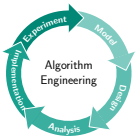


Constructing the Burrows-Wheeler Transform in Distributed Memory

Master's Thesis of

Daniel Brommer

at the Department of Informatics
Institute of Theoretical Informatics, Algorithm Engineering

Reviewer:	Prof. Dr. Peter Sanders
Advisor:	M.Sc. Matthias Schimek
Second advisor:	JProf. Dr. Florian Kurpicz

01. January 2026 – 22. July 2026

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 22.07.2026

. . . . . .

(Daniel Brommer)

Abstract

The BURROWS-WHEELER TRANSFORM (BWT) [10] is an important fundamental algorithm in the world of string algorithms. It serves as the foundation of several algorithms [40, 6, 13], for example, text indices, enabling efficient string pattern matching [16]. The BWT is a permutation of the given string obtained by sorting all cyclic rotations of the string. The amount of genome data parsed using the BWT increases daily because of efforts like the *GenomeTrakr* network [22], which is collecting hundreds of thousands of genome sequences. Therefore, constructing the BWT for large datasets is of interest. There are multiple algorithms to construct the BWT [10, 51, 9, 14], some of which already focus on processing large amounts of data [9, 39, 11]. So far, there have not been efforts to develop BWT construction algorithms that use distributed high-performance computing systems. We present two distributed BWT construction algorithms. One adapts the traditional approach of constructing the BWT by computing the *suffix array* [43] in distributed memory. The other distributed algorithm is based on BIG-BWT [9], a BWT construction algorithm that uses PREFIX-FREE PARSING [9] as a preprocessing step. We modified the BIG-BWT algorithm to efficiently and scalably work in distributed memory. We show that both of our distributed BWT construction algorithms can scale up to 6144 cores. This makes them, to the best of our knowledge, the first distributed BWT construction algorithms to do so. This enables us to use more computation resources to construct the BWT. Our distributed PFP-based algorithm requires only 96 cores to match or surpass the runtime of the state-of-the-art shared-memory parallel BWT construction based on Lyndon words [51] running on a single 48-core compute node. Using 64x more cores, our distributed PFP-based construction algorithm achieves a speedup of up to 20.9x over the Lyndon-based construction running on a single compute node.

Zusammenfassung

Die BURROWS-WHEELER TRANSFORMATION (BWT) [10] ist ein wichtiger Grundlagenalgorithmus in der Welt der textbasierten Algorithmen. Sie dient als Grundlage für einige Algorithmen [40, 6, 13], etwa für Algorithmen, die einen Textindex erstellen, mithilfe dessen Muster in Texten effizient gefunden werden können [16]. Die BWT ist eine Permutation des Eingabetexts, die berechnet wird, indem alle zyklischen Rotationen des Textes sortiert werden. Die Menge an Genomdaten, die mithilfe der BWT verarbeitet werden, nimmt täglich zu. Projekte wie das *GenomeTrakr network* [22] sammeln hunderttausende Genomsequenzen. Deshalb ist es interessant, die BWT auf großen Datensätzen zu berechnen. Es gibt verschiedene Ansätze zur Berechnung der BWT [10, 51, 9, 14], wobei sich einige auf die Verarbeitung großer Datensätze spezialisiert haben [9, 39, 11]. Bisher gibt es keine Algorithmen, um die BWT auf verteilten Hochleistungsrechnern zu berechnen. Wir präsentieren zwei verteilte BWT-Konstruktionsalgorithmen. Der eine basiert auf dem traditionellen Ansatz und konstruiert die BWT mittels verteilter Berechnung des *Suffix Arrays* [43]. Der andere verteilte BWT-Konstruktionsalgorithmus basiert auf dem BIG-BWT Algorithmus [52]. Der BIG-BWT Algorithmus selbst verwendet PREFIX-FREE PARSING (PFP) [9] als Vorverarbeitungsschritt. Wir haben den BIG-BWT Algorithmus modifiziert und optimiert, sodass er effizient in verteiltem Speicher skalieren kann. Beide unsere verteilten BWT-Konstruktionsalgorithmen skalieren bis zu 6144 Kernen. Dies macht sie, nach unserem besten Wissen, zu den ersten verteilten BWT-Konstruktionsalgorithmen, die so skalieren. Dadurch können wir mehr Rechenressourcen einsetzen, um die BWT zu konstruieren. Unser verteilter PFP-basierter BWT-Algorithmus benötigt nur 96 Kerne, um die Laufzeit des aktuell schnellsten BWT-Algorithmus für gemeinsamen Speicher, der auf Lyndon-Wörtern basiert [51], zu schlagen. Mit 64x so vielen Kernen erreicht unser verteilter, PFP-basierter BWT-Algorithmus eine $20,9$x kürzere Laufzeit als der Lyndon-basierte Algorithmus auf einem einzelnen Rechenknoten.

Acknowledgments

I want to thank my supervisors, Matthias Schimek and Florian Kurpicz, for their continuous support throughout this thesis, and for the valuable insights and advice they provided along the way.

I gratefully acknowledge the Gauss Centre for Supercomputing e.V. [1] for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre [2]

In accordance with the Deutsche Forschungsgemeinschaft statement[3] on the use of generative large language models, we disclose that such models were used for the following purposes:

- We used Claude Opus to assist with implementing the described algorithms in C++. All suggestions made by the model were reviewed by the author and adopted only where appropriate.
- We used Claude Opus to help write the plotting scripts for the plots throughout this thesis.

[1] https://www.gauss-centre.eu/
[2] https://www.lrz.de/
[3] https://www.dfg.de/de/aktuelles/neuigkeiten-themen/info-wissenschaft/2023/info-wissenschaft-23-72

Contents

1 Introduction

1.1 Motivation

The number of genomic sequences researchers around the world have access to increases daily. In 2008, the *1000 Genomes Project* was initiated to collect a comprehensive catalog of human genetic variation. The project was completed in 2015 after the genomes of 2504 individuals from around the world had been collected [1]. In 2012, the *100,000 Genomes Project* was announced with the aim of collecting 100,000 whole-genome sequences [57]. Until the project's end in 2018, the genomes of around 85,000 individuals had been collected [48]. This increase in data to be processed is evident not only in human genomic data but also in other fields. Since 2013, the *GenomeTrakr* network has collected around 1.8 million bacterial sequences, with the count increasing daily [22]. Thereof, roughly 800,000 sequences are from the bacterium *Salmonella enterica*. As each assembled Salmonella genome comprises roughly 5 million base pairs [46], and a base pair being stored using one byte, the sequence data for the Salmonella entries alone amounts to around 4 TB.

At the same time, computing resources increased heavily. In 2008, the fastest high performance computing (HPC) system on the TOP500 list, *Roadrunner*, had a performance of 1.105 Peta Floating Point Operations Per Second (FLOPS)[54]. In June 2026, the fastest HPC system on the TOP500 list is *LineShine*, with a performance of 2.198 Exa FLOPS, an increase by a factor of 1,000 over 2008 [55]. Systems at this scale are based on distributed-memory architectures. Using a shared memory system can make it challenging to process large amounts of data because of its limited main memory.

For algorithms operating in main memory, processing large amounts of data requires substantial main memory, which large-scale distributed HPC systems provide. Hence, it makes sense to combine these two developments and develop efficient algorithms for large-scale HPC systems to aid research on large genomic databases. There has been prior work on distributed algorithms for problems related to large strings [29, 37]. Especially in bioinformatics, the BURROWS-WHEELER TRANSFORM [10] (BWT) is a fundamental algorithm used in many applications. The BWT is a reversible permutation of an input string that yields beneficial algorithmic properties. It can be used to compress these large datasets and serves as a building block for indices that enable efficient pattern matching over them. There has been work on parallel Burrows-Wheeler Transform construction algorithms [9, 51] and distributed algorithms on data structures closely related to the Burrows-Wheeler Transform [29, 19]. First steps towards developing distributed Burrows-Wheeler Transform construction algorithms have been made [24], but to our knowledge, there have been no efforts to

explicitly construct the Burrows-Wheeler Transform on a scale of hundreds of gigabytes in distributed memory using thousands of cores. Therefore, we investigate and present the changes needed to existing Burrows-Wheeler Transform algorithms to enable them to scale on distributed HPC systems.

1.2 Contribution

The main contribution of this thesis is the development of two distributed BWT construction algorithms. One builds the BWT on top of the *suffix array*. The other one is based on the BIG-BWT algorithm [9]. The BIG-BWT algorithm was adapted to run efficiently on distributed memory. The distributed BWT construction algorithms were implemented and extensively evaluated, proving that they can scale up to 6144 cores. To our knowledge, these are the first distributed BWT construction algorithms able to scale with thousands of cores. Our implementations are compared with the shared-memory parallel BIG-BWT [9] algorithm and the state-of-the-art shared-memory parallel BWT construction algorithm for repetitive strings, which is based on Lyndon words [51].

The evaluation on multiple real-world datasets showed that the developed distributed algorithms scale up to 6144 cores. This enables us to use more computation resources to construct the BWT. Our distributed PFP-based algorithm requires only 96 cores to match or surpass the runtime of the BWT construction based on Lyndon words [51] running on a single 48-core compute node. As our distributed PFP-based approach can be scaled up to 6144 cores, we achieve a speedup of up to 20.9x over the Lyndon-based construction on a single 48-core node.

1.3 Structure of Thesis

In Chapter 2, the fundamentals of strings and their notation are introduced. The Burrows-Wheeler Transform and the related data structures are defined. Furthermore, the distributed $\alpha\beta$ computation model on which our work is based is introduced. This chapter closes with an overview of related BWT construction algorithms.

Chapter 3 exhibits the building blocks on which our work is based. The PREFIX-FREE PARSING algorithm is introduced alongside (distributed) string sorting and distributed suffix array construction. Lastly, some repeating MPI patterns are presented.

The main contribution of this thesis is the distributed BWT construction algorithms presented in Chapter 4. First, the BWT construction based on the suffix array is introduced. Then, the sequential BWT construction based on PREFIX-FREE PARSING is introduced. Changes made to this algorithm to work on distributed memory systems are presented. The chapter closes with several notes on optimizations that improve the distributed construction of the BWT in distributed memory.

The implementation of the distributed BWT construction is evaluated in Chapter 5. We provide an overview of the implementation before presenting and comparing the performance of our distributed BWT construction algorithms to the shared-memory competitors.

Lastly, Chapter 6 concludes our work and gives an overview of future work on this topic.

2 Preliminaries

This chapter first introduces the string notation and string-based data structures used throughout this thesis in Section 2.1. Next, Section 2.2 introduces the computational model used and its communication methods. Other related BWT construction algorithms are presented in Section 2.3.

2.1 General Definitions

String Notation. A *string* T is made up of elements from a set, called an *alphabet* $\Sigma = \{0, 1, 2, \ldots, \sigma\}$. The sentinel \$ is used to mark the end of a string. *length* of T is denoted with n, so $T \in \Sigma^n$. $T[x]$ is the element of T at the position x with $0 \leq x \leq n-1$. Given two strings T, T', TT' is the *concatenation* of strings such that $TT' = T[0]T[1]\ldots T[n-1]T'[0]\ldots T'[n-1]$. $T[x..y]$ is a *substring* of T, starting at position x and ending with the character at position y: $T[x..y] = T[x]T[x+1]\ldots T[y]$. The substring $T[0..i-1]$ is called the *prefix* of size i. $T[n-1-j..n-1]$ is the *suffix* of size j.

A string u is the prefix of another string v if $\exists w \in \Sigma^* : uw = v$. Analogously, a string u is a suffix of another string v if $\exists w \in \Sigma^* : wu = v$. In both cases, the remainder string w could be the *empty string* ϵ. To differentiate, a string u is a *proper prefix* of v if $\exists w \in \Sigma^+ : uw = v$ and it is a *proper suffix* if $\exists w \in \Sigma^+ : wu = v$

The order of two strings is defined using a *lexicographic order*. Let $a, b \in \Sigma$ and T, T' be strings, this order implies if $aT \leq bT'$ then $a \leq b$ or $T \leq T'$.

Two strings T, T' are called *conjugate* if $T = uv$ and $T' = vu$ for $u, v \in \Sigma^*$. In other words, T and T' are cyclic rotations of each other.

Burrows-Wheeler Transform. The BURROWS-WHEELER TRANSFORM (BWT) was introduced in 1994 by M. Burrows and D.J. Wheeler as part of a lossless data compression algorithm [10]. The BWT itself is not a compression method but a reversible transform of the given data that makes it easier to compress. The main idea of the BWT is to cyclically rotate the input data and sort these rotations.

The following Example 1 shows the process of constructing the BWT of the \$ terminated string $S = \texttt{algorithm\$}$ with $N = 9$ and the alphabet $\Sigma = \{a, g, h, i, l, m, o, r, t, \$\}$ with the resulting $BWT(S) = \texttt{m\$ltrahgoi}$. First, all cyclic rotations of the input are

generated in a matrix. Then these rotations are lexicographically sorted. The BWT is defined as the characters in the last column of the sorted rotations matrix.

All rotations		**Sorted rotations**		
algorithm$		**$**algortih*m*		
lgorithm$a		**algorithm$**		
gorithm$al		**gorithm$***al*		
orithm$alg	$\Longrightarrow$	**hm$**algorit	$\Longrightarrow$	*m$ltrahgoi*
rithm$algo	sort	**ithm$**algo*r*	last column	
ithm$algor		**lgorithm$***a*		
thm$algori		**m$**algorit*h*		
hm$algorit		**orithm$**alg		
m$algorith		**rithm$**algo		
$algorithm		**thm$**algor*i*		

Example 1: BWT construction of the word algorithm$ using a sorted rotation matrix.

For the BWT to be reversible, the end of the input data in the transformation needs to be known. In their original proposal for the BWT, Burrows and Wheeler did not use an end-of-file symbol \$ but instead introduced an index I that points to the position of the rotation in the original input data. In their definition, the BWT returns the pair $(BWT(S), I)$.

A later definition by Manzini [45] introduced the unique end-of-file symbol \$ that is appended to the input data and included in all rotations. I now points to the row starting with the end-of-file symbol \$.

Properties of the BWT. For data compression using the BWT, the following property is used. Given a substring t of T, all other substrings of T that have t as a prefix will appear below one another in the sorted rotation matrix. A practical example of this behavior can be found in written text. The word the will appear often in written text. All rotations starting with he will appear below one another in the sorted rotation matrix. For all of these he that belong to the, the last character of the rotation is t. So all t belonging to the will appear grouped in the BWT [10].

A substring $BWT[i..j]$ is a *run* of the same character if for all $n \in [i, j] : BWT[n] = BWT[i]$, $BWT[i-1] \neq BWT[i]$ and $BWT[j+1] \neq BWT[j]$. Usually the number of runs is much smaller than the number of characters in a text. Therefore, *run-length encoding* can be used for compression. It stores character pairs with their corresponding run lengths [15].

An important data structure built on top of the BWT is the FM-INDEX [16]. Using the FM-INDEX, it is possible to perform a *backward search* to find the number of occurrences of a given pattern and their starting positions in the text.

Suffix Array. The *suffix array* (SA) [43] contains all lexicographically sorted suffixes of a given string T. The SA is defined as the index of the sorted suffixes of T, where $SA[n]$ points to the n-th smallest suffix of T. In other words, given the SA of T, for all $i \leq j \in [0, n) : T[SA[i]..n-1] \leq T[SA[j]..n-1]$ holds.
Given $T =$ `distributed`, Example 2 shows how the SA of T is constructed:

	Suffixes			**Sorted suffixes**			**Suffix array**
0	distributed		6	buted			6
1	istributed		10	d			10
2	stributed		0	distributed			0
3	tributed		9	ed			9
4	ributed	$\Longrightarrow$	5	ibuted	$\Longrightarrow$		5
5	ibuted	sort	1	istributed			1
6	buted		4	ributed			4
7	uted		2	stributed			2
8	ted		8	ted			8
9	ed		3	tributed			3
10	d		7	uted			7

Example 2: Suffix array construction of the word `distributed`.

BWT and SA. After creating all cyclic rotations of the input T and sorting them, this sorting step also sorts all suffixes, assuming the end-of-file symbol \$ was appended to T. The rotated phrases contain all strings that are conjugate to T. Let S be the set of all suffixes of T, and let T^i_{rotate} be the i-th cyclic rotation of T. Then $\forall s \in S \exists i \exists x$ prefix of T : $T^i_{rotate} = sx$.
This implies a definition of the BWT in terms of the SA. After implicitly sorting all suffixes, the BWT is defined as the last character of each conjugate; given $T^i_{rotate} = sx$, this is $x[|x|-1]$, for $x \neq \epsilon$. Because T^i_{rotate} is a conjugate of T, $x[|x|-1]$ is the character in T that precedes the suffix s. This leads to the following definition of the BWT based on the SA:

Definition 2.1.1 (Burrows-Wheeler Transform)**.**

$$BWT[i] = \begin{cases} T[SA[i]-1] & SA[i] \geq 1 \\ \$ & \textit{otherwise} \end{cases}$$

This connection between BWT and SA is illustrated in the previous Example 1. The suffix part of the sorted rotation phrases is bolded.

Longest Common Prefix Array. A data structure related to the suffix array is the *longest common prefix array* (LCP array) [28]. The *longest common prefix* is defined as $lcp(i, j)$. It stores the length of the common prefix of the suffixes starting at $T[i]$ and $T[j]$, respectively. The LCP array contains the lcp values for all consecutive suffixes of the suffix array. The lexicographically smallest suffix has no preceding phrase. Therefore, the wildcard $\perp$ is used for the first entry in the LCP array:

$$LCP[j] = \begin{cases} lcp(T[SA[j]..|T|-1], T[SA[j-1]..|T|-1]) & j > 1 \\ \perp & \text{otherwise} \end{cases}$$

For the above example $T =$ `distributed` the LCP array is:

Sorted suffixes		**LCP array**
`buted`		$\perp$
`d`		0
`distributed`		1
`ed`		0
`ibuted`	$\Longrightarrow$	0
`istributed`	$lcp(i,j)$	1
`ributed`		0
`stributed`		0
`ted`		0
`tributed`		1
`uted`		0

Example 3: LCP construction of the word `distributed`.

2.2 Model of Computation

This thesis describes algorithms implemented for distributed memory systems. A distributed memory system consists of multiple nodes. Each node houses multiple cores. Within a node, each core has its own private L1 and L2 caches, but they share the same physical L3 cache and main memory. The nodes do not share their main memory, but are connected via a network. Figure 2.1 shows an overview of a generic distributed memory system. In distributed computing, the term processing element (PE) is used to describe a single processing unit. Typically, a single core is mapped to one PE. Depending on the physical memory assigned to a PE, its access time to that memory varies across PEs. This effect of different memory access times for PEs is called *nonuniform memory access* (NUMA) [53].

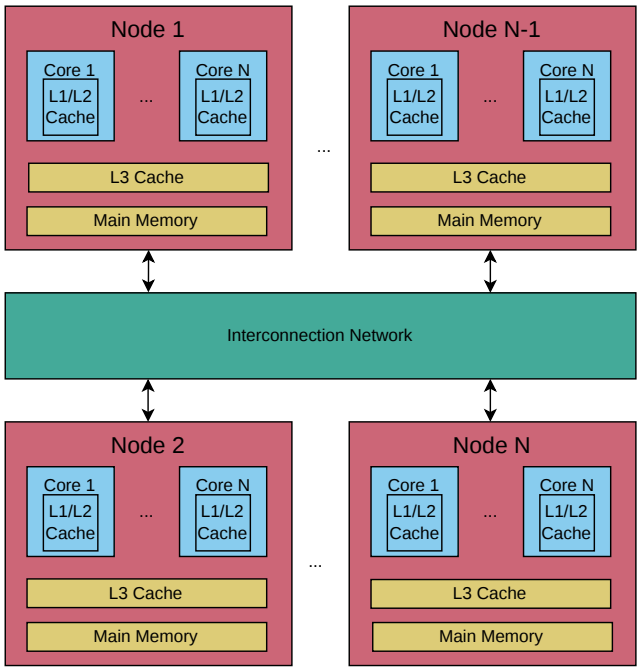


Figure 2.1: Schematic overview of a generic distributed memory system.

Because accessing memory on another node is not possible, messages are exchanged via the interconnection network to share data among nodes and PEs. The time needed to exchange a message of length n in a point-to-point communication between two PEs is described using the linear compute model [23]:

$$T = \alpha + n\beta$$

The constant α describes the start-up cost and β the propagation time of one data unit, which depends on the used network and its performance.

In this thesis, multiple different modes of communication between the PEs are used. In the following, these modes are introduced and visualized in Figure 2.2. In the following paragraphs, k is the number of PEs, and n is the size of the message m.

