first, we find the number z of symbols that belong to the *left half* of $[a, b]$ (line 6). We then scan over T from left to right and construct the string T^0 of length z and the string T^1 of length $n - z$. These strings correspond to the texts displayed in Figure 1. For each symbol c in T, we test whether c belongs to the left half of $[a, b]$ and if this is the case, write a 0-bit into B_v and append c to T^0 (line 13). Otherwise, if c belongs to the right half, write a 1-bit to B_v and append c to T^1 (line 17). Now, T is no longer needed and can be discarded. We construct the wavelet subtrees of the left and right child of v recursively using T^0 and T^1.

We invoke *simpleWT* for the root node $v = 0$ and the entire effective alphabet $[a, b] = [0, \sigma)$ to construct the full wavelet tree for an input T.

Algorithm 1: *simpleWT* – sequentially constructs the wavelet tree in its node-based representation.

Input : node BFS rank $v \in \mathbb{N}$, interval $[a, b] \subseteq [0, \sigma)$, text $T \in [a, b]^n$
Output: bit vectors $B_u : \mathbb{B}^*$ for each node u in the subtree of v

Function *simpleWT(v, a, b, T)*
  if $a = b$ **then**
    return ϵ *// leaf*
  $m \coloneqq \lfloor \frac{a+b}{2} \rfloor$
  $z \coloneqq 0$
  // precompute size of T^0 and T^1
  for $i = 0$ **to** $n - 1$ **do**
    if $T[i] \leq m$ **then**
      $z{+}{+}$
  allocate $T^0 \in [a, m]^z$ and $T^1 \in [m + 1, b]^{n-z}$
  $z \coloneqq 0$
  for $i = 0$ **to** $n - 1$ **do**
    if $T[i] \leq m$ **then**
      // $T[i]$ is in the left half
      $B_v[i] \coloneqq 0$
      $T^0[i] \coloneqq T[i]$
      $z{+}{+}$
    else
      // $T[i]$ is in the right half
      $B_v[i] \coloneqq 0$
      $T^1[i - z] \coloneqq T[i]$
  free T
  simpleWT$(2(v + 1) - 1, a, m, T^0)$ *// recursively construct left child*
  simpleWT$(2(v + 1), m + 1, b, T^1)$ *// recursively construct right child*
  return $(B_v, B_{2(v+1)-1}, B_{2(v+1)}, \dots)$

The algorithm requires total time $\mathcal{O}(n \log \sigma)$, because we scan a total of n symbols on each of the $\lceil \log \sigma \rceil$ levels. We need $\mathcal{O}(n \log \sigma)$ bits of space to store the temporary strings T^0 and T^1.

Various advancements have been made in the field of sequential wavelet tree construction. Munro et al. [22] show that the wavelet tree can be constructed in time $\mathcal{O}(n\lceil \frac{\log \sigma}{\sqrt{\log n}} \rceil)$ by first constructing an L-ary wavelet tree (with $L = 2^{\sqrt{\log n}}$) and then converting it to a binary tree. Claude et al. [6] give an algorithm that constructs the wavelet tree using only $\mathcal{O}(\log n \log \sigma)$ bits of additional space by permuting truncated representations of the symbols of the input text. Furthermore, they develop an in-place variant of that algorithm that replaces the input text and uses $n + \mathcal{O}(\log n \log \sigma)$ bits of extra space. Finally, da Fonseca and da Silva [7] show how to construct the wavelet tree online, i.e., without prior knowledge of the input text and alphabet.

2.2.4 Parallel Construction

In recent years, there has been quite some research on the *parallel* construction of wavelet trees in multi-core, shared memory settings [10, 19, 29, 30, 31, 32]. In this section, we briefly summarize the practically most relevant strategies employed by the algorithms presented thus far. We describe in greater detail in chapter 3, where we adopt them for our distributed algorithms.

Domain decomposition. Sepúlveda et al. [29] present an easily conceivable idea to efficiently distribute work across the available processors. Using the *domain decomposition* approach, the input text is partitioned and each processor locally computes the entire wavelet tree for one part of the text using any sequential algorithm (e.g., Algorithm 1). In order to retrieve the wavelet tree for the full text, the partial wavelet trees are merged node by node by concatenating their bit vectors from left to right in the order of the processor ranks.

Recursive with parallel split. The algorithm *recursiveWT* [19] due to Labeit et al. constructs the wavelet tree recursively in a way similar to Algorithm 1. The input text is partitioned and processed in parallel using p processors to construct the bit vector. The computation of T^0 for recursing on the left subtree and T^1 for the right subtree is done with thea *parallel split* operation using the previously constructed bit vector. Then, $\frac{p}{2}$ processors recurse to construct the left and right subtree for T^0 and T^1, respectively.

Stable sorting. While the domain decomposition and recursive strategies construct the wavelet tree with focus on the nodes, the algorithm *sortWT* first proposed by Shun [31] and later improved by Fischer et al. [10] constructs each level as a whole. The bit vector is computed as in *simpleWT* (Algorithm 1) from the current text of length n in parallel (e.g., the input text for the first level). After constructing level ℓ, we *stably sort* the text by the $(\ell + 1)$-bit prefixes of its symbols. Because the first $\ell + 1$ bits of a symbol encode the first $\ell + 1$ steps in the wavelet tree towards its leaf and we already constructed ℓ levels, the sorted text corresponds to the order of bits on the following level.

2.3 Distributed Computing

Distributed computing requires an extended view on algorithms in terms of performance evaluation compared to the classical sequential computing models. In sequential computing, a single *processing element* executes an algorithm in order to compute a result. Using Flynn's taxonomy [11], this corresponds to the *Single Instruction Stream – Single Data Stream* (SISD) model. We evaluate SISD algorithms according the number of steps it re-

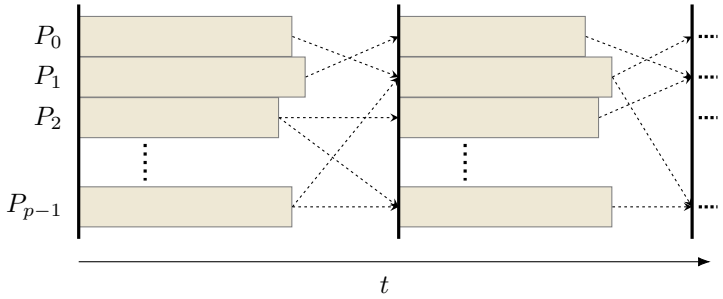


Figure 2: Distributed computation of processing element P_0 to P_{p-1} over time t in the BSP model. The bars represent the local computations of the processing elements, the dashed arrows represent their communication. The thick vertical lines mark synchronization barriers (including the start of the distributed algorithm) and thus separate the computation into supersteps.

quires to get to said result (running time), as well as how much memory (RAM) needs to be allocated for its temporary storage of data during computation.

In contrast, distributed computing means that multiple processing elements, aware of one another, work together to compute a cohesive result. These processing elements, however, may be physically separate machines, connected only via a medium of communication (e.g., ethernet or InfiniBand). There is no concept of shared memory, so a processing element P_1 cannot see what another processing element P_2 is doing unless P_2 tells it, i.e., unless *communication* takes place in the form of messages being sent from P_2 to P_1 over the medium.

Each processing element may execute the same or different local algorithms. Without any communication, they work in a completely isolated manner and thus independently of the other processing elements. This corresponds to Flynn's *Multiple Instruction Stream – Multiple Data Stream* (MIMD) model [11]. The combination of the local algorithms and communication with the goal of computing a cohesive result, however, form a *distributed algorithm* (see also [20]).

Communication may be *expensive* due to a myriad of reasons (e.g., due to limited bandwidth on the medium) and naturally causes computational overhead. Therefore, the amount of communicated information is a key measure in the analysis of distributed algorithms.

2.3.1 The Bulk Synchronous Parallel Model

This notion is implemented in the *bulk-synchronous parallel* (BSP) model introduced by Valiant [35]. In the BSP model, a distributed algorithm is thought to be separated in so-called *supersteps*, each of which consists of three phases: *local computation*, *communication* and *barrier synchronization*. This is visualized in Figure 2.

Local computation. The first phase corresponds to the aforementioned local algorithms that each processing element executes independently. Eventually, the execution arrives at a point where either communication is necessary in order to continue, or the result needs to be communicated in order to form the cohesive result. The duration of the local computation may differ for each processing element — be it because of different hardware, different algorithms being executed, different parts of the input being processed or simply statistical fluctuations.

Communication. The second phase is an abstraction from the fact that any processing element may dispatch messages at any point during its local computation. To this end, we imagine that all messages from all processing elements are dispatched simultaneously after all local computation is done.

Barrier synchronization. In order for the messages to be processed by their recipients, a synchronization needs to take place. This can be thought of the point in time where the last message sent during the communication phase has traversed the communication medium and arrived at its destination. In the BSP model, this is considered a barrier because only at this point, processing elements may continue with local computation, processing the received messages. The synchronization concludes a superstep; any local computation processing the received messages belong to the next superstep.

Cost analysis. We define the *BSP costs* of a distributed algorithm to reflect the complexity of the three phases. Let n be the size of the input. The BSP costs of an algorithm consist of (1) the maximum (worst-case) number $W(n)$ of *local work* steps that processing elements are required to perform, (2) the number $H(n)$ of *words sent* over the communication medium during the distributed computation and (3) the number $S(n)$ of synchronization barriers that occur during the computation.

To express the number of words sent, we define three types of word lengths that depend on the underlying computer and network architecture:

1. the length $w_{\mathbb{N}}$ of one *integer* — e.g., $w_{\mathbb{N}} = 8$ bytes for 64-bit integers,
2. the length w_{Σ} of one *symbol* — e.g., $w_{\Sigma} = 1$ byte in ASCII encoding,
3. the length $w_{\mathbb{B}}$ of one *bit*[1] — which, for example, we could substitute by $\frac{1}{8}$ to express the number of bytes being sent.

To give an example, when a processing element sends a message containing m integers, we write $m \cdot w_{\mathbb{N}}$ *words*. We assume all three word lengths to be in $\mathcal{O}(1)$.

[1] Note that in common network architectures, it is not possible to send messages consisting of single bits. Instead, multiple bits are packed into bytes or other types of network payloads. However, some algorithms that we analyze in this thesis send ranges of bit vectors and we use $w_{\mathbb{B}}$ in order to distinguish from other word types.

2.3.2 MPI

Message passing is a paradigm commonly followed in the development of distributed algorithms and constitutes the means of how communication between different processing elements is achieved: by sending and receiving *messages* over a communication medium, which is assumed to be reliable (i.e., the user / programmer does not need to cope with communication failures).

The *Message-Passing Interface* (MPI) establishes a standard for writing message passing programs with the goal of defining an application programming interface allowing for efficient, thread-safe communication and implementations portable to heterogeneous environments [21]. While the semantics of MPI operations are language independent, the standard was introduced with C and Fortran bindings in mind.

An established open-source implementation of MPI is *Open MPI* [12]. For our practical evaluation, we use the *Intel MPI Library*[2], which is optimized for Intel clusters.

Communicators. A key concept of MPI is the *communicator*, which can roughly be understood as a group of workers executing a distributed algorithm and are allowed to send and receive messages from one another. To address a worker, its rank within the respective communicator is used.

Message passing operations. Message passing is done using the straightforwardly named operations *Send* and *Recv*, which causes the executing worker to send a message to a certain worker or receive one, respectively. The *Probe* operation allows to test for an incoming message and check it before actually receiving it, e.g., in order to allocate a buffer for it to be stored. Alternatively, its reception may be canceled altogether.

The *Send*, *Recv* and *Prob*e operations come in two major variants: *blocking* and *non-blocking*. A blocking *Send* will block the execution flow of the sending worker until the recipient receives the message and vice versa, a blocking *Recv* will block until a matching message is sent by another worker. The non-blocking variants, on the other hand, can be thought of as asynchronous operations, immediately returning the execution flow.

Collective operations. MPI defines several collective operations, which involve all workers of a communicator. The *Barrier* operation, for example, resembles an explicit barrier synchronization according to the BSP model: the local execution is halted until all workers (of the corresponding communicator) reach the synchronization point.

[2]Intel MPI Library: `https://software.intel.com/en-us/mpi-library`

2.3.3 Thrill

Thrill is a general purpose framework for distributed processing of big data, setting the focus on data flow as opposed to message passing [2, 3]. It can be compared to the well-known frameworks Apache Spark [36] and Apache Flink [4] in that it is a high-level framework for processing data streams. In contrast to those frameworks, however, Thrill is (1) based on C++, which allows for native machine code optimized for the executing hardware and a much more fine-grained control over memory allocations as opposed to frameworks based on the Java Virtual Machine, and (2) its main data structure is an array rather than a multiset, which allows operations such as sorting, prefix summing and zipping to be implemented in the framework. Thrill outperforms both Apache Spark and Apache Flink in common benchmarks in terms of total running time and network throughput [3].

Distributed Immutable Arrays. The primary data structure of Thrill is the *Distributed Immutable Array* (DIA). It constitutes the notion of the data being processed to exist in an *array* that is *distributed* over any set of workers in the network. This data may be of any serializable type: primitives such as integers, tuples or complex user-defined structures. Thrill makes it transparent to the user where any part of the data is currently stored or even whether it currently exists at all — Thrill makes use of lazy evaluation wherever possible. Independent of these factors, DIAs allow the distributed data to be thought of as a consecutive sequence of items as described in section 2.1.2, which allow for high-level operations to be defined on them. However, it is important to note that Thrill provides no means whatsoever to access or modify single items of a DIA. The only way to modify a DIA is by performing an operation which transforms it into a new DIA, i.e., the represented array is *immutable*.

Building blocks. These DIA operations are the building blocks on which Thrill applications are based. Among several others, Thrill implements all the operations defined in Table 2. DIA operations are categorized as either (a) *source operations*, which initially produce a DIA to start working with — e.g., read from a file or generate an indexed sequence of integers, (b) *local operations*, which are trivial in the sense that no network communication is required in order to perform them — e.g., the Map operation, which maps items using a map function, or (c) *distributed operations*, which do require communication in order to be executed — e.g., the Sort operation, where only the local sequence can be sorted on each worker before communication is necessary to sort globally.

2.4 Common Distributed Algorithms

We introduce two problems that commonly occur in distributed computing: *prefix summing* and *AllReduce*. Both MPI and Thrill provide implementations to solve these problems, but

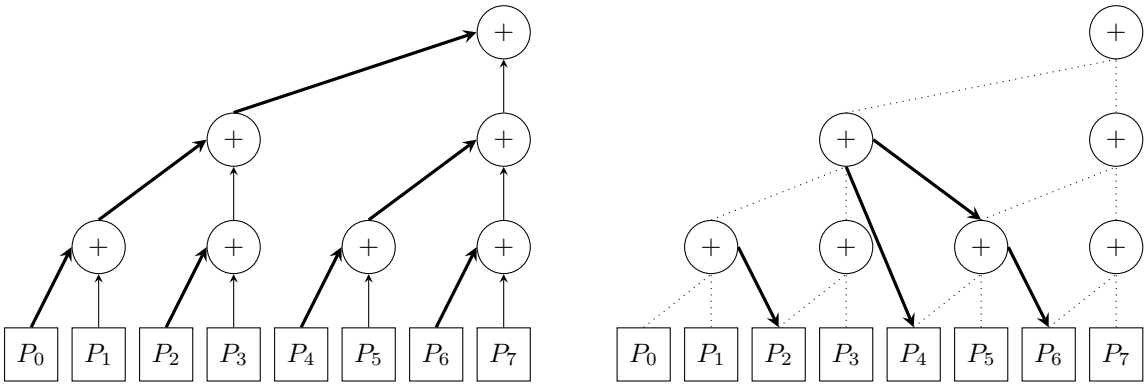
it is not documented exactly which algorithms they implement. Therefore, we take a look at known optimal algorithms.

Let p be the number of processing elements and let $x = (x_0, x_1, \dots x_{p-1}) \in \mathbb{X}^p$ be a *distributed tuple* over some set $\mathbb{X}$. We assume that processing element i — the i-th processing element in the network with $i \in [0, p)$ — only knows the *local* item x_i, but none of the *remote* items x_j with $j \neq i$. Let $\oplus$ be an associative operator over items from $\mathbb{X}$.

2.4.1 Prefix Summing

A frequently required operation in distributed computing is for each processing element i to find the *prefix sum* $X_i := x_0 \oplus x_1 \oplus \cdots \oplus x_{i-1}$. An optimal parallel algorithm to compute prefix sums uses a *binary merge tree* as described by JáJá [16, section 2.1.1]. We can easily apply it to our distributed scenario: a processing element with rank $2k$ sends its local value x_{2k} to processing element $2k + 1$, the right sibling in the merge tree. Processing element $2k+1$ computes the sum $x_{2k} + x_{2k+1}$ and forms the parent node k in the merge tree. This process is repeated recursively in a bottom-up manner, after which $\frac{p}{2}$ processing elements know their respective prefix sums. In a second phase, the inner nodes communicate in top-down manner to deliver the prefix sums to the remaining $\frac{p}{2}$ processing elements. An example is shown in Figure 3.

Corresponding to the costs of the parallel algorithm shown in [16], the prefix sum computation using a binary merge tree has $2(\lceil \log p \rceil - 1)$ BSP barriers — one between each level of the merge tree and in both directions. We need to send a total $\mathcal{O}(p) \cdot w_{\mathbb{N}}$ words (assuming $\mathbb{X} \subseteq \mathbb{N}$) and require $\mathcal{O}(p)$ steps of local work on each processing element.



(a) First phase (bottom-up) of the distributed prefix sum computation.

(b) Second phase (top-down) of the distributed prefix sum computation.

Figure 3: Example of the two phases of a distributed prefix sum computation of eight processing elements using a binary merge tree. The x-axis represents the processing elements that hold a respective value and the y-axis represents BSP barriers. Only the thick arrows actually mean communication; values passed using the thin arrows do not need to be sent over the network.

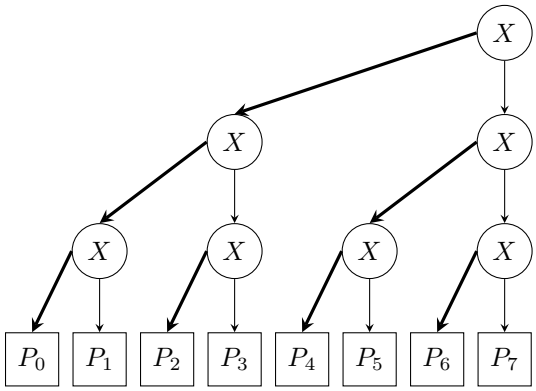


Figure 4: Example of the broadcast phase of the AllReduce operation following Figure 3a. Only the thick arrows actually mean communication; values passed using the thin arrows do not need to be sent over the network.

In the scope of this thesis, we only need the *exclusive* prefix sum $X'_i \coloneqq X_i - x_i$, which is the prefix sum sans the local value x_i.

2.4.2 AllReduce

Using the binary tree communication pattern, we can also perform the distributed operation known as *AllReduce*, which computes the *reduction* $X = x_0 \oplus x_1 \oplus \cdots \oplus x_{p-1}$ and broadcasts it back to all processing elements. It is easy to see that $X = X_{p-1}$, i.e., the reduction equals the prefix sum of all $p - 1$ items.

We can compute it using the tree topology as shown in Figure 3a. The processing element forming the root node (processing element 7 in the figure) then holds the reduction X. To broadcast X to the other processing elements, we simply invert the direction of communication and have every node send X to its two children as shown in Figure 4. Finally, all processing elements know X.

The BSP costs of this algorithm are the same as for prefix summing. Even though it is optimal in terms of local work and communication, there have been many optimizations of the AllReduce protocol for practical uses (e.g., [27] and [26]).

3 Distributed Wavelet Tree Construction

In this chapter, we adopt the briefly introduced ideas for parallel wavelet tree construction in multiprocessor, shared memory settings to develop distributed algorithms and analyze their theoretical performance in the BSP model.

Let us first specify our scenario: we want construct the wavelet tree for the input text T of length n over the *effective* alphabet $[0, \sigma)$ using p processing elements which are aware of each other and may communicate over a network. In a preliminary step, T is *partitioned* so that processing element i — the i-th processing element in the network with $i \in [0, p)$ — initially holds part $T_i = T[i\lceil \frac{n}{p} \rceil \dots (i+1)\lceil \frac{n}{p} \rceil - 1]$ of length $\lceil \frac{n}{p} \rceil$. In the case that n is not a multiple of p, the last processing element may process a slightly shorter part of length $n \mod p$. We consider that case an implementation detail and will not regard it any further.

Our desired output form of the *global* wavelet tree for T is the levelwise representation consisting of $\lceil \log \sigma \rceil$ bit vectors $B_0, \dots, B_{\lceil \log \sigma \rceil - 1}$ of length n each. The bit vectors should be *balanced* so that each processing element has an equal number of $\lceil \frac{n}{p} \rceil$ bits per level, i.e., $\lceil \log \sigma \rceil \cdot \lceil \frac{n}{p} \rceil$ bits in total.

