Scalable Text Index Construction

Timo Bingmann[1], Patrick Dinklage[2], Johannes Fischer[2], Florian Kurpicz[2], Enno Ohlebusch[3], and Peter Sanders[1]

[1] Karlsruher Institut für Technologie, Germany tb@panthema.net, sanders@kit.edu
[2] Technische Universität Dortmund, Germany patrick.dinklage@tu-dortmund.de, johannes.fischer@cs.tu-dortmund.de, florian.kurpicz@tu-dortmund.de
[3] Universität Ulm, Germany enno.ohlebusch@uni-ulm.de

Abstract. We survey recent advances in scalable text index construction with a focus on practical algorithms in distributed, shared, and external memory.

Keywords: text indices, suffix array, suffix tree, wavelet tree, Burrows-Wheeler transform, FM-index, distributed memory, shared memory, external memory

1 Introduction

Texts occur in many different domains, ranging from natural language texts over source code to DNA and protein sequences, and their amount is ever-increasing. The field of algorithm and data structure research on strings is often referred to as *Stringology*. One important aspect within this line of research is the efficient construction of text indices. A text index is a data structure that provides additional information for a given text to speed up answering different types of queries, e.g., pattern matching queries that ask if (or how often, or where) a pattern occurs in the text. We focus on full-text indices for possibly unstructured texts, which allow the user to query for arbitrary patterns (this excludes, e.g., inverted indices). Real-world applications of text indices can be found, for example, in computational biology where text indices are a crucial part of the software for DNA alignment [134]. However, the amount of textual data is increasing significantly faster than the computational capacity of ordinary computers. For example, in 2008 the 1000 Genomes Project (1KGP) was launched to collect and sequence the genomes of thousands of people, whereas, in 2020, the 1+Million Genomes Initiative (1+MG) started to collect at least one million genomes, making this collection 1000 times larger. Therefore, scalable construction algorithms that can handle the massively growing amount of text are necessary.

In this survey, we discuss the current state of the art in scalable text index construction. We focus on distributed, external, and shared memory construction algorithms for different text indices and their applications. While there already exist surveys focussing on particular indices (e.g., suffix arrays [28,172] or wavelet trees [63 SPP,149,160]), or models of computation (e.g., external memory [56,23]), this chapter tries to give a more unified view. To this end, we point out common techniques that are used in different models of computation or in the computation of different text indices.

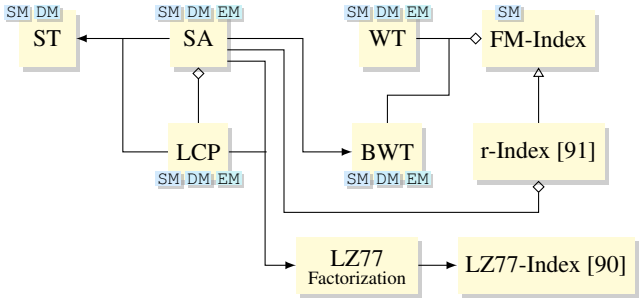


Fig. 1: Relations of text indices. In this article, we consider text indices that have scalable construction algorithms. The labels SM, DM, EM mark whether such construction algorithms in shared, distributed, and external memory exist. Note that the LZ77 factorization itself is a text compression and not an index. We use arrows (→) to denote indices that are used (in practice) to compute the targeted text index or if they are a special case (⊸) of the targeted index. Diamonds (⊸) are used to denote indices that are part of the targeted text index.

This survey is structured as follows. First, in Section 2, we introduce models of computation and give an overview of (string) sorting algorithms and further building blocks that are required as basic tools for text index construction. The main body of work can be found in Section 3. Here, we discuss the scalable construction of different text indices. We start with the suffix array (*SA*), one of the most well-researched text indices, and the longest common prefix (*LCP*) array, which often accompanies the SA. Next, we take a look wavelet trees (*WT*) and the Burrows-Wheeler Transform (*BWT*), which both are important parts of the FM-index, a compressed text index frequently used in practice. Then, we discuss algorithms for the suffix tree (*ST*) and space efficient representations thereof. See Fig. 1 for an overview of the text indices and their relations. While most of the discussed work solely focuses on the construction of the text indices, we also show approaches to *answer* queries on text indices in distributed memory. Finally, in Section 4, we show real-world applications of text indices in bioinformatics and text compression before we address future challenges in Section 5.

2 Preliminaries

Let $T = T[0] \dots T[n-2]\$$ be a text of length n over an alphabet $\Sigma = [0, \sigma)$, where we assume that T is terminated with an end-of-file or sentinel symbol \$ with $\$ \notin \Sigma$ and $\$ < \alpha$ for all $\alpha \in \Sigma$. A text over an alphabet of size $\sigma = 2$ is called *bit vector*. Usually, bit vectors do not contain a sentinel. We call $T[i..j) = T[i] \dots T[j-1]$ a *substring* of T for $i, j \in [0, n]$. The substrings $T[0..i)$ and $T[j..n)$ are called *prefix* and *suffix* for $i, j \in [0, n]$.

2.1 Models of Computation

In this section, we introduce models of computation that are relevant for the rest of this chapter and give pointers to software libraries that are commonly used to implement

algorithms in those models. The starting point is the sequential *random access machine* (RAM) model [182], where we have a single *processing element* (PE) that contains multiple registers to perform operations on data and a main memory, which can be accessed in constant time. However, real-world systems are often more complex and require more sophisticated models.

One of these models is the *external memory* (EM) model [4]. Here, we have an internal memory of size M words and an external memory of unlimited size that is much slower to access randomly. To compensate for this, transfer between EM and RAM happens in blocks of B consecutive words. Such a transfer is called *I/O operation* (I/O for short). The cost of external memory algorithms is then described by the number of required I/Os, e.g., scanning through N elements requires $\Theta\left(\frac{N}{B}\right)$ I/Os, and sorting N elements requires $\text{sort}(N) := \Theta\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ I/Os. The software libraries STXXL [57] and TPIE [9] implement the most commonly used external memory algorithms and data structures. A (practical) relaxation of the model is the *semi-external* model, where we allow random access to either the input or output, but not both. The Succinct Data Structure Library (SDSL) [94] provides implementations of semi-external construction algorithms for various data structures.

We also consider two parallel machine models, where by p we always denote the number of available PEs. The first is the *parallel random access machine* (PRAM), where all PEs have access to the same (shared) memory. There are various PRAM variants differentiating between which types of concurrent memory reads/writes are allowed; for practical algorihms on a multi-core processor one should only use exclusive writes, implying that the *Concurrent Read Exclusive Write* (CREW) model is best for analyzing algorithms. In the analysis, the *work* and *depth* are of interest. The former is the total number of operations performed, and the latter is the longest sequence of sequential dependencies in the algorithm. When implementing shared memory algorithms, Cilk [38] (now deprecated), OpenMP [53], Intel's TBB [174], Microsoft's Parallel Patterns Library (PPL), or built-in concurrency features of the programming language, e.g., `thread` in C++11, are often used to express parallelism. The Multi-Core Standard Template Library (MCSTL) [188] provides parallel algorithms and can be used as the *parallel mode* of the GNU C++ Standard Library. Recently, ParlayLib [36] was introduced as a library containing efficient implementations of the parallel algorithms in the C++ Standard Library.

The *distributed memory* model is our second parallel machine model. Here, communication between different PEs is conducted by sending messages over a network, and PEs have only local memory. Often, the cost of such a message is given as a startup cost plus a cost that depends on the size of the message. This is also reflected in the *bulk-synchronous parallel* model [200], where algorithms are divided into a sequence of supersteps consisting of three phases: local work, communication, and synchronization. The cost of an algorithm is then the sum of the costs of all supersteps. In practice, there are two flavors of frameworks for developing distributed algorithms: low-level interfaces provided by the *message passing interface* (MPI)[4] with its open-source implementations Open MPI [89] and MPICH [98], and frameworks providing a more high-

[4]MPI standard: `https://www.mpi-forum.org/docs` (last accessed 2020-07-14).

level functionality, e.g., Apache Flink [5], Apache Hadoop (based on MapReduce [54]), Apache Spark [210], and Thrill [29 SPP].

2.2 Building Blocks

Sorting. Sorting is a fundamental and well-studied topic in computer science, and the many results fill entire volumes [129,146] of related work. Hence, we will only review recent results for sorting integers in this section, which can be used in various of the following text indexing algorithms. In applications, sorting is most often still performed using classic sequential algorithms [107,159], despite existing more cache- or instruction-efficient variants [180,205,12,65] and well-developed modern parallel algorithms for shared-memory machines such as IPS^{4}o [17], or the sorters in the MCSTL [188], Intel's TBB [174], the PBBS [186], ParlayLib [36], or Microsoft's PPL. Another method of accelerating sorting is by vectorizing comparisons or operations using SIMD instructions [87,110,207,41,108,209,35].