Broadcast. A *broadcast* is used to send a message m to all PEs. The algorithms in our work do not use broadcast directly, but it serves as the basis for other communication modes. Broadcast uses *pipelining*, which is done by splitting a message m into b packets of size $\lceil n/b \rceil$. These packets are spread to all PEs using a binary tree. In the binary tree, each non-leaf PE must send the packets to both of its child nodes. If the binary tree is perfectly

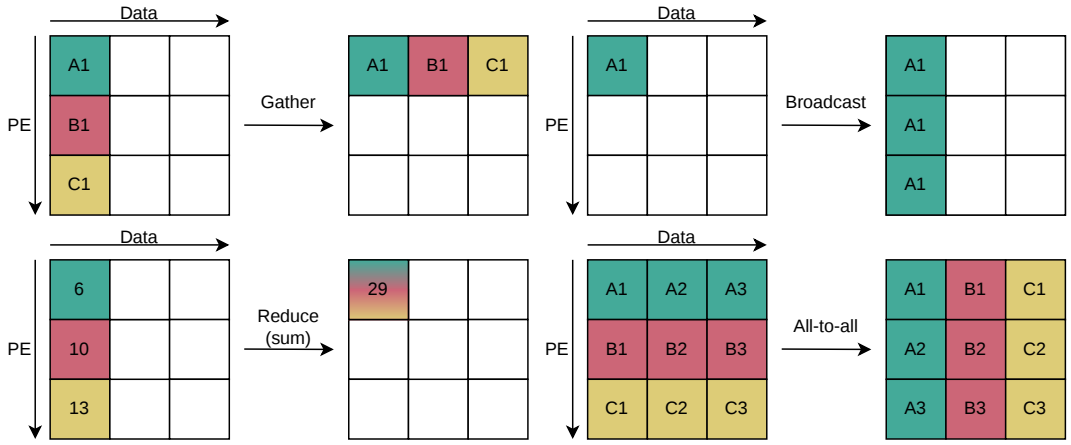


Figure 2.2: Communication patterns: Gather, Broadcast, Reduce and All-to-all.

balanced, which holds if the length of the longest root-to-leaf path is $\lfloor \log k \rfloor$ [53], we get the following runtime for broadcast:

$$T_{broadcast} = O(\beta n + \alpha \log k)$$

Reduction. A reduction is used to aggregate messages from different PEs into a single global result. The aggregation is described using an associative operator $\otimes$, such that $\forall a, b, c : (a \otimes b) \otimes c = a \otimes (b \otimes c)$. Given the message m_i on PE i, the reduction computes $\otimes_i m_i$ for all PEs i participating in the reduction. Using pipelining and a perfectly balanced binary tree, the communication time for the reduction is [53]:

$$T^{\otimes} = O(\beta n + \alpha \log k)$$

All-Reduce. The basic reduction returns the global aggregation only to one PE. To distribute the global result to all PEs, all-reduce is used. All-reduce computes the reduction result and then broadcasts it.

Scan. If the aggregation $\otimes$ is only needed for the PEs preceding PE i, a scan or a prefix sum is used. The result of a scan on PE i is $\otimes_{j \leq i} m_i$. To exclude the message m_i, the exclusive scan is used: $\otimes_{j<i} m_i$. The communication cost for a scan using pipelining and a binary tree is [53]:

$$T_{scan} = O(\alpha n + \beta \log k)$$

Gather. The goal of the gather operation is to concatenate the messages m_i from all PEs i into a single root PE j, such that j holds $[m_0 m_1 \dots m_{p-1}]$. If all messages m_i have the same length, gather can be viewed as a reduction with the (associative) concatenation operator. In case the messages have different lengths, using a binomial tree, the communication time of the gather operation is [53]:

$$T_{gather} = \lceil \log k \rceil \alpha + (k-1)n\beta$$

To gather the resulting $[m_0 m_1 \dots m_{p-1}]$ across all PEs, an all-gather is used, which is a gather followed by a broadcast.

All-to-all. To move messages from every PE to every other PE, the all-to-all operation is used. For all PEs j, PE i has a message m_{ij} it wants to send to PE j. In case all massages m_{ij} have the same length, the communication cost is [53]:

$$T_{all\_to\_all} \leq (k-1)(n\beta + \alpha)$$

For the all-to-all exchange, all PEs need to communicate with every other PE. These communication costs are accounted for in the *telephone model*, in which all communication occurs between pairs of PEs. It must be ensured that there are no deadlocks caused by PEs sending rather than receiving. To resolve this, the sequence $s = 2r \bmod p$, with $r \in [0, \dots, k-1]$, is used. Two PEs i and j communicate with each other in round r if $i + j = s$. This ensures that every PE communicates with every other PE (including itself) exactly once. Note that this only works for an odd k. For even k there is no PE that communicates with itself each round. Instead, this PE communicates with PE $k-1$.

A widely used variation of the standard all-to-all is the all-to-all-v, in which the messages m_{ij} have different lengths. In this case, the communication cost depends on the *h-relation*, which specifies the maximum amount of data that a single PE can send or receive. With the *bottleneck communication volume* $h = \max_i \max(\sum_j |m_{ij}|, \sum_j |m_{ji}|)$ the communication cost is: [53]:

$$T_{all\_to\_all\_v} \leq 2T_{all\_to\_all}(\frac{h}{k} + 2k^2) \approx 2k\alpha + (2h + 4k)\beta$$

This communication cost is achieved by using two all-to-all exchanges. Each PE splits its messages m_{ij} into k equally sized pieces. For the first all-to-all exchange, PE i sends all pieces m_{i*h}, which are all messages sent from PE i to other PEs via PE h. In the second exchange, PE h aggregates the received messages into m_{*jk} and sends them to PE j [53].

2.3 Related Work

There have been efforts to improve the performance of the BWT construction before. In the following, some of these approaches are briefly introduced.

GPU-enhanced BWT Construction. Zhao et al. presented a GPU-based approach for genomic data [59]. They used the blockwise computation of the SA and the BWT proposed by Kärkkäinen [38]. The idea is not to compute the SA fully at once, but to select several splitting suffixes and distribute suffixes into blocks based on the splitting suffixes. Each block is sorted, and the corresponding BWT block is computed. By GPU-parallel radix sorting the suffixes of each block and then computing the corresponding BWT block in main memory, they achieved a speed-up of 16.2 on a whole human genome compared to the CPU-based implementation by Kärkkäinen [59].

Ilya Grebnov implemented LIBCUBWT [27], a CPU-based implementation of an SA-based BWT construction. The suffix array is built by combining *prefix doubling* and the DC3 algorithm. In Section 3.3 these techniques will be briefly introduced.

External Memory BWT Construction. With BWT-DISK, Ferragina et al. present a BWT construction algorithm that operates in external memory [17]. They split the input T into blocks $S^n, \ldots, S^1$ of size M. The BWT is constructed by parsing the blocks in reverse order, starting with S^1. The idea is that with $BWT(S^i S^{i-1} \ldots S^1)$, the relative order of the suffixes is keep, while S^{i+1} is inserted to obtain $BWT(S^{i+1} S^i \ldots S^1)$. For an input of size n, this takes $O(n^2/m)$ time.

BWT Variants. The BWT can not only be defined on a single string T, but also on a string collection $S = S_1, \ldots, S_n$. Especially in bioinformatics, where the BWT of multiple individual genomic sequences is computed, viewing the input as a collection of strings is a sensible approach. The extended BWT [44] (eBWT) was introduced by Mantaci et al. as a generalization of the BWT for a multiset of strings. Following the regular BWT, it uses all rotations of all input strings. But instead of sorting them lexicographically, they are sorted by ω-order. Two strings a, b are ω-ordered $a \prec_\omega b$ iff $a^\omega < b^\omega$ which is the infinite repetition of a and b respectively. The eBWT is the last character of all ω-sorted rotations.

The ROPEBWT3 algorithm [39] computes the BWT for string collection $S = (T_1\$_1, \ldots, T_m\$_m)$ with m strings, using m unique delimiters smaller than the used alphabet after each string. To compute the BWT of this collection, the BWT of a subset is computed using the BWT construction library LIBSAIS [26]. This partial BWT is then merged into the existing run-length encoded BWT. This is repeated until the BWT of the whole input is computed.

The BCR-BWT algorithm [5], named after the authors Bauer, Cox, and Rosone, works on string collections. It needs $O(m \log m)$ memory to process m strings, making it efficient when processing many small strings. They also show that the BWT of an existing string

collection can be updated if a string is removed or added to the collection. There exist several implementations based on the BCR-BWT algorithm; for example, the MSBWT algorithm [30] can merge multiple BWTs into one BWT.

BWT Construction based on Lyndon Words. Jannik Olbrich presented the state-of-the-art BWT construction algorithm for repetitive strings [51]. Instead of computing the SA of the input string, it uses a grammar to compress the input before computing the BWT. A *Lyndon word* is a string that is lexicographically smaller than all its proper cyclic rotations. For example, `bab` is not a Lyndon word, because `abb` $<$ `bab`. On the other hand, `aab` is a Lyndon word. According to the *Chen-Fox-Lyndon theorem* [12], every non-empty string T has a unique Lyndon factorization. That is, a sequence of lexicographically non-increasing Lyndon words $v_1 \geq v_2 \cdots \geq v_n$ with $T = v_1 v_2 \dots v_n$. Each of these Lyndon words w is split into two words $w = uv$ with v being the longest proper suffix of w that is a Lyndon word. Recursively applying this split yields a binary tree for each factor, whose nodes are converted into the rules of a grammar. Identical subtrees are assigned the same grammar rule, thereby compressing repetitive substrings in the input. Olbrich showed that the BWT can be constructed by sorting grammar rules lexicographically according to the strings they produce. The algorithm constructs the grammar as it reads the input. It is therefore an online algorithm and reads the input once.

3 Building Blocks

In this chapter, the algorithms that serve as building blocks for the distributed BWT construction are presented. The algorithms are presented in the order the BWT construction uses them. Section 3.1 shows what PREFIX-FREE PARSING is and how it works. Next, Section 3.2 introduces (distributed) string sorting and Section 3.3 distributed suffix array construction. Last, recurring MPI patterns are introduced in Section 3.4.

3.1 Prefix-Free Parsing

PREFIX-FREE PARSING (PFP) is a preprocessing step that aims to compress the input data in a lossless and reversible manner. PFP was first introduced to facilitate the construction of the BWT [9]. Other works have shown that PFP can be used as a preprocessing step for various string algorithms, such as the LEMPEL-ZIV 77 factorization [31], the construction of the Wheeler Graph [25], the construction of the BWT [9] and of the eBWT [8].

PFP computes a set D, called the dictionary, and a list P, called the parse. Given a text $T[0..n-1]$ over an alphabet Σ, and a set $E \subseteq \Sigma^w$ with $w \geq 1$. Let $E' = E \cup \{\#, \$^w\}$ with $\#$ and $\$$ being elements smaller than all elements in Σ. D is defined as the maximum set with the following properties for all phrases $d \in D$:

Definition 3.1.1 (PFP dictionary properties)**.**

- *d is a substring of $\#T\$^w$*
- *Exactly one proper suffix of d is in E'*
- *Exactly one proper prefix of d is in E'*
- *No other substring of d is in E'*

In other words, the dictionary D consists of substrings of $\#T\$^w$ that begin with an element of E', end with an element of E', and contain no other element of E' as a substring. D is computed by iterating over $\#T\$^w$ and matching he splitter strings $e \in E'$ at the current position. If a match is found, the text is split after e. The next phrase starts with e and continues until the next splitter string is found. All elements of E' (apart from the start-of-text symbol $\#$) are of size w. Therefore, each found phrase d in $\#T\$^w$ has an overlap of w at its beginning with the ending of the previously found phrase. After all unique phrases d have been found, D is sorted lexicographically, and each phrase is assigned its lexicographical rank $[0, 1, \ldots, |D| - 1]$.

The parse P is computed by iterating over $\#T\$^w$ and determining the order in which the phrases $d \in D$ occur in $\#T\$^w$. For each occurrence, the rank of d is appended to P.

The resulting relationship between D, P and T is as follows. Given $P[i]$, let $d_i \in D$ be the i-th phrase in the parse. With d_i^w being the phrase obtained by removing the suffix of length w from d_i (which is the w overlap), the relationship is:

$$\#T\$^w = [d_0^w d_1^w \dots d_{|P|-1}^w]$$

Instead of storing T, just the dictionary D and the parse P need to be stored. The relationship shows that the compression achieved by PFP works especially well if T is repetitive. For example, given a set of splitter strings E and three strings a, b, c, each starting and ending with a splitter string $e \in E$. For an input string T made up just of a, b, c, like $T = abbbbcbbbbaaaaabaac$, only the phrases a, b, c as D and P, with one entry for each occurrence of the phrases, need to be stored.

PFP Example. Figure 3.1 shows how the dictionary D and the parse P are computed from an input string T. First, T is split into phrases at the marked splitters. Each phrase ends with a split and starts with the w overlap of the splitter. Then the obtained phrases are deduplicated and sorted. After assigning each phrase its rank, the parse is obtained by encoding the phrases from the first step with their ranks according to the dictionary. This example will be used as the running example throughout this thesis.

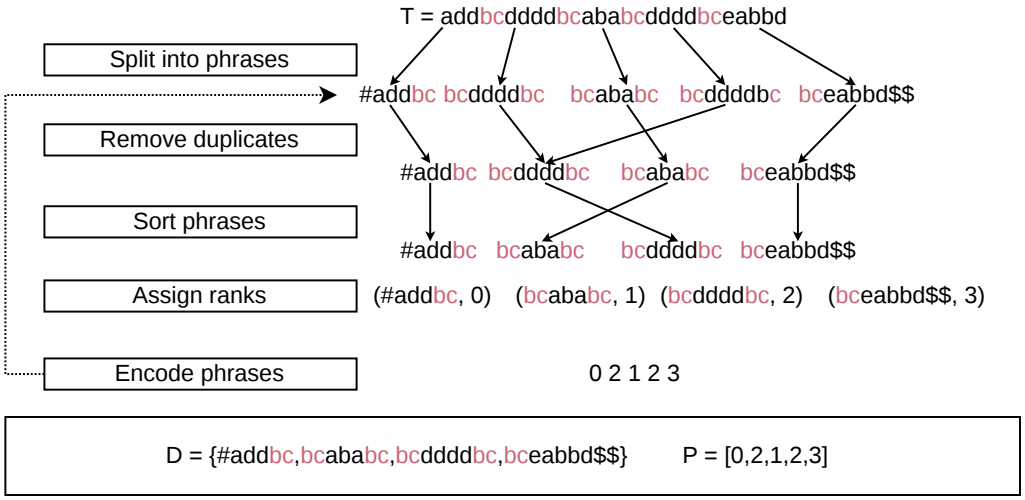


Figure 3.1: Example showing the construction of the dictionary and parse of PFP.

Strictly following this definition, one would have to explicitly define the set of splitter

strings E for every input, matching the alphabet Σ of this input. Therefore, Boucher et al. proposed implicitly defining the splitter set E using a rolling hash [9].

3.1.1 Karp-Rabin Fingerprint

In practice, it is not feasible to explicitly define the set E of splitter strings for PFP. Instead, hashing is used, with the strings splitting whenever the hash over w characters equals a specific value. The hashes in this context are called fingerprints. In their work, Boucher et al. use the Karp-Rabin fingerprint [9].

To determine the positions of the splitter strings, a *linear pattern-matching problem* needs to be solved. Given a text T and a pattern P, determine the indexes i where P occurs in T with:

$$i < |T| - |P| : T_i = P_0, T_{i+1} = P_1, ..., T_{i+|P|} = P_{|P|}$$

This problem can be solved by computing the fingerprint of P and all substrings of T of size $|P|$. Then the fingerprint of P is compared to the fingerprints of all substrings of T. Every match indicates an occurrence of P in T. This approach takes $O(P)$ to compute the fingerprint of each substring. With $|T| - |P|$ substrings of size $|P|$, this would take $O(P \cdot (|T| - |P|))$ just to compute all required fingerprints.

Karp and Rabin found out that, instead of computing the fingerprint of all substrings of length $|P|$ in T, the fingerprint of the substring $T[i, i + |P|]$ can be computed from the fingerprint of $T[i - 1, i - 1 + |P|]$ in $O(1)$ instead of the naive $O(P)$ needed to compute a whole fingerprint [35]. This approach can be used in a sliding window of size w. The fingerprint of the first w characters is computed once in $O(w)$, and the fingerprint of the next window $[1, \ldots, w + 1]$ can then be computed in $O(1)$.

The Karp-Rabin fingerprint $\phi(x)$ of a string x with size w is defined as:

$$\phi_0(x) = \sum_{i=0}^{w-1} x_i c^{w-1-i} \bmod p$$

The sliding window, adding one character to the fingerprint, is defined as:

$$\phi_{i+1}(x) = (\phi_i(x) + p - (x_i c^{w-1} \bmod p))c + x_{i+w} \mod p$$

with c being a random integer and p prime.

PFP and Karp-Rabin Fingerprints. In PFP, Karp-Rabin fingerprints are used to determine where to split the string. This implicitly defines the set of splitter strings $E = [x|\phi(x) \bmod p = 0]$ with p being a constant integer used to control the size of E.

Using the Karp-Rabin fingerprints, there is a small chance of false positives. Given two strings x, y with $x \neq y$, $\phi(x) = \phi(y)$ could hold. In the context of PFP, the fingerprinting is used to determine the splitter strings E, so these rare false positives do not affect the computation.

With the Karp-Rabin fingerprint, PFP is defined in terms of two parameters. First, w is the window size used for the sliding fingerprint. This is the length of implicitly defined elements in E. The other parameter is p, which is the modulus used to determine which fingerprints trigger a split.

Increasing w increases the overlap between the strings, while increasing p reduces the total number of elements in D.

Lemma 3.1.2. *Assuming the sliding Karp-Rabin fingerprint $\phi(x)$ is modeled as a random hash function. The average length of a phrase extracted by PFP is $w + p$.*

Proof. Given a uniformly distributed random input, the average size of the elements in D can be approximated using w and p. Each time a character is added to the Karp-Rabin window, there is a probability π of $1/p$ that the newly computed fingerprint modulo p is 0. This indicates a Bernoulli process with probability $1/p$ of a split and $1 - 1/p$ of no split. The geometric distribution defines the number of trials needed to get a successful Bernoulli experiment. In this case, this is the number of characters scanned before reaching a split. For the geometric distribution, the expected value is $E(split) = 1/\pi$ with $\pi = 1/p$; this boils down to $E(split) = p$. On average, there is a split every p characters. Because each phrase (except the first one) has the previous split window of size w prepended, the expected length of a phrase is $w + p$. □

3.2 Sorting Algorithms

There are several sorting algorithms, each with advantages and disadvantages depending on their use case. The BWT construction algorithm presented in this thesis uses two different distributed sorters. One is used to sort integers distributed across multiple PEs. The other one is used to sort strings distributed across PEs. The algorithms used in the BWT construction are briefly presented below.

Adaptive Multi-level Sample Sort. The general distributed sorting algorithm used in the BWT construction is ADAPTIVE MULTI-LEVEL SAMPLE SORT (AMS-SORT) [2, 3]. AMS-sort assumes that there is a data amount n distributed across k PEs, with the PEs being arranged in r groups. It follows the sample sort approach by first drawing abr random

samples from the data, with a and b being tuning parameters. These samples are sorted in parallel, and $br-1$ equidistant splitters are chosen. After all-gathering the splitters, each PE locally creates br buckets accordingly. The computed buckets are globally load-balanced per group based on the globally all-reduced bucket sizes. With suitable parameters $b = \Omega(1/\epsilon)$ and $a = \Omega(\log r)$, the upper bound on the number of elements in each group is $1 + \epsilon\frac{n}{r}$. The groups are then recursively split into groups of size $\frac{k}{r}$, and the process is repeated until each group contains a single PE, after which the data is sorted locally and globally.

Scalable Distributed String Sorting. The MULTI-LEVEL MERGE SORT (MS) used by Kurpicz et al. [37] adapts the idea of the previously presented AMS-SORT to distributed string sorting. The algorithm is initialized by locally sorting the set of strings on each PE and computing the LCP array. PEs are assigned into r groups of size k/r. Globally, $r-1$ splitters are computed, and each PE locally assigns its data (strings and LCP values) to r buckets $[b_0, \dots b_{r-1}]$. The content of each bucket b_i is then distributed to the corresponding group i. The received sorted string buckets are then merged using the LCP values. This results in each PE having locally sorted its assigned buckets. This is done recursively until the strings are globally sorted.

3.3 Distributed Suffix Array Construction

There are several approaches to computing the suffix array in distributed memory. For this thesis, the current fastest distributed suffix array construction algorithms (SACA) are considered. Haag et al. [29] presented a distributed memory suffix array construction algorithm based on the *difference cover* algorithm introduced by Kärkkäinen and Sanders [33]. The other distributed SACA was introduced by Flick and Aluru [19]. This algorithm is based on *prefix doubling*. Because their algorithm does not have a distinct name, their project will be referred to as Parallel Suffix Array and Tree Construction (PSAC).