In the following, we first take a look at how to retrieve the histogram and effective transformation of the input text in section 3.1. We then develop three distributed algorithms for wavelet tree construction — using domain decomposition (section 3.2), a distributed split operation (section 3.3) and stable sorting (section 3.4) — and analyze them in the BSP model.

3.1 Histogram Computation & Effective Transformation

The histogram of the input text contains crucial information about the wavelet tree. As described in section 2.2.1, we can use it to compute the C array and predict the size of each node. This is required, because some of the algorithms presented in this chapter produce the node-based representation of the wavelet tree as an intermediate result. Furthermore, in a real application, it is unlikely that the input is given in its effective transformation. In this case, we need to transform the input.

We develop the distributed Algorithm 2 to compute the histogram of T in a distributed setting and then produce the effective transformation. First, each processing element i computes the *local histogram* H_i by scanning the local part T_i once (starting at line 2).

Algorithm 2: Distributed computation of the histogram H and the effective transformation T_{eff} for input text T using *AllReduce*.

Input : text $T \in \Sigma^n$
Output: effective alphabet size $\sigma \in \mathbb{N}$, histogram $H \in \mathbb{N}^\sigma$, effective transformation $T_{\text{eff}} \in [0, \sigma)^n$

```
1  Function prepare(T)
       // compute local histogram
2      parfor i = 0 to p − 1 do
3          for c = 0 to |Σ| − 1 do
4              H_i[c] := 0^|Σ|
5          parfor k = 0 to n − 1 do
6              H_i[T[k]]++
7      h ← AllReduce_{i=0}^{p−1}(H_i)
       ⟨BSP: 2(⌈log p⌉ − 1) implicit barriers⟩
       // compute the effective alphabet and mapping
8      σ := 0, eff := 0^|Σ|
9      for c = 0 to |Σ| − 1 do
10         if H[c] > 0 then
11             eff[c] := σ
12             σ++
       // perform effective transformation
13     parfor k = 0 to n − 1 do
14         T_eff[k] := eff[T[k]]
15     return (σ, H, T_eff)
```

The *global histogram* H is the result of an *AllReduce* operation (line 7), i.e., we sum up the local occurrence counts and broadcast them back to all processing elements. Note how we highlight implicit the BSP barriers caused by the *AllReduce* operation in the pseudocode listing. Knowing the global histogram H, we can determine the size σ of the effective alphabet and create a mapping that maps symbols from the input alphabet Σ that occur in T to $[0, \sigma)$ (starting at line 8). In a final scan of T_i (starting at line 13), we use this mapping to perform the effective transformation of the input.

BSP costs. The only distributed operation performed by Algorithm 2 is the *AllReduce* operation, which requires $\mathcal{O}(p)$ steps of local work and $2(\lceil \log p \rceil - 1)$ BSP barriers and sends $\mathcal{O}(p) \cdot w_{\mathbb{N}}$ words (see section 2.4.2). Additionally, $\mathcal{O}(\frac{n}{p})$ steps of local work are needed for the scans of T_i and $\mathcal{O}(|\Sigma|)$ steps for the computation of the effective alphabet mapping. In summary, Algorithm 2 requires $\mathcal{O}(\frac{n}{p} + |\Sigma| + p)$ steps of local work.

3.2 Domain Decomposition

Domain decomposition is a very straightforward technique to split up work across the available processing elements, which has first been used for parallel wavelet tree construction

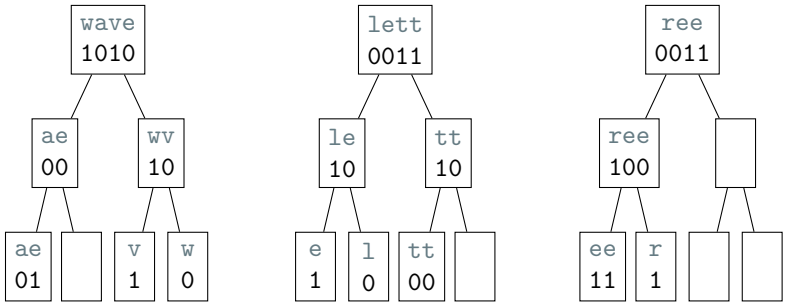


Figure 5: Domain decomposition of the wavelet tree for $T =$ `wavelettree` using three processing elements, where the first processing element computed the wavelet tree for part $T_0 =$ `wave` (leftmost), and so on. The empty nodes occur due to the fact that the *global* alphabet is used for construction and some symbols do not occur in the corresponding parts of the text. Comparing against Figure 1, we see that the left-to-right concatenation of the local bit vectors for each node equals the corresponding global node's bit vector.

by Sepúlveda et al. [29]. Each processing element i computes the entire wavelet tree for its part T_i of the input. Subsequently, these partial wavelet trees are merged into the *global* wavelet tree.

3.2.1 Local Computation

Because each processing element i constructs the wavelet tree for T_i in its entirety and independent of the other processing elements in the network, there are no requirements as to which algorithm is used to do that. However, it is important that the computation is done with respect to the *global* alphabet $[0, \sigma)$ — despite the fact that some c may never occur locally in T_i. We assume that the local wavelet tree is computed after $W(localWT)$ local work steps and that it is produced in its node-based representation so that on computer i, we have a bit vector $B_{v,i}$ for every node $v \in [0, 2\sigma - 1)$.

In case the local algorithm computes the levelwise representation, we can transform it into the node-based representation by computing the node boundaries using the C array as described in section 2.2.1. In this case, we need negligible $\mathcal{O}(\sigma)$ additional steps of local work.

3.2.2 Balanced Merge

Each processing element i now has the wavelet tree for T_i in its node-based representation. Figure 5 shows an example for this scenario.

In order to merge these local wavelet trees to retrieve the global wavelet tree, we make use of the fact that the input has been partitioned in the order of the processing element ranks and that the wavelet tree has been computed with respect to the global effective alphabet $[0, \sigma)$: for each node v, we *concatenate* all the bit vectors $B_{v,i}$ of each processing element i in the order of their ranks.

Algorithm 3: *dddWTMerge* – merges the wavelet tree's local node bit vectors into the global level bit vectors. The function *NodeLength* computes the length of a node using the precomputed C array from the histogram as described in section 2.2.1, which we do not regard here in detail.

Input : effective alphabet size $\sigma \in \mathbb{N}$, histogram $H : \mathbb{N}^\sigma$, bit vectors $B_{v,i}$ for each node $v \in [0, 2\sigma - 1)$ and processing element $i \in [0, p)$.
Output: bit vectors $B_\ell : \mathbb{B}^n$ for each level $\ell \in [0, \lceil \log \sigma \rceil)$.

1 **Function** *dddWTMerge*(σ, H, $B_{v,i}$)
  // prefix sum computation
2 **parfor** $i = 0$ **to** $p - 1$ **do**
3 **for** $v = 0$ **to** $2\sigma - 2$ **do**
4 $\text{off}_i[v] \coloneqq \sum_{j=0}^{i-1} |B_{v,j}|$
  $\langle$***BSP:*** $2(\lceil \log p \rceil - 1)$ ***implicit barriers***$\rangle$
  // copy first level from root node
5 **parfor** $i = 0$ **to** $p - 1$ **do**
6 $d \coloneqq i \cdot \lceil \frac{n}{p} \rceil$
7 $B_0[d,\ d + \lceil \frac{n}{p} \rceil - 1] \coloneqq B_{0,i}$
  // bit redistribution for remaining $\lceil \log \sigma \rceil - 1$ *levels*
8 **for** $\ell = 1$ **to** $\lceil \log \sigma \rceil - 1$ **do**
  // concatenate bit vectors of nodes on level ℓ
9 $\text{off} \coloneqq 0$
10 **parfor** $u = 0$ **to** $2^\ell - 1$ **do**
11 $v \coloneqq 2^\ell + u$ *// node BFS rank*
12 **parfor** $i = 0$ **to** $p - 1$ **do**
13 $d \coloneqq \text{off} + \text{off}_i[v]$
14 $B_\ell[d,\ d + |Bv_i| - 1] \coloneqq B_{v,i}$ *// distribute bit vector range*
  $\langle$***BSP: implicit barrier***$\rangle$
15 $\text{off} \coloneqq \text{off} + \text{NodeLength}(v, H)$
16 **return** $(B_0, \ldots, B_{\lceil \log \sigma \rceil - 1})$

The caveat is that we would like to have the global wavelet tree in its *levelwise* representation and also *balanced* across our p processing elements so that each processing element has $\lceil \frac{n}{p} \rceil$ bits of each level. In order to save additional subsequent steps, our merge operation should take care of both requirements simultaneously. To achieve this, each processing element may have to send different ranges of its bit vectors to different recipients. The difficulty lies in determining those recipients, because exactly which processing element shall receive a certain bit vector range depends on how many bits processing elements with a lower rank will send, which corresponds to the problem of finding *prefix sums*. In the following, we develop Algorithm 3 (*dddWTMerge*) for our balanced wavelet tree merge.

Prefix sums. On processing element i, let $\text{off}_i[v]$ be the starting position of $B_{v,i}$ within B_v for any node v. In other words, $\text{off}_i[v]$ is the starting position of the first local bit of v within v's bit vector in the global wavelet tree. It is the exclusive prefix sum of the sizes

$|B_{v,j}|$ for processing elements $j < i$ (line 4), which we compute using a binary merge tree as described in section 2.4.1. It is important to note that we are computing prefix sums of *vectors* containing up to $2\sigma - 1$ items each (one for each node). Taking this into account, the prefix sum computation sends $\mathcal{O}(p\sigma) \cdot w_{\mathbb{N}}$ words and requires $\mathcal{O}(p\sigma)$ steps of local work. The number of $2(\lceil \log p \rceil - 1)$ BSP barriers remains unaffected by the number of items per vector.

Bit redistribution. With knowledge of $\text{off}_i[v]$, we can distribute the bits across the available processing elements so each processing element has an equally sized part of the corresponding *level* bit vector B_ℓ. Figure 6 visualizes what we would like to achieve.

When processing element i processes node v on level ℓ, it needs the position $\text{off}(v, \ell)$ of the first bit of B_v in B_ℓ (line 15). As seen previously, it can be computed using the C array, which we can precompute from the histogram in negligible time $\mathcal{O}(\sigma)$. The position of the first bit of $B_{v,i}$ in B_ℓ is then $\text{off}(v, \ell) + \text{off}_i(v)$ (line 13).

Since every processing element is supposed to hold exactly $\lceil \frac{n}{p} \rceil$ bits, we can now easily determine the recipient for every bit in $B_{v,i}$. Let $k \in [0, |B_{v,i}|)$ be the position of such a local bit. Then its recipient is processing element

$$j = \left\lfloor \frac{\text{off}(v, \ell) + \text{off}_i[v] + k}{\lceil \frac{n}{p} \rceil} \right\rfloor .$$

Because the local wavelet tree was constructed for a text of length $\lceil \frac{n}{p} \rceil$, it is $|B_{v,i}| \leq \lceil \frac{n}{p} \rceil$. This means that there may be at most two different recipients for the bits of $B_{v,i}$. Therefore, the total number of messages sent by all p processing elements for all the $2\sigma - 2$ nodes is at most $2p(2\sigma - 2) = \mathcal{O}(p\sigma)$ (the root node does not need to be merged). Because the messages

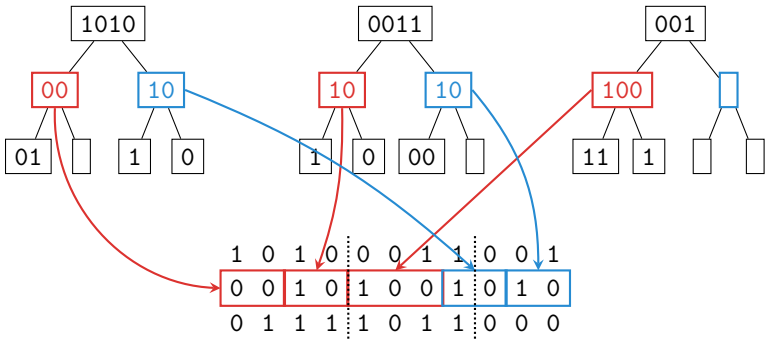


Figure 6: Visualization of the balanced merge operation of the domain decomposition shown in Figure 5. The figure specifically highlights the merge of the second level. Note how the bits of the left local nodes are concatenated in the left part of the second level bit vector and the bits of the right nodes in the right part. The dotted lines mark the boundaries of the bit vectors on each processing element. The first processing element (leftmost) needs to send the two bits of the node with BFS rank 3 to two different processing elements to achieve balance.

contain at most $\lceil \frac{n}{p} \rceil$ bits each, the total number of words sent is in $\mathcal{O}(p\sigma\frac{n}{p}) \cdot w_{\mathbb{B}} = \mathcal{O}(n\sigma) \cdot w_{\mathbb{B}}$. All nodes need to be scanned once locally on each processing element, which can be done in $\mathcal{O}(\sigma\frac{n}{p})$ total steps. Regarding BSP barriers, we perform the redistribution levelwise and require one barrier synchronization for each level. Since we need to merge $\lceil \log \sigma \rceil - 1$ levels, we have $\lceil \log \sigma \rceil - 1$ barriers in total (the first level corresponds to the root node and does not need to be merged).

Summarized BSP costs. In summary, the balanced merge can be done with $\mathcal{O}(p\sigma + \sigma\frac{n}{p})$ local work steps, sending $\mathcal{O}(p\sigma) \cdot w_{\mathbb{N}} + \mathcal{O}(n\sigma) \cdot w_{\mathbb{B}}$ words and requiring $2\lceil \log p \rceil + \lceil \log \sigma \rceil - 3$ BSP barriers.

3.3 Distributed Split

Labeit et al. [19] present the *recursiveWT* algorithm for multiprocessor scenarios with shared memory, which makes use of a *parallel split* operation. The idea is as follows: when we process node v for alphabet $[a, b] \subseteq [0, \sigma]$, we (1) compute the node's bit vector B_v in parallel, so that each processing element only scans $\lceil \frac{n}{p} \rceil$ symbols, and then (2) split up T into T^0 and T^1 and recurse using $\frac{p}{2}$ processing elements for the left and right child of v, respectively. The split of T into T^0 and T^1 is done according to the bits in B_v: if $B_v[k] = 0$ for some k, the symbol $T[k]$ is appended to T^0 and analogously, if $B_v[k] = 1$, $T[k]$ is appended to T^1. Figure 7 shows an example of a split operation.

We develop Algorithm 4 (*dsplitWT*) that computes the wavelet tree in its node-based representation using a distributed split operation for root node v, alphabet interval $[a, b]$ and processing elements i to j. In order to construct the complete wavelet tree, we invoke the algorithm for the root node $v = 0$, the entire effective alphabet $[a, b] = [0, \sigma)$, the full input text T and all p processing elements with $i = 0$ and $j = p - 1$.

In the following, we show how the parallel split can be applied to a distributed scenario. Furthermore, because our goal is the levelwise representation, we need to merge the node bit vectors into level bit vectors, which we describe in section 3.3.2.

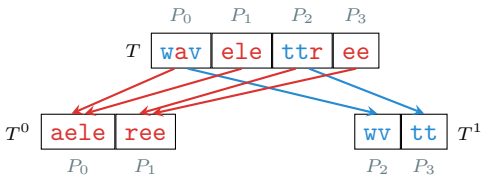


Figure 7: Parallel split on the first level of our running example $T =$ `wavelettree` using $p = 4$ processing elements. Initially, each processing element $i \in [0, 4)$ has part T_i and decides which symbols go to T^0 and which go to T^1. Both T^0 and T^1 will be processed by only $p/2 = 2$ processors and the symbols are communicated between the processing elements accordingly.

Algorithm 4: *dsplitWT* – recursively constructs the wavelet subtree with root v representing alphabet interval $[a, b]$ in its node-based representation using the distributed split operation with processing elements i to j. As shown in section 3.3.2, the output of this algorithm can be forwarded to *dddWTMerge* (Algorithm 3) in order to compute the levelwise representation.

Input : node BFS rank $v \in [0, 2\sigma - 1]$, alphabet interval $[a, b] \subseteq [0, \sigma)$, text $T \in [a, b]^n$, processing element ranks $i, j \in [0, p)$ with $i \leq j$
Output: bit vectors $B_u : \mathbb{B}^*$ for all nodes u in v's subtree

Function *dsplitWT*(v, a, b, T, i, j)
  if $a = b$ **then**
    return ϵ // *leaf*
  if $i < j$ **then**
    $m \coloneqq \lfloor \frac{a+b}{2} \rfloor$
    // *compute bit vector for v*
    parfor $k = 0$ **to** $n - 1$ **do**
      $B_v[k] \coloneqq (T[k] > m)$
    // *only split and recurse if either child will be non-empty*
    if $b - a + 1 > 2$ **then**
      // *split up T according to the bits in B_v*
      $(T^0, T^1) \coloneqq \text{DistributedSplit}(T, B_v)$
      $\langle$***BSP:*** $2\lceil \log(y - x + 1) \rceil$ ***implicit barriers***$\rangle$
      // *recurse on the left and right child using half the processing elements each*
      $i_m \coloneqq \lfloor \frac{i+j}{2} \rfloor$
      pardo
        dddWTMerge(T^0, $2(v + 1) - 1$, a, m, i, i_m) // *left*
        dddWTMerge(T^1, $2(v + 1)$, $m + 1$, b, $i_m + 1$, j) // *right*
    return $(B_v, B_{2(v+1)-1}, B_{2(v+1)}, \dots)$
  else
    // *we are left with only one processing element – compute remaining subtree sequentially*
    return $\text{SequentialWT}(T, a, b)$

3.3.1 The Distributed Split Operation

The core idea of the parallel split operation to split up work across the available processing elements after splitting the input can almost directly be applied to a distributed setting. However, since there is no shared memory between the processing elements, the *distributed split* operation requires communication as depicted in Figure 7.

For a node v, let $B_{v,i}$ be the part of B_v that processing element i has computed. In order to perform the split operation, i needs to decide for every symbol $c \in [a, b]$ of the input where to send it to. In the case that $c \leq \lfloor \frac{a+b}{2} \rfloor$, i.e., if c belongs to the left half of the alphabet, it should be appended to T^0 and thus sent to a processing element in the range $[0, \lfloor \frac{p}{2} \rfloor]$. Otherwise, if $c > \lfloor \frac{a+b}{2} \rfloor$, it is appended to T^1 and sent to a processing element in the remaining range $[\lfloor \frac{p}{2} \rfloor + 1, p)$ of processing elements.

Without loss of generality, let us assume in the following that c belongs to the left half of the alphabet, i.e., it is appended to T^0 and sent to a processing element $j \leq \lfloor \frac{p}{2} \rfloor$. The case where c is added to T^1 can be handled analogously.

T^0 should be balanced across the range of $\frac{p}{2}$ processing elements. This means that each processing element should receive the same number $\left\lceil \frac{|T^0|}{p/2} \right\rceil$ of symbols from T^0. Let $\text{pos}_{T_i}(c)$ be the position where c is located at in the local part T_i of the input. It is not yet possible for our processing element i to determine the recipient j, because it only sees its local part T_i^0 of T^0 and lacks the information about where it lies in T^0 globally. To get this information, it needs to know the length of the concatenation $T_0^0 \cdots T_{i-1}^0$ for the processing elements with rank lower than i, which is the exclusive *prefix sum*

$$Z_i \coloneqq \sum_{k=0}^{i-1} |T_k^0|.$$

To give an example, consider in Figure 7 the first occurrence of `t` in T on processing element P_2: it is clear that `t` is appended to T^1, but not whether to keep it at P_2 or send it to P_3, because we do not know how many symbols P_0 and P_1 will append to T^1 prior to P_2. For this, we need the prefix sum $Z_2 = |T_0^1| + |T_1^1| = 2$. We then see that P_2 already holds enough symbols of T^1 and `t` should be sent to P_3.

We can compute Z_i in a preliminary step using a binary merge tree as described in section 2.4.1. With knowledge of Z_i, it is possible for processing element i to determine the recipient

$$j = \left\lfloor \frac{Z_i + \text{pos}_{T_i}(c)}{p/2} \right\rfloor$$

of symbol c. In the following, we analyze the BSP costs of the distributed split algorithm.

Number of words sent. We send one *substring* to each recipient by concatenating the symbols it should receive, retaining their text order. It is $|T_i^0| \leq |T_i| = \lceil \frac{n}{p} \rceil$ and each recipient, from all processing elements processing T^0, will receive exactly $\left\lceil \frac{|T^0|}{p/2} \right\rceil \leq \lceil \frac{n}{p/2} \rceil$ symbols. This means that processing element i sends the symbols of T_i^0 to at most two different recipients. The same applies to T_i^1, so we have at most four recipients in total. On the global scope, during one distributed split operation, each processing element sends at most four substrings of length at most $\lceil \frac{n}{p} \rceil$, resulting in $4p \cdot \mathcal{O}(\frac{n}{p}) \cdot w_\Sigma = \mathcal{O}(n) \cdot w_\Sigma$ words. In order to construct the wavelet tree, we need to perform a distributed split operation for all nodes except those on the last level, i.e., for $2^{\lceil \log \sigma \rceil - 1} - 1$ nodes. Thus, we send $\mathcal{O}(n\sigma) \cdot w_\Sigma$ words in total for all distributed splits.