For sorting integers, there is also the option of using radix sort algorithms, which have to be implemented carefully for modern CPUs [152,173,123]. Many parallel radix sorts for shared-memory machines are also available [192,138,203,165], and are most prominent on GPUs [181,101,109,154,194].

Sorting of data on external memory is a classic subject [4,58], and implementations are available in specialized libraries like TPIE [8] or STXXL [57].

An entirely different challenge is sorting on highly-scalable distributed shared-nothing machines, where load balancing, communication, and data redistribution have to be devised carefully, as PEs do not share memory. Most distributed memory sorting algorithms are based on either Quicksort [133,1,178,196,16,13] or sample sort [60,37,106,96,193,14 SPP,15,13].

Sorting is often used as a black box for text indexing algorithms, but depending on the model, machine, or scenario, large performance gains are possible by picking a better sorting implementation.

String Sorting. Sorting strings is an interesting special case of sorting, especially for text indexing algorithms, and most classical sorting algorithms have been adapted to multi-component objects or multi-key data [26,152,189,162,123,33]. Early parallel algorithms were formulated in the PRAM model and are based on merging of tries [102,113]. For external memory, theoretical algorithms were proposed, distinguishing short and long strings [7], or using hashing [70]. Many well-developed cache-efficient sequential and shared-memory parallel string sorting algorithms [33,30 SPP,28] are available in the TLX C++ library[5]. The fastest sequential ones are engineered variants of radix sort with very little memory overhead, and the fastest shared-memory parallel one is a string-aware sample sort implementation. These implementations also support outputting the lengths of the longest common prefixes (LCPs) of lexicographically adjacent strings at next to zero extra cost.

[5] TLX website: `https://panthema.net/tlx/` (last accessed 2020-10-18)

While in principle the shared-memory parallel algorithms could be adapted to shared-nothing distributed supercomputers, they neglect that *communication volume* is the limiting factor for the scalability of algorithms to large systems [6,39]. The first distributed string sorting algorithm we developed was a straight-forward adaptation of merge sort for use in a distributed suffix array construction algorithm [78 SPP]. This first version still considered strings as unbreakable objects.

Bingmann et al. therefore developed genuine distributed string sorting algorithms based on multi-way merge sort [34 SPP], which break up the strings into characters. The strings on each PE are first sorted locally. The PEs then collectively execute a distributed partitioning algorithm which yields p ranges of equal size with respect to the entire data. Each range is spread across the p machines in p fragments, and in the next step, each PE sends its misplaced $p-1$ fragments to the corresponding target machine. Finally, each PE merges the received partition fragments. The appeal of multi-way merging for communication-efficient sorting is that the local sorting exposes common prefixes of the local input strings. The *Distributed String Merge Sort* (MS) exploits this by only communicating the length of the common prefix with the previous string followed by the remaining characters. Here, the LCP values also allow us to use the multiway LCP-merging technique previously developed by Bingmann et al. [30 SPP] in such a way that characters are only inspected once.

The second algorithm, *Distributed Prefix-Doubling String Merge Sort* (PDMS), further improves communication efficiency by only communicating characters that may be needed to establish the global ordering of the data (the distinguishing prefix). The algorithm also has optimal local work for a comparison-based string sorting algorithm. The key idea is to apply the communication-efficient duplicate detection algorithm by Sansers et al. [179] to geometrically growing prefixes of each string. Once a prefix has no duplicate anymore, we know that it is sufficient to transmit only this prefix. The same idea was also used to make *any* PRAM algorithm LCP-aware [68 SPP].

An experimental evaluation of MS and PDMS (which are implemented in MPI) on up to 1280 cores shows that these algorithm are often more than five times faster than previous non-string-aware algorithms. In the future, we hope that these algorithms will find their way into general purpose distributed toolkits such as Apache Spark [210] or Thrill [29 SPP].

Further Building Blocks. The *prefix sum* (w.r.t. a binary associative operator $\oplus$) of n elements $A[0],\ldots,A[n-1]$ is an array B of n elements with $B[i] = \bigoplus_{k=0}^{i} A[k]$ for $i \in [0,n)$. In the PRAM model, the prefix sum of n elements can be computed in $\mathcal{O}(\lg n)$ depth and $\mathcal{O}(n)$ work [112, p. 47]. Due to their ubiquity, algorithms for prefix sums are part of frameworks used in different parallel models, e.g., distributed [29 SPP] and shared memory [188].

Rank and *select* data structures for a bit vector of length n allow us to compute the number of set (or unset) bits up to position $i \in [0,n)$ (rank), and the position of the j-th set (or unset) bit for $j \in [1,n]$ (select), respectively. They are an important ingredient of wavelet trees (see Section 3.2). To the best of our knowledge, the only parallel construction algorithms for rank and select data structures are described by Shun [185] and require $\mathcal{O}(\lg n)$ depth and $\mathcal{O}(n/\lg n)$ work if the n bits are packed into $\lceil n/\lg n \rceil$ words.

	0	1	2	3	4	5	6	7	8	9	10	11
T	m	i	s	s	i	s	s	i	p	p	i	$
SA	11	10	7	4	1	0	9	8	6	3	5	2
LCP	0	0	1	1	4	0	0	1	0	2	1	3
	$	i$	ippi$	issippi$	ississippi$	mississippi$	pi$	ppi$	sippi$	sissippi$	ssippi$	ssissippi

Fig. 2: Suffix array and longest common prefix array (see Section 3.1) for the text $T =$ `mississippi$`. Below, we also show the suffixes in lexicographical order, i.e., the suffixes represented in the suffix array. There, we also visualize the longest common prefixes of two lexicographically consecutive suffixes in green (▢).

In practice, only sequential construction has been considered, e.g., [46,147,211]. However, the construction of the data structures proposed by Zhou et al. [211] heavily relies on prefix sums and could thus easily be parallelized.

We can generalize binary rank and select queries for a text T. Then, the function $rank_\alpha(T,i)$ counts, for some character $\alpha \in \Sigma$ and a text position $i \in [0,n)$, the number of occurrences of α in $T[0..i]$, whereas $select_\alpha(T,k)$, for some $k > 0$, finds the position of the k-th occurrence of α in T. Generalized rank/select queries can be answered efficiently using wavelet trees, which reduce them to $\mathcal{O}(\lg \sigma)$ binary rank/select queries (see Section 3.2).

3 Text Indices

A text index provides additional information for a text to speed up answering different types of queries. In the following, we give an overview of different construction algorithms for text indices in the models that we describe in Section 2.1.

3.1 Scalable Suffix Array Construction

One of the best-researched text indices is the *suffix array* (*SA*), which has been introduced by Manber and Myers [150] and independently by Gonnet et al. [95] as the PAT array. The *SA* of a text T of length n is a permutation of $[0,n)$ such that $T[SA[i],n) < T[SA[j],n)$ for all $0 \le i < j < n$, i.e., it lists all suffixes lexicographically. See Fig. 2 for an example. Suffix arrays are a space efficient replacement of *suffix trees* (*ST*) (see Section 3.3). To obtain the same functionality as the *ST*s, *SA*s are often accompanied by additional arrays containing further information. Since suffix array

construction algorithms sort all suffixes of a text, we use the term *suffix sorting* synonymously with suffix array construction.

When both the text and the *SA* fit into memory, the *SA* can be computed in linear time using the *difference cover* algorithm [124]. The idea is to sample suffixes and sort the samples. Using the sorted samples, we can lexicographically compare two suffixes in constant time. First, we compute SA^{12} containing all suffixes starting at positions that are not a multiple of three, i.e., suffixes starting at positions that are multiples of 1 and 2. To this end, we interpret three characters as one (increasing the alphabet size) and recursively call this algorithm until all characters are unique. Then, the SA^0 of all other suffixes is computed using the already computed SA^{12}. To obtain the final *SA*, SA^0 and SA^{12} are merged. The algorithm described above is called *DC3*. It can be generalized to other difference covers modulo $X > 3$; then we refer to it as *DCX*. The DCX algorithm can easily be adapted to several models of computation where it also is asymptotically optimal [124]. However, it often impractical due to substantial constant factor overheads, while *induced sorting* algorithms (Section 3.1) are superior, at least in the sequential computations. But the latter are hard to parallelize. Closing this gap between theory and practice is an interesting open problem for algorithm engineering. Note that all but one [20] sequential linear time suffix sorting algorithms rely on recursion. The *SA* can be constructed sequentially with only constant space overhead while retaining a linear running time [97,141]. For more information on sequential suffix sorting, we point to two extensive surveys [28,172] and a practical evaluation [19 SPP].