3.3.1 Distributed DCX

First, the concept of difference cover has to be introduced. Given an integer X, a subset $D_X \subseteq \{0, \dots, X\}$ covers the set $\{0, \dots, X\}$ if for all $r \in \{0, \dots, X\}$ there are $i, j \in D_X$ with $i - j \mod X = r$ [33]. The SACA algorithm based on difference cover first used DC3 [33]. In this case, $D_X = \{1, 2\}$. The distributed SACA uses the generalized form DCX, with $X = 39$ showing the most promising results [29].

The SACA based on DCX is defined in 3 steps:

(i) A difference cover sample set is created. It contains all suffixes S_i with $i \bmod X \in D_X$. This sample set is lexicographically sorted by the X-length prefix, and each

suffix is assigned its rank in order. If all X-length prefixes are distinct, all suffixes can be unambiguously sorted, and step (ii) can be skipped. Otherwise, the recursive step (ii) is used.

(ii) If the sample suffixes from step (i) can not be distinctly sorted by their X-length prefix, their ranks are concatenated into a new string, and the DCX algorithm is recursively applied to this string to obtain a unique ordering.

(iii) For each suffix s_j the following triple is constructed: Its prefix of length X, the ranks of $|D_X|$ many samples from the sample set of suffixes coming after s_j in T, the text index j. Sorting these triples by the first two components and retrieving the text index j of each triple in order yields the SA.

In the distributed settings, it is not feasible to materialize all length-X prefixes and possibly exchange them for the sorting step. Therefore, $q + 1$ splitters are used to split the to-be-sorted prefixes into q buckets. Then each bucket is distributed string-sorted by itself, with only the suffix in one bucket being materialized at a time. Using random chunking, the expected elements per bucket and per PE are $n/(pq)$, and the probability that a PE receives more than $2n/(pq)$ is $1/p^\gamma$ for $\gamma > 0$.

For a given string T of size n over an alphabet with σ elements, q buckets, the complexity of the distributed DCX SACA operating on p PEs is [29]:

$$DCX(n,p) = O(X\frac{n}{p}\log n + \alpha \log p(q + qk + \sqrt[k]{p}) + \beta X\frac{n}{p}(\log \sigma + \frac{\log n}{\sqrt{X}}))$$

The first summand denotes the local work, the second the start-up latency, and the third the communication cost.

3.3.2 Distributed PSAC

PSAC [19, 20, 21] is the distributed implementation based on *prefix doubling*. The underlying technique was introduced by Karp et al. in 1972 [34]. Manber and Myers later applied it to construct the suffix array [43].

Prefix doubling first sorts all suffixes according to their first character. After this, the 1-ordering of the suffixes is known. The *h-ordering* is the order of suffixes considering only the first h characters: $T[i..n] \leq_h T[j..n] \Leftrightarrow T[i..i+h) \leq T[j..j+h)$. Because this ordering is ambiguous, suffixes with the same first h characters are assigned the same bucket. To obtain the $2h$-ordering, each suffix $T[i..n]$ is sorted by the pair consisting of its own bucket and the bucket of $T[i + h..n]$. The bucket $T[i + h..n]$ is known by the h-ordering of the previous iteration. Doubling the h-prefix in each iteration will yield the correct SA after at most $\lceil \log n \rceil$ iterations.

To compute the BWT from the dictionary and parse generated by PFP, the LCP array of the dictionary is needed. PSAC is the state-of-the-art distributed algorithm able to compute the SA and the LCP array. Therefore, our implementation will use their algorithm.

3.4 MPI Patterns

Request-Response Pattern. The *request-response pattern* is used to exchange data, assuming that each PE holds data and all other PEs need to retrieve some of this data. This is achieved by using all-to-all-v twice. The first all-to-all-v is used to send the requests to the PEs owning the requested data. This requires each PE to know which other PE has to handle the request. After receiving requests, each PE computes the responses and constructs the response message passed to the second all-to-all-v accordingly. The function $\phi(x)$ is used to compute the response for a request x. The pattern is shown in Algorithm 1.

Algorithm 1: Request-Response Pattern

`Data`: Vector of requests ordered by PE R, response function ϕ
`Result`: Responses ordered by PE

```
1 (recvReq, r) ← Alltoallv(R)          // requests, r = recv counts
2 resp ← [ φ(x) : x ∈ recvReq ]        // resolve locally
3 out ← Alltoallv(resp, r)             // responses
4 return out
```

Assuming there are n PEs that have a request volume of $[r_1, r_2, \ldots, r_n]$ with $h^{req} = \max([r_1, r_2, \ldots, r_n])$.The volume of the responses each PE has to send is $[r'_1, r'_2, \ldots, r'_n]$ with $h^{resp} = \max([r'_1, r'_2, \ldots, r'_n])$. The complexity of an irregular all-to-all is $O(\alpha p + h\beta)$. Given the complexity of the response function ϕ as γ, and $h = \max(h^{req}, h^{resp})$ we get:

Lemma 3.4.1. *The complexity of the request-response pattern is*

$$O(\alpha k + h\beta + h^{resp}\gamma)$$

With α being the start-up latency and β denoting the propagation time needed for one data unit.

Irregular Request-Response Pattern. If the response to a request is irregular (not all responses are the same size), e.g., if the response is a dynamically sized vector, the simple request-response pattern cannot be used anymore because it assumes that each response is as large as one data unit. In the irregular case, the responses can be of different sizes, larger than one data unit.

The pattern used for irregular responses is shown in Algorithm 2. An additional all-to-all-v is used to make the response sizes available. This all-to-all-v contains the size for each response. The second all-to-all-v contains the flattened responses. Using the sizes, the response can be reconstructed from the flattened responses.

Algorithm 2: Irregular Request-Response Pattern

`Data`: Vector of requests ordered by PE R, response function ϕ
`Result`: Variable-length responses ordered by PE

1 $(recvReq, r) \leftarrow$ Alltoallv(R) *// requests; r = recv counts*
2 $resp \leftarrow [\, \phi(x) : x \in recvReq \,]$ *// resolve locally*
3 $sizes \leftarrow [\, |y| : y \in resp \,]$ *// size of each response*
4 $recvSizes \leftarrow$ Alltoallv($sizes$, r) *// exchange sizes*
5 $out \leftarrow$ Alltoallv(flatten($resp$), $recvSizes$) *// responses*
6 `return` out

In the regular case, each request results in a response of one response data unit. Therefore, the maximum response volume h^{resp} is the same as the number of times ϕ is applied to get the responses. In the irregular case, this is not the case. One request can result in a response of multiple data units. Let h^{max} denote the maximum number of requests or responses a single PE receives and γ the single largest response produced by ϕ.

Lemma 3.4.2. *The complexity of the irregular request-response pattern is*

$$O(\alpha k + h\beta + h^{max}\gamma)$$

It is worth noting that in the irregular case, the response volume h^{resp} can grow faster than in the regular case. The response function ϕ could translate each request into an array of multiple elements, increasing the response volume $h^{max}\gamma$ by multiple factors compared to the regular case.

4 Distributed BWT Construction

This chapter introduces the distributed BWT construction algorithms. The distributed BWT construction based on the SA is shown in Section 4.1. The distributed PFP-based BWT construction algorithm is presented in Section 4.2. It first shows the sequential approach BIG-BWT as presented by Boucher et al. [9] before introducing the changes made to the BIG-BWT algorithm to work efficiently in distributed memory.

4.1 BWT from Suffix Array

Constructing the BWT can be easily done by computing the SA of the given string and applying the definition of the BWT 2.1.1. Iterating over the SA and emitting the character preceding the current suffix will yield the BWT.

This is directly transferable to the distributed case. First, the SA of T is computed using a distributed SACA. After that, PE i holds a chunk T_i of the input data and a chunk SA_i of the SA. Next, the local SA_i is iterated. Because the input text is distributed across all PEs, a simple lookup for the character immediately preceding the current suffix is not possible if that character is not part of the local chunk T_i.

In Algorithm 3, the following distributed SA-based BWT construction algorithm is described for PE i. To obtain the characters that are not stored locally, the PE that owns them needs to be computed. This is done by computing the prefix sum over the lengths of all local text chunks T_i. For an index j, the PE owning $T[j]$ can be computed by using a binary search on the text-length prefix sum.

When a non-local suffix is found, the owner is computed and stored in a list of requests. After all PEs have finished iterating over the local suffixes, the local requests are used with the request-response pattern introduced in Section 3.4. With the missing characters obtained, each PE can fill the missing BWT characters.

The more PEs are used to run this algorithm, the less likely it is that the required characters are in the local T_i. This can result in high memory consumption by the receive-response pattern, as it has to store the send and the receive buffers. Therefore, the local SA_i is processed in buckets. With b buckets, $|SA_i|/b$ entries of SA_i are processed before each of the b request-responses.

Algorithm 3: Distributed BWT construction from the distributed suffix array.

```
Data: SA_i, T_i, b
Result: BWT_i of T
1  Ps ← Scan(|T_i|)
2  Offset ← Ps[i]
3  BWT ← [⊥, ..., ⊥]                                   /* |SA_i| entries */
4  BucketSize ← ⌈|SA_i|/b⌉
5  for B ← 0 to b − 1 do
6  |  Requests ← [ ]
7  |  for n ← B · BucketSize to min{(B + 1) · BucketSize, |SA_i|} − 1 do
8  |  |  s_n ← SA_i[n]
9  |  |  if s_n − Offset ∈ [0, |T_i|) then              /* T[s_n] local */
10 |  |  |  BWT[n] ← T_i[s_n − Offset]
11 |  |  else                                           /* T[s_n] not local */
12 |  |  |  append (FindOwner(Ps, s_n), s_n) to Requests
13 |  |  end
14 |  end
15 |  Missing ← RequestResponse(Requests)
16 |  n ← 0
17 |  for j ← B · BucketSize to min{(B + 1) · BucketSize, |SA_i|} − 1 do
18 |  |  if BWT[j] = ⊥ then
19 |  |  |  BWT[j] ← Missing[n]
20 |  |  |  n ← n + 1
21 |  |  end
22 |  end
23 end
```

4.2 BWT from Prefix-Free Parsing

This section introduces the distributed PFP-based construction algorithm. To explain how this algorithm works, we first present the sequential approach BIG-BWT as presented by Boucher et al. [9]. Then, we state the changes made to the sequential algorithm to work in distributed memory.

4.2.1 Sequential BWT from Prefix-Free Parsing

Given a string T, PFP computes the dictionary $D^{w,p}$ and the parse $P^{w,p}$, based on the window size w and the used modulus p. For readability, the following will use D and P to denote the parameters. The set S is defined as the set containing all suffixes of elements in D, so $S = \{d[i]..d[|d|-1] \mid \forall d \in D \forall i \in [0, |d|-1)]\}$.

The PFP algorithm is based on the fact that S is a *prefix-free set*. A prefix-free set is a set where no element is a proper prefix of another element:

Definition 4.2.1. *Given a set $K \subseteq \Sigma^*$, K is a prefix-free set if*

$$\forall s, t \in K : s \neq t \implies s \text{ not proper prefix of } t$$

Using this definition and the definition of the set S, we get:

Lemma 4.2.2. *The suffix set S is prefix-free.*

Proof. Suppose there is an element $u \in S$ that is a proper prefix of another element $v \in S$ with $u \neq v$. According to the definition of D, the last w characters of u are in the set of splitter strings E. Then either $u = v$, contradicting their definition, or there is a substring s in v that is neither a proper prefix nor a proper suffix of v, contradicting the definition of D. □

For the correctness of this algorithm, refer to the corresponding proofs in the original paper [9].

The PFP-based BWT construction revolves around the SA of the dictionary D. In the following, this SA is denoted by SA_D. To compute SA_D, the phrases are appended with a delimiter symbol & between phrases that is lexicographically smaller than all other characters. Each suffix $s \in SA_D$ maps to exactly one $d \in D$. Because S is prefix-free, there can not be a suffix s that maps to more than one $d \in D$. Multiple properties can be derived from this suffix-phrase mapping:

Definition 4.2.3 (Suffix-phrase mapping)**.** *The lexicographical rank of the phrase d in D, to which s points, is denoted by $r_D(s)$.*

Definition 4.2.4 (Suffix lengths in D)**.** *The length of a suffix in the phrase it points to is denoted using $|s|_D$. That is, the length of the suffix s in its corresponding phrase d starting at the position indicated by s.*

Definition 4.2.5 (Suffix-suffix mapping)**.** *The phrase suffix starting at position s and ending at the end of the corresponding phrase is denoted by $s_D(s)$*

Definition 4.2.6 (Number of phrase appearances)**.** *The number of times a phrase d appears in the parse is denoted by $a_P(d)$.*

In Figure 4.1 an example of $s_D(s), |s|_D, r_D(s)$ and $a_P(d)$ is shown. This example reuses the example from Section 3.1. Given a suffix $s \in SA_D$, $s_D(s)$ returns the substring from the character s points to until the end of the phrase. The end of the string is reached when the phrase delimiter $\&$ is reached. $r_D(s)$ is obtained by finding the corresponding phrase. The parse is used to determine the number of appearances $a_P(d)$ of that phrase.

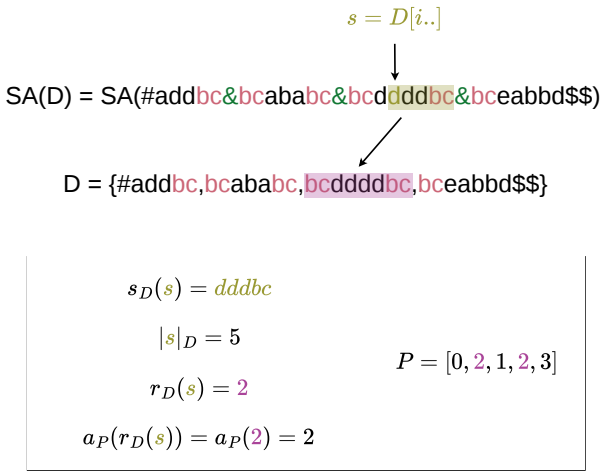


Figure 4.1: Example for $s_D(s), |s|_D, r_D(s)$ and $a_P(d)$.

This suffix-phrase mapping is used to obtain the BWT from the compressed output provided by PFP. For each suffix $s \in SA_D$, the BWT construction algorithm needs to determine which character to emit and how many times. To do this, the definition of SA-based BWT construction 2.1.1 is followed, but using the SA of the PFP dictionary SA_D instead of the SA of T. The BWT definition states that the i-th character of the BWT is the character preceding the i-th suffix. The suffixes $s \in SA_D$ provided by PFP differ from the general suffixes provided by the SA. In the following, these differences are stated and illustrated to ease the understanding of the BWT construction algorithm. For readability, s_{PFP} denotes a suffix from SA_D, and s_T a suffix from the SA of T.

Semantics. s_T points to a suffix of the input string T. s_{PFP} points to a suffix of the dictionary D with phrases being separated with a delimiter $\&$.

Occurrences. The phrase s_{PFP} maps to can appear multiple times in the parse. A suffix s_T only appears once in the given T.

Uniqueness. For a suffix s_T, the character in front of it is unambiguous. It is the character at position $T[SA_T[s_T] - 1]$. For s_{PFP}, the preceding is not necessarily unambiguous. If s_{PFP} maps to the first character of a phrase, then the character preceding it is the last character of the previous phrase. A phrase can occur multiple times in the parse and can have different preceding phrases with different last characters.

Order. For s_T the lexicographical order is well defined. Each subsequent suffix of s_T is equal or smaller than it. For s_{PFP}, this is not given. There can be two suffixes s_{PFP} and s'_{PFP} that map to the same phrase suffix, $s_D(s_{PFP}) = s_D(s'_{PFP})$. In this case, their order is defined by the phrases occurring after $r_D(s_{PFP})$ and $r_D(s'_{PFP})$. Both phrases can appear multiple times in the parse with different subsequent phrases each time, making the order ambiguous.

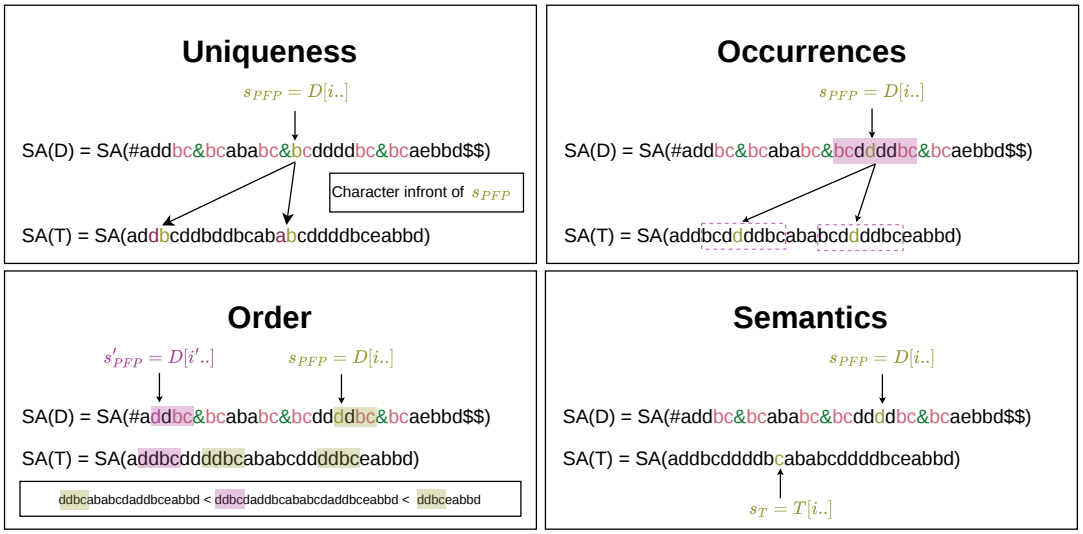


Figure 4.2: Differences between a suffix of SA_D and a suffix of the SA of T.

Figure 4.2 shows the difference using the known example. **Uniqueness**: In case s_{PFP} points to the first character of a phrase `bcdddbc`, the preceding character is not unique, because the phrase appears twice in T with different preceding characters. **Occurrences**: If s_{PFP} maps to the phrase phrase `bcdddbc` the related suffix in T appears twice. **Order**:

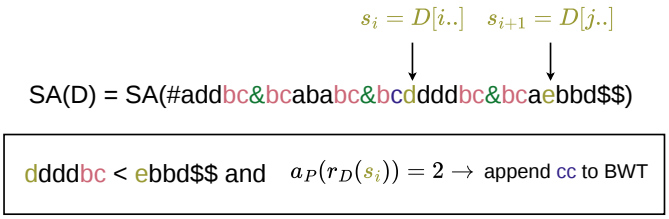


Figure 4.3: Example of suffix s_i being an easy case.

With s_{PFP} and s'_{PFP} both pointing to the phrase suffix `ddbc`, their order in T is not ambiguous. **Semantics**: s_T points to a suffix of T while s_{PFP} points to a phrase in D possible effected by the above cases.

In the SA-based BWT construction, the computed SA is iterated, and $T[SA[i]-1]$ is emitted for each i. In general, the PFP-based BWT construction follows this approach, but instead SA_D is iterated. Hence, the different behavior of the described suffixes needs to be considered. Suffixes s' with $|s'|_D \leq w$ are skipped since they lie within the w overlap of a phrase.

A suffix $s \in SA_D$ can behave in one of three different ways. Given the LCP_D of D, the following cases arise:

(i) ***First-character case*** s_i is a proper prefix of a phrase $d \in D$. In other words $s_D(s) \in D$.

(ii) ***Easy case*** s_i is not a first-character case and $|s_i|_D > LCP_D[s_{i+1}]$. This indicates that the order between s_i and s_{i+1} is unambiguous, with $s_D(s) < s_D(s_{i+1})$.

(iii) ***Hard case*** s_i is not a first-character case and $|s_i|_D \leq LCP_D[s_{i+1}]$. In this case, the order between s_i and s_{i+1} is ambiguous.

In the following, the different cases are reviewed in detail. It is shown how characters are emitted to the BWT.

Easy Case. A suffix s_i is an easy case if it is not a first-char case and $|s_i|_D > LCP_D[s_{i+1}]$. This indicates that the global suffix order for this suffix is unambiguous. In this case, the character preceding $s_D(s_i)$ in $r_D(s_i)$ will be emitted $n = a_P(r_D(s_i))$ times.

Figure 4.3 shows a suffix s_i that is an easy case in the running example. The phrase-suffix s_i maps to is strictly smaller than the phrase-suffix of the next suffix s_{i+1}. Therefore, the character in front of s_i will be emitted to the BWT. The phrase s_i points to appears twice in the parse. Hence, the preceding character `c` is appended to the BWT twice.