This does not yet account for the prefix sum computations of Z_i and its counterpart for T^1, which are needed for the processing elements to determine where to send the symbols of T_i^0 and T_i^1. Following section 2.4.1, the number of words sent for the prefix

sum computation of one distributed split operation is in $\mathcal{O}(p') \cdot w_{\mathbb{N}}$, where p' denotes the number of processing elements in use. On level ℓ, there are 2^ℓ nodes and we use $\frac{p}{2^\ell}$ processing elements for each node. Therefore, from a level perspective, we use all $p' = p$ processing elements and thus send $\mathcal{O}(p) \cdot w_{\mathbb{N}}$ words per level. For the $\lceil \log \sigma \rceil - 1$ levels that we need to perform a split for, this accumulates to $\mathcal{O}(p \log \sigma) \cdot w_{\mathbb{N}}$ words being sent for the prefix sum computations.

In summary, we send an accumulated number of $\mathcal{O}(n\sigma) \cdot w_\Sigma + \mathcal{O}(p \log \sigma) \cdot w_{\mathbb{N}}$ words for the distributed split algorithm.

BSP barriers. Each distributed split operation requires $2\lceil \log p' \rceil$ synchronization barriers in the BSP model when using p' processing elements: $2\lceil \log p' \rceil - 1$ barriers for the prefix sum computation and one additional barrier for the communication of the substrings. We perform a distributed split operation for all nodes except those on the last level. However, because all splits on the same level are performed in parallel, we can summarize the barriers caused by each split for the whole level. Since we use all $p' = p$ processing elements for any of the $\lceil \log \sigma \rceil - 1$ levels, we get $2\lceil \log p \rceil (\lceil \log \sigma \rceil - 1)$ total synchronization barriers.

Local work. The local work is dominated by the scans of T for the computation of the node bit vectors and for sending the substrings. On level ℓ, we process 2^ℓ nodes and require at most $\mathcal{O}(2^\ell \frac{n}{p/2^\ell}) = \mathcal{O}(2^\ell \frac{n}{p})$ steps. For all of the $\lceil \log \sigma \rceil$ levels, we require

$$\sum_{\ell=0}^{\lceil \log \sigma \rceil - 1} \mathcal{O}\left(2^\ell \frac{n}{p}\right) = \mathcal{O}\left(\sigma \frac{n}{p}\right)$$

steps of local work for the scans. For each split, we also need a prefix sum computation, which adds $\mathcal{O}(p)$ steps per node and thus $\mathcal{O}(p\sigma)$ steps for all nodes. In summary, the split operations require $\mathcal{O}(n\sigma + p\sigma)$ total steps of local work.

Sequential subtree construction. In the case of $p < \sigma$, there are not enough processing elements available to split the processing element range after every node, i.e., there will be a point during construction where only one processing element remains for a wavelet subtree (the case $i = j$ in line 15 of Algorithm 4). When this happens, we proceed constructing the remaining wavelet subtree using a sequential algorithm on that processing element.

To that regard, for small p, the local work steps required depends on the complexity of the chosen sequential algorithm. The number of words sent and BSP barriers which we analyzed above are then merely upper bounds.

3.3.2 Merge

After all nodes have been processed using the distributed split approach, the bit vectors are scattered across the p processing elements and we need to merge them in order to get the global wavelet tree in levelwise representation.

It turns out that the scenario is highly similar to that of a domain decomposition: thanks to the prefix sum awareness of the splits, the symbols in the sent substrings retain their original order and processing elements receive the substrings based on their rank in the network. This means that the same initial conditions apply as for the merge of the domain decomposition, i.e., all nodes' local bit vectors only need to be concatenated to the get the global bit vectors. To that end, the merge operation shown in section 3.2 can be reused here without any modifications.

Following that, we add the BSP costs of the domain decomposition merge: $\mathcal{O}(\sigma\frac{n}{p})$ steps of local work, $H(dd) = \mathcal{O}(p\sigma) \cdot w_{\mathbb{N}} + \mathcal{O}(n\sigma) \cdot w_{\mathbb{B}}$ number of words sent and $S(dd) = 2\lceil \log p \rceil + \lceil \log \sigma \rceil - 3$ BSP barriers.

3.3.3 Load Balancing

The motivation Labeit et al. [19] state for using the parallel split operation is to reduce (shared) memory consumption and achieve greater cache-friendliness. Of course, these aspects do not apply in a distributed setting. However, the approach can easily be modified to balance work across the available processing elements to optimize local work in practice.

Let z be the number of 0-bits in B_v, i.e., the length of T^0 and size of the left child of v. Then, $n - z$ is the number of 1-bits, length of T^1 and the size of the right child. Instead of using $\frac{p}{2}$ processing elements each for computing the left and right child, we use $\mathcal{O}(p\frac{z}{n})$ processing elements for the left and $\mathcal{O}(p(1 - \frac{z}{n}))$ processing elements for the right child, with the idea that the number of processing elements used corresponds to the work that remains on either side.

In the case that $\mathcal{O}(p\frac{z}{n}) \leq 1$, we assume that T^0 is small enough so the left wavelet subtree can be computed sequentially by one single processing element and the remaining processing elements can be used to process T^1. The inverse case of $\mathcal{O}(p(1 - \frac{z}{n})) \leq 1$ is handled analogously.

Since we still process the number amount of nodes and levels, the asymptotic BSP costs of the distributed split algorithm remain as analyzed previously.

3.4 Stable Sorting

The stable sorting approach presented by Shun [31] makes use of the fact that the binary representation of a symbol encodes the path to its corresponding leaf in the wavelet tree. Let $c \in [0, \sigma)$ be a symbol from the effective input alphabet. We call the integer represented

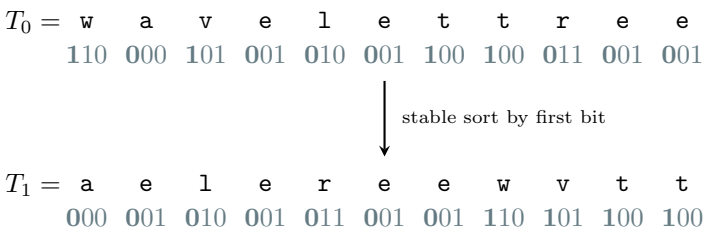


Figure 8: Example of the stable sorting by the one-bit prefix for the first level of our running example $T_0 =$ `wavelettree`. The binary codes of the symbols are written below the symbols with the first bit written in bold. To get T_1, we stably sort the symbols in T_0 by their one-bit prefixes. Note how the transformed text equals the order of symbols on the second level from the example wavelet tree in Figure 1.

by the ℓ most significant bits of c's binary representation the *ℓ-bit prefix* of c and write $\text{bitpref}_\ell(c)$.

We construct the wavelet tree level by level as follows: on level ℓ, from the *current text* T_ℓ (initially $T_0 = T$), we first compute the bit vector B_ℓ. If ℓ is not yet the last level, we compute $T_{\ell+1}$ by *stably sorting* the symbols in T_ℓ in ascending order by their $(\ell+1)$-bit prefixes and proceed with $T_{\ell+1}$. An example is shown in Figure 8.

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 in ascending order moves all symbols whose next bit is 0 to the left and those whose next bit is 1 to the right — which corresponds precisely to the path to their representing leaves in the wavelet tree. Because the sorting is stable, the relative order of the symbols is retained.

In distributed computing. This approach reduces most of the wavelet tree construction to stable integer sorting. Thus, it can be applied to a distributed setting with relative ease when a stable distributed sorting algorithm is available. Ideally, that algorithm also balances the sorted sequence across the p available processing elements. We look at two candidates in section 3.4.1 and section 3.4.2.

Because we construct the wavelet tree in its levelwise representation using the stable sorting approach, no subsequent merge operation is necessary (as opposed to domain decomposition or distributed split). Furthermore, if the sorter balances the sorted text for the next level across the available processing elements, it also indirectly takes care of balancing the corresponding bit vector so that each processing element finally has $\frac{n}{p}$ bits of each level.

3.4.1 Distributed Super Scalar Sample Sort

A popular example for a balancing parallel sorter is *super scalar sample sort* (sss-sort) presented by Sanders and Winkel [28]. It receives two parameters: the number of sort

buckets k and the oversampling factor a (which is explained below). Bingmann et al. implement a distributed variant of sss-sort in the Thrill framework [3]. Their distribution is essentially based on the idea to fix the number of buckets k to the number of available processing elements p, so that each processing element holds exactly one bucket of expected size $\frac{n}{p}$.

Assuming that initially, each processing element has $\frac{n}{p}$ items of the input sequence, the distributed sss-sort proceeds as follows:

1. Each processing element first picks the *sample* of a local items at random (with a the oversampling factor) and sends them to the *master*, the processing element designated to receive all samples (e.g., the processing element with rank 0).
2. The master sorts the $p \cdot a$ samples it received and attempts to choose $p-1$ equidistant *splitters*. This is where the oversampling factor a comes into effect: the higher the factor, the more samples the master receives and the higher becomes the probability that the chosen splitters are equidistant. After choosing the splitters, the master broadcasts them to all processing elements.
3. The splitters are used for *distribution*: each processing element determines a recipient for each local item and sends it there. Because the splitters are sorted, binary search can be employed to speed up the process.
   Based on the samples picked in the first step, the expected number of items each processing element receives after the distribution is $\frac{n}{p}$. It then already holds that $x_i \preceq x_{i+1}$ for any item x_i received by processing element i and any item x_{i+1} received by processing element $i+1$ (where $\preceq$ denotes the relation according to which we are sorting).
4. All that remains now is to *locally sort* the received items on each processing element. Finally, all item have been sorted globally so that processing element i has the i-th part of the sorted sequence.

Sample-based sorting vs. wavelet tree construction. There is one major issue regarding sss-sort in the face of wavelet tree construction. Let us think of the sort operation after constructing the first level of the wavelet tree: we sort the text by the 1-bit prefixes of the symbols. This means that we have only two distinct sort keys (0 and 1).

In sss-sort, this means that there are only two distinct samples and it is impossible to find $p-1$ equidistant splitters if $p > 2$. As a consequence, during the classic distribution described by Sanders and Winkel [28] and also employed by Bingmann et al. [3], all symbols are sent to only two of the p available processing elements, causing a major unbalance that gets worse with higher p. The same applies to the following levels: the sort operation done on level ℓ has at most $2^{\ell+1}$ distinct sort keys — the $(\ell+1)$-bit prefixes of the symbols. Thus, the problem of unbalance exists for the first $\log p$ sort operations.

We can balance items with the same sort key across a range of processing elements using solutions similar to the "investigator" proposed by Khatami et al. [18]. However, this is no longer in the scope of this thesis. Instead, we have a look at another sort algorithm which is inherently suited for wavelet tree construction.

3.4.2 Distributed Bucket Sort

We can turn the problem of low numbers of distinct sort keys into an advantage by re-imagining what the sort buckets represent: in the wavelet tree, because it is a binary tree, level $\ell+1$ has at most $2^{\ell+1}$ nodes — which is equal to the number of possible distinct sort keys after constructing level ℓ.

This leads to Algorithm 5 (*dbsortWT*), in which we employ a distributed *bucket sort*: on level ℓ, we allocate a bucket for each of the $2^{\ell+1}$ nodes of the following level (line 4). We scan the local input T_i from left to right and append each symbol $c \in [0, \sigma)$ to bucket $v = \text{bitpref}_{\ell+1}(c)$ (line 8), which is in the desired range of $[0, 2^{\ell+1})$ and corresponds directly to the index of c's node on level $\ell+1$. Conveniently, the least significant bit $\text{lsb}(v)$ of v

Algorithm 5: *dbsortWT* – constructs the wavelet tree using a stable distributed bucket sort. The binary $\circ$ operator denotes string concatenation.

Input : effective alphabet size $\sigma \in \mathbb{N}$, text $T \in [0, \sigma)^n$
Output: bit vectors $B_\ell : \mathbb{B}^n$ for each level $\ell \in [0, \lceil \log \sigma \rceil)$

```
1  Function dbsortWT(σ, T)
2      for ℓ = 0 to ⌈log σ⌉ − 2 do
           // construct bit vector and assign buckets
3          parfor i = 0 to p − 1 do
4              for v = 0 to 2^{ℓ+1} − 1 do
5                  C_{v,i} := new empty bucket
6              parfor k = 0 to n − 1 do
7                  v := bitpref_{ℓ+1}(T[k])
8                  append T_i[k] to bucket C_{v,i}
9                  B_ℓ[k] := lsb(v)
           // balanced concatenation
10         T := ϵ
11         parfor v = 0 to 2^{ℓ+1} − 1 do
12             C_v := ϵ
13             for i = 0 to p − 1 do
14                 C_v := C_v ∘ C_{v,i}
15             T := T ∘ C_v
               ⟨BSP: implicit barrier⟩
       // construct last level's bit vector
16     parfor k = 0 to n − 1 do
17         B_{⌈log σ⌉−1}[k] := lsb(T_i[k])
18     return (B_0, ..., B_{⌈log σ⌉−1})
```

represents the direction we take to get to c's node on the next level, i.e., it is the bit we set in B_ℓ at c's position (line 9).

One scan requires $\mathcal{O}(\frac{n}{p})$ steps of local work. We need to perform a scan for all $\lceil \log \sigma \rceil$ levels: even though the buckets are not needed for the last level of the wavelet tree — because there is no need to sort afterwards — we still need to compute its bit vector (line 16). In total, the scans require $\mathcal{O}(\frac{n}{p} \log \sigma)$ steps of local work.

All symbols in the same bucket share the same sort key v and are thus already sorted. Because we fill the buckets in a left-to-right scan, their relative order is retained and the sorting is stable. What follows is an operation very similar to the merge we described for the domain decomposition: we perform a *balanced concatenation* of the buckets (line 11 to line 15).

Balanced concatenation. Let $C_{v,i}$ be the bucket that processing element i has filled for node $v \in [0, 2^{\ell+1})$ on level ℓ. We call the concatenation $C_v := C_{v,0} \circ C_{v,1} \circ \cdots \circ C_{v,p-1}$ of each processing element's local bucket for v the *global bucket for* v. Then, the concatenation $T_{\ell+1} = C_0 \circ \cdots \circ C_{2^{\ell+1}-1}$ of the global buckets is the stable sorting of T_ℓ in ascending order by the symbols' $(\ell+1)$-bit prefixes. Figure 9 shows an example.

Our goal is to balance $T_{\ell+1}$ so that each processing element has an equal number of $\lceil \frac{n}{p} \rceil$ symbols for processing the next level. As mentioned above, this is very similar to merging node bit vectors for the domain decomposition. In fact, our concatenation can be seen as a domain decomposition of level $\ell+1$, except that we do not concatenate the bits of $B_{\ell+1}$, but the substrings of $T_{\ell+1}$. Therefore, the same techniques can be applied.

In the following, we adapt the BSP cost analysis of the domain decomposition merge (section 3.2) to our slightly different scenario. The concatenation consists of two phases: (1) computation of prefix sums and (2) redistribution (of substrings instead of bits).

Prefix sums. Assume that we are processing level ℓ: we are building prefix sums of vectors with one dimension per bucket, i.e., $2^{\ell-1}$ integers per vector. Thus, the prefix sum

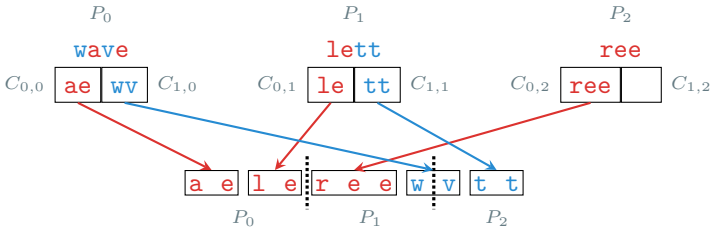


Figure 9: Example of the distributed bucket sort for the first level on our running example $T =$ `wavelettree` using $p = 3$ processing elements. Each processing element i has two sort buckets $C_{0,i}$ and $C_{1,i}$ and already assigned the symbols accordingly. The buckets are then concatenated by sending their symbols to the corresponding processing elements. Note that the symbols in bucket $C_{1,0}$ are sent to two different processing elements (P_1 and P_2) in order to achieve balance.

computation sends $\mathcal{O}(p2^\ell) \cdot w_{\mathbb{N}}$ words, requires $\mathcal{O}(p2^\ell)$ steps of local work and is finished after $2\lceil \log p \rceil - 1$ BSP barriers.

In order to compute the $\lceil \log \sigma \rceil$ levels of the wavelet tree, $\lceil \log \sigma \rceil - 1$ sort operations (concatenations) are required. The number of local work steps required sums up to

$$\sum_{\ell=0}^{\lceil \log \sigma \rceil - 1} \mathcal{O}(p2^\ell) = \mathcal{O}(p\sigma).$$

Accordingly, $\mathcal{O}(p\sigma) \cdot w_{\mathbb{N}}$ words are sent. The total number of BSP barriers sums up to $(\lceil \log \sigma \rceil - 1)(2\lceil \log p \rceil - 1)$.

Redistribution. For the redistribution of substrings, following our analysis in section 3.2.2, we send at most two substrings from each processing element for each of the $2^{\ell+1}$ buckets on level ℓ. Because the messages contain at most $\lceil \frac{n}{p} \rceil$ symbols each, the number of words sent is $\mathcal{O}(n2^\ell) \cdot w_\Sigma$. For all $\lceil \log \sigma \rceil - 1$ levels, this amounts to $\mathcal{O}(n\sigma) \cdot w_\Sigma$ words. The steps of local work required are in $\mathcal{O}(2^\ell \frac{n}{p})$ on level ℓ and thus in $\mathcal{O}(\sigma \frac{n}{p})$ for all levels. On each level, the redistribution induces one BSP barrier and thus we have $\lceil \log \sigma \rceil - 1$ BSP barriers in total.

3.5 Summary & Discussion

We summarize the BSP costs of the algorithms presented in the previous sections in Table 4.

The asymptotic local work steps and number of words sent by *dddWT* (distributed domain decomposition) and *dbsortWT* (bucket sort) reflect the similarity between the merge

Algorithm	Local work	Words sent	Barriers
dddWT	$W(\mathit{localWT})$ $+\mathcal{O}(p\sigma + \sigma\frac{n}{p})$	$\mathcal{O}(p\sigma) \cdot w_{\mathbb{N}} + \mathcal{O}(n\sigma) \cdot w_{\mathbb{B}}$	$2\lceil \log p \rceil + \lceil \log \sigma \rceil - 3$
dsplitWT	$\mathcal{O}(n\sigma + p\sigma + \sigma\frac{n}{p})$	$\mathcal{O}(p \log \sigma) \cdot w_{\mathbb{N}}$ $+\mathcal{O}(n\sigma) \cdot w_\Sigma$ $+H(\mathit{dddWT})$	$2\lceil \log p \rceil (\lceil \log \sigma \rceil - 1)$ $+S(\mathit{dddWT})$
dbsortWT	$\mathcal{O}(\frac{n}{p} \log \sigma + p\sigma + \sigma\frac{n}{p})$	$\mathcal{O}(p\sigma) \cdot w_{\mathbb{N}} + \mathcal{O}(n\sigma) \cdot w_\Sigma$	$2\lceil \log p \rceil \lceil \log \sigma \rceil$
prepare	$\mathcal{O}(\frac{n}{p} + \lvert\Sigma\rvert + p)$	$\mathcal{O}(p) \cdot w_{\mathbb{N}}$	$2(\lceil \log p \rceil - 1)$

Table 4: Summary of the BSP costs of the three approaches. The final entry, *prepare*, stands for the initial computation of the histogram and effective transformation of the input (section 3.1), which os not included in the other rows. For *dddWT*, we denote the local work required by the local wavelet tree construction by $W(\mathit{localWT})$. For *dsplitWT*, we use $H(\mathit{dddWT})$ and $S(\mathit{dddWT})$ to denote the number of words sent and BSP barriers in the domain decomposition, respectively. Regarding the local work of *dsplit*, we assume $p \geq \sigma$ so that no sequential construction is required.

operation done in domain decomposition and the bucket concatenation in the distributed bucket sort. The only difference lies in the fact that *dbsortWT* distributes symbols, whereas *dddWT* distributes bits, which likely means reduced traffic in practice. Concerning BSP barriers, *dddWT* has the big advantage that only one prefix computation is required that takes care of the whole wavelet tree, whereas in *dbsortWT*, we require one prefix sum computation per level. Another advantage of *dddWT* is that it is completely independent of the locally employed sequential wavelet tree construction algorithm. To that end, research advancements in the sequential or multi-core parallel, shared memory construction can be applied to *dddWT* directly.