We now give an overview of suffix sorting algorithms in external memory, in shared memory (briefly touching also GPUs), and in distributed memory. Later, we take a look at the LCP array, one of the arrays often supplementing the *SA*.

External Memory. Crauser and Ferragina [52] and Dementiev et al. [56] present EM prefix doubling algorithms with discarding. The idea of *prefix doubling* [150] is to sort all suffixes based on the *h-order* $\leq_h$, defined by $T[i,n) \leq_h T[j,n) \Leftrightarrow T[i,i+h) \leq T[j,j+h)$ ($=_h$ and $<_h$ are defined analogously). The *h-rank* of a suffix is the number of suffixes that are strictly smaller w.r.t. the *h*-order. Now, during the *k*-th iteration, we compute the 2^k-ranks using the 2^{k-1}-ranks: for all suffixes $T[i,n)$, we use the ranks of $T[i,i+2^{k-1})$ and $T[i+2^{k-1},i+2^k)$, which are known from the previous iteration. We stop when ranks are unique; then, each rank is the position of that suffix in the *SA*. In practice, we can *discard* those *h*-ranks that are unique and not needed to compute other ranks any more, which can speed up the sorting, as it reduces the number of elements that we have to sort. For texts with small alphabets, prefix doubling algorithms are in practice often sped up by *alphabet reduction* in combination with *word packing*, e.g., [32 SPP,56,78 SPP,81]. Here, an alphabet of size σ is first mapped to $[0,\sigma')$ such that $\sigma' \leq \sigma$ each character of the new alphabet occurs at least once in the text and they retain their original order. Then, each character is augmented such that it not only stores i, but also the following $\lfloor b/\lg\sigma' \rfloor$ characters for some suitable bit-width b. This makes sense, for example, when there are unused bits already reserved in the binary representation of the characters, as with DNA ($\sigma' = 4$) stored in bytes ($b = 4$). This allows prefix doubling algorithms to skip the first $\lfloor \lg(\lfloor b/\lg\sigma' \rfloor) \rfloor$ iterations. Dementiev et al. [56] also generalize prefix doubling to α-tupling, i.e., considering α^k-ranks during the $(k+1)$-th

iteration and present experimental results for their implementations. Here, EM DC3 is superior to all prefix doubling/quadrupling ($\alpha = 2$ and $\alpha = 4$) algorithms w.r.t. running time and I/Os. They also show that for small alphabets, DCX can yield further improvements when using difference covers of size 31.

Induced sorting (see [144] for a detailed overview) is another prominent approach for EM suffix sorting. It is also used in the fastest sequential main memory suffix sorting algorithms [19 SPP] that are called SAIS [164] and DivSufSort[6]. This technique has also been generalized to compute the *SA* of collections of strings [145]. The general idea of all EM induced sorting suffix sorting algorithms is to: (1) classify all suffixes into two classes, which can be done in a single scan of the text, (2) sort at most $n/2$ special suffixes, which are suffixes from one of the classes that are (in text order) next to a suffix from the other class, and (3) induce the lexicographical order of all other suffixes using an EM priority queue. The two most prominently used classification schemes are by Itoh and Tanaka [111] and Nong et al. [164]. All following external memory algorithms make use of the latter classification scheme.

Bingmann et al. [31] propose *eSAIS* following the ideas described above. Additionally, eSAIS can also be used to compute the *LCP* array, which we define later in this section. Another EM induced sorting algorithm *DSAIS* is presented by Nong et al. [163]. However, this algorithm assumes that $n = \mathcal{O}\left(M^2/B\right)$, which limits the scalability, as the input size is still bounded by the size of the main memory (it is also not faster in practice than eSAIS [122]). An improved version *DSAIS+* by Wu et al. [206] is reported to be faster than eSAIS and also requires around half the disk space. Another EM induced sorting algorithm, called *fSAIS*, is presented by Kärkkäinen et al. [122]. The fSAIS algorithm introduces multiple improvements compared with eSAIS and DSAIS. First, it uses the classification by Nong et al. [164] but switches the classes when it comes to determining the special class, which resolves some corner cases, because now the last suffix $T[n-1..n)$ cannot be in the special class. Then, a stable priority queue is used, making timestamps to keep track of the order of the induced suffices unnecessary (compared to eSAIS) and thus reducing the I/O volume. Finally, to avoid random access on the text, a simplified *blockwise preinducing* [163] is used, i.e., the text is split into fixed sized blocks and the characters in each block are ordered in the same way they are accessed during the inducing phase. In addition to fewer random access, this makes it unnecessary to store the text positions from which the suffixes is induced. All these improvements halve the I/O volume of the algorithm compared to eSAIS. Han et al. [103] recently presented *nSAIS*, which reduces the I/O volume and required disk space even further.

Another idea for EM suffix sorting is to split the text into consecutive blocks such that the *SA* of the block can be computed in main memory. These *partial SA*s (plus additional information that helps later on) are then merged to obtain the final *SA* [117]. This approach can be parallelized [121] in EM.

[6] Original implementation without publication: `https://github.com/y-256/libdivsufsort` (last accessed 2020-10-18). Fischer and Kurpicz give a detailed description of the algorithm and extend it to also compute the *LCP* array [77 SPP].

Shared Memory and GPGPU. On a PRAM, we are only aware of induced sorting algorithms. Labeit et al. [132] present a parallel implementation of DivSufSort. Lao et al. present a parallel version of SAIS [136] and SACAK [137], the latter being a simplified version of SAIS. Both are faster on repetitive texts than the parallel DivSufSort. An improved parallel SACAK algorithm, by Xie et al. [208], is the fastest algorithm on most inputs (in their evaluation, the parallel DivSufSort is only faster on two of the non-repetitive inputs).

Finally, we also want to mention *SA* construction using graphics cards (general purpose computation on graphics processing unit, GPGPU). Due to the limited amount of memory available on graphics cards, these algorithms do not scale well. The dominant techniques used in GPGPUs are prefix doubling: either heavily relying on prefix sums [195] or using radix sort [169,202]. DCX algorithms have been presented by Deo and Keely [59] and Wang et al. [202] but are outperformed in practice by the prefix doubling approaches. The latter also present a DCX-prefix-doubling hybrid, which is the fastest GPGPU suffix sorting algorithm.

Distributed Memory. In distributed memory, suffix sorting becomes harder than in RAM, as we have to communicate to obtain access to text that is not locally available on a PE; we want to avoid random access on data that is not local. There exist distributed suffix sorting algorithms that are based on merge-sort [128], quicksort [161], and radix sort [2,88]. The DCX algorithm has also been practically evaluated in distributed memory [32 SPP,131,155].[7]

In practice, variants of prefix doubling are most often used, with different implementations of how the new ranks are computed. Kitajima and Navarro [127] presented an early distributed version of Manber and Myers's [150] prefix doubling algorithm, but it requires a lot of bookkeeping. Flick and Aluru's distributed prefix doubling algorithm [81] makes use of the inverse *SA* that is partly computed based on the currently considered h-ranks. A further practical improvement is that the algorithm switches to a different strategy for refining the ranks for small groups of suffixes with the same rank; this reduces communication even further. In addition, this algorithm is the only distributed algorithms that supports the computation of the *LCP* array. Two distributed prefix doubling algorithm are presented by Bingmann et al. [32 SPP]. Those algorithms have been implemented in the Thrill framework [29 SPP], which results in some restrictions regarding the access to the distributed data. The first algorithm makes use of a *window* of size 2^k (in the k-th iteration) to obtain the required rank, whereas the second one is a prefix doubling with *discarding* algorithm. This idea was later revisited and implemented using MPI [78 SPP]. Here, the prefix doubling algorithm and distributed string sorting (see Section 2.2) are used as building blocks for a distributed *induced sorting* suffix sorting algorithm, which is the most memory efficient distributed suffix sorting algorithms currently available, but only works efficiently for small alphabets due to a σ^2-factor in space and the number of synchronization steps.

[7] DC3/7/13 implementation without publication is available at `https://github.com/bingmann/pDCX` (last accessed 2020-09-25).

Longest Common Prefix Array. The *SA* is often accompanied by different arrays containing useful information to speed up different types of queries. One of the most important ones is the *longest common prefix* (*LCP*) array. It contains the lengths of the longest common prefixes of lexicographically consecutive suffixes. More formally, $LCP[0] = 0$ and $LCP[i] = \max\{\ell \geq 0 \colon T[SA[i], SA[i]+\ell) = T[SA[i-1], SA[i-1]+\ell)\}$, see Fig. 2. The *LCP* array can be computed sequentially in linear time [125].