Hard Case. A suffix s_i is a hard case, if $|s_i|_D \leq LCP_D[s_{i+1}]$. This means that $s_D(s_i) = s_D(s_{i+1})$, e.g., two subsequent suffixes map to the same phrase suffix. A hard case always

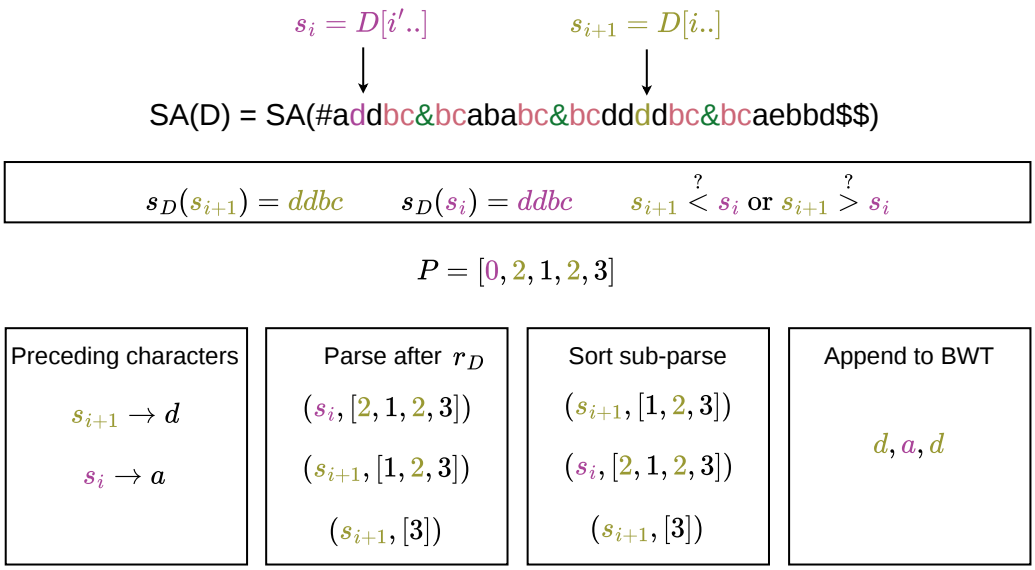


Figure 4.4: Example of two suffixes s_i and s_{i+1} that are hard cases.

consists of at least two suffixes. There can be up to n suffixes in a hard case, if for all $n-1$ suffixes following s $|s_{i+n}|_D \leq LCP_D[s_{i+n+1}]$ holds. In the hard case, the order of suffixes is unknown and possibly ambiguous. Their order depends on what comes after the corresponding phrases in the parse.

Using the known example, Figure 4.4 shows a hard case and how to solve it. The phrase suffixes of s_i and s_{i+1} both map to the same phrase suffix `ddbc`. Therefore, it is not possible to determine the order of the suffixes with regard to T by only considering the phrase suffixes. The point where s_i and s_{i+1} differ comes after their respective phrases. The parse encodes this information. The sub-parses, which are the content of the parses after $r_D(s_i)$ and $r_D(s_{i+1})$, are extracted and sorted. The ranks in the parse are the sorted phrases, so a phrase with a smaller rank is smaller than a phrase with a higher rank. The first index at which the sub-parses differ indicates the order of the respective suffixes. In this example, the first occurrence of the phrase s_{i+1} maps to is succeeded by the phrase with rank 1 in T. Therefore, this occurrence is the smallest considering T. After sorting all sub-parses, the order of the suffixes is known, and the preceding character of each suffix is appended to the BWT in that order.

Computing the sub-parses and sorting all of them for each hard case is very time- and memory-consuming. While in the above example the different sub-parses are sorted, this sorting step can also be viewed as suffix sorting of the parse. This makes it desirable to compute the SA of the parse. The SA does not provide all the needed information. It is made up of indices, so the indices at which each of the phrases appears need to be known.

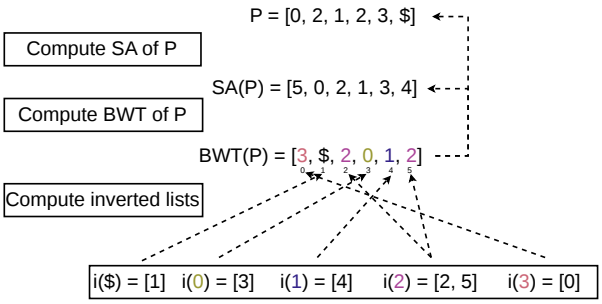


Figure 4.5: Example of the computation of the inverted lists.

Furthermore, the order of the suffixes excluding the phrase itself is needed. Even if the index i of a phrase is known, $SA[i]$ is a suffix beginning with the phrase i, but the order considering the subsequent phrases is needed. This information is encoded in the BWT.

Revisiting the BWT definition (see Definition 2.1.1) with the parse P instead of the general T yields $BWT[i] = P[SA[i] - 1]$. For a phrase at position i in the BWT, that indicates the phrase's subsequent sub-parse of phrases is smaller than any phrase at a position $j > i$. This is the information needed to compute the order of hard case suffixes. To get the order of two or more suffixes in a hard case, iterate over the BWT and collect these phrases. The order in which the phrases are seen in the BWT is the same as the introduced sorted sub-parse order.

To optimize computation and provide a fast lookup, the *inverted list* is computed once instead of iterating the BWT for each hard case.

The inverted list for a phrase d contains all ascending BWT positions where d occurs. Figure 4.5 shows how the inverted lists are computed for our example. The SA of P is computed, and the SA-based approach is used to compute the BWT of P. The inverted list $i(d)$ of a phrase d is then obtained by extracting the indices of all occurrences of d in the BWT in ascending order. The inverted list $i(\$)$ is included for completeness; it will be ignored in the algorithm.

After computing the BWT of P and the inverted list for all phrases, solving hard cases becomes a merging problem. Figure 4.6 shows the same hard case as seen in the previous example in Figure 4.4. For each suffix in the hard case, its corresponding phrase $r_D(s)$ and the corresponding inverted list $i(d)$ are fetched. Then all entries of the inverted lists are placed on a binary min-heap, and the top (smallest) elements are popped until the heap is empty. Each popped element corresponds to a phrase, which in turn corresponds to a character. For each popped element, this character is emitted into the BWT.

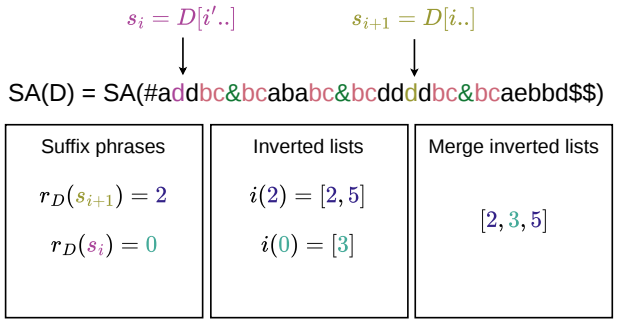


Figure 4.6: Example of solving a hard case using inverted lists.

First-character Case. A suffix $s \in SA_D$ is a first-character case if it is the proper prefix of an element in D. This means that s points to the first character of a phrase $d \in D$. A suffix s can be a proper prefix of at most one phrase $d \in D$.

The character preceding s is the last character of the phrase preceding the corresponding d in T. This information can be extracted from the parse. Given the rank of the phrase s maps to $(r_D(s))$, the parse shows where this phrase occurs in T.

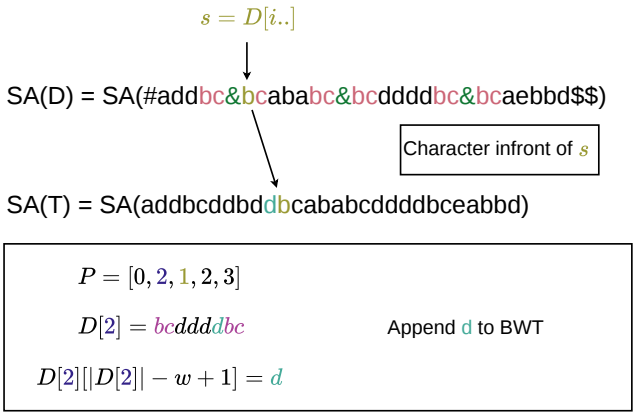


Figure 4.7: Example of a First-character case with a phrase appearing only once in the parse.

If $a_D(r_D(s)) = 1$, the phrase occurs only once in the parse. An example for this case is shown in Figure 4.7. In this case, with $D[2]$ being the phrase preceding $d = r_D(s)$ in the parse, the $w + 1$ last character of $D[2]$ is appended to the BWT. The $w + 1$-last character is emitted because the w-suffix of $D[2]$ overlaps with the w-prefix of d. .

The PFP approach is effective if the input is repetitive and most phrases appear multiple times in the parse. For each of these appearances, the preceding phrase in the parse can be

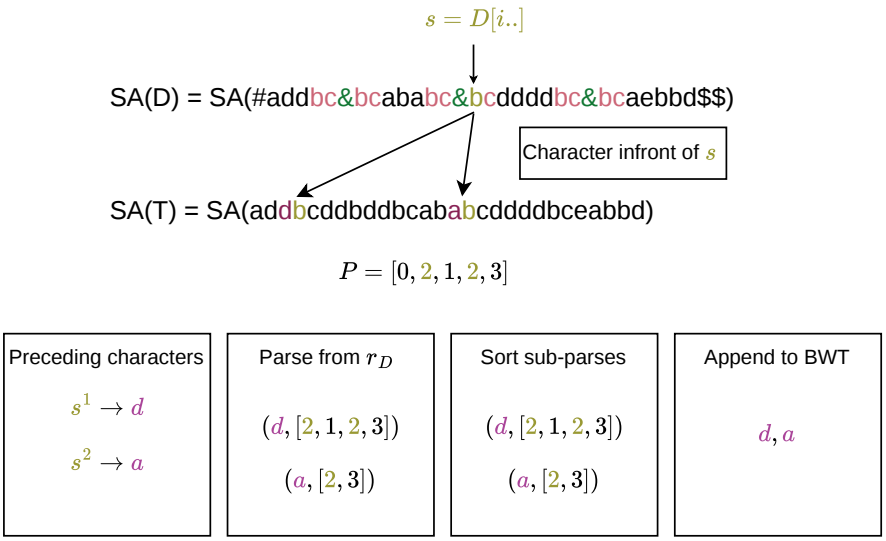


Figure 4.8: Example of a First-character case with a phrase appearing multiple times in the parse.

different. So the correct order in which the $w + 1$ last characters of each preceding phrase are emitted to the BWT needs to be computed. The global suffix ordering must be inferred from the smaller SA_D ordering.

An example of the first-character case of a phrase appearing multiple times is shown in Figure 4.8. Phrase 2 appears twice in the parse, with different preceding phrases. To solve this case, the $w + 1$ last character of each preceding phrase is fetched. For the first occurrence of phrase 2, the $w+1$ last character of the preceding phrase is `d`. For the second appearance, the preceding character is `a`. To get the correct order of the appearances with regard to T, the sub-parses starting at phrase 2 are extracted and sorted. After sorting, the order of the appearances is known, and the corresponding characters are appended to the BWT in this order.

Materializing all sub-parse for all phrases and sorting them is not viable. Therefore, the BWT of the parse and a helping structure W are computed to directly obtain the preceding phrases in SA order for each first-character suffix.

The running example in Figure 4.9 shows how a first-character case is solved using the W list. First, the last character of each phrase is computed. Because of the w overlap, the actual last character is the $w + 1$-th character from the end, which is exactly preceding a suffix pointing to the first character of a phrase. A list L of last characters according to P is created with $L[i]$ being the last character of phrase $P[i]$. To obtain the W list, L and the SA of P are used with $W[i] = L[SA[i] - 2]$. The case $SA[i] = 0$ is ignored as it points to the end-of-file symbol. For $W[i]$ with $SA[i] = 1$, $L[|L| - 1]$ is stored. $SA[i] - 1$ is the phrase the suffix points to, so $SA[i] - 2$ is the phrase preceding it. This maps the

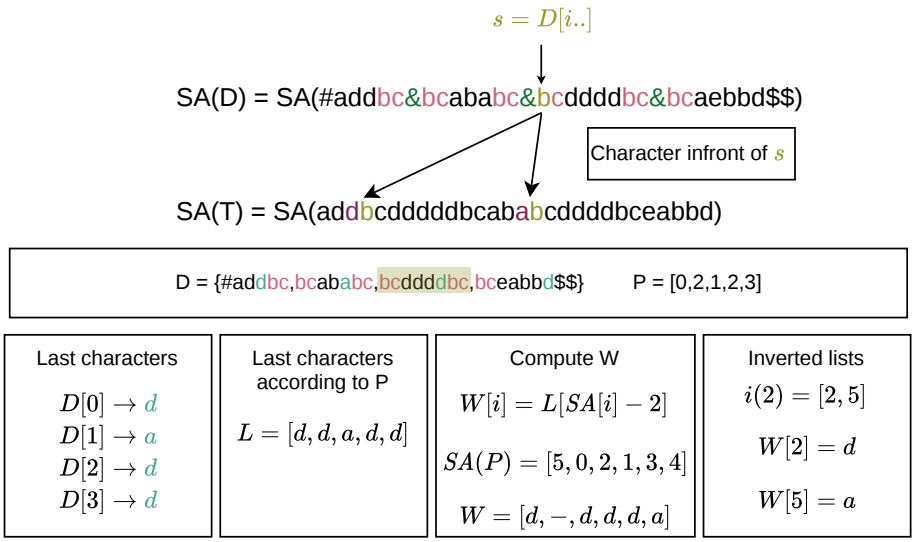


Figure 4.9: First-character case solved by using the W list that contains the last characters in SA order.

entries of W to the last characters of the phrases in BWT-order. To retrieve the characters in the correct order for a phrase d, their inverted list $i(d)$ is used, and the characters at $W[i(d)[0]], W[i(d)[1]], \ldots$ are appended to the BWT.

The full Picture. A high-level overview of the PFP-based BWT construction is given in Algorithm 4. Assuming that P, D, the SA of P and D, and the LCP array of D have already been computed. The following steps are needed to construct the BWT based on PFP.

(i) Compute the BWT of P.

(ii) Compute the inverted lists.

(iii) Compute the first-character list W.

(iv) The main loop iterating over $s_i \in SA_D$.

a) Solve first-character cases by appending characters from W indexed by the inverted list of phrase $r_D(s_i)$.

b) Solve easy cases by appending $T[s_i - 1]$ to the BWT $a_P(r_D(s_i))$ times.

c) Solve hard cases by merging the respective inverted lists and appending $T[s' - 1]$ to the BWT for each popped suffix s'.

Algorithm 4: PFP-based BWT construction

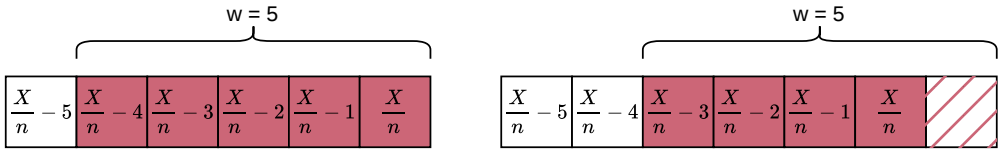
```
Data: D, P from PFP, SA_D, LCP_D, SA_P
Result: BWT of T
BWT_P ← BWT(P, SA_P)
inv ← inv(BWT_P)
W ← permute_w(D, BWT_P)
pos ← 0
for i ← 0 to |SA_D| − 1 do
    s_i ← SA_D[i]
    if |s_i|_D < w then
        continue
    else if s_i ∈ D then                          /* s_i is a first-character case */
        for j : inv[r_D(s_i)] do
            BWT[pos] ← W[j]
            pos ← pos + 1
        end
    else if |s_i|_D > LCP_D[i + 1] then           /* s_i is an easy case */
        for n : a_P(r_D(s_i)) do
            BWT[pos] ← T[s_i − 1]
            pos ← pos + 1
        end
    else                                          /* s_i is part of a hard case */
        S ← [ s_i ]
        while |s_{i+1}|_D ≥ LCP_D[i + 2] do
            i ← i + 1
            s_i ← SA_D[i]
            append s_i to S
        end
        for s^merge : merge({ inv[r_D(s)] : s ∈ S }) do
            BWT[pos] ← T[s^merge − 1]
            pos ← pos + 1
        end
    end
end
```

4.2.2 Distributed Prefix-Free Parsing

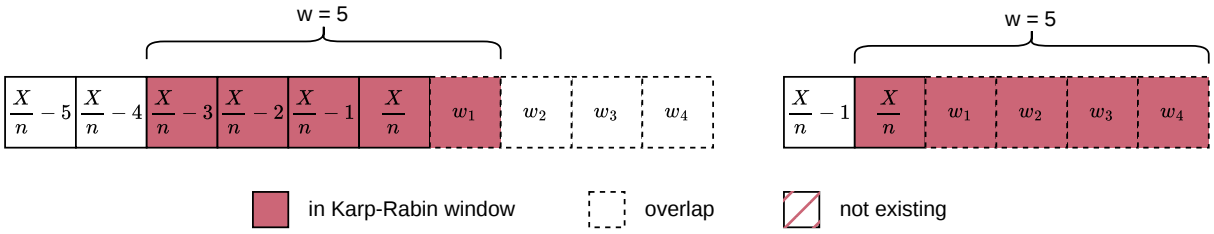
The first step of PFP is embarrassingly parallel. The input T is distributed on all available k PEs such that each PE holds

$$\ell_i = \left\lfloor \frac{X}{k} \right\rfloor + (w-1) + \begin{cases} 1 & \text{if } i < X \bmod k \\ 0 & \text{otherwise} \end{cases}$$

The input data overlaps with $w-1$ characters. In Figure 4.10, the need for this overlap is visualized. Without the overlap, the Karp-Rabin fingerprint for the last $w-1$ characters can not be computed, as there are not enough characters to fill the Karp-Rabin window; this is shown in Figure 4.10a. With the overlap, the fingerprint can be correctly computed for all $l_i - w - 1$ characters, because the overlap is used to fill the Karp-Rabin window, see Figure 4.10b.



(a) Karp-Rabin window without an overlap



(b) Karp-Rabin window with $w-1$ overlap

Figure 4.10: Karp-Rabin window of size 5 in case data is distributed without an overlap (a) and in case an overlap of size $w-1$ is used (b).

The distributed PFP algorithm is described below and visualized in Algorithm 5. The dictionary is computed on each PE by sliding a Karp-Rabin fingerprint window of size w over its assigned data and checking whether the fingerprint modulo p is zero. This results in a local dictionary D_i on each PE. For each phrase $d \in D_i$, the fingerprint of the whole phrase is stored. To get the ranks of each phrase, the local dictionaries D_i are globally sorted using the distributed string sorter described in Section 3.2. This results in each PE having a sorted chunk of the global dictionary. To ensure the phrases in the global

dictionary are unique, each PE locally deduplicates its local dictionary. This deduplication is done by fingerprinting each phrase in the parse and comparing its fingerprint with that of the next phrase. The fingerprint of each unique phrase is stored for the next step. To detect duplicates spanning across PE borders, the send-receive communication is used. Each PE i sends the fingerprint of its last phrase to the next PE $i + 1$. If the fingerprint received by PE $i + 1$ matches its first fingerprint, it deletes its local first phrase.

With the dictionary being globally deduplicated, an exclusive scan over the number of phrases on each PE is used to determine the global rank of each phrase locally. Using the previously stored fingerprints, rank-fingerprint pairs are created. At this point, each PE holds its chunk of the globally unique and sorted phrases, rank-fingerprint pairs corresponding to the sorted phrases, and the fingerprints of the phrases in the local unsorted dictionary.

To determine the global parse, each PE needs to know the global rank of the phrases it found locally. For this, the previously stored rank-hash pairs are globally sorted by fingerprint. Each PE now holds pairs with an increasing fingerprint value. Using an all-gather, each PE sends its local first hash to all other PEs. By using a binary search on the all-gathered fingerprints, a PE can determine which PE holds the rank corresponding to a local fingerprint. When each PE has computed the PE that holds the rank for its local hashes, the request-response pattern introduced in Section 3.4 is used to retrieve the phrase ranks for all local phrases. After this step, the parse is computed and already globally distributed across all PEs.

There is a chance that the fingerprints computed for two different phrases are the same. In case both phrases are on the same PE after the global sort, this would result in one of the two phrases being removed in the deduplication step, resulting in a wrong parse. If the colliding phrases are on two different PEs, the collision can be detected after globally sorting the hashes, as a single PE would hold the same fingerprint twice. In this case, the collision can be resolved by, for example, recomputing the fingerprint of the two phrases using a different prime number. The BIG-BWT authors argue that by using 64-bit fingerprints, the chance of a collision is very low and therefore do not handle collisions [9].

4.2.3 Distributed BWT from Prefix-Free Parsing

As in the sequential approach, the first step in the BWT construction based on PFP is to compute the SA and LCP arrays of the dictionary D and the SA of the parse P. In the distributed case, each PE holds a chunk of P and D, and after computing the SA_D, SA_P and the LCP array of D, each PE holds a chunk of both SAs and the LCP array.