However, *dbsortWT* and *dsplitWT* (distributed split) both have a property that *dddWT* does not have: they split up the text, put it in the order of the bits on the current level of the wavelet tree and balance the re-ordered text across the processing elements. This induces a locality of data that belongs to the same wavelet subtree: in the case of *dsplitWT*, this is done explicitly by design, whereas it happens implicitly in *dbsortWT* by using a balanced concatenation of buckets. Depending on the network topology, this may be an advantage if processing element ranks are assigned in a way that resembles the communication costs in the network, e.g., if processing elements that are physically far from each other also have a high difference in their ranks.

4 Implementation

In this chapter, we discuss the implementations of the distributed wavelet tree construction algorithms presented in chapter 3 which we use for the practical evaluation in chapter 5. The source code is written in C++14 and is available in a public repository[3].

We implement algorithms for two different *worlds*: the *Thrill* world is based on the Thrill C++ framework for distributed big data processing developed by Bingmann et al. [3] and makes use of its high-level programming interface. The *MPI* world revolves around the message-passing interface that comes as C libraries[4].

Note that there is no implementation of the distributed split algorithm for Thrill. This is because Thrill's high-level programming interface provides no control over which processing elements are used to process a certain DIA. However, this kind of control is essential for the distributed split algorithm, which is why there is no way to implement it using Thrill's API. Furthermore, there is no implementation of wavelet tree construction using sss-sort in MPI. The reason for this is the inherently poor load balancing of sss-sort in the face of wavelet tree construction as described in section 3.4.1. Despite this, there is a Thrill implementation using sss-sort as it was the first implementation written in the scope of this thesis (before the load balancing issues were discovered). We keep the *thrill-sss-sort* implementation to demonstrate the issue in our practical evaluation.

4.1 Prerequisites

We first cover the implementation of non-trivial operations and data structures required for the implementations of the algorithms.

4.1.1 Output Format Specification

For the sake of reusability of the output produced by the implementations (e.g., for decoding and verification), we first specify the output format of wavelet trees on a distributed file system (DFS).

[3] All source code related to the implementations of thesis is available at `https://github.com/pdinklag/distwt`.

[4] We use the widespread *Open MPI* (`https://www.open-mpi.org/`) throughout development but the *Intel MPI Library* (`https://software.intel.com/en-us/mpi-library`) for the practical evaluation as it is optimized for Intel clusters. MPI source code is typically compatible to both MPI implementations.

Level bit vectors. As specified in chapter 3, we want the wavelet tree in its levelwise representation, consisting of $\lceil \log \sigma \rceil$ bit vectors of length n each. Because p processing elements will be writing bit vectors to the DFS simultaneously, it would be problematic, in practice, to have them write to the same file. Instead, per bit vector (i.e., per level), we allow each processing element to write into its own file in a common directory on the DFS. To that end, each level of the wavelet tree will be stored in p files of $\lceil \frac{n}{p} \rceil$ bits. If needed, these files can be concatenated — with possible byte-alignment bits in mind — in a subsequent step that we do not regard any further.

Auxiliary information. In order to be able to decode the wavelet tree and restore the original input text T (e.g., for the purpose of verification), we also need to store auxiliary information about T. Conveniently, the *histogram* of T implicitly contains all the information we need:

- The sum of the occurrences of all symbols equals the text length n.
- The number of non-zero entries in the histogram equals the size of the effective alphabet σ, which can be used to compute the height of the wavelet tree.
- Furthermore, the non-zero entries can be used to map the effective alphabet $[0, \sigma)$ back to the input alphabet Σ.
- As shown in section 2.2.1, the histogram can be used to compute the sizes of the wavelet tree's nodes and therefore the entire tree structure.

Since the histogram is required in order to find σ and compute the effective transformation of the input, we choose to store it in a single file on the DFS next to the bit vectors after it has been computed.

4.2 Thrill Implementations

One of the initial goals of this thesis was to implement a wavelet tree construction algorithm in Thrill. For this reason, we first look at the Thrill implementations before proceeding to MPI.

4.2.1 Thrill-Specific Prerequisites

We recall some properties of Thrill's distributed immutable arrays (DIA, see section 2.3.3): there is no information exposed as to where (i.e., on which processing element) or — because of lazy evaluation — even *whether* their entries currently exist. The only way to process a DIA is by using the sequence-based operations provided by the Thrill API. This has big consequences on the way we design algorithms. In this section, we see how we cover some basics using Thrill's toolset.

The input text can be read from disk using the *ReadBinary* source node, which will provide it in a DIA of characters.

Histogram and effective transformation. Thrill already provides all the tools we need to implement Algorithm 2 (from section 3.1) to retrieve the histogram and effective transformation of the input.

The local histogram computation can be done with one scan of the input DIA using the local *Map* operation. As the mapping function, we use an extended identity mapping that also increases the local occurrence counter for each processed character. Thrill provides an implementation of *AllReduce*, which we use to compute the global histogram from the local counters. Finally, the effective transformation is naturally a mapping function by itself. We apply it using another *Map* operation on the input DIA.

Representation of wavelet trees. We are dealing with two representations of the wavelet tree: the node-based representation as an intermediate result during the domain decomposition algorithm, and the levelwise representation for the final result.

In either case, the contents of nodes or levels are bit vectors. Because Thrill does not provide a bit DIA, we use DIAs of r-bit integers to simulate them. On r-bit architectures (e.g., $r = 64$), this maximizes the number of bits packed into a single register. An alternative way to implement bit DIAs would be DIAs of boolean values. However, alignment rules would cause each item to be stored in a byte, essentially wasting seven bits per DIA element.

We use arrays[5] of bit DIAs for our wavelet tree representations. For the levelwise representation, we can simply use an array of $\lceil \log \sigma \rceil$ bit DIAs. Because the wavelet tree is a binary tree of size $2\sigma - 1$, we can use an array of size $2\sigma - 1$ bit DIAs to store the node-based representation, where the i-th entry represents the node with BFS rank i and its children are located at positions $2(i + 1) - 1$ and $2(i + 1)$.

DIA Concatenation. A key task in all of the algorithms from chapter 3 is *concatenation*: for the merge step in the domain decomposition algorithm, we concatenate bit vectors and for bucket sort, we concatenate buckets (strings).

Unfortunately, concatenation of DIAs is a very heavy operation in Thrill. This is due to the fact that Thrill does not keep any central information about where any part of a DIA is located (because it may not even exist yet). While Thrill does provide a distributed *Concat* operation, it is recommended not to use it if there are other ways to achieve a concatenation. In our use cases, because the sizes of wavelet tree nodes can be precomputed from the histogram, we can use the following sequence of operations:

1. We map each element x of the input DIAs to a tuple (k, x), where k is the item's index in the *concatenated* sequence. Thrill provides the operation *ZipWithIndex* that does this for a single DIA. We add our precomputed offsets for each DIA to the indices provided by Thrill in order to compute k.

[5] In practice, we use `std::vector` from the Standard Template Library.

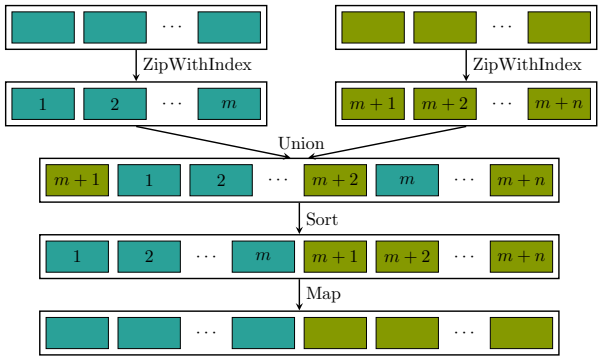


Figure 10: Four-step concatenation of two DIAs of sizes m and n, respectively. The numbers represent the augmentation of items x to indexed tuples (k, x). The coloring is only for visualization purposes and does not represent any information available to Thrill. The order of elements in the DIA resulting from the *Union* operation is arbitrary and only exemplified in this figure. It may be any other order in practice, making the sorting necessary.

2. We then use Thrill's local *Union* operation to merge the input DIAs. The result is a single DIA that contains all elements (our tuples (k, x)) in unspecified order.
3. The concatenation can now be done using the *Sort* operation. We sort the union created in the previous step by the indices k that we gave to each element in the first step. The result of the sort operation is the concatenated DIA. Because Thrill uses sss-sort (section 3.4.1), the concatenation can also be expected to be balanced across our processing elements.
4. We no longer need the indices k and thus map all tuples (k, x) back to just x.

Figure 10 shows an example. While this seems like a long detour to achieve concatenation, it is actually less expensive than a generic concatenation, because we give Thrill valuable extra information by adding the precomputed node sizes to the indices. In a generic concatenation, this information needs to be extracted from the DIAs first, which is a very costly endeavor as we confirm in our practical evaluation.

Concatenation of bit DIAs. The concatenation of bit DIAs, as required by the domain decomposition algorithm, holds an additional challenge: because we use r-bit words to back bits, each part of every bit DIA on every processing element may contain $a < r$ *alignment bits* that need to be purged by the concatenation.

Because Thrill has no explicit support for bit DIAs, we are not aware of a better way to achieve this other than mapping the bit DIA to a temporary DIA of boolean values excluding the alignment bits[6], and then map the result back to a bit DIA. The temporary waste of seven alignment bits for each element due to using a boolean DIA needs to be tolerated.

[6]Thrill provides a *FlatMap* operation, where the mapping function may emit an arbitrary number of elements to map a value to. We make use of this to emit zero elements for the alignment bits, which effectively removes them.

4.2.2 Stable Sorting in Thrill

Thrill features a distributed *Sort* operation to sort a DIA according to a given order (defined by a comparison function). However, this sort operation is not stable. Thrill's sorter is a distributed implementation of super scalar sample sort (sss-sort, see section 3.4.1). We recall the single stages of the sorting:

1. Each processing element randomly selects a *sample* of local items and sends it to the master.
2. The master selects *splitters* and sends them to all processing elements. The splitters essentially represent sort bucket boundaries and each processing element is assigned one of these buckets.
3. Each processing element distributes their local items according to the splitters, sorting them on a processing element rank level.
4. Finally, each processing element locally sorts its respective bucket. The items are then globally sorted.

We analyze Thrill's implementation of sss-sort and identify the following causes for the sorting not being stable:

- The local sorting of the buckets is done using `std::sort` by default, which is not stable.
- In case any bucket becomes too large for the local sorting to be done in memory, Thrill externalizes it into several local files. These files are sorted one by one and the sorted sequences are then merged in a multiway merge step. This multiway merging is not stable by default.
- The distribution of items to the corresponding processing elements according to the splitters is done asynchronously. This means that a processing element scanning a late part of the input sequence may send an item to its bucket before a processing element scanning an early part of the input does. In this case, the respective bucket is no longer in input order and stability is lost.

We implement a *SortStable* operation that has since been merged into the official repository[7]. To achieve this, we apply the following modifications to Thrill's original *Sort*.

Local sorting. The first cause for instability – the use of `std::sort` for local sorting – is easily rectified by using `std::stable_sort` instead.

Local multiway merging of files. We enable stable multiway merging for the case that a processing element's bucket is split into multiple files. A file with a lower index contains earlier items from the input sequence. Therefore, stability can be achieved by extending

[7]The official code repository for Thrill is found at `https://github.com/thrill/thrill`. The stable sort operation was incorporated into Thrill due to the following pull request: `https://github.com/thrill/thrill/pull/181`.

Algorithm 6: Wavelet tree construction using stable sorting in Thrill. This first variant uses our *StableSort* operation, which is a stable implementation of distributed super scalar sample sort.

Input : effective alphabet size $\sigma \in \mathbb{N}$, text DIA $T \in [0, \sigma)^n$
Output: bit DIA $B_\ell : \mathbb{B}^n$ for each level $\ell \in [0, \lceil \log \sigma \rceil)$

```
1 Function thrillSssSortWT(σ, T)
2     for ℓ = 0 to ⌈log σ⌉ − 1 do
3         r := ⌈log σ⌉ − 1 − ℓ   // right shift operand
          // construct bit vector
4         B_ℓ := Map(T, {c ↦ ((c ≫ r) AND 1)})
5         if ℓ is not the last level then
              // sort text
6             T := StableSort(T, {(c1, c2) ↦ ((c1 ≫ r) < (c2 ≫ r))})
7     return (B_0, ..., B_{⌈log σ⌉−1})
```

the comparison function of elements: if two items from different files are equal, the one from the file with the lower index is taken first.

Item distribution. The asynchronous distribution of items to their respective buckets needs to be synchronized so that items received from processing elements with lower ranks appear first in the bucket. Thrill internally provides different types of *streams* which are used to control this. We replace the *MixStream* used for reception of items by a *CatStream*, which operates exactly as desired: it orders received items according to the rank of the sending processing elements.

4.2.3 Using Stable Sorting in Thrill

With stable sorting now available, we implement *thrillSssSortWT* (Algorithm 6) for constructing the wavelet tree in Thrill using the stable sorting approach as described in section 3.4.

We proceed levelwise for each of the $\lceil \log \sigma \rceil$ levels of the wavelet tree. For level ℓ, we determine the *right shift operand* r (line 3). The result of shifting any symbol $c \in [0, \sigma]$ to the right by r (i.e., perform an integer division of c by 2^r) is the ℓ-bit prefix of c. We use the bit prefix (a) to compute the bit vector B_ℓ (line 4) by mapping each symbol to its ℓ-least significant bit, and (b) as the key for the stable sorting done in line 6. The reordering of the text is not necessary after constructing the final level of the wavelet tree.

4.2.4 Using Bucket Sort in Thrill

We have thoroughly described the issues of sample sort in the scope of wavelet tree construction (section 3.4.1). In contrast, we saw that bucket sort suits wavelet tree construction in a natural manner. We implement *thrillBsortWT* (Algorithm 7) in a way similar to the

Algorithm 7: Wavelet tree construction using a stable bucket sort in Thrill.

```
   Input  : effective alphabet size σ ∈ ℕ, text DIA T ∈ [0, σ)^n
   Output: bit DIA B_ℓ : 𝔹^n for each level ℓ ∈ [0, ⌈log σ⌉)
1  Function thrillBsortWT(σ, T)
2      for ℓ = 0 to ⌈log σ⌉ − 1 do
3          r := ⌈log σ⌉ − 1 − ℓ   // right shift operand
           // construct bit vector
4          B_ℓ := Map(T, {c ↦ ((c ≫ r) AND 1)})
5          if ℓ is not the last level then
               // fill sort buckets
6              for v = 0 to 2^ℓ − 1 do
7                  C_v := Filter(T, {c ↦ ((c ≫ r) = v)})
               // concatenate buckets
8              T := Concat(C_0, C_1, ..., C_{2^ℓ − 1})
9      return (B_0, ..., B_{⌈log σ⌉−1})
```

previous algorithm, but replace the stable sorting by filling and then concatenating sort buckets.

Each sort bucket C_v for node $v \in [0, 2^\ell)$ on level ℓ is a DIA of characters. We produce the buckets by *filtering* characters from the current text according to their ℓ-bit-prefixes (line 7). To stably reorder the text, we then only need to *concatenate* the buckets (line 8, in the manner described in section 4.2.1).

4.2.5 Domain Decomposition in Thrill

The *Filter* operation we used for bucket sort is a local operation in Thrill, that is, it is applied to a DIA's local part on each processing element. We can make use of this to implement the domain decomposition algorithm. To construct the wavelet tree locally for a processing element's part of the input text DIA, we use the recursive algorithm *simpleWT* described by Navarro (Algorithm 1 in section 2.2.3) and apply it to Thrill in Algorithm 8 (*thrillRecursiveWT*). We initiate *thrillRecursiveWT* for the root node $v = 0$, the entire effective alphabet $[a, b] = [0, \sigma)$ and the full input text T.

We use the alphabet's middle m to compute the bit vector for node v using a mapping function (line 5), as well as to split up T into T^0 and T^1 for recursing on the left and right child of v using filters (line 7 and line 9). The *Map* and *Filter* operations are local operations in Thrill and cause no communication. Therefore, Algorithm 8 is a sequential algorithm.

In order to construct the wavelet tree, the local wavelet trees of all processing elements need to be merged. As we have analyzed in section 3.2, this is done by concatenating the bit vectors of all nodes on the same level. We implement *thrillWTMerge* (Algorithm 9).

Algorithm 8: Sequential recursive wavelet tree construction in Thrill.

Input : node BFS rank $v \in [0, 2\sigma - 1]$, alphabet interval $[a, b] \subseteq [0, \sigma)$, text DIA $T \in [a, b]^n$
Output: bit DIA $B_u : \mathbb{B}^*$ for all nodes u in v's subtree.

```
1  Function thrillRecursiveWT(v, a, b, T)
2      if a = b then
3          return ϵ // leaf
       // construct bit vector
4      m := ⌊(a+b)/2⌋
5      B_v := Map(T, {c ↦ (c > m)})
       // recurse on left child
6      T^0 := Filter(T, {c ↦ (c ≤ m)})
7      thrillRecursiveWT(T^0, a, m, 2(v + 1) − 1)
       // recurse on right child
8      T^1 := Filter(T, {c ↦ (c > m)})
9      thrillRecursiveWT(T^1, m + 1, b, 2(v + 1))
10 return (B_v, B_{2(v+1)−1}, B_{2(v+1)}, ...)
```

Algorithm 9: Merge of a node-based wavelet tree to the levelwise wavelet tree in Thrill.

Input : effective alphabet size $\sigma \in \mathbb{N}$, bit DIA B_v for each node $v \in [0, 2\sigma - 1)$.
Output: bit DIA $B_\ell : \mathbb{B}^n$ for each level $\ell \in [0, \lceil \log \sigma \rceil)$

```
1  Function thrillWTMerge(σ, B_0, ..., B_{2σ−2})
2      for ℓ = 0 to ⌈log σ⌉ − 1 do
3          u := 2^ℓ − 1  // first node on level ℓ
4          k := 2^ℓ  // number of nodes on level ℓ
           // concatenate node bit vectors on level
5          B_ℓ := Concat(B_u, B_{u+1}, ..., B_{u+k−1})
6      return (B_0, ..., B_{⌈log σ⌉−1})
```

The *Concat* operation (line 5 of Algorithm 9) is replaced by the concatenation method described previously. Since we are concatenating bit vectors, we also need to take care of alignment bits that occur for each processing element.

4.2.6 Verification of Wavelet Trees

To verify the correctness of our implementations in practice, it is crucial to be able to *verify* a wavelet tree after construction. The wavelet tree is an encoding of the input text, since it can be used to answer *access* queries for every position $i \in [0, n)$. We use this property to *decode* the wavelet tree and conclude its correctness if the decoded text equals the original input text.

We recall that the bits of the binary representation of $T[i] \in [0, \sigma)$ encode the path from the wavelet tree's root to the node that represents $T[i]$. Following that, each level of the wavelet tree contains one bit of $T[i]$. On the first level, we conveniently find the most

Algorithm 10: Decoding of the wavelet tree to restore the original text in Thrill.

Input : effective alphabet size σ, bit vectors $B_\ell \in \mathbb{B}^n$ for each level of the wavelet tree
Output: effective transformation $T_{\text{eff}} \in [0, \sigma)^n$ of the input text

1 **Function** *thrillDecodeWT*(σ, B_0, …, $B_{\lceil \log \sigma \rceil - 1}$)
2 $S := ((0,1),(0,2),\ldots,(0,n))$
3 **for** $\ell = 0$ **to** $\lceil \log \sigma \rceil - 1$ **do**
4 $l := \lceil \log \sigma \rceil - 1 - \ell$ *// left shift operand*
// read the ℓ-most significant bit from B_ℓ into c components
5 $S := \text{Zip}(B_\ell, S, \{((c,k),b) \mapsto (c \text{ OR } (b \ll l), k)\}$
// stably sort S according to the ℓ-bit prefixes
6 $S := \text{SortStable}(S, \{((a,k_a),(b,k_b)) \mapsto (a < b)\})$
// put S back into its original order
7 $S := \text{Sort}(S, \{((a,k_a),(b,k_b)) \mapsto k_a < k_b\})$
// map tuples back to the effective symbols
8 **return** $\text{Map}(S, \{(c,k) \mapsto c\}$

significant bit at position i. On level $\ell + 1$, however, the bits are reordered according to the bits of B_ℓ, as can be seen easiest in the stable sorting algorithm.

We transform the stable sorting algorithm to decode the wavelet tree level by level using tuples $(c, k) \in [0, \sigma) \times [0, n)$, where c is the *symbol* component used to fill the bits of the k-th symbol from the original input text. Initially, we have $S = ((0,0),(0,1),\ldots,(0,n-1))$, i.e., all symbols are initialized as zero. On level ℓ, we read the ℓ-most significant bit of each symbol from B_ℓ into the symbols of S. Then, we *stably sort* S according to the ℓ-bit prefixes of its symbols — which are equal to the current c components — and proceed to level $\ell + 1$. Following the idea of the stable sorting algorithm, the i-th bit is now the next bit of $S[i]$'s symbol. After processing all of the $\lceil \log \sigma \rceil$ levels in this manner, each symbol in S has its $\lceil \log \sigma \rceil$ bits. In a final step, we sort S according to the k component, which mark the original positions of the symbols in the input text. We can then drop these indices and have restored the input. We implement this in *thrillDecodeWT* (Algorithm 10).