There exist *LCP* array construction algorithms based on prefix doubling in distributed memory [81]. In external memory, the *LCP* array can be constructed while executing eSAIS [31]. Alternatively, it can be computed after the computation of the *SA* [115,116]. This EM computation can also be parallelized [118,119]). In GPGPUs, there exists a parallel version of Kasai et al.'s [125] algorithm [59]. We refer to [183] for a extensive evaluation of different shared memory *LCP* array construction algorithms. The *LCP* array construction has also been generalized to collections of strings [67,145].

3.2 Compressed Full-Text Index

In the following, we consider a space-efficient alternative to the *SA*, the FM-index. We first look at the construction of its two main building blocks, the Burrows-Wheeler transform and the wavelet tree, and then how it can be combined to finally obtain the FM-index.

Burrows-Wheeler Transform. The *Burrows-Wheeler transform* (*BWT*) [42] of a text T of length n is defined by $BWT[i] = T[SA[i]-1 \mod n]$. A different, more verbatim definition of the *BWT* is that we sort the strings $S_0 = T[0]\dots T[n-1], S_1 = T[1]\dots T[n-1]T[0], \dots, S_{n-1} = T[n-1]T[0]\dots T[n-2]$ (the *shifts* of T) lexicographically. Then *BWT* is the last character of each of the shifts, when the shifts are read in lexicographic order. We call this the *naive* approach. See Fig. 3a for an example of the *BWT*. The first definition of the *BWT* can be translated to a simple construction algorithm based on the *SA*—for which we have seen many construction algorithms in different models of computation in Section 3.1. However, there are many algorithms that do not require the computation of the *SA*. In RAM, the best main memory algorithm can compute the *BWT* in time $\mathcal{O}\left(n \lg \sigma / \sqrt{\lg n}\right)$ for alphabets of size $\sigma \leq \sqrt{\lg n}$ [126].

On a PRAM, Hayashi and Taura [104] present a construction algorithm that is based on the divide-and-conquer paradigm. They first recursively split the text into consecutive slices (until the size of a slice falls below a threshold). After that, *partial BWT*s are computed for the slices. These partial *BWT*s are then merged in parallel. To speed up merging, additional information, like *SA* samples and *WT*s, is used. Liu et al. [143] present an algorithm that does not merge the partial *BWT*s directly, but only computes partial *SA*s and merges those. However, unlike Hayashi and Taura, they use a single dedicated PE to merge the partial *SA*s, which are computed by all other PEs. Again, additional information, like the *LCP* array (see Section 3.1), is used. The *BWT* is then obtained using the final *SA*. Fuentes-Sepúlveda et al. [86] present a parallel version of [157] that considers consecutive slices of size $\Delta = \lceil \lg_\sigma n \rceil$ as meta-symbols. The *SA* of the concatenation of S_1 and S_2 (of size $2n/\Delta$) is used to compute a partial *BWT*. Then, all other shifts S_i ($\Delta - 2$ many) are merged (each in parallel) with S_1. Additional information obtained by the merging is used to update the partial *BWT*.

Ohlebusch et al. [167] consider the *reverse BWT* (BWT^{rev}), i.e., the *BWT* of the reverse text $T^{\text{rev}} = T[n-1]T[n-2]\dots T[0]$ that is of interest for short read mapping (cf. Section 4). The sequential version of the algorithm makes use of the wavelet tree of the *BWT* of the text, the *SA*, and the text itself. This leads to independent intervals in BWT^{rev} that can easily be computed in parallel. Gilchrist and Cuhadar [93] show that for many applications (cf. Section 4.2), the *BWT* is only required for slices of the text. The *BWT* construction for independent slices of the text is easy to parallelize.

Menon et al. [153] give a distributed *BWT* construction algorithm based on MapReduce. Another distributed algorithm based on merging is presented by Wang et al. [201]. This algorithm is tuned for large collection of DNA reads and first partitions the text with respect to a common prefix, and then computing the *BWT* for partitions with a common prefix—similar to the domain decomposition for wavelet trees (cf. Section 3.2). Ferragina et al. [72] present an EM version of [104] that is based on merging. Also in EM, *prefix free parsing* [40,130] is used, which is a technique similar to the one used for the asymptotically best sequential *BWT* construction algorithm [126]. The naive *BWT* construction has also been parallelized with FPGAs by Trinidad et al. [198]. Here, the disadvantage is that we actually have to store all shifts S_i. This approach is also considered on GPGPUs by Patel et al. [170].

As with *SA* and the *LCP* array, the *BWT* has also been generalized for a collection of strings and there exist external memory algorithms for its construction [67,145].

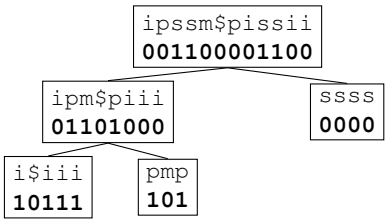
Wavelet Trees. For our compressed full-text index, we need to answer generalized rank and select queries (see Section 2.2) on the *BWT* efficiently. The *wavelet tree* (*WT*), introduced by Grossi et al. [99], is a binary tree data structure that allows answering both queries in time $\mathcal{O}(\lg\sigma)$ and can be stored in $n\lg\sigma + o(n)$ bits of memory. Each node of the tree represents an interval $[a,b] \subseteq \Sigma$ and is labeled by a bit vector that contains one bit for each text position i, in text order, where $T[i] \in [a,b]$. The bit is set iff $T[i] > \lfloor (a+b)/2 \rfloor$. The root node represents the entire alphabet $[a,b] = \Sigma$ and is therefore labeled by n bits, corresponding to the entire text T. A node has two children if $|[a,b]| \geq 2$. Then, recursively, the left child represents the interval $[a, \lfloor (a+b)/2 \rfloor]$, and the right child represents $[\lfloor (a+b)/2 \rfloor + 1, b]$. Finally, the leaves represent intervals of size one or two. Because the alphabet is split in two halves at every node, the tree has height $\lceil \lg\sigma \rceil$. Fig. 3b shows an example.

Instead of comparing a character to the interval's middle to determine its bit in a node, it is more common to look at the $\lceil \lg\sigma \rceil$ bits of the characters' binary representations, starting with the most significant bit. Each bit tells whether to go left (zero) or right (one), i.e., characters encode a path down the *WT* starting from the root. In that regard, different codes can be used. A prominent example for using a code other than binary is the *Huffman-shaped WT*, which is constructed based on the characters' canonical Huffman codes. The bit vectors labelling the nodes then require only as much space as the Huffman-compressed text.

Apart from text indexing, the *WT* has applications in more areas, as described in various surveys on the topic [73,100,149,160]. An alternative representation of the *WT*—the wavelet *matrix* (*WM*)—introduced by Claude et al. [47], is a more efficient choice when dealing with large alphabets. It only requires negligible extra space compared

	0	1	2	3	4	5	6	7	8	9	10	11
T	m	i	s	s	i	s	s	i	p	p	i	$
SA	11	10	7	4	1	0	9	8	6	3	5	2
BWT	i	p	s	s	m	$	p	i	s	s	i	i

(a) Burrows-Wheeler transform.



(b) Wavelet Tree.

α	code
$	000
i	001
m	010
p	011
s	100

(c) Binary representation.

Fig. 2: Burrows-Wheeler transform of the text $T = \texttt{mississippi\$}$ in (a). In (b), we show the wavelet tree (assuming $\sigma = 8$) of the Burrows-Wheeler transform depicted in (a). The binary representation of the characters is given in (c). Together, the Burrows-Wheeler transform and the wavelet tree are the FM-index, which we briefly describe in Section 3.2.

to the *WT* and can be used to answer the same queries in the same asymptotic time. However, when answering queries, fewer constant-time binary rank queries are needed on the bit vectors than in the *WT*, making it faster in practice. The similarities and differences between *WT* and *WM* are studied in more in detail by Dinklage [61]. The remainder of this section focuses on algorithms to construct the *WT* in the computational models introduced in Section 2.1. We will refer to *levels* of the *WT*, where level ℓ describes the set of nodes with depth ℓ.