Next, the number of occurrences of each phrase needs to be computed. This is what the function $a_P(d)$ models. For this, each PE locally computes the number of occurrences. Using a reduce operation, the local counts are globally summed. Another required data structure is the prefix sum of the lengths of all phrases in D on each PE. The prefix sum is needed to compute the phrase to which a given suffix belongs. This is what $r_D(s)$ models.

Algorithm 5: Distributed PFP Computation

`Data`: Input chunk T_i, window size w, split modulus p
`Result`: Local chunk of PFP of T

```
1  (D, F_D) ← KarpRabin(T_i, w, p)                    /* local dictionary and fingerprints */
2  D^sort ← globalSort(D)
3  D^uniq ← []; F ← []
4  for i ← 0 to |D^sort| − 1 do                        /* deduplicate D^sort */
5  |  f_i ← fingerprint(D^sort[i])
6  |  if i = 0 or f_i ≠ f_{i−1} then
7  |  |  append D^sort[i] to D^uniq
8  |  |  append f_i to F
9  |
10 end
11 borderCheck(D^uniq, F[|F| − 1])                   /* remove duplicate across PE boundaries */
12 D^sort ← D^uniq
13 offset ← exScan(|F|)
14 pairs ← []
15 for i ← 0 to |F| − 1 do
16 |  append (offset + i, F[i]) to pairs
17 end
18 P^sort ← globalSort(pairs)                        /* fingerprint is sort key */
19 F^last ← allgather(P^sort.last.fingerprint)
20 requests ← []
21 for i ← 0 to |F_D| − 1 do
22 |  append binarySearchOwner(F^last, F_D[i]) to requests
23 end
24 parse ←requestResponse(requests, P^sort)
25 return parse, D^sort
```

Both the number of occurrences and the prefix sum of the phrase length need to be accessed for every $s \in SA_D$ a PE has to process. Therefore, these data structures are replicated on each PE. In Section 5.1.1 it is shown that these data structures can be stored once per node, weakening the requirement to store a copy on every PE.

Inverted Lists. In the distributed approach, it is desirable to have the inverted lists (see Section 4.2.1) distributed across all PEs. The BWT of P is computed globally using the distributed SA of P as introduced in Section 4.1. For each phrase in the local BWT, the corresponding index is stored. These local BWT indices create a partially local inverted list for each phrase that is present in the local BWT chunk. Inside this partially local inverted list, the ordering is already correct. The local partial inverted lists are globally distributed

using the all-to-all-v collective. Each PE is assigned a range of phrases for which it will store the inverted list. The inverted list of a phrase d has exactly as many entries as the number of times d has appeared in the parse ($a_P(d)$). Hence, the assigned phrases need to take $a_P(d)$ into account. The following paragraph introduces the partitioning used to balance the inverted list across all PEs.

SA-based Partitioning. Balancing is needed not only for the workload of each PE but also for the data stored by each PE. Both the inverted lists and the W list used for the first-character case grow with $O(P)$. It is not desirable to split those two data structures by size so that each PE holds an equally sized part of each list. Splitting the inverted lists just by size would most certainly result in an inverted list being split across PEs. Which, in turn, would add communication costs whenever this particular inverted list is required, since two PEs own a chunk of it.

The inverted lists and W are both structures with $|D|$ entries for each phrase d_i. Each entry is of size $a_P(d_i)$. Instead of load balancing solely by total size, it is balanced based on both total size and per-entry size. Building on this idea, the following load-balancing pattern for inverted lists is introduced. The same pattern applies for both the inverted lists and the W list.

Given phrases $D = \{d_0, d_1, \ldots, d_{m-1}\}$ with each phrase d_i occurring $a_P(d_i)$ times in the parse. The total size of all inverted lists is denoted by $\sum_i a_P(d_i)$. With n PEs, the desired amount of inverted list entries per PE is $\tau = \frac{\sum_i a_P(d_i)}{n}$

An inverted list should not be split across PEs but should live entirely on a single PE. Therefore, a linear partitioning is used, assigning each PE at least τ elements without splitting inside an inverted list. Let b_i be the number of inverted list entries on PE i and let $P(k) = \sum_{j=0}^{k-1} a_P(d_i)$ be the inclusive prefix sum over the inverted list sizes for the first k phrases. Using a greedy approach, the inverted lists are split after the z_i-th individual inverted list:

$$z_0 = \min\{k \in [0, m] : P(k) \geq \tau\}$$
$$z_{i+1} = \min\{k \in (z_i, m] : P(k) - P(z_i) \geq \tau\}$$

yielding

$$b_i = P(z_{i+1}) - P(z_i)$$

Because each PE has access to the number of occurrences of each phrase, it is possible to compute the same partitioning locally on each PE. Each PE sends its partial inverted list for phrase d_i to the PE that owns this phrase, according to the computed partition. For this, an all-to-all-v is used. The all-to-all-v collective returns the received data in PE order. The PEs also hold their chunk of the BWT in order. Therefore, the order of each inverted list received by a PE is already correct.

Distributed W Permutation. The list W, which contains the last characters of each phrase, needs to be permuted and stored in a distributed manner. The prerequisite is that the distributed SA of P and P itself are known. The following paragraphs describe the individual steps of this algorithm and analyze their complexity, annotated with the corresponding lines of Algorithm 6.

1. Fetch remote SA Entries. In the first step, each PE fetches the phrase $P[SA_P[i]]$ and the preceding phrase $P[SA_P[i]-1]$ that each local suffix in SA_P points to. This is done using the request-response pattern presented in Section 3.4. In the following, $P[SA_P[i]]$ is called the base phrase. This step corresponds to the lines 1 through 10.

Complexity: For each entry of the local SA_P chunks, the PE(s) owning the parse index $SA_P[i]$ and $SA_P[i]-1$ need to be computed. Let SA_P^i be the SA_P chunk on PE i with $SA_P^{max} = \max_i(|SA_P^i|)$ and k the number of PEs, then the complexity to find the owners for all requests is: $O(SA_P^{max} \log k)$. Adding the request-response with ϕ being an array lookup in $O(1)$. The complexity of this step is $O(\alpha k + \beta h + SA_P^{max} \log k)$. With h being the maximum number of requests or responses sent by a PE. The maximum number of requests a PE can send is $|SA_p^i|$, and the maximum number of responses a PE can return is $|P^i|$. Because $|SA_P| = |P|$ and assuming SA_P and P is equally distributed, we get $h = SA_P^{max}$. In total this boils down to $O(\alpha k + \beta SA_P^{max} + SA_P^{max} \log k)$

2. Create local Triples. An exclusive scan over $|SA_P^i|$ is used to determine the offset of $|SA_P^i|$ for each PE. PE 0 has offset 0, PE 1 has an offset of $|SA_P^0|$, etc. Using this offset, each PE iterates over its local SA_P^i chunk and creates triples of the form $\langle \text{offset} + j, P[SA_P^i[j]], P[SA_P^i[j]-1] \rangle$ using the $P[...]$ fetched in the previous step. This step corresponds to the lines 11 through 15.

Complexity: The exclusive scan has an complexity of $O(\alpha+\beta \log k)$, because each message is the size of SA_P^i which is one data type. Creating the triples takes $O(SA_P^{max})$, resulting in a total complexity of $O(\alpha + \beta \log k + SA_P^{max})$.

3. All-to-all-v Triples. The $\langle \text{offset} + i, P[SA_P[i]], P[SA_P[i]-1] \rangle$ triples are routed to the PE owning the base phrase ($P[SA_P[i]]$) using an all-to-all-v collective. To balance the resulting W map, the SA-based partitioning introduced in Section 4.2.3 is used. W shows the same behavior as the inverted lists, since each phrase appears exactly as many times as it occurs in P entries in W. This step corresponds to the lines 17 through 21.

Complexity: This steps complexity is defined by the all-to-all-v collective with $O(\alpha k + \beta SA_P^{max})$.

4. Sort Triples locally. After receiving all triples containing a base phrase that is part of the PEs partition, the triples are grouped by base phrase. Then each group is locally sorted according to the first component (offset $+ i$). This sorting step ensures that all triples

of a group are in SA_P order, which is the order needed to emit the characters to the BWT in the correct order. This step corresponds to the lines 22 through 26.

Complexity: Sorting the triples can be done in $O(t \log t)$ time, where t is the maximum number of triples on a single PE. Assuming the partition introduced in Section 4.2.3 is balanced, each PE has the same number of local triples. The total number of triples equals the number of elements in the parse. Therefore, $t = SA_P^{max}$ can be assumed.

5. Fetch Characters. To ease access to W and make the subsequent steps of the PFP-based BWT construction more convenient, the actual characters that the triples point to are fetched. At this point, each group is sorted according to SA_P and contains the preceding phrases ($P[SA_P[i] - 1]$) for each entry. The array containing the last character of each phrase is distributed across all PEs. For all local triples, the last character of the phrase $P[SA_P[i]-1]$ is requested using the regular request-response pattern. This step corresponds to the lines 27 through 31.

Complexity: The character look-up itself can be done in $O(1)$. Using the same assumption as in step four, the maximum number of triples on a single PE is SA_P^{max}. Then $O(SA_P^{max} \log k)$ time is needed to compute the PEs that own the characters. With the request-response pattern, this step results in complexity $O(\alpha k + \beta SA_P^{max} + SA_P^{max} \log k)$.

6. Store Characters in W. The last step is to store the previously fetched characters in a phrase-to-array of last characters mapping. This is done by replacing the preceding phrase ($P[SA_P[i] - 1]$) with the actual character and storing it in the W map. This step corresponds to the lines 32 through 37.

Complexity: This is a $O(1)$ pass over all triples. With the previous assumption on the number of triples, this step takes $O(SA_P^{max})$.

Total Complexity. Combining the complexity of the above steps, we get the complexity of the whole distributed W permutation with:

$$\begin{aligned} &O(\alpha k + \beta SA_P^{max} + SA_P^{max} \log k + \\ &\quad \alpha + \beta \log k + SA_P^{max} + \\ &\quad SA_P^{max} \log SA_P^{max}) \end{aligned}$$

Because for all sufficiently large inputs $k << SA_P^{max}$:

Lemma 4.2.7. *The complexity of the distributed W permutation assuming equally balanced P, SA_P and partition 4.2.3 is:*

$$O(\alpha k + \beta SA_P^{max} + SA_P^{max} \log SA_P^{max})$$

Algorithm 6: Distributed W Computation

```
Data: SA_P, P, last-char array L
Result: Map W: phrase ↦ vector of chars (SA order)
requests ← []
for i ← 0 to |SA_P| − 1 do
    append SA_P[i] to requests
    if SA_P[i] = 0 then
        append |P|_global − 1 to requests
    else
        append SA_P[i] − 1 to requests
    end
end
resp ← requestResponse(requests, P)
offset ← exScan(|SA_P|)
triples ← []
for i ← 0 to |SA_P| − 1 do
    append (offset + i, resp[2i], resp[2i + 1]) to triples
end

dest ← []
for i ← 0 to |triples| − 1 do
    append binarySearchOwner(triples[i].cur) to dest
end
triples ← allToAllv(triples, dest)
groups ← {}
foreach t ∈ triples do
    append (t.idx, t.prev) to groups[t.cur]
end
foreach g ∈ groups do sort g by idx
req ← []
foreach g ∈ groups do
    foreach (idx, prev) ∈ g do append prev to req
end
chars ← requestResponse(req, L)
W ← {}; k ← 0
foreach (ph, g) ∈ groups do
    foreach (idx, prev) ∈ g do
        append chars[k] to W[ph]; k ← k + 1
    end
end

return W
```

Main BWT Loop. In the main BWT loop, the local SA_D^i chunk is iterated. Each suffix is classified into a case and solved accordingly.

First-character Case. The first-character cases are handled first. With a first pass over SA_D^i, the suffixes that are in a first-character case and their respective phrases are determined. Using the irregular request-response pattern, the W entries for the local first-character phrases are fetched from the owning PEs. If, in the main loop, a first-character suffix is reached, the characters from the W entry are emitted to the BWT at the current position.

Easy Case. For each easy case suffix s_i, the character at $D[s_i - 1]$ needs to be emitted $a_P(r_D(s_i))$ times. In most cases, $D[s_i - 1]$ will not reside on this PE, so this character must be fetched remotely. If $D[s_i - 1]$ resides on this PE, the character is written directly into the BWT $a_P(r_D(s_i))$ times at the current position, and the position is adjusted accordingly. If $D[s_i - 1]$ does not reside on the current PE, some bookkeeping data for the case is stored, and it will be resolved once the main loop is passed. The data stored is the suffix s_i in question, the phrase $r_D(s_i)$, and the current BWT position. The position itself is increased by $a_P(r_D(s_i))$, so it points to where the next suffix case should be placed in the BWT. After the main loop, the corresponding character $D[s_i - 1]$ will be fetched for all easy cases at once. This is done by binary searching the PE that owns the D chunk containing $D[s_i - 1]$ and using the regular request-response pattern to obtain the remote character $D[s_i - 1]$. Each obtained character is written at its previously stored position in the BWT, $a_P(r_D(s_i))$ many times.

Hard Case. A hard case h contains multiple suffixes $[s_1^h, s_2^h, \ldots, s_n^h]$. In the sequential algorithm, the inverted list of each suffix $l(s_i^h)$ is inserted into a min-heap. Popping from the heap yields a BWT index given by l. This BWT index is mapped back to the corresponding phrase d_x. Given the suffix s_x with $r_D(s_x) = d_x$, its preceding character $D[s_x - 1]$ is emitted to the BWT $a_P(d_x)$ times.

In the distributed case, the preceding character $D[s_i - 1]$ for each s_i appearing in a local hard case and the inverted lists $l(d_i)$ for all $r_D(s_i)$ the local hard case suffixes point to are not necessarily known locally on each PE. The missing characters are fetched after the main loop, using the same request-response approach as for the easy cases. To fetch the non-local inverted lists, the irregular request-response pattern is used, because each inverted list $l(d_i)$ contains multiple elements depending on $a_P(d_i)$. The inverted lists are distributed according to the partition described in 4.2.3. The requests are computed by retrieving the PE that owns the required inverted list. Each request receives a copy of the requested inverted list. ´

After this communication, each PE locally stores $D[s_i - 1]$ and $l(r_D(s_i))$ for each suffix in the local hard cases. Each inverted list contains monotonically increasing values.

Therefore, a *Loser Tree* [36] is used to merge the inverted lists.

We refrain from detailing the complexity of the entire distributed PFP-based BWT construction algorithm. At many points in the algorithm, the runtime depends on the compression achieved by PFP and the size of the resulting intermediate data structures, such as the parse or the dictionary. It is not possible to determine sensible bounds for the sizes of these data structures, because the worst case will always be an input where no split is found, resulting in a single phrase in the dictionary. This would yield a theoretically correct runtime but would not account for the fact that this algorithm is designed for highly repetitive inputs, nor would it provide further insights into the algorithm's actual theoretical performance.

4.2.4 Optimizations

We apply multiple optimizations to the distributed PFP-based BT construction algorithms described in Section 4.2.3.

Inverted Map Bucketing. Strictly following the previously described approach to solve the hard cases can lead to heavily unbalanced communication. The inverted lists are balanced by the size of the individual inverted lists. The number of appearances of a phrase d_i equals the number of elements in its inverted list. But the size of an inverted list does not correlate with how often a suffix of its phrase appears in a hard case. There could be long phrases without a hard case and short phrases with each suffix being part of a hard case. This behavior can lead to a highly above-average workload for a PE who owns many hot inverted lists that are requested frequently. In the worst case, a single large inverted list is requested multiple times by different PEs. To send an inverted list l from one PE i to multiple other PEs j, it needs to create a message m_{ij}^l for each PE j that requests l. All of these messages m_{ij}^l will have the same content, leading to high memory consumption and longer computation of the complete response from PE i.

The number of times an inverted list is replicated is upper-bounded. In the worst case, every PE requests every inverted list on PE i. For n PEs to request the same inverted list, the corresponding phrase needs to have at least $n+1$ characters (excluding the w overlap). The total size of the inverted lists on a single PE is, in case they are perfectly balanced:

$$\frac{\sum_i a_P(d_i)}{n}$$

If every PE requests all local inverted lists, it must replicate each local inverted list k times, leading to a worst-case memory usage of $O(k\frac{\sum_i a_P(d_i)}{n})$ for the responding PE.

Request Bucketing

To prevent this memory- and compute-intensive peak, bucketing is used. Instead of a single irregular request-response pattern, there are b_{req} rounds of the irregular request-response

pattern. Each PE computes the total size of the inverted lists it needs to fetch from remote PEs. A PE can compute this because they have access to the phrase occurrence list. In the following, R_i denotes the requests from PE i. Instead of all-to-allv-ing all requests at once, PE i splits R_i into b_{req} equally sized chunks $[R_i^1, R_i^2, \ldots, R_i^{b_{req}}]$. In the j-th round of request-responses, PE i uses the request chunk R_i^j. Using the responses, it can solve all cases corresponding to the requests R_i^j. Splitting the requests of a hard case would render it impossible to solve. Therefore, PE i needs to ensure that the request chunks R_i^j do not split a hard case but end with a complete hard case.

Response Bucketing

The same bucketing idea is applied to the response side. This reduces the workload on PEs that hold hot inverted lists requested multiple times per request-response round. Instead of computing and sending all responses at once, they are returned in b_{resp} buckets. To aid the PEs receiving the responses, the total size of the responses is precomputed. This can be achieved by using a size function σ along the response function ϕ that, for a given request r, returns the size of ϕ: $\sigma = |\phi(r)|$. Let P_i denote the responses sent by PE i. Following the request bucketing approach, P_i is split into equal-sized buckets $[R_i^1, \ldots, R_i^{b_{resp}}]$. Because the PEs receiving the responses need to assemble the full response from each bucket, responses can be split by size. This reduces the above-stated upper bound for the responding PE by a factor of b_{resp}. With response bucketing, a single PE that receives requests for each local inverted list from all other PEs will use worst-case memory of

$$O(\frac{k \sum_i a_P(d_i)}{n b_{resp}})$$

Load Balancing the SA. For efficient distributed computation of the BWT, the workload must be balanced across all PEs. Previously, it was shown how the important data structures are distributed. But the actual work itself needs to be distributed as well. Much of the work is done in the main loop, iterating over the local SA_D^i chunk and then solving the suffix cases. Splitting SA_D into equally sized chunks will not balance the work. There are two reasons why this approach would not balance the work. First, a single suffix $s_i \in SA_D^j$ can correspond to any number of characters being written in the BWT. Using equally sized SA_D^j would lead to an imbalance in the corresponding BWT chunks BWT^j. The other reason for imbalance is that a suffix can imply a different amount of work needed. A suffix s could be an easy case, with $D[s-1]$ being local, or it could be part of a hard case in which a large inverted list must be fetched remotely. To achieve satisfactory load balancing, both of these possibilities need to be considered. Hence, the following balancing based on the number of occurrences of each phrase is proposed.

Each PE computes the cost of the suffixes in its local SA chunk SA_D^i. To compute the cost, each suffix is multiplied by the number of times the corresponding phrase appears in the parse, $a_P(r_D(s))$. The resulting product is weighted according to the case the suffix

belongs to, using the three weights w_F, w_E, w_H for the first-character case, the easy case, and the hard case. This leads to the following local cost c_i:

$$c_i(SA_D^i) = \sum_{j=0}^{|SA_D^i|-1} a_P(r_D(SA_D^i[j]))w(SA_D^i[j])$$

$$w(s) = \begin{cases} w_F, s \text{ First-character case} \\ w_H, s \text{ Hard case} \\ w_E, s \text{ Easy case} \end{cases}$$

With the local cost c_i of each PE, the total cost $c = \sum_i c_i$ can be all-reduced. Using the number of PEs k, the desired balanced cost per PE is $c_{target} = c/k$. The target cost c_{target} is used to cost-balance SA_D as follows. An exclusive prefix sum of c_i is used to obtain the preceding cost. PE i iterates over its local SA_D^i and computes the weighted per-suffix cost. Based on c_{target} and the exclusive prefix sum, PE i can determine which PE j each suffix should be assigned to achieve the correct cost balance. To ensure that each PE can solve its local suffix cases, the balancing will move suffixes from the same hard case to the same PE.

As hard cases are not split across PEs and because there will be phrases with $a_P(D) > 1$, the resulting cost balance will be slightly skewed.

5 Experimental Evaluation

In this chapter, the performance and behavior of the distributed BWT construction based on PFP are analyzed. This approach is compared to existing BWT construction implementations. In Section 5.1, an overview of the implementation is given. Furthermore, important implementation details are highlighted. The datasets and experimental settings used for the evaluation are shown in Section 5.2. The experiments and their results are presented in Section 5.3.