We use Thrill's *Zip* operation to combine S with bit vector B_ℓ on each level (line 5). To set the ℓ-most significant bit for each tuple, we use the *left shift operand* l. Because the index components of S are distinct, the final sort to put S back into text order (line 7) needs not be stable. We finally drop the indices using a *Map* operation.

Note that from the wavelet tree itself, we can only restore the *effective transformation* T_{eff} of the input text. To regain the original input text, we need to revert the effective mapping, which can be restored from the histogram that we store besides the level bit vectors. We will not regard this here any further.

Using bucket sort. The balancing issues of sss-sort for wavelet tree construction due to a low number of distinct sort keys also apply to the verification. We can solve the issue the

same way as we do for construction and replace the stable sort operation in Algorithm 10 (line 6) by bucket sort to achieve better load balancing.

4.3 MPI Implementations

Thanks to the high abstraction level, a big advantage of Thrill is the simplicity with which algorithms can be expressed. The merge step of the domain decomposition is a notable example comparing Algorithm 3 (*dddWTMerge*) and Algorithm 9 (*thrillWTMerge*). However, these conveniences come at the cost of less control over what a certain processing element is doing. To that regard, the distributed split algorithm for wavelet tree construction is a good example that demonstrates the limitations of Thrill: there is no way to implement it using Thrill's high-level API.

As these limitations became clear during work on the thesis, the idea came up to also provide low-level implementations of wavelet tree construction algorithms using MPI. We discuss these implementations in this section, albeit in less detail compared to the Thrill implementations. In general, they are much closer to the pseudocode listings of chapter 3 than the Thrill implementations.

4.3.1 MPI-Specific Prerequisites

Looking at a low-level interface like MPI, it becomes clear how much functionality Thrill has out of the box. The following are some of the requirements we need to implement from the ground up using the very basic toolset provided by MPI:

- Context management — we need an overview over the available processing elements. This also includes management of communicators, which we use for the distributed split implementation.
- Input partitioning — we need to partition the input file so each processing element initially holds a part of the same size.
- Statistics tracking — MPI does not count the number of bytes sent and received by itself. This information is important for our practical evaluation. We implement wrappers around MPI's send, receive and advanced operations to track or estimate[8] the number of bytes sent/received.
- Message management — the buffers used to send and receive messages need to be allocated and managed manually; MPI provides no functionality to that regard. Furthermore, messages containing bit vectors need to be packed into data types supported by MPI (e.g., integers).

[8] MPI does not specify what algorithms are used for advanced operations such as *AllReduce* or prefix sums. Therefore, we estimate the traffic caused by these operations based on the theoretical bounds given in section 2.4.

We are not going into greater detail about these topics. For some common distributed operations, MPI does provide implementations out of the box. The functions *MPI_Allreduce* and *MPI_Exscan* (exclusive prefix summing) are of particular interest for us. We use the *MPI_Allreduce* to implement the preliminary histogram computation and effective transformation according to Algorithm 2 (section 3.1).

Representation of wavelet trees. Contrary to the Thrill API, we are not bound to an abstract concept like DIAs for our MPI implementations. To represent bit vectors, we choose the bit vector data type provided by the Standard Template Library[9] (STL) to store bit vectors. Furthermore, following the same ideas as for the Thrill implementations, we use arrays of bit vectors to represent the levels or nodes of the wavelet tree.

For sending bit vectors as MPI messages, they need to be packed into arrays of integer types MPI is able to handle. In order to maximize the number of bits packed into one element, we choose `MPI_UNSIGNED_LONG_LONG` which, at the time of writing, corresponds to unsigned 64-bit integers on 64-bit architectures.

Range message protocol. Many of the messages sent during the wavelet tree construction algorithms are substrings or bit vector ranges. The recipient of these messages needs to know the *interval boundaries* of these ranges in order to be able to place them at the correct position locally.

We define a simple *range message protocol* that consists of two MPI messages. The *header* message contains two integer values: (1) the *global* position of the left interval boundary, which the recipient translates back to a local position using its rank, and (2) the *length* of the range. The *payload* message then contains the actual substring or bit vector range. Because the overhead is constant for each message, use of this protocol does not affect any of the asymptotic word counts analyzed in chapter 3.

4.3.2 Domain Decomposition in MPI

The MPI-relevant part of the domain decomposition revolves around the balancing merge operation, since the sequential construction of local wavelet trees requires no communication.

Sequential construction. For sequential construction, we implement the fastest known practical algorithm *pcWT* due to Fischer et al. [10]. *pcWT* constructs the wavelet tree in a bottom-up manner, using the histogram of the input text to compute node boundaries for each level. This way, bit vectors for each level can be computed directly from the input text, which needs no longer be reordered, saving both time and memory.

[9] Common STL implementations provide a space-efficient implementation of bit vectors in `std::vector<bool>`, see `https://en.cppreference.com/w/cpp/container/vector_bool`.

Balanced range transmission. The merge of local node bit vectors into the global level bit vector consists primarily of the bit redistribution in order to concatenate the node bit vectors in a balanced manner (see section 3.2.2). The key operation performed here is line 14 in Algorithm 3: in the pseudocode, we simply assign the local bit vector range into the global bit vector. In practice, this assignment is more complex: the executing processing element first needs to determine the recipients of the bits and then send them.

We implement the algorithm *sendRangeBalanced* (Algorithm 11) that refines the mentioned pseudocode line of Algorithm 3. The algorithm sends the local item range $[a, b]$ from an array A_i (i.e., we send $A_i[a], A_i[a+1], \ldots, A_i[b]$) to recipients based on the *offset* d. The offset marks the position of $A_i[0]$ in the *global* array A (with $|A| = n$) and may be the result of a prefix sum computation, as is the case in Algorithm 3.

Each processing element has $\lceil \frac{n}{p} \rceil$ items before the redistribution and will receive $\lceil \frac{n}{p} \rceil$ items. Because of this, the local range $[a, b]$ (also of length at most $\lceil \frac{n}{p} \rceil$) will be sent to at most two different recipients j_a and j_b, which we determine using the offset d in line 3. In the case that we have only one recipient, we send the whole local range $A_i[a, \ldots, b]$ to it using the range message protocol defined earlier (line 6). Otherwise, if there are two recipients, we first need to find the *splitter* $d_b := j_b \lceil \frac{n}{p} \rceil$ (line 9), which is the smallest

Algorithm 11: On processing element i, sends a range of at most $\lceil \frac{n}{p} \rceil$ local items from A_i to recipients so the global array A is balanced across all p processing elements.

Input: local node rank $i \in [0, p)$, local array A_i of length at most $\lceil \frac{n}{p} \rceil$, offset $d \in \mathbb{N}$, local interval $[a, b] \in \mathbb{N} \times \mathbb{N}$.

1 **Function** *sendRangeBalanced(i, A_i, d, a, b)*
2 **if** $a < b$ **then**
3 $j_a := \left\lfloor \frac{d+a}{\lceil n/p \rceil} \right\rfloor$, $j_b := \left\lfloor \frac{d+b}{\lceil n/p \rceil} \right\rfloor$ *// determine recipients*
4 **if** $j_a = j_b$ **then**
 // the whole range goes to the same recipient
5 $m := b - a + 1$
6 **Send** $(d + a, m)$ **to** j_a *// header*
7 **Send** $A_i[a, \ldots, b]$ **to** j_a *// payload*
8 **else**
 // there are two different recipients, but it holds that $j_b = j_a + 1$
9 $d_b := j_b \lceil \frac{n}{p} \rceil$ *// splitter*
10 $m_a := d_b - (d + a)$ *// number of items to j_a*
11 $m_b := (d + b) - d_b$ *// number of items to j_b*
 // send first part to j_a
12 **Send** $(d + a, m_a)$ **to** j_a *// header*
13 **Send** $A_i[a, \ldots, a + m_a]$ **to** j_a *// payload*
 // send second part to j_b
14 **Send** (d_b, m_b) **to** j_b *// header*
15 **Send** $A_i[a + m_a, \ldots, b]$ **to** j_b *// payload*

global position in A that belongs to the second recipient. Using d_b, we partition the local interval $[a, b]$ into two parts and send each part to the corresponding recipient.

Algorithm 11 is useful in all cases where we need to reorder an array A in a balanced manner, one of these cases being the domain decomposition merge. We use the algorithm refine the bit distribution in Algorithm 3 for our MPI implementation.

4.3.3 Using Distributed Split in MPI

We implement the load balancing variant of the distributed split operation described in section 3.3.3. For this, we make use of *communicators*, which form an isolated communication group of a set of processing elements. After performing a split operation, the processing elements that process T^0 or T^1 — the symbols that were assigned a 0-bit or 1-bit on the current level, respectively — are grouped into a corresponding communicator as shown in Figure 11.

The use of communicators allows MPI to optimize communication between the grouped processing elements, but it also makes the implementation itself easier: processing element ranks within a communicator are re-mapped locally so that rank zero corresponds to the first processing element in the communicator, regardless of its actual global rank. In case a communicator is left with only one single processing element (e.g., on the bottom level in Figure 11), we use *pcWT* [10] for sequential construction of the remaining wavelet subtree.

4.3.4 Using Stable Sorting in MPI

Due to the load-balancing issues of distributed super scalar sample sort in the scope of wavelet tree construction, we only implement the algorithm using bucket sort (Algorithm 5 in section 3.4.2) in MPI.

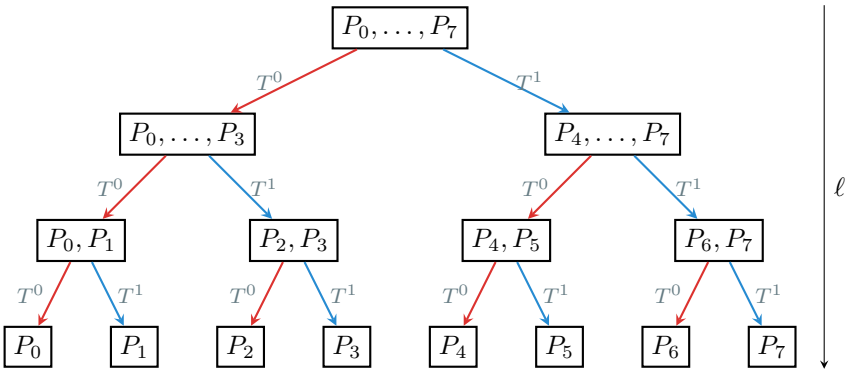


Figure 11: Overview of MPI communicators over the course of the distributed split algorithm implementation in MPI using eight processing elements. Each node represents a communicator, labeled by the processing elements that belong to it. On the top level, the entire text is processed and split T into T^0 and T^1 using all eight processing elements. Processing elements P_0 to P_3 then continue processing T^0 recursively and P_4 to P_7 recurse on T^1 on the second level and so on.

Similar to the implementation of the domain decomposition's merge operation, there is a part in the pseudocode listing of Algorithm 5 that proves to be more complex in practice. In this case, it is the *balanced concatenation* of buckets (line 11 to line 15). We have at most $\lceil \frac{n}{p} \rceil$ symbols in the local buckets and because the concatenation is balanced, all processing elements will have $\lceil \frac{n}{p} \rceil$ symbols after the concatenation. To that end, the same preconditions apply as for the concatenation of node bit vectors in the domain decomposition's merge operation. Therefore, we employ the *sendRangeBalanced* algorithm (Algorithm 11) in the MPI implementation of the bucket sort algorithm.

5 Practical Evaluation

In the following, we present the practical evaluation of the implementations from chapter 4. We conduct weak and strong scaling experiments in order to analyze their scalability and get an insight on what algorithms are the most suitable for the distributed construction of wavelet trees. Furthermore, we conduct a select few experiments specifically for our Thrill implementations for deeper insight.

5.1 Preparations

Before we proceed to the evaluation itself, we discuss some necessary additions to our implementations that are not related to the implemented algorithms, but rather serve as a preparation of our evaluation.

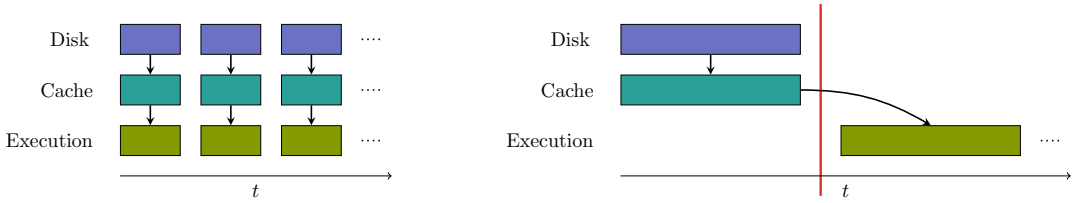
Lazy evaluation in Thrill vs. benchmarking. Thrill makes use of lazy evaluation wherever possible. This means that no operation is actually executed before its result is needed (e.g., if a distributed operation is due on a DIA) and if possible, items are pipelined through these operations instead of producing the result DIA as a whole and then proceed with the next operation. This also affects the *ReadBinary* source operation to read the input from disk.

Because reading from disk may be subject to heavy performance fluctuations[10], it is common to exclude the initial read from benchmarks for practical evaluations. This is usually done by starting the timer used for measurements only after the input has been initially read from disk and cached. However, that point in time does not exist in Thrill's concept of lazy evaluation and pipelining: data items would not be read from disk until they are actually being used. This results in unpredictable scattering of I/O operations over the program flow, distorting the benchmarking process. This behavior is visualized in Figure 12a.

To resolve this clash of interests and to create a fixed point in the program flow where we can be sure that the input has been read from disk and cached as depicted in Figure 12b, we have to trick[11] Thrill into believing that the data is being used before starting our

[10] In our case, we use a distributed file system with no control over how many parties are using it simultaneously.

[11] This issue has been discussed publicly with the authors of Thrill here: `https://github.com/thrill/thrill/issues/188`.



(a) Processing a DIA from disk using lazy evaluation. The actual reading is done when necessary, so that there is no specific point in time where the entire DIA is cached.

(b) Forcing the data to be read from disk and cached before starting the actual program. The vertical line marks the point in time at which the DIA has been cached in its entirety.

Figure 12: Abstract visualization of disk data flow in lazy evaluation against our desired data flow for benchmarking over time. We visualize three layers: the *Disk* layer shows data stored on disk with costly access and unpredictable delays (e.g., a distributed file system), the *Cache* layer shows data that is cached and ready to be used by the program (e.g., in RAM or a local file system) and the *Execution* layer displays the data usage in the program execution (e.g., in registers or CPU caches).

actual program: we perform an *identity mapping* on the input and request its size. Then, we use the identity-mapped DIA as the input of our actual program. This forces Thrill to read from disk, apply the mapping and count the number of elements, which are then cached for the rest of the program execution.

5.2 Experimental Setup

We conduct experiments for the algorithms developed in chapter 3 and implemented in chapter 4. Table 5 gives an overview.

Implementation	Description
thrill-dd	Domain decomposition in Thrill.
thrill-sss-sort	Construction using stable sorting with sss-sort in Thrill.
thrill-bsort	Construction using stable sorting with bucket sort in Thrill.
mpi-dd	Domain decomposition in MPI, using the *pcWT* algorithm due to Fischer et al. [10] for local sequential wavelet tree construction.
mpi-dsplit	Construction using the load-balancing distributed split operation in MPI. As for mpi-dd, *pcWT* is used for local sequential computations in case only one processing is left to process a wavelet subtree.
mpi-bsort	Construction using stable sorting with bucket sort in MPI.

Table 5: Overview over the implementations of distributed wavelet tree construction algorithms written in the scope of this thesis.

Text	n	σ	Description / Source
CommonCrawl	137.56 GiB	242	Texts of web crawl data from Common Crawl with markup commands and meta data removed (`http://commoncrawl.org/`).
DNA	149.31 GiB	4	Raw DNA string from 1000 Genomes with FASTQ information removed (`http://www.internationalgenome.org/`).
Wiki	127.72 GiB	213	XML-formatted Wikimedia of English, Finnish, French, German, Italian, Polish and Spanish Wikipedia articles (`https://dumps.wikimedia.org/`).

Table 6: Selected statistics and descriptions of the input text collection used for evaluation, where n is the total length of the available file and σ the size of the effective alphabet.

Cluster. The evaluation is conducted on the Linux cluster of the TU Dortmund University[12]. More precisely, we use the cluster's two-socket nodes, of which there are 316 in total. Each of these nodes has an Intel Xeon E5-2640v4 processor (ten cores running at 2.4 GHz with 25 MB of L3 cache) and 64 GB of RAM. The nodes are interconnected to a parallel file system (BeeGFS) via InfiniBand QDR (40 Gbit /s).

Software versions. We use the Thrill snapshot as of September 27, 2018 and modify the Slurm invocation scripts in order to make them work in our cluster. We compile Thrill, as well as the implementations of chapter 4, with the GNU `g++` compiler version 7.3.0 and link against the Intel MPI Library 2018.3.

Input text collection. We construct wavelet trees for (prefixes of) the three input texts listed in Table 6. All texts are stored as ASCII encoded files on the distributed file system.

[12]Linux Cluster Dortmund, 3rd generation (LiDO3). More information (in German) can be found here: `https://www.itmc.tu-dortmund.de/cms/de/dienste/hochleistungsrechnen/lido3/index.html`.

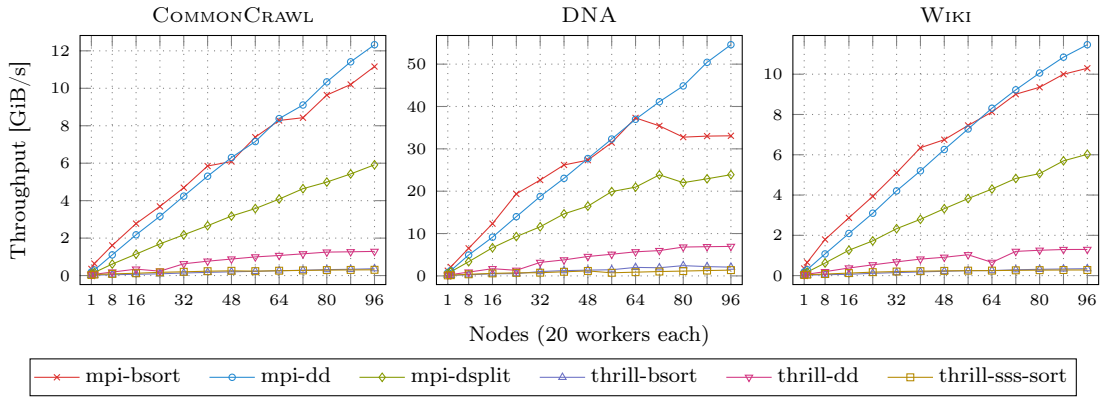


Figure 13: Median throughput over five iterations in the weak scaling experiments.

5.3 Weak Scaling Results

We conduct *weak scaling* experiments as follows: for N nodes with 20 processing elements per node, we process prefixes of length $n := N \cdot \text{GiB}$ of each input file, i.e., one gibibyte per node. We perform five iterations for each experiment and measure

1. the median *throughput* (i.e., the running time divided by the input size) of the histogram computation and construction of the wavelet tree,
2. the total *network traffic* caused by all nodes, excluding internal shared memory traffic between processing elements on the same node, and
3. the average *memory usage* (RAM) per node.

In the following sections, we present and discuss the results for each of the measures.

5.3.1 Throughput

Figure 13 shows the measured median throughputs of our implementations over five iterations.

Throughput of MPI implementations. The highest throughputs are achieved by the MPI implementations *mpi-dd* and *mpi-bsort*, where *mpi-bsort* seems to be slightly faster overall up to 64 nodes. The similarity of throughputs stems from the fact that *mpi-bsort* is practically just the merge operation of *mpi-dd* without the previous local wavelet tree construction. Besides, communication is more local in *mpi-bsort* as discussed in section 3.5. This results in more fast shared memory communication compared to *mpi-dd*. However, *mpi-bsort* requires more data to be communicated as it is sending substrings (one byte per character) rather than packed bit vectors (eight bits per byte), increasing the amount of handled data by a factor of eight, which likely dampens the throughput a bit compared to *mpi-dd*.

On DNA, the throughput of *mpi-bsort* appears to plummet going past the 64 node mark. An explanation is the small alphabet size of four, meaning we only construct two wavelet tree levels and perform only one bucket sort after constructing the first level. In this sort operation, there are only two very large buckets scattered over *all* nodes that need to be redistributed, i.e., the advantage of locality is not yet in effect here. Because there are no further levels, the locality advantage never applies as opposed to inputs with larger alphabets (CommonCrawl and Wiki). A similar effect is slightly less visible for *mpi-dsplit*, which also achieves data locality on deeper levels.