We first consider *sequential* construction algorithms. There are various improvements to naïve algorithms to construct the *WT*: Claude et al. [48] and Tischler [197] give the most space-efficient algorithms using only $\mathcal{O}(\lg n)$ bits, but do not provide a competitive implementation. Da Fonseca and da Silva [84] give an *online* construction algorithm, i.e., one where no prior knowledge of the input alphabet is required, that runs in time $\mathcal{O}(n \lg \sigma)$ and uses $n\lceil \lg \sigma \rceil + o(n \lg \sigma)$ bits of space. The fastest known algorithms in theory require time $\mathcal{O}\left(n \lg \sigma / \sqrt{\lg n}\right)$ and were given by Babenko et al. [18] and Munro et al. [158]. The latter was implemented by Kaneta [114], proving that the use of modern CPU instructions can reflect theoretical improvements also in practice. Kaneta's results are competitive with the currently known fastest and most space-efficient algorithm to construct the *WT*, which has been developed by Fischer et al. [79 SPP]: it is based on *prefix counting* and, except for the topmost level, constructs the *WT* bottom-up as described in the following. In a first scan of *T*, we compute the

histogram of T, i.e., the frequencies of all characters, as well as the topmost level of the *WT*, which consists of the characters' most significant bits in the same order in which they occur in T. For each remaining level $\ell \in [2, \lceil \lg \sigma \rceil)$, starting with the bottommost level, we first compute the histogram of T. This is done by combining the frequencies of every pair in the previous histogram: because a node combines the two intervals of the alphabet represented by its children, the total frequency of its represented characters is the sum of the respective frequencies of its children. The histogram for level ℓ allows us to easily compute the positions of the first bit for every node on level ℓ. In one scan of T, we can then compute the bits for all nodes on level ℓ and directly write them to the correct positions. The algorithm requires total time $\mathcal{O}(n \lg \sigma)$ and $\sigma \lceil \lg n \rceil$ bits of space in addition to the input and output. The same technique can be used to construct the Huffman-shaped *WT*, where it also yields the best practical results in terms of speed and space usage.

We now regard the *parallel WT* construction in the *shared memory* model. Labeit et al. [132] gave a recursive algorithm based on the *parallel split* operation. Here, the available PEs process T in parallel to compute the bits for the root node. These bits are then used to perform a parallel split of T for the left and right child, which are recursively processed in parallel. The number of PEs used to process each child is proportional to the sizes of the children. Two further techniques for parallel *WT* construction stand out: *domain decomposition* and an algorithm based on *sorting*. The use of domain decomposition for *WT* construction has first been proposed by Sepúlveda et al. [85]. The input T is partitioned such that every PE receives a slice of size n/p. and computes the entire *WT* for its slice using any sequential algorithm, e.g., prefix counting. In a subsequent step, these *WT* are merged into the *WT* for T, which can be done efficiently by concatenating the bit vectors contained in the corresponding nodes. Because an arbitrary sequential construction algorithm can be used locally, domain decomposition can be tuned to have a very low memory footprint. The algorithm based on sorting, first proposed by Shun [184], constructs the *WT* top-down, level by level, and makes use of stable integer sorters, which are well studied for all practically relevant computational models. The bits of the topmost level can be computed in an initial parallel scan of T, similar to the (sequential) prefix counting algorithm. Then, before proceeding to some level $\ell > 1$, the text is reordered by stably sorting the characters according to their ℓ-bit prefixes, which puts them in the correct positions to compute that level's bit vector in a parallel scan of the reordered text. To that end, the algorithm only requires $\lceil \lg \sigma \rceil$ parallel scans of T. For both algorithms, Shun presents techniques that allow for different trade-offs between work and time [185]. The best known implementations were given by Fischer et al. [79 SPP], concluding that domain decomposition is the fastest approach in practice, also for constructing the Huffman-shaped *WT*.

The parallel construction in *distributed memory* has been studied by Dinklage et al. [64 SPP], confirming the practical relevance of domain decomposition, which yields the fastest running times and best memory efficiency in practice. An important measure for distributed memory algorithms is the communication volume. During the distributed domain decomposition, only the merging phase requires communication between the PEs. They also adapted Shun's parallel sorting algorithm [184] to distributed memory and achieved nearly as good running times, albeit requiring more communication. Be-

cause the individually constructed levels need not be partitioned into nodes, the sorting algorithm has furthermore been found to be better suited than domain decomposition for constructing the *WT* for large alphabets.

Finally, we look at *WT* construction in *external memory*. Ellert and Kurpicz [69 SPP] present sequential and parallel external memory algorithms. The sequential algorithm is based on sorting and works similar to the corresponding parallel algorithm. Using only a constant amount of main memory, it requires two scans of T for each level of the *WT*. They also provide various semi-external algorithms with similar properties, all of which outperform the semi-external *WT* construction algorithms from the Succinct Data Structure Library (SDSL) [94]. Finally, their parallel algorithm makes use of domain decomposition to distribute work on the available PEs, each PE using a sequential in-memory algorithm (e.g., prefix counting) to construct a partial *WT*. Because the p parts of T may not fit into main memory, each PE furthermore partitions its part into segments of size k such that a segment and its *WT* does fit in main memory. They then process their part segment by segment. The algorithm requires four scans over T for each level, plus σ random I/O operations for each segment. Naturally, because of the necessary synchronizations with external memory, the algorithm only scales well up to a limited number of PEs. Yet, the parallelization achieves a notable speedup in practice.

FM-Index. The FM-index [74] combines the *BWT* and (Huffman shaped) *WT*s to a compressed full-text index. It is widely used, in particular in most DNA read aligners [134] and in Bioinformatics in general (cf. Section 4.1).

To *locate* a pattern using the FM-index, a *backward search* is performed. Using the C array (for each $\alpha \in \Sigma$, $C[\alpha]$ is the overall number of occurrences of characters in *BWT* that are strictly smaller than α, i.e., the rank of α in Σ) and the *WT* of the *BWT* to answer $rank_\alpha(i)$ (on the *BWT*, cf. Fig. 2) it is possible to search backwards for a pattern in T [74]: Given an ω-interval $[i,j]$ (i.e., ω is a prefix of $T[SA[k]..n)$ if and only if $i \leq k \leq j$) and $\alpha \in \Sigma$, the procedure called $backwardSearch(\alpha,[i,j])$ returns the $\alpha\omega$-interval $[lb,rb]$, where $lb = C[\alpha] + rank_\alpha(i) + 1$ and $rb = C[\alpha] + rank_\alpha(j+1)$. If $lb > rb$, the pattern does not occur in T.

Note that any combination of *BWT* and *WT* construction algorithms can be combined to compute the FM-index (in any model of computation). Still, there exist dedicated practical PRAM FM-index construction algorithms by Labeit et al. [132] and Lio et al. [143]. The former combines their parallel *SA* (see Section 3.1) and *WT* (see Section 3.2) construction algorithms to compute an FM-index (in parallel), whereas the latter provides a parallel algorithm that computes both the *BWT* and the FM-index.

3.3 Suffix Trees

A suffix tree (*ST*) for a string of length n is a compact trie storing all the suffixes of T, i.e., the concatenation of the edge labels on the path from the root to leaf i exactly spells out the suffix $T[i..n)$; see Fig. 3a for an example. Weiner [204] showed that it can be constructed in linear time provided that the underlying alphabet has constant size. Farach-Colton et al. [71] gave the first suffix tree construction algorithm that is optimal

for all alphabets. It has linear run-time for alphabets consisting of integers in a polynomial range. The *ST* is one of the most powerful data structures in string processing, with applications in fields like bioinformatics or information retrieval, e.g., [21,45].

Abouelhoda et al. [3] showed that there is a one-to-one correspondence between the set of all lcp-intervals and the set of all internal nodes of the *ST* of T. Let us define the concept of lcp-intervals (see Fig. 3a). An interval $[i, j]$ in the *LCP* array—for simplicity, we now assume that $LCP[0] = -1 = LCP[n]$—is called an *lcp-interval of lcp-value* ℓ if (1) $LCP[i] < \ell$, (2) $LCP[k] \geq \ell$ for all k with $i < k \leq j$, (3) $LCP[k] = \ell$ for at least one k with $i < k \leq j$, and (4) $LCP[j+1] < \ell$. Every index k ($i < k \leq j$) with $LCP[k] = \ell$ is called *ℓ-index* or *lcp-index*. A leaf in the *ST* corresponds to a *singleton interval* $[k,k]$. The parent interval of an lcp-interval $[i, j]$ (or a singleton interval) is the smallest lcp-interval that contains $[i, j]$ but does not coincide with $[i, j]$.