5.1 Implementation

The distributed SA-based and the PFP-based BWT construction algorithms have been implemented as part of this work. This implementation is publicly available [1]. For a convenient MPI implementation, the (near) zero-overhead C++ wrapper KaMPIng has been used [58]. For the SA-based approach, the distributed DCX algorithm [29] is used to compute the SA of the input. The PFP-based approach uses PSAC [19] for the SA and LCP computation of the dictionary and the distributed DCX algorithm [29] to compute the SA of the parse. Scalable Distributed String Sorting [37] is used for the initial distributed sorting of all phrases in the dictionary. AMSSORT [2] provided by the Karlsruhe Distributed Sorting Library (KaDiS) is used to globally sort the fingerprints during the PFP computation. For generic local sorting, IN-PLACE PARALLEL SUPER SCALAR SAMPLESORT (IPS^{4}o) is used [4]. The shared memory string sorter and the Loser Tree implementation provided by the tlx collection have been used [7]. The correctness has been verified by computing the BWT using the shared-memory BIG-BWT implementation on the same input and comparing the resulting BWTs character by character.

5.1.1 Implementation Details

Difference encoded Inverted Lists. Solving the hard cases using inverted lists can require a large amount of memory. This is especially the case if a single PE receives requests from multiple other PEs for the same inverted list. In this case, the responding PE needs to replicate the inverted list for each requesting PE in its send buffer. An algorithmic optimization to reduce this memory footprint by using bucketing is given in Section 4.2.4.

[1] `https://github.com/dabrommer/distributed-prefix-free-parsing`

Another approach to reduce the actual memory usage is to optimize the way an inverted list is stored. Each inverted list consists of five-byte entries. Inside an inverted list, the values are monotonically increasing. Hence, a difference encoding is used to reduce the total size of an inverted list. This encoding stores the first absolute value of an inverted list and, for subsequent entries, only stores the difference between each entry and the previous one. The memory gained from the encoding is that differences can be stored in at most five bytes, but are usually stored in fewer than five bytes. The following experiments will show that, on average, only about one byte is used per inverted list entry.

Sharing replicated Vectors. As described in Section 4.2.3, the list containing the number of times each phrase occurs and the dictionary-length prefix sum are needed on all PEs and need to be accessed multiple times at different points during the algorithm's execution. Distributing those two structures and requesting access only when needed from a remote PE is not feasible due to the overhead it incurs. Both structures, the phrase prefix list and the phrase occurrence list, are of size $|D|$. The size of an entry in the prefix list depends on the size of the dictionary D extracted by PFP. It will use the smallest data type that can fit $|D|$, so it is either 16-bit, 32-bit, or 64-bit. For the occurrence list, the used data type is 32-bit. [2]

To prevent each PE from storing $4 \cdot |D|$ bytes for the occurrence list and between $4 \cdot |D|$ and $8 \cdot |D|$ bytes for the prefix sum, these lists can be stored once per node (or once per NUMA node) using MPI windows. First, a communicator per (NUMA) node is created, and the root of each communicator allocates a shared memory window using *MPI_WIN_ALLOCATE_SHARED*. Using *MPI_WIN_SHARED_QUERY*, the non-root PEs in each communicator obtain a pointer to the base of the allocated shared MPI window.

For both lists, the way they are computed changes slightly when using the MPI shared memory windows. In the case of the phrase prefix sum, each PE computes its local prefix sum, and the PEs' global phrase offset is computed using an exclusive scan. Then the local prefix sum is copied into the shared window, accounting for the offsets. Lastly, an all-gather-v is performed between the roots of each (NUMA) node communicator to place the complete phrase prefix sum into each shared memory window.

For the shared phrase occurrence computation, each PE computes its local occurrences, which are then reduced within each communicator. After the node local all-reduce, the node's roots perform a second all-reduce to obtain the global phrase occurrences and store them in the MPI shared memory window.

Local Deduplication. Before the local dictionary D is globally sorted, each PE sorts its local dictionary D^i and removes duplicates. The remaining duplicates are removed from the global dictionary after sorting it. The local deduplication has two effects. First,

[2] As a side note, this marks the limits of inputs accepted by this implementation. The dictionary D cannot be larger than $2^{64} - 1$, and no single phrase can appear more than $2^{32} - 1$ times in the parse.

the global string-sorting volume is reduced, thereby improving performance. After global sorting, each PE receives a chunk D^i of the sorted D. Without the pre-sorting local deduplication, it could be that all phrases in D^i are the same, so after deduplication this PE would contain only one phrase. In contrast, a PE can have a chunk D^i in which all phrases differ, leading to the globally deduplicated D being stored in a highly unbalanced manner. With local deduplication before the global sort, a single phrase can appear in the global dict D at most k times, where k is the number of PEs. The worst case would be a chunk D^i that contains every phrase exactly k times, which would improve the balance of the deduplicated D compared to not using local deduplication.

Local deduplication is used before most uses of the request-response pattern to prevent duplicate requests. Therefore, the requested data is sorted, and duplicates are removed. Two examples of usage are: In the parse building, deduplication of the fingerprint used to retrieve the rank for a given phrase. If a phrase has been found multiple times locally, its rank only needs to be requested once. Deduplication of the requested inverted lists when solving hard cases, as phrases may appear in multiple hard cases needing the same inverted list.

Long Phrase Separation. Testing showed that long phrases can lead to highly unbalanced results in the distributed string sorting, with the chunks D^i of some PE being much larger than the average chunk size $(\sum_i D^i)/k$. Another issue with long phrases is that performance degradation is observed in distributed string sorting. This degradation is most likely related to large splitter strings being sorted. To address this issue, long phrases are removed locally from D^i before the global sort. Then the dictionary, without long phrases, is globally sorted, and the extracted phrases are sorted separately using the RQUICK algorithm [37]. After the global sort, the last regular phrase of each PE is all-gathered. Using the all-gathered last phrases, the long phrases can be routed to their owning PEs via binary search. A PE receiving long phrases uses binary search to merge them into its local sorted regular phrases.

An important parameter for this mechanism is the threshold after which a phrase is considered long. A smaller threshold will result in more long phrases, which will degrade performance because the binary-search-based merging of long phrases is not performant if many long phrases need to be merged. The threshold can be user-defined. A reasonable threshold appears to be $10(w + p)$, which is ten times the expected average phrase size given in Lemma 3.1.2.

5.2 Experimental Setup

For the experimental evaluation, the five datasets *Cere*, *Linux-kernel*, *Chromosome 19*, *Salmonella*, and *SARS-CoV-2* presented in Table 5.1 are used.

Dataset	Size	Alphabet Size	Notes
Cere	800 GB	5	Concatenation of copies of *cere* provided by the Pizza&Chili Repetitive Corpus [18]
Linux-kernel	536 GB	227	Concatenation of .h and .c files of the first 2775 commits on the Linux kernel repository [56]
Salmonella	508 GB	76	Concatenation of 105,000 whole-genome sequences of *Salmonella enterica* retrieved from NCBI [49]
SARS-CoV-2	95 GB	81	Concatenation of 3,162,365 *SARS-CoV-2* genome sequences retrieved from NCBI [50]
Chromosome 19	59 GB	19	Concatenation of 1,000 human chromosome 19 haplotype sequences [32]

Table 5.1: Datasets used for the evaluation.

The *Cere* dataset is obtained by concatenating multiple copies of the *cere* dataset provided by the Pizza&Chili repetitive corpus. The base dataset is a concatenation of 36 sequences of the yeast *Saccharomyces cerevisiae* provided by the Saccharomyces Genome Resequencing Project [18]. To obtain a large artificial repetitive text, this base dataset, with 440 MiB of data, was self-concatenated until it reached 800 GB.

For the genomic datasets *Salmonella, Chromosome 19* and *SARS-CoV-2*, there are usually five characters used: four for the bases (*A, C, G, T*) and one wildcard for any base (*N*). In addition, each sequence has its own header, including a sequence ID, which creates an alphabet larger than five. For example, a header in the *Chromosome 19* dataset is `>HG00105.1.19`, while the *Salmonella* dataset has longer headers, such as `>AAGBRO01 0000006.1SalmonellaentericastrainPNUSAS046030SAMN09664796-ri d6072313.denovo.06,wholegenomeshotgunsequence` explaining the different alphabet sizes. The datasets *Salmonella, Chromosome 19* and *SARS-CoV-2* have been edited such that each sequence is on its own line, with no line breaks within a sequence.

5.2.1 Repetitiveness

As mentioned before, PFP is especially good at compressing repetitive data. There are many different measures to describe the repetitiveness of data, and there is no consensus on the generally accepted approach [47]. For this thesis, the following two repetitiveness measurements are of interest:

Dataset	$\lvert T\rvert/r(T)$	$\lvert T\rvert/\pi_{20,200}$
Cere	69,170	209.2
Linux-kernel	13,109	189.8
Salmonella	1,059	51.1
SARS-CoV-2	743	4.2
Chromosome 19	1,287	10.4

Table 5.2: Repetitive measurements for the used datasets scaled by their size.

Number of BWT runs. A run in the BWT is the occurrence of the same element multiple times. The number of runs r denotes the number of runs in the BWT [47], yielding:

$$r = r(T) = r(BWT(T))$$

PFP based $\pi_{w,p}$. Lucà et al. introduced a repetitive measurement that directly relies on PFP [42]. They argue that the size of the dictionary and parse computed by PFP directly correlates with the repetitiveness of the input string. Hence, they proposed the following repetitiveness measurement:

$$\pi_{w,p}(T) = ||D_{w,p}|| + |P_{w,p}|$$

With w being the used window size, p the modulus, $||D_{w,p}||$ denoting the total size of all phrases in the dictionary computed by PFP with the parameters w and p, and $|P_{w,p}|$ the size of the resulting parse.

Because the datasets differ in size, the number of runs $r(T)$ and their $\pi_{w,p}$ are scaled by their size. The measurements for the datasets used in the evaluation are shown in Table 5.2. The π measurement is reported for the settings $w = 20, p = 200$. The artificially created dataset *Cere* shows a high repetitiveness, compared to the other datasets.

5.2.2 Environment

All experiments are conducted on the SuperMUC-NG high-performance computing system, using Intel Xeon Platinum 8174 processors running at a base frequency of 3.10 GHz. Each node consists of two processors with 48 cores in total. Every node is equipped with 96 GB of main memory. The nodes are connected using Intel Omni-Path at 100 Gbit/s [41]. *IntelMPI* in version 2025.3.0 was used to run the implementation, and the code itself was compiled with *GCC* 14.3.0. The compiler options *-O3, -DNDEBUG* and *-march=native* have been used.

To reduce the memory footprint of the MPI application, the following settings recommended by the Leibniz Rechenzentrum are used [3]

```
// size of forward cells
export I_MPI_SHM_CELL_FWD_NUM=0

// total number of extended cells per computational node
export I_MPI_SHM_CELL_EXT_NUM_TOTAL=0

// size of backward cells
export I_MPI_SHM_CELL_BWD_SIZE=65536

// number of backward cells per rank
export I_MPI_SHM_CELL_BWD_NUM=64

// disable Intel MPI custom allocator of private memory
export I_MPI_MALLOC=0

// disable Intel MPI custom allocator of shared memory
export I_MPI_SHM_HEAP_VSIZE=0
```

5.3 Distributed BWT Construction Evaluation

In the following, the experimental results of the distributed PFP-based BWT construction algorithm are presented. The experiments have been conducted on all five data sets introduced in Section 5.2. Four algorithms have been compared:

- The shared-memory parallel BWT construction based on Lyndon words, as introduced in Section 2.3 [4].
- The parallel shared-memory implementation of the PFP-based BWT construction BIG-BWT [5]
- Our distributed SA-based BWT construction using the distributed DCX algorithm, as presented in Section 4.1
- Our distributed PFP-based BWT construction, as presented in Section 4.2.3

[3] https://doku.lrz.de/intel-mpi-10746523.html#IntelMPI-Settingsfor lowmemoryfootprint

[4] https://gitlab.com/qwerzuiop/lyndongrammar

[5] https://gitlab.com/manzai/Big-BWT

The values presented in the following section are average values of five iterations for our distributed algorithms and of three iterations for the shared-memory algorithms.

5.3.1 Weak Scaling

In the weak scaling experiment, both the number of PEs and the amount of data input to the algorithm are increased simultaneously. For the following experiments, 100 MB of data per PE is used. For 48 PEs, this corresponds to 4.8 GB of input data; for 96 PEs, it is 9.6 GB, and so on.

The Lyndon-based approach and the reference implementation of BIG-BWT are both developed for shared-memory systems only. Therefore, they always use the same number of cores but with an increasing amount of input data. Both implementations support multi-threading. The maximum number of threads that still achieve a performance increase varies with the input and the hardware. On the SuperMUC-NG hardware, performance increases are observable up to a full 48-core shared-memory node. Hence, the following experiments with BIG-BWT and the Lyndon approach are conducted on 48 cores.

In this experiment, all datasets except the *Cere* dataset will be fully loaded at some PE count. The smallest dataset, *Chromosome 19*, is 59 GB. At 100 MB per PE, it will be fully loaded if there are more than 590 PEs. For the *SARS-CoV-2* dataset, the data is fully processed when using more than 950 PEs. With more PEs, this experiment becomes a strong-scaling experiment for the distributed algorithms, since the data does not change as the number of PEs increases.

Weak Scaling Runtime. The runtimes of the weak scaling experiment with 100 MB per PE for our PFP-based algorithm, our SA-based algorithm, the shared-memory Lyndon-based algorithm, and the shared-memory Big-BWT algorithm are shown in Figure 5.1. The vertical gray line indicates the PE count at which the whole input is loaded. For the distributed algorithm, the experiment becomes a strong-scaling experiment from this point onward. Missing data points indicate a crash or the execution time limit being reached.

100 MB per PE is close to the maximum amount of data that the distributed DCX algorithm can process and is therefore the limit for our SA-based approach. This is evident in the plot for the *Cere* dataset: because DCX runs out of memory, no runtime is recorded for the distributed SA-based approach. A similar situation can be observed in the *Chromosome 19* plot. Our SA-based algorithm crashes when used with 384 or fewer PEs, but works with 768 PEs because, at this point, the entire input is distributed across more PEs, resulting in less than 100 MB per PE, which is sufficient for the SA-based algorithm to work.

Another thing to note is the performance of the BIG-BWT implementation. The first data point for 48 PEs is higher across all datasets than the other three algorithms. As more data is loaded, the runtime increases significantly. The most likely reason for this behavior is that the BIG-BWT algorithm operates semi-externally. Multiple intermediate results

are stored on disk and loaded when needed. On the SuperMUC-NG system, a distributed file system tuned for high bandwidth is used. An example is the intermediate file *.dict*, which is written phrase-by-phrase and contains all unique phrases. For the distributed file system, many of these small file writes will degrade performance. This leads to very long computation times, making BIG-BWT using this system and data uncompetitive.

The Lyndon implementation always uses 48 cores but has to process an increasing amount of data. When comparing distributed algorithms to shared-memory algorithms, it needs to be investigated how many PEs the distributed algorithm needs to be faster than the shared-memory one.

On the *Linux-kernel* dataset, our distributed PFP-based BWT construction is faster on all PE counts. With 48 PEs and an input size of 4.8 GB, Lyndon takes 37.2 seconds, and our PFP BWT takes 23.1 seconds, resulting in a 1.6 times faster execution. As expected, Lyndon's execution times increase with higher PE counts because more data is processed on the same 48 cores. At 3072 PEs and 307.2 GB of input data, Lyndon takes 326.7 seconds, and our PFP BWT takes 20.20 seconds. At this point, Lyndon comes closer to the distributed SA-based approach, which takes 442.63 seconds.

For the *Cere*, *Chromosome 19* and *Salmonella* datasets, our distributed PFP BWT needs 96 PEs to equal Lyndon's performance. With 96 PEs and 9.6 GB of input data, PFP BWT takes 18.9 seconds for *Cere* while Lyndon takes 23.1 seconds. For the *Salmonella* dataset, performance is equal: PFP BWT takes 52.9 seconds, and Lyndon takes 53.3 seconds. For the *Chromosome 19* dataset, PFP BWT on 96 PEs is already faster than Lyndon, with 30.4 seconds versus 43 seconds.

The highest speedup across all datasets is achieved using 3072 PEs on the *Cere* dataset. Our distributed PFP-based algorithm needs 16.2 seconds to process 307.2 GB, while Lyndon needs 339.3 seconds, resulting in a speedup of 20.9.

Our distributed PFP BWT algorithm has the least competitive performance on the *SARS-CoV-2* dataset. While Lyndon needs 13.7 seconds to process 4.8 GB, PFP BWT takes 38.1 seconds. Only when using 384 PEs does PFP BWT outperform Lyndon, with 67.1 seconds versus 93 seconds. In this dataset, the strong scaling of the distributed DCX algorithm is evident. On 6144 PEs, it comes close to the PFP BWT. The *SARS-CoV-2* dataset contains 95 GB of data, so with 6144 PES, this amounts to 15.5 MB per PE. At this point, the SA-based approach takes 90.8 seconds, and PFP BWT takes 61.1 seconds.

Prefix-free Parsing Parameters. Prefix-free parsing heavily depends on the used parameters w for the window size and p for the splitter modulus. Using a smaller p will result in many small phrases, while a large p generates longer phrases. No sweeping statement can be made about these parameters, as the performance heavily depends on the input data. It could be the case that a large p creating large phrases will perform very well, because the extracted phrases appear very often in the input data, leading to fewer phrases and a shorter parse. But if the long phrases do not appear multiple times in the data, there will be a lot of

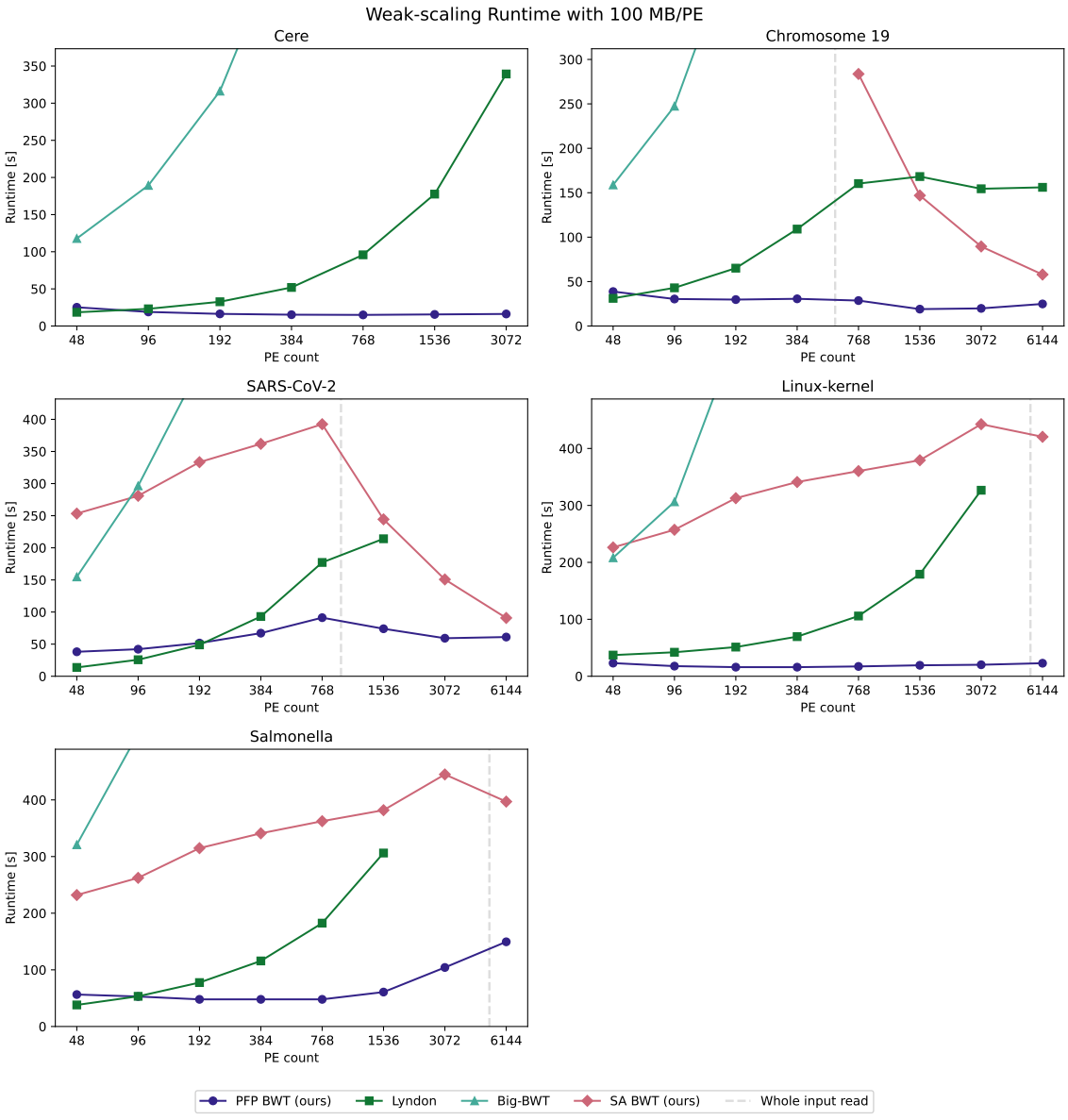


Figure 5.1: Weak scaling runtime with 100 MB/PE comparing our PFP-based algorithm, our SA-based algorithm, the shared-memory Lyndon-based algorithm, and Big-BWT.

long phrases in the dictionary. The same arguments apply to a small p and to the window size w.