Among the MPI implementations, *mpi-dsplit* fares worst concerning throughput, and it is relatively easy to see why: it requires communication similar to *mpi-bsort* during construction (the split text T^0 and T^1, for a whole level, corresponds to the sort buckets, and after construction, it needs to perform a merge operation similar to *mpi-dd*. To that regard, *mpi-dsplit* is the combination of the heaviest operations of *mpi-bsort* and *mpi-dd*, which becomes visible in the throughput.

We see that for larger alphabets, all MPI implementations scale very well: doubling the number of nodes also nearly doubles the throughput. To exemplify this, on CommonCrawl, *mpi-dd* achieves a median throughput of 1.106 GiB /s using eight nodes on an 8 GiB prefix and a median throughput of 8.380 GiB /s using 64 nodes on an 64 GiB prefix, i.e., using eight times as many nodes here yields a median throughput that is about 7.6 times higher.

Thrill vs. MPI. The Thrill implementations have a notably lower throughput than the MPI implementations. We recall that a key operation in distributed wavelet tree construction is the balancing concatenation of bit vectors or strings. In our MPI implementations, we use the precomputed wavelet tree structure to simplify the concatenation: after a distributed prefix sum computation, the transmission of items to their recipients is done in a single linear pass over the local sequence. In Thrill, however, we can only make use of this information doing the detour via sorting of the entire sequence.

Throughput of Thrill implementations. We display the throughputs of the Thrill implementations with a higher scale in Figure 14 for better visibility.

Among the Thrill implementations, *thrill-dd* has the highest throughput and also scales well up to about 72 nodes: increasing the number of nodes by a factor of eight from 8 to 64 for CommonCrawl, we increase the throughput by a factor of nearly six (from 0.187 GiB /s to 1.072 GiB /s). The domain decomposition minimizes the number of concatenated items: after local computation of the wavelet trees, we pack the bit vectors of each node into 64-bit words. We concatenate only these packs, of which there are $\lceil \frac{n}{64} \rceil$ per concatenation and eliminate alignment bits afterwards.

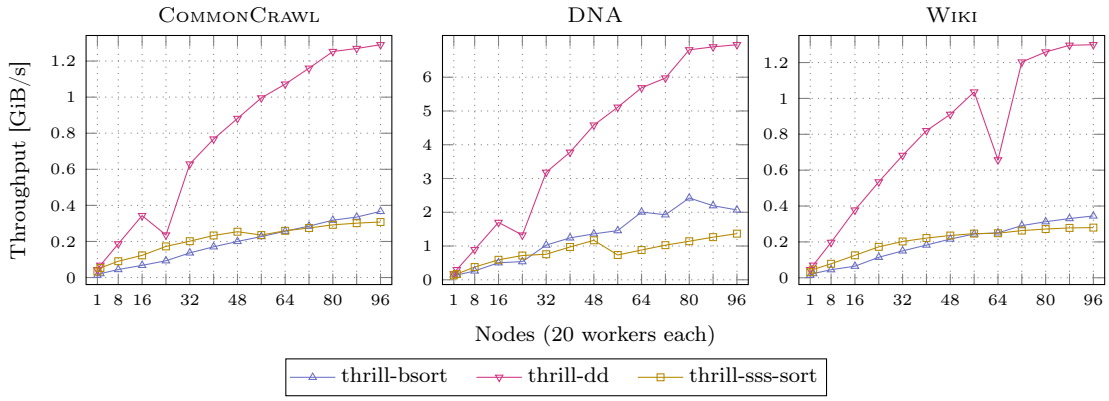


Figure 14: Median throughputs of Figure 13, but only displaying the Thrill implementations.

While we perform the same number of concatenations in *thrill-bsort*, we concatenate substrings there and the observations we already made for the MPI counterparts apply: we can only pack eight symbols (in 8-bit ASCII representation) into a 64-bit word, which results in $\lceil \frac{n}{8} \rceil$ packs to be concatenated, i.e., a factor of eight compared to *thrill-dd*. For this reason, the throughput of *thrill-bsort* is lower overall. However, we observe similar scalability: going from 8 to 64 nodes, we also increase the throughput by a factor of nearly six (from approximately 45 MiB /s to approximately 263 MiB /s).

For *thrill-sss-sort*, we see that going beyond a certain number of nodes does not help a lot to handle larger inputs. We recall that due to sss-sort's poor load balancing properties for wavelet tree construction discussed in chapter 3, there is a greater unbalance with an increasing number of processing elements after constructing the first level. The expected effect for our weak scaling experiments is well visible for Wiki: up to about 24 nodes, *thrill-sss-sort* scales linearly (going from eight nodes with a throughput of approximately 80 MiB /s to a throughput of approximately 177 MiB /s for 24 nodes). However, beyond that point, adding more nodes no longer increases the throughput that much (e.g., approximately 254 MiB /s for 64 nodes).

5.3.2 Network Traffic

Figure 15 shows the measured network traffic caused by our implementations. As one would expect, the amount of traffic rises with an increasing number of nodes for every implementation and every input text. We take a look at the differences.

Thrill vs. MPI. The domain decomposition implementations (*mpi-dd* and *thrill-dd*) cause the least amount of traffic. As previously noted, the amount of data that needs to be communicated is minimized naturally in this approach. The entire wavelet tree is constructed locally and the subsequent merge only requires transmission of bit vectors, which we pack into words of 64 bits each.

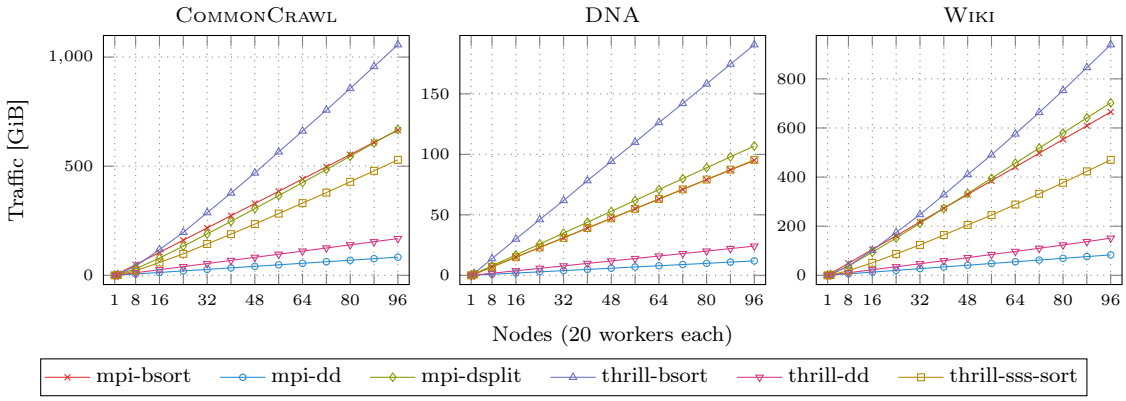


Figure 15: Network traffic in the weak scaling experiments.

Yet, the traffic of *thrill-dd* is about twice as high as that of *mpi-dd*. This is an effect of the way we concatenate bit vectors in Thrill: we need to build *indexed tuples* and the index means an additional word for each bit back word. We use 64-bit words to store the indices, thus we have precisely twice as much data to transmit. However, the fact that *thrill-dd*'s traffic is still second lowest indicates that traffic is not the cause for the poor throughputs of the Thrill implementations, but more likely the computational complexity that comes from the sorting required for concatenations.

Traffic of Thrill implementations. The traffic of *thrill-bsort* exceeds that of *thrill-dd* by a factor of approximately eight, which we can explain by the concatenations as well: in *thrill-dd*, we send $\frac{n}{64}$ indexed bit pack words of 64 bits each. With 64-bit indices, this results in $2n$ total bits. In *thrill-bsort*, we send substrings and may only pack eight characters into a 64-bit word (ASCII representation), each of which is extended by an index as well. Thus, we send $\frac{n}{8}$ indexed pack words of 64 bits each plus index — $16n$ bits in total.

The midfield is occupied by *thrill-sss-sort*. The only traffic it causes is for the stable sorts due to sss-sort. While the communication of bucket boundaries is negligible on a gigabyte scale, there is a lot of randomized shuffling of text characters between all of the nodes with no predictable data locality.

Traffic of MPI implementations. Comparing the traffic of *mpi-bsort* and *mpi-dd*, we can use a similar reasoning as previously for the Thrill counterparts: while we do not need any indices for communication and traffic is generally lower, the factor of eight between these two implementations remains because we send substrings instead of bit vectors.

The equivalence of the split texts in *mpi-dsplit* to the buckets of *mpi-bsort* is very visible in the traffic footprints. While *mpi-dsplit* requires a merge operation after construction and *mpi-bsort* does not, there is perfect data locality during construction in *mpi-dsplit*, because T^0 or T^1 are guaranteed to stay within the same communicator. This results in

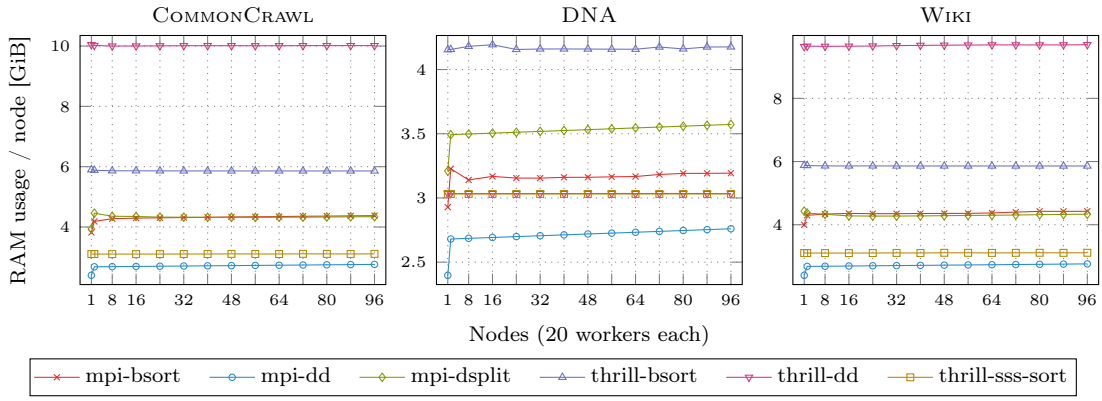


Figure 16: Average RAM usage per node in the weak scaling experiments.

more fast shared memory traffic that is not counted in the figure. Even though there are slight differences between the two implementations in terms of caused traffic, it is not really possible to favor either.

5.3.3 Memory Usage

Figure 16 shows the average per-node memory usage of our implementations in the weak scaling experiment.

Memory usage of Thrill implementations. We should note that we do not have complete insight into Thrill's memory management for DIAs and therefore, we cannot do a very precise memory analysis of the Thrill implementations. However, *thrill-dd* is clearly the least memory efficient implementation for texts with larger alphabets (CommonCrawl and Wiki). To further analyze this, we take a closer look at the RAM utilization for a 32 GiB prefix of CommonCrawl using 32 nodes in Figure 17.

We observe a peak of RAM utilization shortly after the 25 second mark. At this point, the local wavelet tree has been constructed and is cached in RAM. We are also performing the first merge operation, where we use the detour described in section 4.2.1 for concatenation. At the point during this first merge that resembles the peak, we additionally store indexed bit vector packs ($2n$ bits) and a temporary DIA of boolean values ($8n$ bits) that we build from the indexed packs to eliminate alignment bits.

A similar situation arises during *thrill-bsort*. Here, however, we did not previously compute a node-based representation of the wavelet tree, which explains the reduced memory usage compared to *thrill-dd* for texts with a larger alphabet.

Contrary to *thrill-dd* and *thrill-bsort*, *thrill-sss-sort* is very sparse concerning memory usage. In addition to the input and the already constructed wavelet tree, it only requires additional memory for the samples and splitters for sss-sort's distribution phase, as well as additional memory for local sorting with `std::stable_sort`.

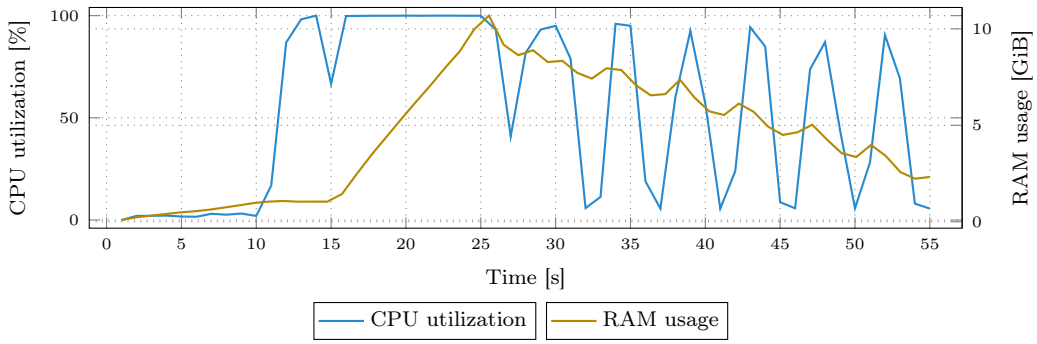


Figure 17: CPU and RAM utilization of the first node over time executing *thrill-dd* for 32 nodes on a 32 GiB prefix of CommonCrawl. The data has been measured using Thrill's own logging features. From the curves, we can estimate the phases of the implementation: before the 15 second mark, the input is being read from disk so CPU utilization is very low. Between the 15 and 25 second marks, the local wavelet tree is being constructed, utilizing the CPU intensively and using an increased amount of RAM over time. After that, we merge the seven levels of the wavelet tree one by one, causing alternating periods of CPU utilization and network synchronization.

Memory usage of MPI implementations. Thanks to the very low memory requirements of *pcWT*, *mpi-dd* requires the lowest amount of RAM overall. For the merge operation after construction, we only need one buffer for packed bit vectors to be transmitted, which we reuse for each level.

Much like in the traffic analysis, we see that *mpi-bsort* and *mpi-dsplit* have an extremely similar memory footprint because the buckets in *mpi-bsort* correspond to the split texts T^0 and T^1 in *mpi-dsplit*. However, *mpi-dsplit* requires a subsequent merge operation, which apparently dominates the memory usage for small alphabets: for DNA, we can see clearly that the difference in RAM usage between *mpi-dsplit* and *mpi-bsort* correlates directly to the memory usage of *mpi-dd*. We stress here that the merge operation for *mpi-dd* and *mpi-dsplit* are, in fact, the same routine in the implementations.

5.4 Strong Scaling Results

We conduct *strong scaling* experiments by constructing the wavelet tree for a prefix of fixed length $n = 128$ GiB for each input file. Note that the full length of Wiki is slightly less than 128 GiB, so we process the whole file instead of a prefix. We only vary the number of nodes used for constructing the wavelet tree (again with 20 processing elements per node) and perform five iterations of each experiment to measure the same attributes as for our weak scaling experiments. We present the strong scaling results in Figure 18.

Missing data points. The first observation is that several data points are missing for a low number of nodes. The respective experiments have been canceled by the operating system due to RAM limitations. The only MPI implementation that succeeded in all experiments

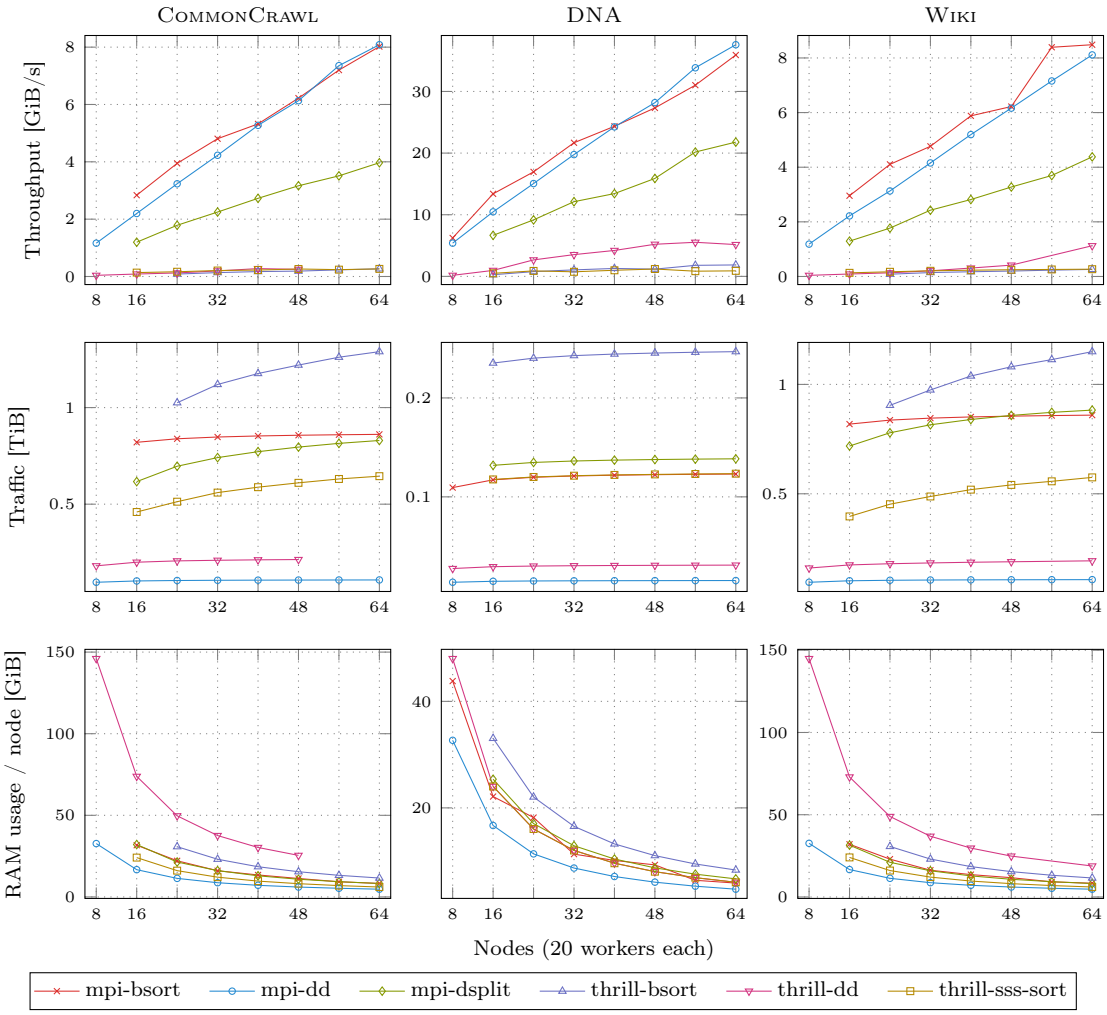


Figure 18: Median throughput over five iterations (first row), network traffic (second row) and average amount of used per-node RAM (third row) of our implementations in the strong scaling experiments for each input text (columns). Missing data points indicate failure of all iterations of the respective experiments. Note that Thrill makes use of external memory and counts it towards RAM usage, so that the RAM usage of Thrill implementations may exceed the physically available 64 GiB.

is *mpi-dd* thanks to the low memory profile of *pcWT*. The Thrill implementations make use of external memory to some extent in order to cope with RAM limitations.

Furthermore, some data points are missing for *thrill-dd* on CommonCrawl and Wiki for 56 and 64 nodes. These experiments failed with memory corruptions for which we could not find a cause and which seem out of our scope. It should be noted that these issues disappeared sporadically for some runs. Therefore, since the implementations are in fact functional (all weak scaling experiments succeeded using the same implementations), we suppose some sort of misconfiguration related to external memory usage in the cluster.

MPI implementations. Considering the results of the weak scaling experiments, the MPI implementations behave as expected in the strong scaling experiments: *mpi-dd* and *mpi-bsort* are going head-to-head in terms of throughput, though *mpi-dd* causes much less traffic and requires less memory. While *mpi-dsplit* causes less traffic than *mpi-bsort* on larger alphabets and uses roughly the same amount of RAM in all instances, it only achieves about half as much throughput in all experiments. The strong scaling experiments also confirm the good scalability of the MPI implementations regarding throughput: going from 16 to 64 nodes on CommonCrawl, the throughput increases by a factor of 3.7 for *mpi-dd* (2.198 GiB /s to 8.084 GiB /s), by a factor of 3.3 for *mpi-dsplit* (1.200 GiB /s to 3.968 GiB /s) and by a factor of 2.8 for *mpi-bsort* (2.836 GiB /s to 8.020 GiB /s).

Thrill implementations. Similar to the weak scaling experiments, the overall throughputs of the Thrill implementations are much lower than those of the MPI implementations. In the strong scaling experiments, this difference is amplified by the fact that Thrill starts using external memory as soon as one third of the available RAM is being used. This threshold is currently fixed in Thrill and the authors explain it by noting that a DIA consists of three logical layers, which reserve a third of the available RAM each. The use of external memory, albeit on the local file systems of the respective nodes, naturally slows down the execution. For better visibility of the Thrill throughputs, we display them on a different scale in Figure 19.

Despite the overall lower throughput, we note that *thrill-dd* and *thrill-bsort* scale very well for CommonCrawl: going from 24 to 48 nodes, the throughput increases by a factor of almost precisely two for *thrill-dd* (approximately 127 MiB /s to approximately 254 MiB /s) and even by a factor of 2.2 for *thrill-bsort* (approximately 87 MiB /s to approximately 192 MiB /s).