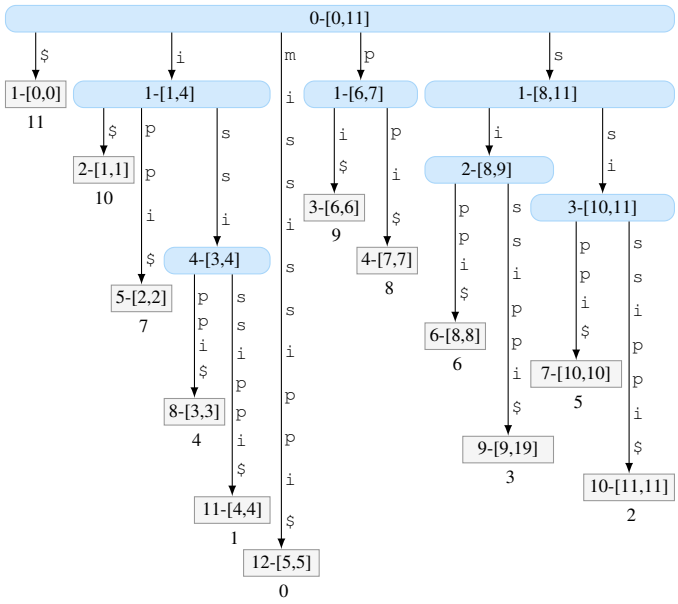
The drawback of *ST*s is their huge space consumption: even carefully engineered implementations require 8–20 bytes per input character. It is possible to save a lot of space by representing the *ST* topology by a sequence of balanced parentheses. The sequence BPS, for instance, can be constructed by a depth first search traversal of the (uncompressed) *ST* as follows. At each node v (starting at the root), write an opening parenthesis, recursively process the child nodes of v, and write a closing parenthesis afterwards (see Fig. 3b). Since the *ST* has n leaves and up to $n-1$ internal nodes, the BPS needs up to $4n-2$ bits. Based on the BPS, all navigational operations on the *ST* can be supported with data structures that require only $o(n)$ bits [177].

The BPS can be constructed in parallel on a shared memory architecture in the CRCW PRAM model with the help of the *LCP* array as follows; see [22] for details, where also the necessary adjustments for the CREW model are explained. Create two arrays C_o and C_c of size n, enumerate all lcp-intervals in parallel, and increment $C_o[i]$ and $C_c[j]$ for each lcp-interval $[i, j]$. After that, compute the prefix sum *PS* of $sum[i+1] = C_o[i] + C_c[i]$, and write $C_o[i]$ opening followed by $C_c[i]$ closing parenthesis at position $PS[i]$ into the bitvector BPS (in parallel). It is possible to enumerate all lcp-intervals (in parallel) with the help of the arrays *PSV* (previous smaller value) and *NSV* (next smaller value), which are defined as follows:

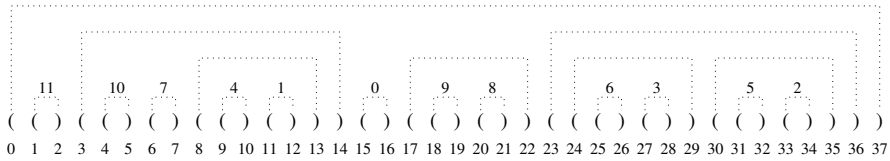
$$PSV[i] = \max\{j \mid 0 \leq j < i \text{ and } LCP[j] < LCP[i]\}$$
$$NSV[i] = \min\{j \mid i < j \leq n \text{ and } LCP[j] < LCP[i]\}$$

Table 1 shows an example (an entry $\perp$ means that the value is undefined). The key observation is that for any index i with $0 < i < n$ and $LCP[i] = \ell$ the interval $[PSV[i], NSV[i] - 1]$ is an lcp-interval of lcp-value ℓ and i is one of its lcp-indices; for a proof see, e.g., [166, Lemma 4.3.8]. A problem of this approach is that an lcp-interval with multiple ℓ-indices will occur more than once in the enumeration. To overcome this problem, such an interval is reported if and only if i is the first (leftmost) ℓ-index of the interval. To this end, previous smaller values (*PSV*) are replaced with previous smaller or equal values (*PSEV*), where the array *PSEV* is defined by $PSEV[i] = \max\{j \mid 0 \leq j < i \text{ and } LCP[j] \leq LCP[i]\}$. Then $[PSV[i], NSV[i] - 1]$ appears in the enumeration if and only if $LCP[i] \neq LCP[PSEV[i]]$.

The problem of computing previous smaller and next smaller values, also known as the all-nearest-smaller-value problem (ANSV), was already solved by Berkman et al.



(a) Suffix tree.



(b) BPS of the suffix tree.

Fig. 3: In (a), the suffix tree for T = `mississippi$` (an annotation ℓ-$[i,j]$ within a node shows the corresponding lcp-interval, i.e., ℓ is the string-depth of the node and $[i,j]$ is the corresponding interval). The number below the leafs is the starting position of the corresponding suffix. For the corresponding *SA* and *LCP* array see Fig. 2. The BPS of the suffix tree is shown in (b). Matching parentheses are connected by dotted lines. A leaf in the suffix tree is represented by an opening parenthesis that is immediately followed by a closing parenthesis; its leaf number (leaf i represents suffix $T[i..n)$) is depicted above the two parentheses. The last row shows the positions of the parentheses in the BPS.

[27] with $\mathcal{O}(n)$ work and in $\mathcal{O}(\log\log n)$ time using $\mathcal{O}(n/\log\log n)$ processors on a CRCW PRAM.

In the following, we will focus on distributed memory. He and Huang [105] presented a bulk-synchronous parallel adaption of the algorithm invented by Berkman et al. [27]. Flick and Aluru [82] improved their work in various directions by introducing a generalized version of the ANSV problem. They showed how to handle duplicate val-

Table 1: The *LCP* array with $LCP[0] = -1 = LCP[n]$ for $T =$ `mississippi$` (cf. *LCP* array in Fig. 2) and the corresponding arrays *NSV*, *PSV*, *PSEV*, and *PFE*.

	0	1	2	3	4	5	6	7	8	9	10	11	12
LCP	−1	0	1	1	4	0	0	1	0	2	1	3	−1
NSV	⊥	12	5	5	5	12	12	8	12	10	12	12	⊥
PSV	⊥	0	1	1	3	0	0	6	0	8	8	10	⊥
PSEV	⊥	0	1	2	3	1	5	6	6	8	8	10	⊥
PFE	⊥	⊥	1	2	2	1	1	1	1	1	1	10	⊥

ues, generalized the communication structure, and provided novel proofs. Based on the improvements on the ANSV problem, they presented a parallel *ST* construction algorithm using the suffix- and *LCP* array that runs in $\mathcal{O}(n/p+p)$ time, which is work optimal for $p = \mathcal{O}(\sqrt{n})$. In a first phase, they represent the *ST* as an array *E* of edges $(i, parent(i))$. This approach requires a unique representative index for each node *v* in the *ST*. Since *v* corresponds to an lcp-interval, one can choose the first (leftmost) lcp-index of that lcp-interval as a representative. Moreover, Lemma 1 shows that the representative of the parent interval can be computed with the help of "previous-furthest-equal" values, defined for all *i* with $1 < i < n$ as follows:

$$PFE[i] = \min\{j \mid PSV[k] < j < i \text{ and } LCP[j] = LCP[k], \text{ where } k = PSEV[LCP[i]]\}$$

Lemma 1. *Recall that, for any index i with $1 < i < n$, the interval $[lb, rb]$, where $lb = PSV[i]$ and $rb = NSV[i] - 1$, is an lcp-interval and i is an lcp-index of $[lb, rb]$. In the following, let $m = PFE[i]$. If $LCP[m] = LCP[i]$, then m is the representative lcp-index and i is a different lcp-index of $[lb, rb]$. From now on we assume $LCP[m] < LCP[i]$. In this case, we have $LCP[m] = LCP[PSV[i]]$. If $LCP[PSV[i]] < LCP[NSV[i]]$, then $NSV[i]$ is the representative lcp-index of the parent interval of $[lb, rb]$; see [166, Lemma 4.3.9]. Otherwise, $PSV[i]$ is an lcp-index of the parent interval of $[lb, rb]$ and m is the representative lcp-index of that parent.*

Flick and Aluru's algorithm assumes that all inputs are distributed equally across processors with n/p elements per process. It computes *PFE* and *NSV* in $\mathcal{O}(n/p+p)$ time. Since the processor for the range $[\frac{n}{p}j, \frac{n}{p}(j+1)-1]$ has the corresponding portions of *LCP*, *PFE*, and *NSV* in local memory, it can compute edges $(i, parent(i))$ in its range based on Lemma 1. The parents of leaf nodes in its range can be computed similarly; see [82] for details. In the second phase of their algorithm, Flick and Aluru show how edges can be inverted (and analyse the communication complexity). This is because for pattern matching applications, instead of having parent pointers, each internal node should point to its children. Other algorithms for distributed *ST* include [44,50,212].