Therefore, an experiment was conducted to compare the performance of different w and p parameters. For w the values $[5, 10, 20]$ and for p the values $[50, 100, 200]$ are used. For each possible combination, a weak scaling experiment with 100 MB per PE is run. Lucà et al. used randomly generated values for w and p and showed that using a modulus with $50 < p < 200$ and a window size of less than 50 yields the smallest parse-dictionary combination on the tested genomic data [42].

Figure 5.2 shows the results of this experiment. Different w and p settings affect the performance of the distributed PFP-based BWT construction. On the *Salmonella* dataset, these settings do not seem to influence performance. While no parameter configuration outperforms across all datasets, some degrade performance and should not be used. The worst performance in terms of runtime surfaces for the parametrization $w = 5$, even crashing if combined with $p = 200$. Adding up all the runtimes, the lowest total runtime across all datasets and all PE counts is $w = 20, p = 200$, with $w = 20, p = 50$ close behind. Hence, for all experiments, the setting $w = 20, p = 200$ has been used.

Memory Usage. While memory usage was not a main concern when developing the distributed PFP-based BWT construction algorithm, it should not be ignored. Hence, the 100 MB per PE weak scaling experiment using the determined parameters $w = 20, p = 200$ was conducted, and the peak memory usage per node was recorded. The peak memory is determined by reading the *ru_maxrss* field from the *rusage* struct populated by the Linux kernel. This field reports the maximum resident set size for the current process. All PEs of a node sum their individual peak memory values to get the per-node peak memory usage.

The recorded peak memory usage is shown in Figure 5.4. It reports the highest peak memory used by a single PE. The *blowup* is the factor by which the used memory is larger than the input data. On the SuperMUC-NG system, each node consists of 48 PEs. Hence, the blowup is computed by dividing the number of PEs by 48 to get the number of nodes used. This number is multiplied by the highest peak memory usage to get the total peak memory usage. The total peak memory usage is then divided by the amount of data processed per PE count to obtain the blowup. This blowup is shown in Figure 5.4 as well.

For the datasets *Cere, Chromosome 19* and *Linux-kernel*, the blowup until the whole file is processed is between 5 and 6. In the case of *SARS-CoV-2* and *Chromosome 19*, the blowup increases while peak memory usage decreases. This increase in blowup is expected because even though the same amount of data is being processed, more PEs are used, which in turn need larger bookkeeping structures, for example, to determine the owning PE of a specific piece of data. Another example of why the blowup increases is the per-node-replicated phrase-prefix sum and phrase-occurrence list.

The only dataset showing a blowup of more than 6 is *Salmonella*. In this case, the memory load is unbalanced. The average peak memory usage per node across all PE counts is

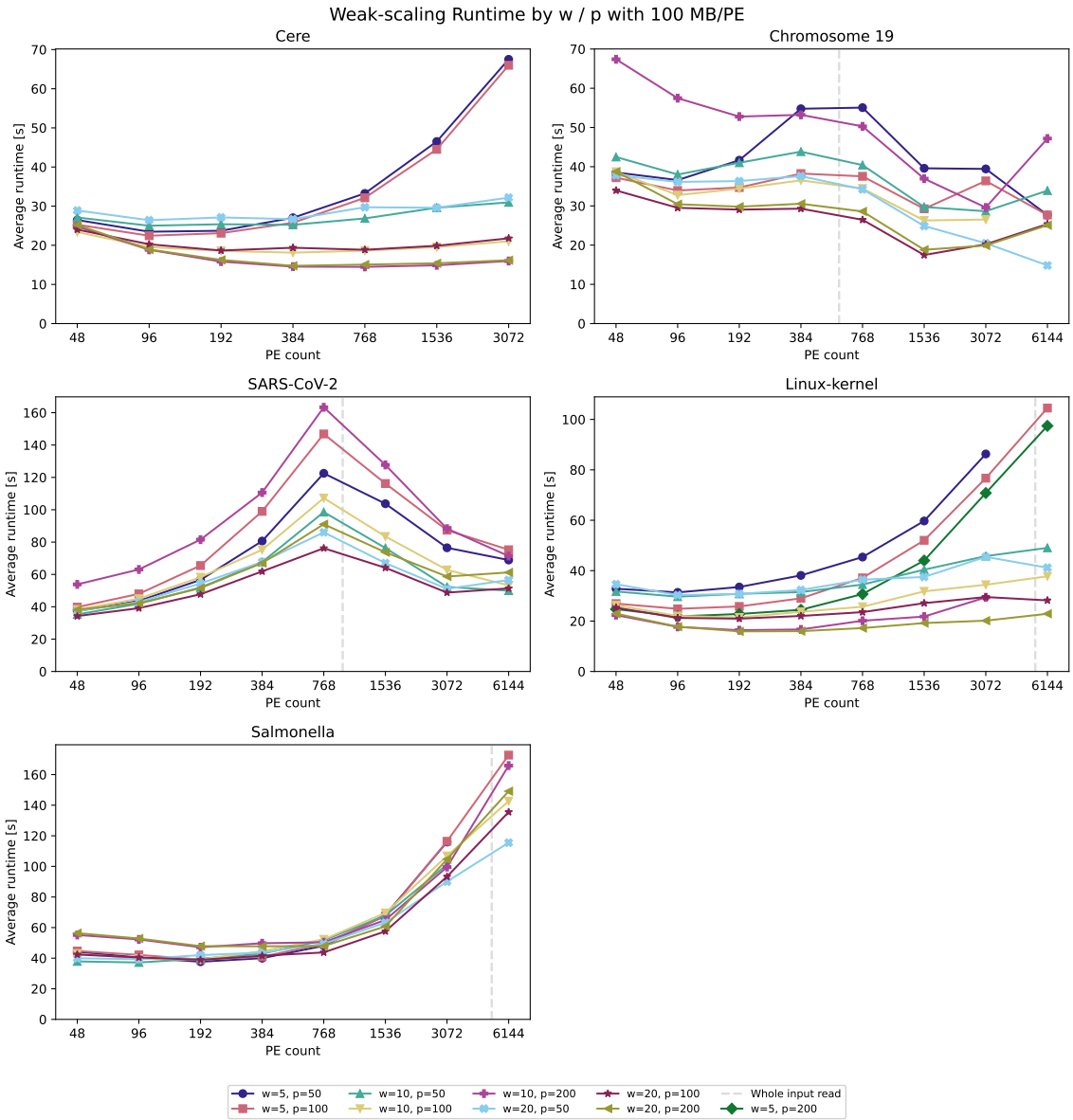


Figure 5.2: Runtime using different w and p parameters for a weak scaling experiment with 100 MB per PE.

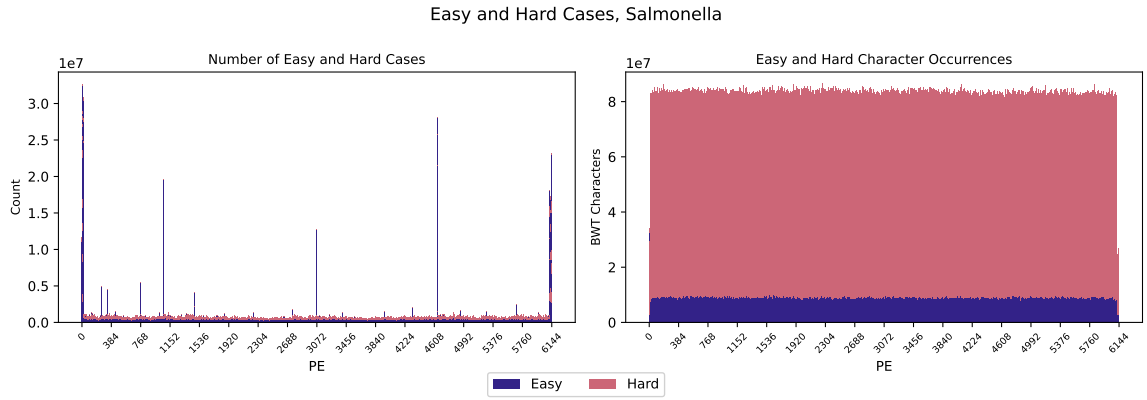


Figure 5.3: Number of easy and hard cases and the number of occurrences of the corresponding characters.

between 25 GB and 30 GB, while the highest peak memory usage shown in the graph ranges from 30 GB to 60 GB. The reason is that, in this dataset, PFP extracts many phrases that occur only once, as well as many that appear multiple times. The load balancing described in Section 4.2.4 tries to balance the load in a way that the resulting BWT is balanced across all PEs. This can lead to imbalances on the SA_D distribution, because a PE with many suffixes that appear only once has the same cost as one suffix appearing multiple times. The implementation stores per-suffix metadata. Therefore, a highly unbalanced SA_D will lead to unbalanced memory usage, but the resulting BWT will be balanced. This imbalance is visible in Figure 5.3. The *Number of Easy and Hard Cases* plot shows how SA_D is balanced across the PE and which cases they belong to. *Easy and Hard Character Occurrences* shows the number of characters each PE emits to the BWT on its local SA_D chunk. The resulting BWT is balanced across all PEs, therefore, the SA_D is not balanced.

Runtime Composition. As described in Chapter 4, the algorithm can be subdivided into the following main steps with regard to runtime:

(i) **PFP** - The computation of the dictionary D and the parse P.

(ii) **SA and LCP of D** - The computation of the suffix array and the LCP array of the dictionary.

(iii) **SA of P** - The computation of the suffix array of the parse.

(iv) **W and Inverted List** - This includes the computation of the BWT of the parse, the W permutation, and the computation of the inverted lists.

(v) **Main Loop** - The iteration over all local suffixes and solving of first-character-cases.

(vi) **Solve Cases** - This includes the steps needed to solve the easy and the hard cases.

For the 100 MB per PE weak scaling experiment, the per-step runtimes are shown in Figure 5.5.

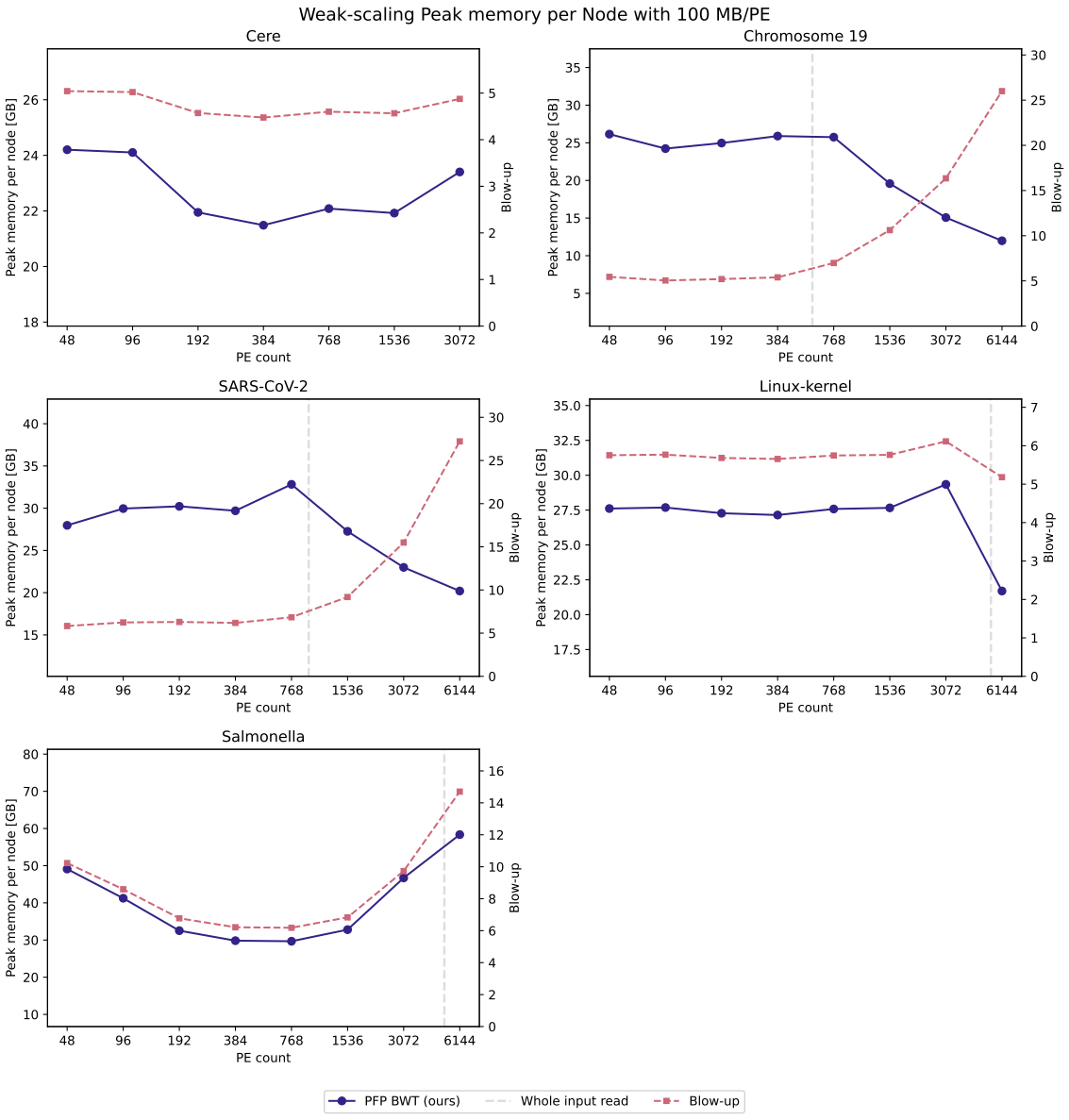


Figure 5.4: Weak scaling peak memory usage with 100 MB per PE.

In the figures for *Cere* and *Linux-kernel*, the advantages are evident. With increasing PE count, and therefore more input data, the runtime for computing the SA and LCP of the dictionary decreases. This is because, even though more input is loaded, more replication of already existing phrases is loaded as well, so the dictionary does not grow with the input. But the PE counts are increasing, so more PEs are used to compute the SA and LCP of a dictionary that does not grow as fast, resulting in a decrease in runtime. At the same time, the runtime of the SA computation grows, because the size of the parse grows with the input. Because the SA and LCP computation of the dictionary is costlier than the computation of the SA of the parse, this is a net gain in runtime.

In *Salmonella*, we observe a similar trend up to 1536 PEs. After that, the performance of the SA and LCP computations of D decreases. There is no clear evidence on why this happens. The size of the dictionary is growing as expected compared to the other datasets. We assume that implementation limits of the PSAC algorithm used are being hit.

Another performance bottleneck is the Solve Cases step. It depends on the input data whether this step is really a bottleneck. Only for the *SARS-CoV-2* and *Salmonella* datasets does this step become a bottleneck. For both datasets, PFP extracts far more easy cases than hard ones. But the suffixes corresponding to the phrases corresponding to the hard cases appear more often in the input data. Hence, many large inverted lists are exchanged to handle hard cases, which reduces performance. This discrepancy between easy and hard cases is evident in the previously introduced Figure 5.3.

5.3.2 Breakdown Experiments

With the breakdown experiment, we want to analyze how much data the distributed PFP-based BWT construction algorithm can process. For this experiment, 384 PEs are used. Unlike in the weak scaling experiments, the PE count remains the same, while the amount of data loaded per PE increases until the program crashes due to out-of-memory. As these experiments are run on the thin nodes of the SuperMUC-NG cluster, each node has 96 GB of main memory available. This experiment is conducted only on the three large datasets, *Cere*, *Linux-kernel*, and *Salmonella*. The two smaller datasets, *Chromosome 19* and *SARS-CoV-2*, are omitted from the results due to their small size. With 384 PEs, the full data will be loaded at some point, and the program will not run out of memory.

Figure 5.6 shows the results of the breakdown tests. For the *Linux-kernel* and *Salmonella* datasets, the PFP-based approach can handle up to 350 MB per PE. With 384 PEs, this amounts to 134.4 GB of input data before the program crashed. The SA-based BWT construction can process only 100 MB per PE, or 3.84 GB in total, across these two datasets. In both cases, the program runs out of memory during the distributed DCX computation. This corresponds to DCX's reported maximum of 130 MB per PE [29]. For the *Cere* dataset, the PFP-based implementation can process 400 MB per PE, totaling 153.6 GB of input data.

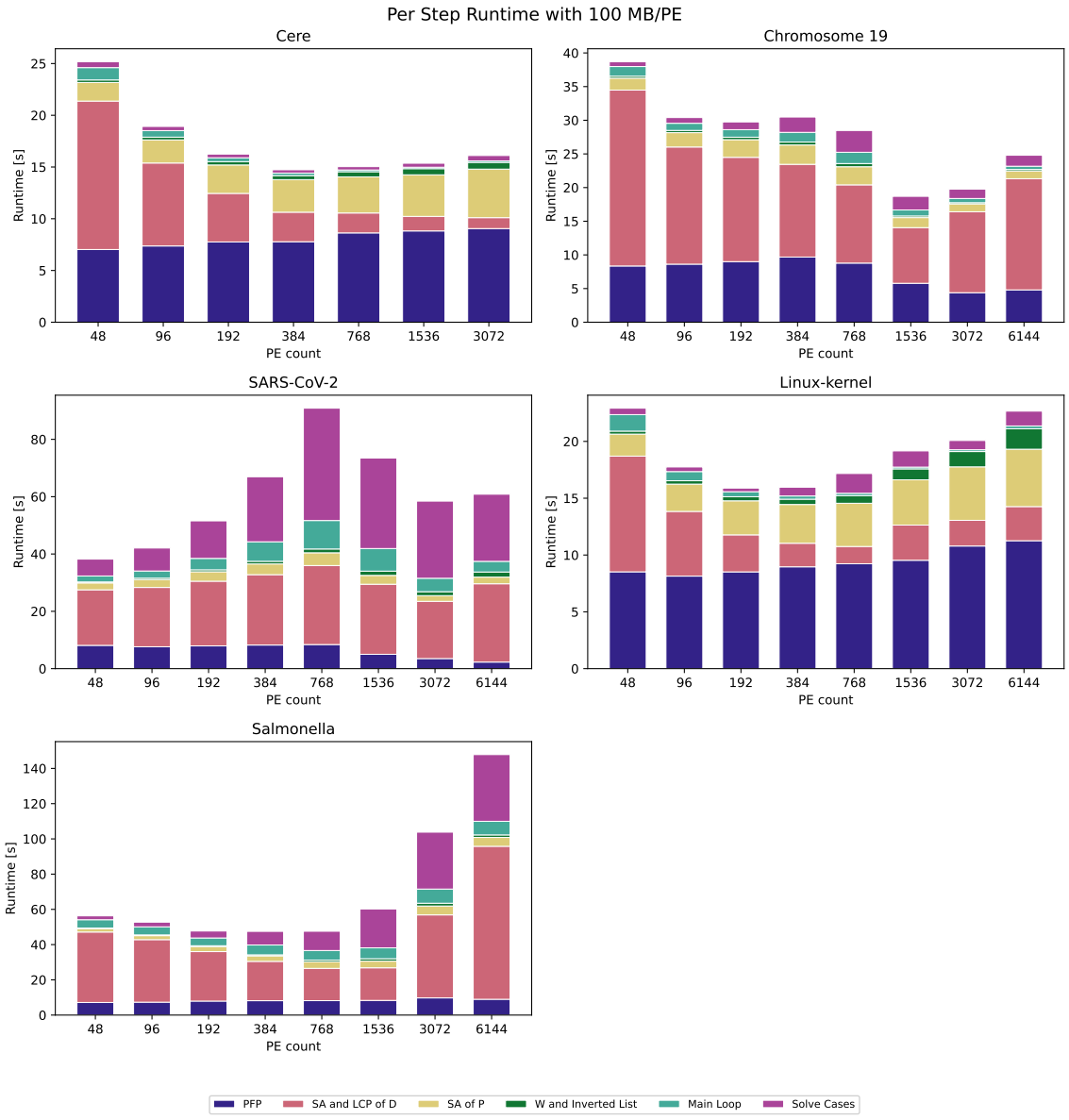


Figure 5.5: Weak scaling runtime per step of our distributed PFP-based algorithm with 100 MB per PE.