Even *thrill-sss-sort* does well to that end, increasing the throughput by a factor of 1.6 (from approximately 170 MiB /s to approximately 268 MiB /s) on CommonCrawl going

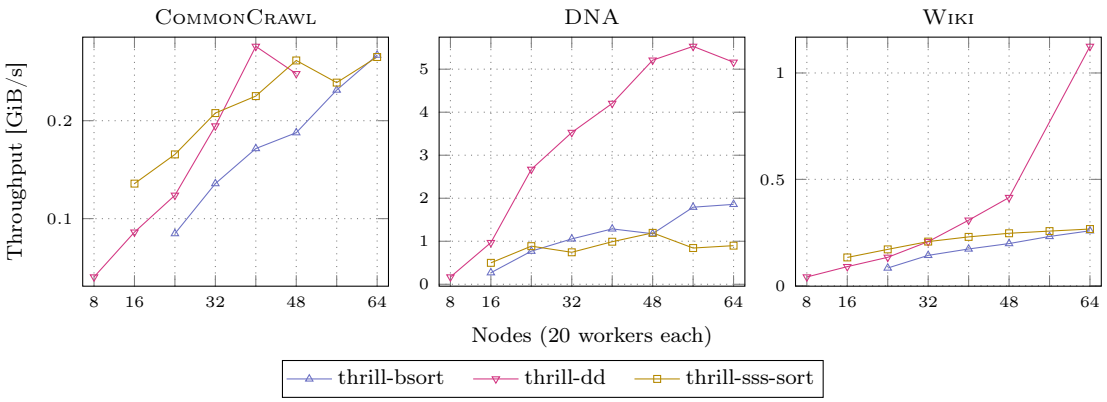


Figure 19: Median throughputs of Figure 18, but only displaying the Thrill implementations.

from 24 to 48 nodes. For DNA, however, where only one stable sort is conducted, we see that the balancing issues of sss-sort come into effect: only two nodes are responsible for constructing the entire second bit vector, which results in bad scaling going from 24 nodes to 48 (factor 1.35 from 0.886 GiB /s to 1.194 GiB /s)

The reason for the sudden rise of *thrill-dd*'s throughput at the 64 node mark on Wiki is unknown, but likely a statistical outlier. We note that only one iteration of the *thrill-dd* experiment succeeded for Wiki using 64 nodes. The other four iterations, as well as all iterations using 56 nodes, failed with the aforementioned memory corruption for undisclosed reasons.

5.5 Thrill-Specific Experiments

We conduct further experiments specifically for our Thrill implementations to confirm (a) the balancing issues of sss-sort regarding wavelet tree construction and (b) that the detour via *ZipWithIndex*, *Union* and *Sort* operations for concatenation fares better than Thrill's generic *Concat* operation in our use cases. To measure the data presented in this section, we use Thrill's own logging functionality that measures data points in fixed time intervals. This causes a slight overhead in the execution times and was disabled for the weak and strong scaling experiments.

5.5.1 Super Scalar Sample Sort vs. Bucket Sort

We execute *thrill-sss-sort*, our implementation to construct the wavelet tree using the stable sss-sort, with 32 nodes on a 32 GiB prefix of CommonCrawl and log the CPU utilization and network traffic of the first node (rank 0) and the last node (rank 31) over time in Figure 20.

The sorting of the text after constructing the first level's bit vector begins shortly beyond the 40 second mark. We observe how from here, node 0 has much longer intensive CPU phases than node 31: after receiving the splitters for item distribution, node 31 sends all local items away to node 0 and node 1 but does not receive any due to the previously described unbalance. For that reason, it is practically idling up until shortly after the 70 second mark where the second sort commences. Meanwhile, node 0 needs to locally sort the received items first.

We noted in section 3.4.1 how the unbalance affects the first $\lceil \log p \rceil$ sort operations. In this case, we have $p = 32$ and thus look at the first $\lceil \log p \rceil = 5$ sorts between 40 and 166 seconds: the intensive CPU phases of node 0 become shorter in logarithmic fashion while node 31 receives more work only later into the process. Thus, we can confirm the balancing issues of sss-sort in the face of wavelet tree construction.

For comparison, we conduct the same experiment for our bucket sort implementation *thrill-bsort* and show the results in Figure 21. It is clearly visible that *thrill-bsort* has

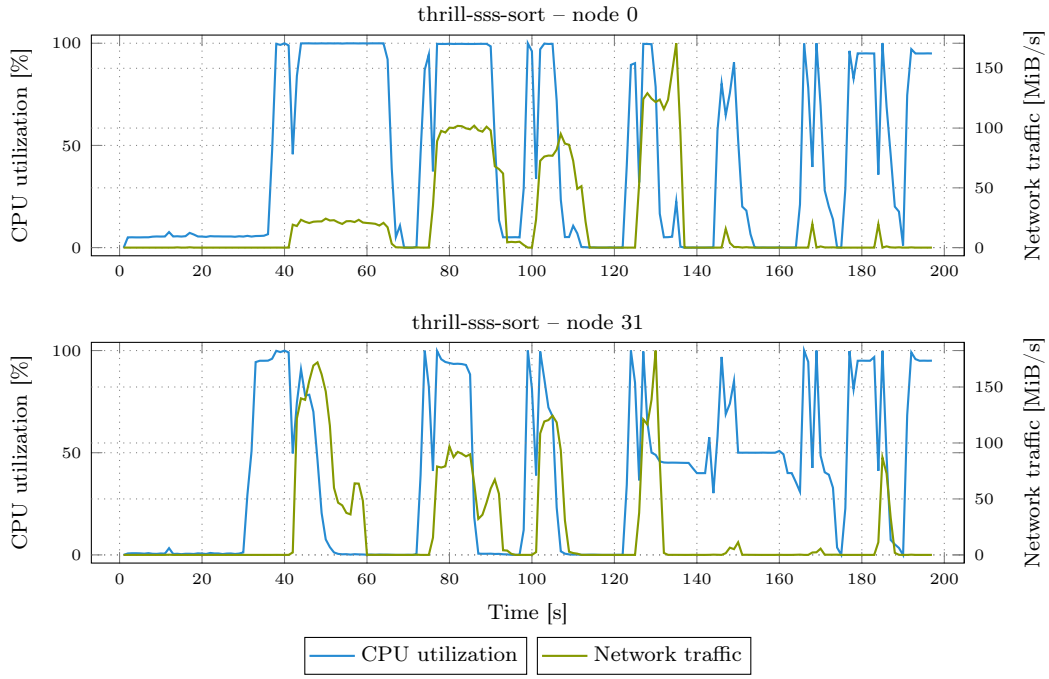


Figure 20: CPU utilization and network traffic of node 0 (upper chart) and node 31 (lower chart) *thrill-sss-sort* using 32 nodes on a 32 GiB prefix of CommonCrawl.

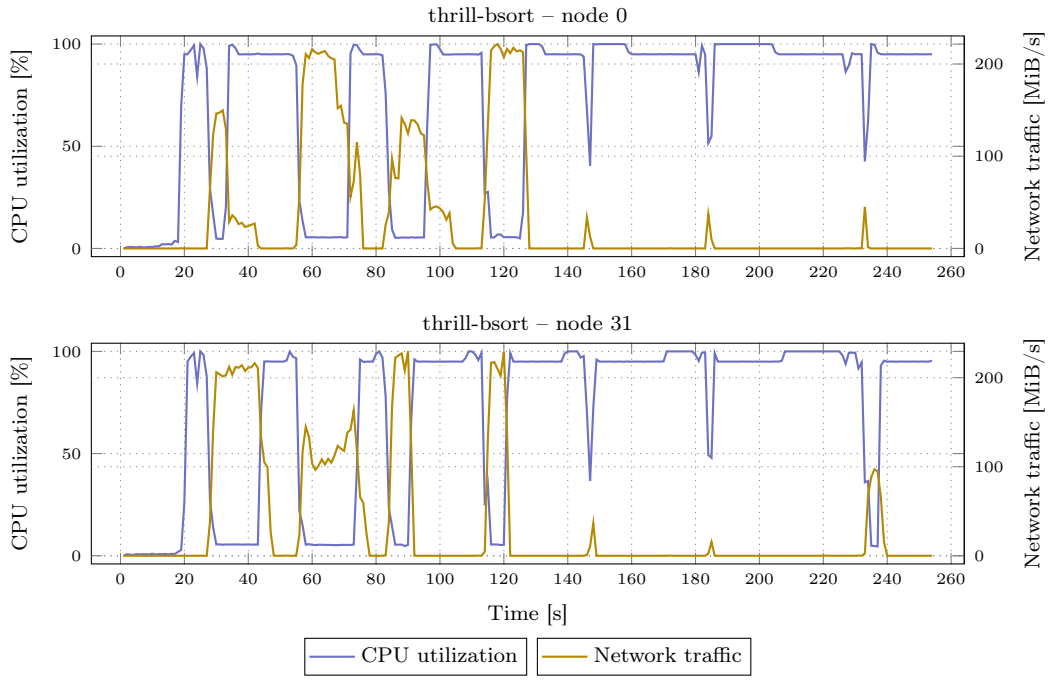


Figure 21: CPU utilization and network traffic of node 0 (upper chart) and node 31 (lower chart) *thrill-bsort* using 32 nodes on a 32 GiB prefix of CommonCrawl.

better load balancing properties, making intensive use of the CPU on the observed nodes. However, as also seen in the weak and strong scaling experiments, *thrill-bsort* takes longer overall than *thrill-sss-sort* in most instances (without regarding the initial phases with very low CPU utilization, where the input is being read from disk). The factor that comes into play here is that *thrill-bsort* requires concatenations, which we already identified to be a heavy operation in Thrill that we solve ultimate by sorting. In contrast, *thrill-sss-sort* only sorts the text without filling buckets first.

We conclude that *thrill-sss-sort* is the faster implementation overall despite the load balancing issues of sss-sort, but *because* of the load balancing issues, it has worse scaling properties than *thrill-bsort*, as evidenced in our scaling experiments.

5.5.2 Concatenation

In order to endorse the alternative concatenation method presented in section 4.2.1 that we use for our Thrill implementations, we execute two variants of *thrill-dd* (domain decomposition) using 32 nodes on a 32 GiB prefix of CommonCrawl and compare the CPU utilization and network traffic over time during the wavelet tree merging. The first variant

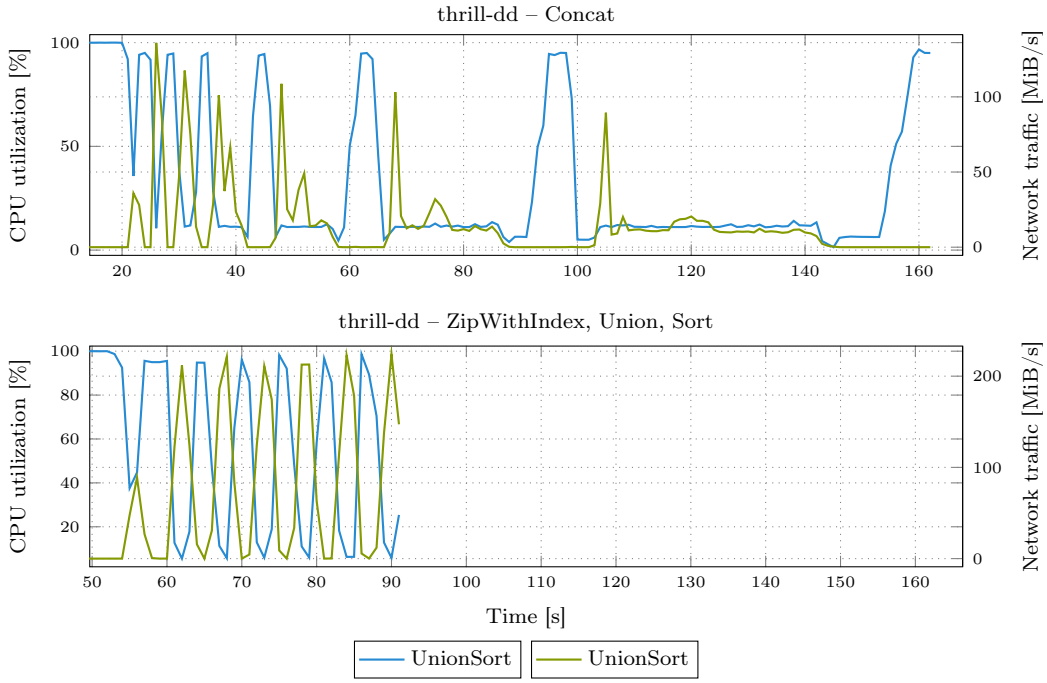


Figure 22: CPU utilization and network traffic of the first node for two variants of *thrill-dd* using 32 nodes on a 32 GiB prefix of CommonCrawl. For the upper chart, we used Thrill's *Concat* operation and in the lower chart, we use the detour via *ZipWithIndex*, *Union* and *Sort*. For a fair comparison of running times, we adjust the x-axis so that both curves start at the point where the merging of local wavelet trees into the global wavelet tree commences (at about 20 seconds for *Concat* and at about 50 seconds for the alternative approach).

uses Thrill's generic *Concat* operation and the second variant does the alleged detour via *ZipWithIndex*, *Union* and *Sort*. The results are shown in Figure 22.

We note that the merge using our alternative concatenation is more than twice as fast as the generic concatenation using Thrill's *Concat* operation. *Concat* requires long phases of communication in order to collect the information that the alternative concatenation gets for free thanks to our precomputed wavelet tree structure, which we use to reduce the problem to sorting. The authors of Thrill also presume that the randomized communication of *Sort* allows for better utilization of the network, transmitting data between multiple nodes simultaneously while *Concat* follows a linear communication pattern in the order of node ranks. Regardless, this experiment confirms our claim from section 4.2.1 that the alleged detour to achieve concatenation results in faster processing overall.

5.6 Summary

Our scaling experiments show that most of our implementations have good scaling properties. The *domain decomposition* algorithms, implemented in *mpi-dd* and *thrill-dd*, achieve the overall best scaling results where doubling the number of nodes also nearly doubles their throughput.

The MPI implementation of the domain decomposition algorithm, *mpi-dd*, not only achieves the highest overall throughputs of all implementations (along with *mpi-bsort* in most experiments), but it also causes the least amount of network traffic and requires the lowest amount of memory.

The Thrill implementations achieve lower throughputs overall than the MPI implementations due to the way that bit vectors and strings are concatenated. Despite *thrill-sss-sort* being faster in most experiments than *thrill-bsort*, its scaling properties are the worst among our implementations because of the balancing issues arising from sss-sort with a low number of sort keys.

6 Construction of Wavelet Trees vs. Wavelet Matrices

In this chapter, we take a step away from distributed computing to get some theoretical insight into the construction of wavelet trees and wavelet matrices. More specifically, we continue the research of Fischer et al. [10] on the topic of whether construction algorithms for the wavelet tree and the wavelet matrix can be modified efficiently to construct the other data structure, respectively.

Wavelet matrix. The *wavelet matrix* can be thought of as an alternative representation of the wavelet tree. In the wavelet tree, in order to retrieve the bit vector B_ℓ^T for level ℓ, we concatenate the bit vectors of the single nodes on that level from *left to right*. In the wavelet matrix, the nodes are concatenated in a different order to obtain bit vector B_ℓ^M: all left children of their respective parents are moved to the left and all right children of their respective parents are moved to the right [24, Ch. 6.2.5]. Figure 23 shows an example. This re-ordering corresponds to the bit-reversal permutation of the node ranks on the respective level [10].

Bit reversal. For any string S, let S^R denote its reversal. For a bit string $B \in \mathbb{B}^*$, let $(B)_\mathbb{N} \in \mathbb{N}$ denote the integer that B is the binary representation of. For $k > 0$ and an integer $i < 2^k$, we call $(i)_{\mathbb{B},k} \in \mathbb{B}^k$ the k-bit binary representation of i.

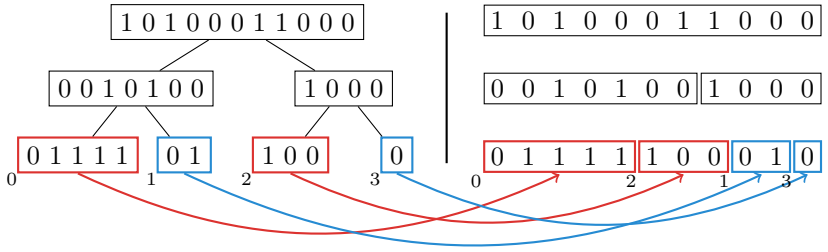


Figure 23: Comparison of the node ordering in the wavelet tree (left) and the wavelet matrix (right). Due to the nature of the bit reversal permutation, the ordering on the first two levels remains the same in the wavelet matrix. On the third level, we observe how nodes 0 and 2 (left children of their respective parents) go to the left part of the corresponding wavelet matrix bit vector and nodes 1 and 3 (right children of their respective parents) go to the right.

i	$(i)_{\mathbb{B},2}$	$((i)_{\mathbb{B},2})^R$	$\text{bitrev}_2(i)$
0	00	00	0
1	01	10	2
2	10	01	1
3	11	11	3

(a) Bit-reversal permutation for $k = 2$.

i	$(i)_{\mathbb{B},3}$	$((i)_{\mathbb{B},3})^R$	$\text{bitrev}_3(i)$
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

(b) Bit-reversal permutation for $k = 3$.

Table 7: Breakdowns of the bit-reversal permutations for $k = 2$ (left) and $k = 3$ (right). The first column contains the integers $i < 2^k$, the second shows their k-bit binary representations, the third shows the reversals and the final column contains the k-bit reversal of i.

We define the *k-bit reversal* $\text{bitrev}_k(i) \coloneqq (((i)_{\mathbb{B},k})^R)_{\mathbb{N}}$ as the integer represented by the reversal of i's k-bit binary representation. For a fixed k, the *bit-reversal permutation* maps each integer $i < 2^k$ to its k-bit reversal. To give examples, Table 7 shows the bit-reversal permutations for $k = 2$ and $k = 3$.

Wavelet tree vs. wavelet matrix construction. The wavelet matrix can be used to answer the same queries as the wavelet tree in the same asymptotic time. However, its relevance in practice stems from the fact that a lower constant number of rank and select queries is required to answer those queries, apart from other storage-related advantages that we will not regard here any further.

We are interested in how a construction algorithm for either data structure (wavelet tree or matrix) can be modified *efficiently* to construct the other, i.e., without worsening any of the present asymptotic time and space boundaries. Fischer et al. [10] show that there is a data structure that can be used to efficiently transform any construction algorithm for the wavelet tree to construct instead the wavelet matrix. However, it was not known whether an efficient transformation in the inverse direction exists, i.e., whether there is an efficient way to construct the wavelet tree using a construction algorithm for the wavelet matrix. In this chapter, we propose a solution to this problem, albeit with a limitation.

6.1 Problem Definition

Formally, let us consider the situation where, during the construction of the wavelet tree, the i-th bit is set in bit vector B_ℓ^T of level ℓ of the wavelet tree (assuming, without loss of generality, a level-wise representation). Fischer et al. [10] present a data structure to efficiently compute a function $f : (\ell, i) \mapsto (\ell, j)$ so that j is the corresponding position for the bit to be set in bit vector B_ℓ^M of the wavelet matrix. For input length n and

alphabet size σ, their data structure occupies $n + \sigma + (\sigma + 2)\lceil \log n \rceil$ bits of space and can be constructed in time $\mathcal{O}(n + \sigma)$ using $o(n + \sigma)$ bits of memory.

In this chapter, we describe a data structure that computes function f^{-1}, which maps (ℓ, j) back to (ℓ, i) with the same asymptotic time and space boundaries as f. Before we proceed there, we observe various properties of the wavelet tree that lead to a similar result for f as that of Fischer et al.. This will help us develop the novel data structure.

6.2 Locating Nodes and Bit Offsets

As previously mentioned, the re-ordering of nodes between the wavelet tree and matrix can be described by the bit-reversal permutation. This knowledge makes it easy to translate a node *ID* (the node's BFS rank) between the two data structures. Based on that, we employ the following strategy to find data structures for functions f and f^{-1}: given the level and position of the bit to be written, we attempt to find (a) the ID of the node that the bit belongs to and (b) the position of the node's first bit in the level bit vector. With this information available, we are going to see that both f and f^{-1} are easy to compute in constant time.

Bottom level node sizes. In section 2.2.1, we described the relation between the C array and the sizes of the wavelet tree's nodes. This relation is especially interesting regarding the *virtual* bottom-most level $h = \lceil \log \sigma \rceil$ of a full binary wavelet tree. We call this level virtual, because all bits on it would be zero and we never actually store it. On this level, each node corresponds to a single symbol from the input alphabet. Let node v_c on level h correspond to symbol $c \in [0, \sigma)$. We have $|B_{v_c}| = C[c+1] - C[c] = \mathrm{occ}_{T_{\mathrm{eff}}}(c)$, which means that the size of v_c matches the number of occurrences of c.