3.4 Query Answering

Up to this point, we only have considered the construction of different full-text indices. Since all full-text indices that we have looked at have their origin in RAM, they can

easily be used there by allocating the incoming queries in a round robin fashion to the PEs. However, in external or distributed memory, the obstacle is that neither the whole text nor the whole index can be accessed in a random access manner, as in the construction algorithms. In this section, we take a look at different approaches to answer queries in such a setting.

Clifford [49] show how to use a suffix tree in distributed memory to answer different types of queries. They build the suffix tree using Ukkonen's algorithm [199]. For this purpose, the whole text is required at each PE, limiting the scalability of this approach significantly.

Mäkinen et al. [148] use the *compressed* suffix array (CSA) [176] in distributed and external memory. The CSA requires roughly the same space as the *compressed* text but also does not need the text to answer queries (unlike the *SA*); it is a *self-index*. In main memory, queries of length m can be answered in $\mathcal{O}(m \lg n)$ time. They improve query times by sampling ℓ-length strings instead of characters and encoding the supporting data structures using Elias delta encoding in combination with lookup tables. This allow for constant time access to the supporting data structures (not queries). In EM, their approach can search for a pattern of length m in $\mathcal{O}(m \lg_B n)$ I/Os (which can be reduced to $\mathcal{O}((m \lg n)/B)$ if $\mathcal{O}(n)$ bits can be stored in main memory). In distributed memory, m supersteps are required to answer such a query. During each superstep only a constant number of words have to be communicated and $\mathcal{O}(\lg n)$ local work is required.

Arroyuelo et al. [10] compare different layouts of the *SA* for pattern matching in distributed memory. When each PE holds a consecutive slice of the *SA*, we have the *global* layout. In addition to the *SA*, pruned suffixes are stored to speed-up querying at the local PE. To speed up queries, a trie for the suffixes at the beginning and end of each slice is built at each PE in order to distribute queries to the PE that can answer it locally. Next, in the *local* layout, each PE holds a consecutive slice of the text and builds a *SA* only for this local slice. Here, each PE must answer the query locally and return the result, requiring only a constant number of supersteps but significant local work (as all PEs always have to search for the query). The *multiplex* layout is an intermingled global layout, where the i-th entry of the global *SA* is stored at PE $i \mod p$ in consecutive fashion, i.e., the i-th and $(i+p)$-th entry are stored consecutively at the same PE. Corresponding pruned suffixes are stored as in the global layout. The multiplex layout (and in some cases the global layout) is the most efficient one in their experiments. They also propose two additional layouts that, however, perform not as well in practice.

The global layout is extended by Fischer et al. [80 SPP]. Instead of answering the query directly on the *SA*, a Patricia trie [156] is constructed for each local slice. To this end, the *LCP* array (see Section 3.1) is required. Furthermore, a global trie is used to distribute the query to the corresponding PE. These two tries together allow queries to be answered with a constant number of supersteps.

Flick and Aluru [83] further improve the above two-level designs by developing the *distributed enhanced SA* (DESA). One improvement of DESA is to eliminate the explicitly stored tree structure of the two-level indices. Also, the DESA does not partition the text into consecutive slices of the same size (where queries may have to be answered on multiple PEs) but partitions the text into more fine grained intervals, such that all intervals can be processed on a single PE. This approach currently scales best in practice.

The local search is an adapted version of Fischer and Heun's [75] query algorithm for enhanced *SA*s [3]. Hence, they only need the *SA*, *LCP* array, *range minimum queries* (RMQs, returning the positions of the smallest element in a given range), and some additional information. To help load balancing queries, the top level trie is dynamically (based in the input) constructed such that each bottom level index, corresponding to a leaf in the top level trie, covers intervals of size n/cp for a constant c. To this end, the ANSV problem (cf. Section 3.3) is solved. This approach is significantly better than a static top level lookup table and overall the best in practice.

4 Applications

All previously described text indices have more applications than (exact) pattern matching. They can also be used to answer approximate queries, i.e., when allowing differences between the pattern and the matched positions. Pockrandt [171] shows how to transform those queries into exact queries. This is also used in practice in the SeqAn library [175], which contains efficient algorithms and data structures for the analysis of biological data (and strings in general). Furthermore, they can be used to compute succinct de Bruijn graphs, all pairs suffix–prefix overlaps, and maximal repeats [67]. In the following, we take a more detailed look into two fields where text indices are of great importance—*Bioinformatics* (Section 4.1) and *lossless compression* (Section 4.2).

4.1 Bioinformatics

The most successful application of index structures in bioinformatics is backward search based on an FM-index [74] (e.g., in form of the *WT* of the *BWT* of the input string [99], cf. Section 3.2). For information on k-mer-based tools, we refer to the recent survey by Marchet et al. [151].

The most important application of backward search in bioinformatics is read mapping. Ultra-high-throughput next-generation sequencing technologies (NGS) have been commercially available since 2005. In NGS, DNA is fragmented into small pieces, of which the first few bases are sequenced, yielding several millions of short "reads", each 30 to 400 base pairs ("DNA characters") long. The read mapping task is now to align these reads to a reference genome, i.e., to the known, nearly complete chromosomal DNA sequences of the organism in question (which may be up to several billion base pairs long); see [43] for an overview article.

Short read mappers like Bowtie [135] or BWA [140] must be able to deal with (sequencing) errors. Inexact matching is either based on recursive algorithms that use backtracking or on the seed-and-extend strategy (exact matches are used as seeds and the shared seeds are then extended into longer, inexact alignments). The same approach has also been successfully applied in genome assembly [187] (sequence assembly refers to aligning and merging reads in order to reconstruct the original sequence). Here, the fastest implementations only utilize a few threads [24]. Those read mappers usually do not use parallel construction algorithms, as the reference sequences are short, allowing a space-efficient sequential algorithm to compute the index in less than an hour.

Alignments of longer sequences (ranging from long read mapping to whole genome alignment) are also obtained by exact matching and the seed-and-extend method. One of the earliest tools in comparative genomics is based on suffix trees (and later on suffix arrays) [55], but there are also tools using the *BWT* [142]. The major principle of comparative genomics is that common features of two organisms will often be encoded within the DNA that is evolutionarily conserved between them. Therefore, comparative genomic approaches start with making some form of alignment of genome sequences. Then, they look for orthologous sequences (sequences that share a common ancestry) in the aligned genomes and check to what extent those sequences are conserved. Nowadays, one tries to take multiple genomes simultaneously into account; see [51] for an overview of pangenomics. When it comes to the alignment of longer sequences, scaling algorithms are used, e.g., multithreaded semi-external prefix-doubling algorithms [190] or building multiple partial indices (in parallel) and merging them [191]. Compressed suffix trees and FM-indices have been used in indexing variation graphs [92] and for graphical pangenome analysis [21]. In particular, the balanced parentheses sequence BPS from Section 3.3 was used for indexing variation graphs [190] (using the algorithm described in [168]). Using a dynamic FM-index, sequences can be inserted in batches, which can easily be parallelized [139].

4.2 Compression

Text indices have been successfully applied to text compression, most notably to compressors based on the *BWT* (see Section 3.2) and on different variants of the Lempel-Ziv parsing of the text. Intuitively, this link between indexing and compression seems plausible, as in both cases one tries to 'group' similar substrings; in the former for listing occurrences, in the latter for exploiting the repetitiveness to somehow save space. We only consider compressors that operate over the *full* text (*not* restricted to small sliding windows/blocks); this is important for highly repetitive texts such as DNA collections of individuals from the same species.