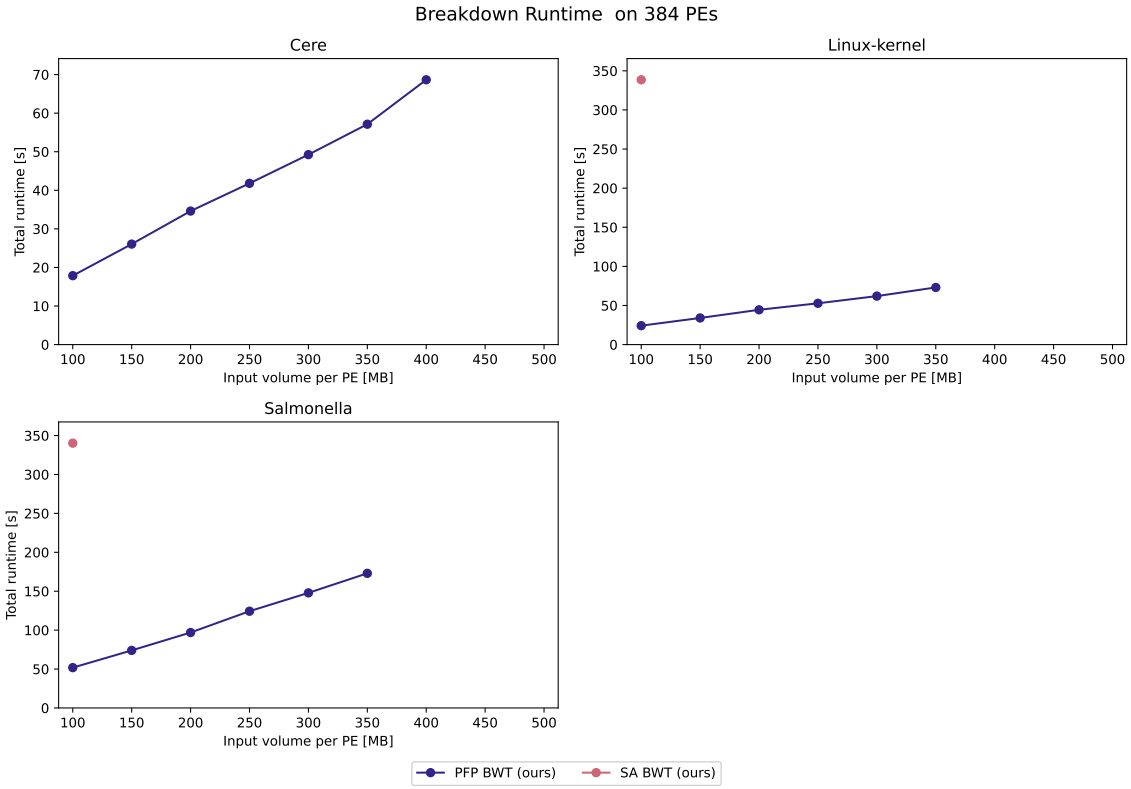


Figure 5.6: Breakdown runtime on 384 PEs.

This shows that the PFP-based approach can process far more data per PE than the SA-based approach.

5.3.3 Optimizations Evaluation

Difference-encoded Inverted Lists. In Section 5.1.1, the difference encoding for the inverted lists is presented. Instead of storing each entry as a raw 5-byte integer. The first value of each list is stored using as few bytes as possible, and each subsequent value is encoded as the difference between it and the next value using the least number of bytes possible.

The average number of bytes used for an entry in the inverted list using difference encoding is shown in Figure 5.7. The same experimental setup as for the weak scaling experiment was used; 100 MB per PE is loaded. For all datasets except *SARS-CoV-2*, the average number of bytes decreases slightly as the amount of data being processed increases. The largest jump is made by *Salmonella*, which uses 1.19 bytes on average for a total input of 4.8 GB and 1.05 bytes when processing the whole input with 508 GB.

With the average number of bytes used approaching 1, this indicates that the average dis-

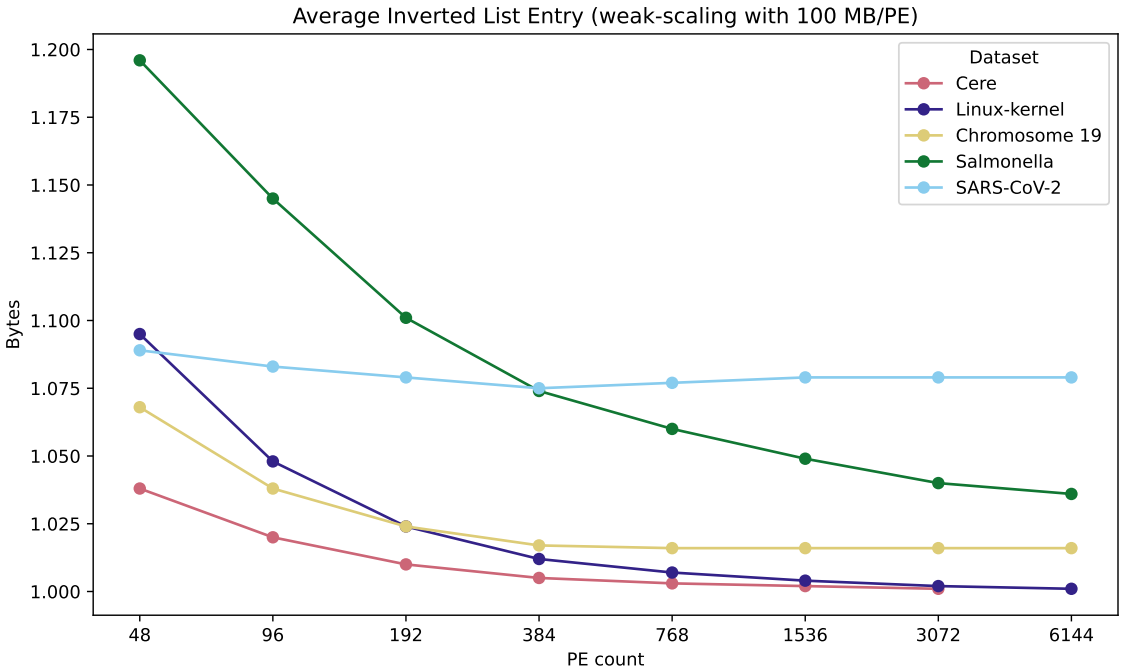


Figure 5.7: Average number of bytes for an entry in the inverted list using difference encoding.

tance between phrases in an inverted list is close to $2^7 = 128$. For each difference byte, seven bits are used to encode the number, while the eighth bit is reserved as a marker for whether the number continues on the next byte or not. Another factor that decreases the average number of bytes used is the total number of inverted lists, which reflects how often phrases appear in the parse. The more different phrases there are, the more inverted lists there are. With the first entry being stored as an absolute value, this can already increase the average number of bytes when there are only short inverted lists.

6 Discussion

6.1 Conclusion

The BURROWS-WHEELER TRANSFORM [10] (BWT) is a string permutation with applications in compressed full-text indices and compression. This makes the BWT a commonly seen tool in bioinformatics, where it is used as a preprocessing step to compress large datasets and serves as a building block for text indices [16] that enable efficient pattern matching.

Since the size of available genomic datasets is ever-increasing, an efficient and scalable implementation of the BWT is required. One possible approach is the use of large-scale distributed memory systems. To the best of our knowledge, there are no distributed algorithms that explicitly construct the BWT in distributed memory and scale with thousands of cores and hundreds of gigabytes of input.

We present two algorithms: a direct construction and a heavily engineered construction based on PREFIX-FREE PARSING [9]. The first one uses the suffix array-based approach to construct the BWT in distributed memory. The second distributed BWT construction algorithm is based on the shared-memory BIG-BWT [9] algorithm, which constructs the BWT using PREFIX-FREE PARSING (PFP) as a preprocessing step. The BIG-BWT algorithm was modified for the distributed setting and optimized to balance memory usage and workload. Using a PFP-based approach to construct the BWT works especially well for highly repetitive inputs like genomic data.

We show that our distributed BWT construction algorithms are able to scale to 6144 cores, making them, to our knowledge, the first distributed BWT construction algorithms that scale on distributed memory systems.

We compare our distributed BWT construction algorithms with the shared-memory parallel algorithm BIG-BWT [9] and the state-of-the-art shared-memory parallel algorithm for constructing the BWT of repetitive inputs, which is based on Lyndon words [51]. Extensive performance evaluation between the two reference implementations and our distributed BWT construction algorithms were performed and are documented in Section 5

Our distributed PFP-based approach was able to surpass the runtime performance of the shared-memory implementations running on a 48-core node when using 96 or more cores. For a repetitive source code dataset, our distributed PFP-based approach is faster than all evaluated competitors, even on a single 48-core compute node. Because our distributed algorithms scale with distributed memory, we are able to achieve a speedup of up to 20.9x

using our distributed PFP-based BWT construction algorithm on 64x more cores compared to the fastest shared-memory parallel competitor, which is based on Lyndon words.

Our distributed BWT construction based on computing the SA with the distributed DCX algorithm [29] is outperformed by our distributed PFP-based approach in every setting we evaluated. Using 6144 cores, the distributed PFP-based algorithm achieves a speedup of up to 18.4x on repetitive data compared to the distributed SA-based approach. The distributed PFP-based BWT construction algorithm uses less than 6 bytes per input byte on four of the five evaluated datasets. In comparison, the distributed DCX algorithm uses around 14 bytes per input byte.

6.2 Future Work

Oliva et al. argue that computing the BWT of the parse P is one of the bottlenecks in PFP-based BWT computation and propose the RECURSIVE PREFIX-FREE PARSING algorithm [52]. Instead of using an SA-based BWT construction of the parse P, it uses the PFP-based approach to compute the BWT of P. It is worth considering whether this recursive use of PFP proves itself in distributed settings.

In the evaluation, it became apparent that one of the time-consuming steps of the computation is the SA and LCP computation of the dictionary D. For this purpose, the current state-of-the-art distributed algorithm, PSAC, is used. The distributed DCX algorithm appears to scale better with increasing numbers of PEs, but it cannot compute the LCP array. It should be possible to obtain the LCP array as a byproduct during the SA computation. Hence, adding the LCP array computation to the distributed DCX algorithm could also improve the distributed PFP-based BWT.

Another potential bottleneck identified in the evaluation is solving the hard cases. Depending on the input data, this step can account for a substantial portion of the runtime of the distributed PFP-based BWT construction algorithm. Research could focus on a completely different approach to solving the hard cases, without relying on inverted lists. A less substantial change would be to explore possible adjustments to the current approach, for example, by addressing the hard cases on its own using a different load-balancing approach.

Bibliography

[1] 1000 Genomes Project Consortium, Adam Auton, Lisa D Brooks, Richard M Durbin, Erik P Garrison, Hyun Min Kang, Jan O Korbel, Jonathan L Marchini, Shane McCarthy, Gil A McVean, and Gonçalo R Abecasis. A global reference for human genetic variation. *Nature*, 526(7571):68–74, October 2015.

[2] Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. Practical massively parallel sorting. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, page 13–23, New York, NY, USA, 2015. Association for Computing Machinery.

[3] Michael Axtmann and Peter Sanders. *Robust Massively Parallel Sorting*, pages 83–97.

[4] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. Engineering in-place (shared-memory) sorting algorithms. Computing Research Repository (CoRR), Sept. 2020.

[5] Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the bwt of string collections. *Theoretical Computer Science*, 483:134–148, 2013. Special Issue Combinatorial Pattern Matching 2011.

[6] Timo Beller, Simon Gog, Enno Ohlebusch, and Thomas Schnattinger. Computing the longest common prefix array based on the Burrows–Wheeler transform. *J. Discrete Algorithms (Amst.)*, 18:22–31, January 2013.

[7] Timo Bingmann. TLX: Collection of sophisticated C++ data structures, algorithms, and miscellaneous helpers, 2018. `https://panthema.net/tlx`, retrieved Oct. 7, 2020.

[8] Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original ebwt faster, simpler, and with less memory. In Thierry Lecroq and Hélène Touzet, editors, *String Processing and Information Retrieval*, pages 129–142, Cham, 2021. Springer International Publishing.

[9] Christina Boucher, Travis Gagie, Alan Kuhnle, and Giovanni Manzini. Prefix-Free Parsing for Building Big BWTs. In Laxmi Parida and Esko Ukkonen, editors, *18th International Workshop on Algorithms in Bioinformatics (WABI 2018)*, volume 113 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[10] Michael Burrows, D J Wheeler D I G I T A L, Robert W. Taylor, David J. Wheeler, and David Wheeler. A block-sorting lossless data compression algorithm. 1994.

[11] Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, and Giovanna Rosone. Computing the optimal bwt of very large string collections. In *2023 Data Compression Conference (DCC)*, pages 71–80, 2023.

[12] K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, iv. the quotient groups of the lower central series. *Annals of Mathematics*, 68(1):81–95, 1958.

[13] Anthony J Cox, Markus J Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform. *Bioinformatics*, 28(11):1415–1419, June 2012.

[14] Diego Díaz-Domínguez, Lavinia Egidi, Veronica Guerrini, Felipe A Louza, Giovanna Rosone, et al. Algorithms for computing very large bwts: a short survey. *OPEN ACCESS SERIES IN INFORMATICS*, 131, 2025.

[15] Peter M. Fenwick. Block sorting text compression – final report. 1996.

[16] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, page 390, USA, 2000. IEEE Computer Society.

[17] Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, July 2012.

[18] Paolo Ferragina and Gonzalo Navarro. Pizza&Chili Corpus – Compressed Indexes and their Testbeds — pizzachili.dcc.uchile.cl. `https://pizzachili.dcc.uchile.cl/real-stats.html`. [Accessed 14-07-2026].

[19] Patrick Flick and Srinivas Aluru. Parallel distributed memory construction of suffix and longest common prefix arrays. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[20] Patrick Flick and Srinivas Aluru. Parallel construction of suffix trees and the all-nearest-smaller-values problem. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 12–21, 2017.

[21] Patrick Flick and Srinivas Aluru. Distributed enhanced suffix arrays: Efficient algorithms for construction and querying. In *SC19: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17, 2019.

[22] Food and Drug Administration (FDA). Total number of sequences in the genometrakr database. `https://www.fda.gov/media/192147/download?attachment`. [Accessed 15-07-2026].

[23] Pierre Fraigniaud and Emmanuel Lazard. Methods and problems of communication in usual networks. *Discrete Applied Mathematics*, 53(1):79–133, 1994.

[24] Ylenia Galluzzo, Raffaele Giancarlo, Mario Randazzo, and Simona E. Rombo. Burrows wheeler transform on a large scale: Algorithms implemented in apache spark. *Data*, 11(3), 2026.

[25] Adrián Goga and Andrej Baláž. Prefix-Free Parsing for Building Large Tunnelled

Wheeler Graphs. In Christina Boucher and Sven Rahmann, editors, *22nd International Workshop on Algorithms in Bioinformatics (WABI 2022)*, volume 242 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:12, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[26] Ilya Grebnov. libsais. `https://github.com/IlyaGrebnov/libsais`. [Accessed 21-07-2026].

[27] Ilya Grebnov. GitHub - IlyaGrebnov/libcubwt: libcubwt is a library for GPU accelerated suffix array and burrows wheeler transform construction. — github.com. `https://github.com/IlyaGrebnov/libcubwt`, 2025. [Accessed 19-07-2026].

[28] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[29] Manuel Haag, Florian Kurpicz, Peter Sanders, and Matthias Schimek. Fast and Lightweight Distributed Suffix Array Construction. In Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman, editors, *33rd Annual European Symposium on Algorithms (ESA 2025)*, volume 351 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 47:1–47:18, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[30] James Holt and Leonard McMillan. Merging of multi-string bwts with applications. *Bioinformatics*, 30(24):3524–3531, 12 2014.

[31] Aaron Hong, Massimiliano Rossi, and Christina Boucher. *LZ77 via Prefix-Free Parsing*, pages 123–134.

[32] Enno Ohlebusch Jannik Olbrich, Thomas Büchler. PANAMA - Universität Ulm — uni-ulm.de. `https://www.uni-ulm.de/en/in/institut-fuer-theoretische-informatik/research/seqana/panama/`, 2024. [Accessed 20-07-2026].

[33] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, November 2006.

[34] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, page 125–136, New York, NY, USA, 1972. Association for Computing Machinery.

[35] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[36] Donald Knuth. *The Art Of Computer Programming, vol. 3: Sorting And Searching*. Addison Wesley Longman Publishing Co., Inc, 1973.

[37] Florian Kurpicz, Pascal Mehnert, Peter Sanders, and Matthias Schimek. Scalable distributed string sorting. *arXiv preprint arXiv:2404.16517*, 2024.

[38] Juha Kärkkäinen. Fast bwt in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007. The Burrows-Wheeler Transform.

[39] Heng Li. BWT construction and search at the terabase scale. *Bioinformatics*, 40(12):btae717, November 2024.

[40] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, July 2009.

[41] Leibniz-Rechenzentrum (LRZ). Hardware of SuperMUC-NG Phase 1 — doku.lrz.de. `https://doku.lrz.de/hardware-of-supermuc-ng-phase-1-11482553.html`. [Accessed 14-07-2026].

[42] Simone Lucà, Francesco Masillo, and Zsuzsanna Lipták. Measuring genomic data with prefix-free parsing. *Computational Biology and Chemistry*, 122:108870, 2026.

[43] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[44] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the burrows–wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007. The Burrows-Wheeler Transform.

[45] Giovanni Manzini. The burrows-wheeler transform: Theory and practice. In Mirosław Kutyłowski, Leszek Pacholski, and Tomasz Wierzbicki, editors, *Mathematical Foundations of Computer Science 1999*, pages 34–47, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[46] M McClelland, K E Sanderson, J Spieth, S W Clifton, P Latreille, L Courtney, S Porwollik, J Ali, M Dante, F Du, S Hou, D Layman, S Leonard, C Nguyen, K Scott, A Holmes, N Grewal, E Mulvaney, E Ryan, H Sun, L Florea, W Miller, T Stoneking, M Nhan, R Waterston, and R K Wilson. Complete genome sequence of salmonella enterica serovar typhimurium LT2. *Nature*, 413(6858):852–856, October 2001.

[47] Gonzalo Navarro. Indexing highly repetitive string collections, part i: Repetitiveness measures. *ACM Comput. Surv.*, 54(2), March 2021.

[48] Joshua J. Nolan, Jamie Forrest, and Elizabeth Ormondroyd. Additional findings from the 100,000 genomes project: A qualitative study of recipient perspectives. *Genetics in Medicine*, 26(6):101103, 2024.

[49] National Library of Medicine. Genome Salmonella enterica — ncbi.nlm.nih.gov. `https://www.ncbi.nlm.nih.gov/datasets/genome/?taxon=28901`, 2026. [Accessed 20-07-2026].

[50] National Library of Medicine. Genome Severe acute respiratory syndrome coronavirus 2 — ncbi.nlm.nih.gov. `https://www.ncbi.nlm.nih.gov/datasets/genome/?taxon=2697049`, 2026. [Accessed 20-07-2026].

[51] Jannik Olbrich. Fast and Memory-Efficient BWT Construction of Repetitive Texts Using Lyndon Grammars. In Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman, editors, *33rd Annual European Symposium on Algorithms (ESA 2025)*, volume 351 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 60:1–

60:19, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[52] Marco Oliva, Travis Gagie, and Christina Boucher. Recursive prefix-free parsing for building big bwts. In *2023 Data Compression Conference (DCC)*, pages 62–70, 2023.

[53] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox*. Springer Publishing Company, Incorporated, 1st edition, 2019.

[54] TOP500. TOP500 List - November 2008 | TOP500 — top500.org. `https://top500.org/lists/top500/list/2008/11/`, 2008. [Accessed 16-07-2026].

[55] TOP500. TOP500 List - June 2026 | TOP500 — top500.org. `https://top500.org/lists/top500/list/2026/06/`, 2026. [Accessed 16-07-2026].

[56] Linus Torvalds. GitHub - torvalds/linux: Linux kernel source tree — github.com. `https://github.com/torvalds/linux`, 2026. [Accessed 20-07-2026].

[57] Clare Turnbull, Richard H Scott, Ellen Thomas, Louise Jones, Nirupa Murugaesu, Freya Boardman Pretty, Dina Halai, Emma Baple, Clare Craig, Angela Hamblin, Shirley Henderson, Christine Patch, Amanda O'Neill, Andrew Devereau, Katherine Smith, Antonio Rueda Martin, Alona Sosinsky, Ellen M McDonagh, Razvan Sultana, Michael Mueller, Damian Smedley, Adam Toms, Lisa Dinh, Tom Fowler, Mark Bale, Tim Hubbard, Augusto Rendon, Sue Hill, and Mark J Caulfield. The 100 000 genomes project: bringing whole genome sequencing to the nhs. *BMJ*, 361, 2018.

[58] Tim Niklas Uhl, Matthias Schimek, Lukas Hübner, Demian Hespe, Florian Kurpicz, Daniel Seemaier, Christoph Stelz, and Peter Sanders. Kamping: Flexible and (near) zero-overhead c++ bindings for mpi. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–21, 2024.

[59] Zhiheng Zhao, Jianping Yin, and Wei Xiong. Gpu-accelerated burrow-wheeler transform for genomic data. In *2012 5th International Conference on BioMedical Engineering and Informatics*, pages 889–892, 2012.