In order to use this property, the wavelet tree must be a *full* binary tree: if it was not, there would be leaves on level $h - 1$ and not all nodes on level h would exist. Therefore, without loss of generality, let us assume from now on that $\sigma = 2^h$ for some integral $h > 0$, i.e., that the alphabet size is a power of two. Then, the wavelet tree is a full binary tree. In case σ is not a power of two, we introduce artificial symbols that never occur in the input and are lexicographically *larger* than all symbols that do occur in the input. This way, the empty nodes for these symbols are moved to the far right of the wavelet tree and can be ignored in the following.

6.2.1 Wavelet Tree

We consider the situation where a wavelet tree constructor sets the i-th of bit vector B_ℓ^T. Let $v(\ell, i)$ be the rank of the wavelet tree node on level ℓ to which the i-th bit belongs. We represent $v(\ell, i)$ relative to the number of the first node on level ℓ, i.e., $v(\ell, 0) = 0$ and $v(\ell, n-1) = 2^\ell - 1$. This representation requires ℓ bits, because there are precisely $2^\ell - 1$

c	a	e	l	r	t	v	w	⊤	
	0	1	2	3	4	5	6	7	
$\mathrm{occ}_T(c)$	1	4	1	1	2	1	1	0	
$C[c]$	0	1	5	6	7	9	10	11	11

Figure 24: The histogram and the C array for $T =$ `wavelettree`. We added the artificial symbol $\top$ so $\sigma = 8$ is a power of two. The new symbol never occurs in T and is lexicographically larger than the other symbols.

nodes on level ℓ. Furthermore, let $p(\ell, v)$ be the position of the first bit in B_ℓ^T that belongs to node v and let

$$\delta_v(\ell, i) \coloneqq i - p(\ell, v(\ell, i))$$

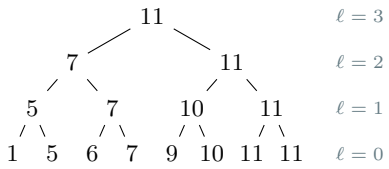
be the distance of i from that position.

We take a closer look at v and p on the virtual level h and observe that $v(h, i) = \min\{x \mid C[x] > i\} - 1$. This is because each node on this level corresponds to precisely one symbol from the input alphabet and the C array encodes, for every c, the number of symbols in the input that are lexicographically smaller than c. This corresponds to the accumulated sizes of the node's left siblings. An example of this relation can be seen comparing Figure 24 and Figure 25b (in row $\ell = 3$). The node that i belongs to on level h is left of the first node whose accumulated size — its entry in the C array — exceeds i. We can immediately conclude that the first bit that belongs to node v is located at position $p(h, v) = C[v]$.

How do v and p on level h relate to those on the other levels $\ell < h$ that we are actually interested in? To answer this, we make use of the fact that the wavelet tree is a full binary

	0	1	2	3	4	5	6	7	8	9	10
$\ell = 0$:	w	a	v	e	l	e	t	t	r	e	e
$\ell = 1$:	a	e	l	e	r	e	e	w	v	t	t
$\ell = 2$:	a	e	e	e	e	l	r	v	t	t	w
$\ell = 3$:	a	e	e	e	e	l	r	t	t	v	w
B_C:	1	1	0	0	0	1	1	1	0	1	1

(a) The text re-ordering on each level and the bit vector B_C. The vertical lines mark the boundaries of the wavelet tree's nodes.



(b) The accumulated sizes of each of the wavelet tree's nodes. Note that the rightmost node on the bottom level corresponds to our artificial symbol $\top$ from Figure 24.

Figure 25: Display of the wavelet tree's text re-ordering on each level, including the virtual level $h = 3$, the bit vector B_C and the accumulated node sizes for our running example text $T =$ `wavelettree`.

tree: the size of a node equals the sum of its children's sizes. This is because the children partition the alphabet interval of a node. As a consequence, the *accumulated* size of any node is retained in its right child, as can be seen in Figure 25b. Since the C array encodes the accumulated sizes of the nodes on level h, it also implicitly encodes the accumulated sizes of all nodes on levels $\ell < h$. Following this notion, we can conclude the following relations:

$$v(\ell, i) = \left\lfloor \frac{\min\{x \mid C[x] > i\} - 1}{2^{h-\ell}} \right\rfloor$$

and

$$p(\ell, v) = C[v \cdot 2^{h-\ell}]. \tag{6.1}$$

The array C is stored in ascending order, so the minimum query required to find v could be answered in time $\mathcal{O}(\log \sigma)$ using binary search. However, we seek a computation in constant time. We construct the bit vector B_C of length n by setting $B_C[k] \coloneqq 1$ if $C[c] = k - 1$ for some c and $B_C[k] \coloneqq 0$ otherwise and prepare it for constant-time rank queries. This can be done in time $\mathcal{O}(n)$ and requires $n + o(n)$ bits of additional space (see section 2.1.3). In B_C, we practically mark the node boundaries on level h of the wavelet tree. See Figure 25a for an example. We can now compute

$$v(\ell, i) = \left\lfloor \frac{\mathrm{rank}_1(B_C, i) - 1}{2^{h-\ell}} \right\rfloor \tag{6.2}$$

in constant time.

We now know that the i-th bit in B_ℓ^T corresponds to the (δ_v)-th bit in the v-th node on level ℓ in the wavelet tree. We can compute v, p and δ_v in constant time using the array C and rank-enhanced bit vector B_C, which together occupy $\sigma\lceil \log n \rceil + n(1 + o(1))$ bits of space. Asymptotically, this space boundary matches that of the data structure presented by Fischer et al. [10].

Example 1. Figure 25, in combination with Figure 24, shows an example of the data structure for $T = \texttt{wavelettree}$. Assume that we are interested in locating the node for bit $i = 9$ on level $\ell = 2$. With Equation 6.2, we get $v(2, 9) = \left\lfloor \frac{\mathrm{rank}_1(B_C, 9) - 1}{2^{3-2}} \right\rfloor = \lfloor \frac{5}{2} \rfloor = 2$. This means that the bit belongs to the third node on level 2 (because we start counting at zero). Furthermore, with Equation 6.1, we get $p(2, 2) = C[2 \cdot 2^{3-2}] = C[4] = 7$. This means that the third node on level 2 starts at position 7. Finally, it is $\delta_v(2, 9) = 9 - p(2, 2) = 9 - 7 = 2$, so bit 9 on level 2 ultimately corresponds to the third bit of the third node on that level.

6.2.2 Wavelet Matrix

Given the observations in section 6.2.1, it is natural to ask how a similar locating can be done for the wavelet matrix. As described previously, its bit vector B_ℓ^M can be viewn as the concatenation of the wavelet tree nodes on level ℓ in bit-reverse order. To that regard, the wavelet matrix can be represented as a tree just as well with the nodes re-ordered accordingly. Even though there are no practical advantages of storing the wavelet matrix as a tree, this notion will help us find an efficient data structure for computing f and f^{-1}.

We consider the situation where a wavelet matrix constructor sets the j-th bit of bit vector B_ℓ^M of the wavelet matrix and are interested in the node to which this bit belongs. Analogously to v, p and δ_v in the previous section, we define $u(\ell, j)$, $q(\ell, u)$ and $\delta_u(\ell, j) \coloneqq j - q(\ell, u(\ell, i))$ as the node into which the written bit belongs, the position of the node's first bit in B_ℓ^M and the distance of j from the node's first bit, respectively.

Due to the re-ordering of the nodes, the correspondences between their accumulated sizes and the C array, which we observed for the wavelet tree, are no longer valid for the wavelet matrix. As a consequence, we need to find a different way to compute u and q.

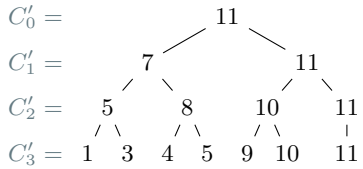
We make use of the following observation to find u: in either the wavelet tree or the wavelet matrix, all occurrences of a symbol $c \in [0, \sigma)$ belong to the same node on any level. Since this is also true for level h, in order to find the node to which any occurrence of c belongs on level h, it suffices to know to which node the *first* occurrence of c belongs. This first occurrence of c on level h is always located at position $C[c]$. As seen previously, once the node for level h is known, it is easy to narrow it down to any level $\ell < h$. Of course, we then have the node in the wavelet *tree*, but in the wavelet matrix, the nodes are simply permuted in bit-reverse order. Let c be the symbol from which we computed the bit that we are setting in B_ℓ^M. If c is known when the bit is being set[13], we can express

$$u(\ell, j, c) = \mathrm{bitrev}_\ell(v(\ell, C[c])). \tag{6.3}$$

It remains to compute q. As stated above, the C array cannot be used directly to compute the accumulated node sizes for the wavelet matrix, because nodes are permuted. However, the node sizes themselves remain the same and thus, with awareness of the bit-reversal ordering of nodes on every level, it is easy to precompute the accumulated node sizes for all nodes of the wavelet matrix using the C array in time $\mathcal{O}(\sigma)$. Since we are dealing with a full binary tree of height $h = \log \sigma$, the accumulated wavelet matrix node sizes can be stored in an array C' of length $2^h - 1 = \sigma - 1$ (since σ is a power of two), occupying $(\sigma - 1)\lceil \log n \rceil$ bits of space. Figure 26b shows an example. We consider C' a

[13] We address the consequences of this requirement in section 6.3.

(a) The text re-ordering on each level of the wavelet matrix. The vertical lines mark the boundaries of the *nodes* of the wavelet matrix.



(b) The accumulated sizes of each of the *nodes* of the wavelet matrix. Note that for C'_3, the bottom level, is not actually needed and depicted only for the sake of completeness.

Figure 26: Display of the wavelet matrix's text re-ordering on each level for running example text $T = \texttt{wavelettree}$.

set of arrays C'_ℓ for each level ℓ, so that the first entry of C'_ℓ contains the size of the first node on level ℓ. Then, q can be determined in a very simple way:

$$q(\ell, u) = \begin{cases} 0 & \text{if } u = 0. \\ C'_\ell[u-1] & \text{if } u > 0. \end{cases} \tag{6.4}$$

We then know that the j-th bit in B^M_ℓ of the wavelet matrix corresponds to the δ_u-th bit in the u-th node's bit vector on level ℓ. We can compute u, q and δ_u in constant time using the arrays C and C' and rank-enhanced bit vector B_C, which, in total, occupy $(2\sigma - 1)\lceil \log n \rceil + n(1 + o(1))$ bits of space.

Example 2. Figure 26, in combination with Figure 25 and Figure 24, shows an example for the data structure for $T = \texttt{wavelettree}$. Assume that we are interested in locating the node for bit $j = 9$ on level $\ell = 2$ of the wavelet matrix. The symbol for which the bit is written is $c = \texttt{r}$ (see Figure 26a). With Equation 6.3, we get $u(2, 9, \texttt{r}) = \text{bitrev}_3(v(2, C[\texttt{r}])) = \text{bitrev}_3(v(2, 6)) = \text{bitrev}_2(1) = 2$. This means that the bit belongs to the third node on level 2. Furthermore, with Equation 6.4, we get $q(2, 2) = C'_2[2 - 1] = 8$. This means that the third node on level 2 starts at position 8. Finally, it is $\delta_u(2, 9) = 9 - 8 = 1$, so bit 9 on level 2 ultimately corresponds to the second bit of the third node on that level.

6.3 Converting Between Wavelet Tree and Wavelet Matrix Construction

Using the locating data structures described above, we can express functions f and f^{-1} as follows:

$$f(\ell, i) = q(\ell, \text{bitrev}_\ell(v(\ell, i))) + \delta_v(\ell, i),$$
$$f^{-1}(\ell, j) = p(\ell, \text{bitrev}_\ell(u(\ell, j, c))) + \delta_u(\ell, j, c).$$

Both f and f^{-1} can be computed in constant time using the arrays C, C' and rank-enhanced bit vector B_C. These occupy $\sigma\lceil\log n\rceil + (2\sigma - 1)\lceil\log n\rceil + n(1 + o(1))$ bits of space can be constructed in time $\mathcal{O}(\sigma + n)$.

We impose the restriction that for f^{-1}, the symbol c, for which a bit is being set in B_ℓ^M, has to be known when setting the bit. Even though this bit must ultimately have been computed from c, there are construction algorithms for the wavelet tree that redistribute the bits of c before constructing the bit vectors [1, 17, 22, 34]. Due to the existence of our function f alone, such techniques may as well be used for the construction of the wavelet matrix. In this case, c is *not* known when setting the bit in question and f^{-1} cannot be used.

7 Conclusion

In this thesis, we reached the goal of developing the first distributed algorithms for the construction of wavelet trees. We implemented these algorithms against Thrill and MPI and performed a practical evaluation constructing wavelet trees for large inputs. In this concluding chapter, we summarize the insights we gained and give an outlook for possible future research topics.

7.1 Insights

Distributed computing is a practicable option for constructing wavelet trees for large texts. We conclude so following the evaluation of the weak and strong scaling experiments of chapter 5.

The approach that works best overall is the *domain decomposition*, where we partition the input and distribute it evenly across the available processing elements. Each processing element constructs an entire wavelet tree for their local part of the input. In a subsequent step, the local wavelet trees are merged to form the wavelet tree of the entire input. This approach minimizes the amount of necessary communication both in theory (as analyzed in chapter 3) and in practice (see chapter 5). It is independent of the algorithm used for local wavelet tree construction on each processing element and thus allows for making use of any future advancements regarding the sequential or parallel (multi-core, shared memory) wavelet tree construction. For our implementation, we use the fastest known practical algorithm *pcWT* due to Fischer et al. [10] for sequential wavelet tree construction, which, thanks to its low memory profile, allows us to process large inputs with only few processing elements. In the strong scaling experiments conducted in chapter 5, we observed near-perfect scalability with an increasing number of nodes used.

Balancing concatenation. We observe that a key operation required for the distributed construction of wavelet trees is the *balancing concatenation* of distributed sequences. All distributed approaches analyzed in chapter 3 require an operation where a sequence of bits or symbols, distributed over the available processing elements, are concatenated so that each processing element maintains an equal number of items of the concatenation with the goal of load balancing.

Thanks to the fact that we can precompute the structure of the wavelet tree in negligible time using just the input's histogram, the balancing concatenations in our scenarios can be realized easily using one prefix sum computation per concatenation to determine recipients for each of the items in the sequence.

Thrill and distributed wavelet tree construction. For Thrill, the concatenations pose a problem due to its concept of *distributed immutable arrays* that abstract from the locality of their items. This requires us to use a detour via sorting in order to avoid a costly generic concatenation that cannot make use of our precomputed information. While Thrill excels in some areas of distributed computing (e.g., k-means clustering or page rank computations [3]), our practical evaluation hints that it is not very suitable for the distributed construction of wavelet trees.

However, thanks to Thrill's high-level API and its large set of tools provided out of the box, we could design and implement the first working distributed wavelet tree construction algorithm (*thrill-dd*) in a very short time compared to our MPI implementations. Furthermore, in early evaluations of just the Thrill implementations, it already became visible that the domain decomposition is probably the best approach to go with. To that regard, even in areas where Thrill's design may not be inherently suitable to solve the problem at hand, it provides a solid, easy to learn toolset for quick sketching of distributed algorithms and preliminary evaluations.

Wavelet trees and wavelet matrices. In chapter 6, we went a step back from distributed computing and attempted to solve an open theoretical problem concerning wavelet trees and wavelet matrices. We described a data structure that can be used to extend a construction algorithm for the wavelet matrix to construct instead the wavelet tree with constant time overhead. We can construct this data structure in time $\mathcal{O}(\sigma + n)$ time and it requires $\mathcal{O}(\sigma \log n + n)$ bits of memory, matching the asymptotic time and space requirements of the data structure described by Fischer et al. [10] for the inverse direction (transforming wavelet tree construction into wavelet matrix construction). However, we impose a restriction for our data structure that makes it unsuitable for a class of wavelet matrix construction algorithms that do not keep the entire binary representation of the input symbols when computing the bit vectors.

7.2 Outlook

We conclude the thesis by giving an outlook for potential future work on related topics.

Alternative wavelet tree representations. We provided algorithms to construct the levelwise representation of the original wavelet tree described by Grossi et al. [13]. However,

they also describe an entropy-compressed variant of the wavelet tree, the *Huffman-shaped wavelet tree*. In this representation, the bits on the path to a symbol's leaf represent its Huffman code [15] rather than its $\lceil \log \sigma \rceil$-bit binary representation. This causes the wavelet tree to adopt the shape of the Huffman tree for the input text. Furthermore, the *wavelet matrix*, briefly described in chapter 6, is a different representation of the wavelet tree where the nodes are re-ordered according the bit-reversal permutation on each level. This is of practical use as it allows for queries to be answered using less rank and select queries on the bit vectors compared to the wavelet tree. A variant of the wavelet matrix uses canonical Huffman codes in order to achieve an entropy-compressed representation similar to Huffman-shaped wavelet trees.

In future work, one could extend upon the distributed wavelet tree construction algorithms developed in this thesis to construct these practically more relevant representations.

Answering queries. Another interesting topic is how the queries supported by the wavelet tree and matrix (e.g., rank, select and access) can be answered efficiently in a distributed setting. This would be substantial for a distributed FM index [8].

Extending Thrill. Thrill provides a solid and very helpful toolset for quickly sketching and implementing practically functional distributed algorithms. However, we saw some difficulties in the scenario of constructing wavelet trees. To make Thrill more suitable for wavelet tree construction, one could extend it by a specialized concatenation operation that can make proper use of precomputed information such as the wavelet tree structure. It is conceivable that such an operation may be useful in other scenarios as well. Furthermore, the lack of support for bit DIAs requires us to use temporary DIAs of boolean values in order to get rid of alignment bits after concatenation. Each boolean value, which in theory only requires one bit, is byte-aligned in practice and thus takes up eight bits, causing a spike in memory usage where seven bits per item are wasted. To that regard, actual bit DIAs would be a useful addition to Thrill.

Lifting restrictions of the WT / WM constructor transformation. Finally, the theoretical problem of how to efficiently transform a wavelet matrix construction algorithm to construct instead the wavelet tree has been solved in this thesis only by imposing the restriction that the wavelet matrix constructor requires the entire binary representation of a symbol when writing a bit. It is still open whether this restriction can be lifted without worsening the asymptotic time and space boundaries of our data structure.

Bibliography

[1] Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana A. Starikovskaya. Wavelet trees meet suffix trees. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 572–591. SIAM, 2015.

[2] Timo Bingmann. *Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2018. URL `https://publikationen.bibliothek.kit.edu/1000085031`.

[3] 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.

[4] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink^{TM}: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[5] Francisco Claude and Gonzalo Navarro. The wavelet matrix. In *String Processing and Information Retrieval (SPIRE)*, pages 167–179. Springer, 2012.

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

[7] Paulo G. S. da Fonseca and Israel B. F. da Silva. Online construction of wavelet trees. In *Symposium on Experimental Algorithms, (SEA)*, pages 16:1–16:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[8] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 390–398. IEEE Computer Society, 2000.

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

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

[11] Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54 (12):1901–1909, 1966.

[12] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings*, pages 97–104. Springer, 2004.

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

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

[15] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[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 *String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings*, pages 218–232. Springer, 2018.

[18] Zahra Khatami, Sungpack Hong, Jinsoo Lee, Siegfried Depner, Hassan Chafi, J. Ramanujam, and Hartmut Kaiser. A load-balanced parallel and distributed sorting algorithm implemented with PGX.D. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2017, Orlando / Buena Vista, FL, USA, May 29 - June 2, 2017*, pages 1317–1324. IEEE Computer Society, 2017.

[19] 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.

[20] Leslie Lamport and Nancy A. Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 1157–1199. Elsevier, 1990.

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

[22] J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. Fast construction of wavelet trees. *Theor. Comput. Sci.*, 638:91–97, 2016.

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

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

[25] Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *Symposium on Experimental Algorithms, (SEA)*, pages 295–306. Springer, 2012.

[26] Pitch Patarasuk and Xin Yuan. Bandwidth efficient all-reduce operation on tree topologies. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–8. IEEE, 2007.

[27] Rolf Rabenseifner. Optimization of collective reduction operations. In *Computational Science - ICCS 2004, 4th International Conference, Kraków, Poland, June 6-9, 2004, Proceedings, Part I*, pages 1–9. Springer, 2004.

[28] Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *Algorithms - ESA 2004, 12th Annual European Symposium, Bergen, Norway, September 14-17, 2004, Proceedings*, pages 784–796. Springer, 2004.

[29] José Fuentes Sepúlveda, Erick Elejalde, Leo Ferres, and Diego Seco. Efficient wavelet tree construction and querying for multicore architectures. In *Symposium on Experimental Algorithms, (SEA)*, pages 150–161. Springer, 2014.

[30] 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.

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

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

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

[34] German Tischler. On wavelet tree construction. In *Combinatorial Pattern Matching (CPM)*, pages 208–218. Springer, 2011.

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

[36] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 28.03.2019

Patrick Dinklage