Lempel-Ziv in External Memory. The LZ77-factorization [213] of a text T is defined as follows: suppose $T[0..i)$ has already been parsed into LZ77-phrases. Then the next LZ77-phrase is the longest prefix of $T[i..n)$ that has an occurrence in T starting strictly before i (but possibly ending in $T[i..n)$), or a single character if $T[i]$ does not occur before. Given a text index on T, this prefix can be located by iteratively querying for $T[i..i+1)$, $T[i..i+2)$, ..., as long as an occurrence starting before i exists. In main memory, Fischer et al. [76] have the most space efficient implementation of this idea using compressed variants of the suffix tree, needing only $(1+\varepsilon)n\log n + \mathcal{O}(n)$ bits of space and running in $\mathcal{O}(n/\varepsilon)$ time. The difficulty in EM is, of course, that such repeated querying causes too many I/Os. Kärkkäinen et al. [120] avoid this in two ways: in their EM-LPF algorithm, they first compute the array of *longest previous factors* in EM, from which the LZ77 factorization is easily obtained in total $\mathrm{sort}(n)$ I/Os. Their second algorithms, EM-LZScan, divides T into blocks of size $\Theta(M)$ and then computes *matching statistics* [166, Sect. 5.5.4] of the current block w.r.t. the prefix of T up to the current block. EM-LZScan needs $\mathcal{O}\left(\frac{n^2 \log \sigma}{BM \log n}\right)$ I/Os in theory, but is significantly faster

in practice than EM-LPF for highly repetitive texts. A different approach was taken by Dinklage et al. [62 SPP], who show that the flexibility of allowing factor occurrences also be to the *right* of their starting position (so-called *bidirectional parsings*) leads to a much better throughput than EM-LZScan, while achieving similar compression rates. Their algorithm plcpcomp has been successfully applied to texts of size 128 GiB on a machine with just 16 GiB of RAM. Considering *de*compression, Belazzougui et al. [25] show the I/O-complexity to be $\mathrm{sort}(n/\log_\sigma n)$ I/Os and also give a practical implementation; however, this algorithm cannot be applied to the bidirectional variant, which is much slower at decompression. Other variants of LZ exist, but have so far not been successfully applied to large datasets, although promising approaches exist for LZ78 that might lead to semi-external solutions [11].

Parallel Burrows-Wheeler-Based Compression. In Section 3.2, we already mentioned the relevant literature for computing the *BWT L*. This output can be postprocessed to compute a compressed version of T, as characters following a similar preceding context are grouped in L. The postprocessing consists of computing the move-to-front numbers when processing L from left to right, followed by a Huffman encoding of the resulting numbers. On the PRAM, Edwards and Vishkin [66] show how to perform those latter steps in $\mathcal{O}(\log n)$ parallel time and $\mathcal{O}(n)$ work, and report good speedups on FPGA-hardware over popular tools such as bzip2, although only using moderately-sized inputs. They also show how to decompress the resulting file within the same complexities. At their core, the algorithms are reduced to the building blocks prefix sums (cf. Section 2.2) and list ranking. Geared more towards practice, Patel et al. [170] have similar ideas and show GPGPU implementations; however, they use mergesort for computing the *BWT* and report this as their main bottleneck.

We are not aware of any algorithms in external or distributed memory implementing the full *BWT* compression pipeline, despite that algorithms for computing the *BWT* exist in these models of computation (see Section 3.2).

5 Conclusion and Future Work

Advanced text index data structures such as suffix trees, suffix arrays and wavelet trees are key to handling large data sets in a range of important applications. A combination of parallel, external, and compressed implementations can approach the requirements for handling the exploding amounts of available data.

In this short survey, we have discussed a number of techniques for building and using such data structures. Our impression is that memory hierarchies and compression by themselves are fairly well understood by now. A range of parallelization approaches are known but they suffer from a tradeoff between asymptotic scalability and efficiency. In particular, the most efficient sequential and external techniques are inherently sequential. Hence, a number of important open problems remain. These involve highly scalable techniques with good constant factors (e.g., for constructing suffix arrays and *LCP* arrays with linear work) as well as integration of parallelism, memory hierarchies, compression and applications. Another interesting research direction is to engineer recent text indices for highly repetitive data [91] for handling large texts. In principle,

big data frameworks such as Thrill [29 SPP] can handle parallelization and memory hierarchies automatically but the question remains whether the involved overheads are acceptable.

Acknowledgements. This preprint has not undergone peer review (when applicable) or any post-submission improvements or corrections.

References

1. Abali, B., Özgüner, F., Bataineh, A.: Balanced parallel sort on hypercube multiprocessors. IEEE Trans. Parallel Distrib. Syst. **4**(5), 572–581 (1993). https://doi.org/10.1109/71.224220
2. Abdelhadi, A., Kandil, A.H., Abouelhoda, M.: Cloud-based parallel suffix array construction based on MPI. In: MECBME. pp. 334–337 (2014). https://doi.org/10.1109/MECBME.2014.6783271
3. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms **2**(1), 53–86 (2004). https://doi.org/10.1016/S1570-8667(03)00065-0
4. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988). https://doi.org/10.1145/48529.48535
5. Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M.J., Schelter, S., Höger, M., Tzoumas, K., Warneke, D.: The stratosphere platform for big data analytics. VLDB J. **23**(6), 939–964 (2014). https://doi.org/10.1007/s00778-014-0357-y
6. Amarasinghe, S., Campbell, D., Carlson, W., Chien, A., Dally, W., Elnohazy, E., Hall, M., Harrison, R., Harrod, W., Hill, K., et al.: Exascale software study: Software challenges in extreme scale systems. DARPA IPTO, Air Force Research Labs, Tech. Rep pp. 1–153 (2009)
7. Arge, L., Ferragina, P., Grossi, R., Vitter, J.S.: On sorting strings in external memory (extended abstract). In: STOC. pp. 540–548. ACM (1997). https://doi.org/10.1145/258533.258647
8. Arge, L., Procopiuc, O., Vitter, J.S.: Implementing i/o-efficient data structures using TPIE. In: ESA. pp. 88–100. Springer (2002). https://doi.org/10.1007/3-540-45749-6_12
9. Arge, L., Rav, M., Svendsen, S.C., Truelsen, J.: External memory pipelining made easy with TPIE. In: BigData. pp. 319–324. IEEE Computer Society (2017). https://doi.org/10.1109/BigData.2017.8257940
10. Arroyuelo, D., Bonacic, C., Costa, V.G., Marín, M., Navarro, G.: Distributed text search using suffix arrays. Parallel Comput. **40**(9), 471–495 (2014). https://doi.org/10.1016/j.parco.2014.06.007
11. Arroyuelo, D., Cánovas, R., Navarro, G., Raman, R.: LZ78 compression in low main memory space. In: SPIRE. pp. 38–50. Springer (2017). https://doi.org/10.1007/978-3-319-67428-5_4
12. Aumüller, M., Dietzfelbinger, M.: Optimal partitioning for dual-pivot quicksort. ACM Trans. Algorithms **12**(2), 18:1–18:36 (2016). https://doi.org/10.1145/2743020
13. Axtmann, M.: Robust Scalable Sorting. Ph.D. thesis, Karlsruhe Institute of Technology, Germany (2021). https://doi.org/10.5445/IR/1000136621

14 SPP. Axtmann, M., Bingmann, T., Sanders, P., Schulz, C.: Practical massively parallel sorting. In: SPAA. pp. 13–23. ACM (2015). https://doi.org/10.1145/2755573.2755595

15. Axtmann, M., Sanders, P.: Robust massively parallel sorting. In: ALENEX. pp. 83–97. SIAM (2017). https://doi.org/10.1137/1.9781611974768.7
16. Axtmann, M., Wiebigke, A., Sanders, P.: Lightweight MPI communicators with applications to perfectly balanced quicksort. In: IPDPS. pp. 254–265. IEEE Computer Society (2018). https://doi.org/10.1109/IPDPS.2018.00035
17. Axtmann, M., Witt, S., Ferizovic, D., Sanders, P.: Engineering in-place (shared-memory) sorting algorithms. ACM Trans. Parallel Comput. **9**(1), 2:1–2:62 (2022). https://doi.org/10.1145/3505286
18. Babenko, M.A., Gawrychowski, P., Kociumaka, T., Starikovskaya, T.: Wavelet trees meet suffix trees. In: SODA. pp. 572–591. SIAM (2015). https://doi.org/10.1137/1.9781611973730.39
19 SPP. Bahne, J., Bertram, N., Böcker, M., Bode, J., Fischer, J., Foot, H., Grieskamp, F., Kurpicz, F., Löbel, M., Magiera, O., Pink, R., Piper, D., Poeplau, C.: Sacabench: Benchmarking suffix array construction. In: SPIRE. pp. 407–416. Springer (2019). https://doi.org/10.1007/978-3-030-32686-9_29
20. Baier, U.: Linear-time suffix sorting - A new approach for suffix array construction. In: CPM. pp. 23:1–23:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.CPM.2016.23
21. Baier, U., Beller, T., Ohlebusch, E.: Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform. Bioinform. **32**(4), 497–504 (2016). https://doi.org/10.1093/bioinformatics/btv603
22. Baier, U., Beller, T., Ohlebusch, E.: Space-efficient parallel construction of succinct representations of suffix tree topologies. ACM J. Exp. Algorithmics **22** (2017). https://doi.org/10.1145/3035540
23. Barsky, M., Stege, U., Thomo, A.: A survey of practical algorithms for suffix tree construction in external memory. Softw. Pract. Exp. **40**(11), 965–988 (2010). https://doi.org/10.1002/spe.960
24. Bauer, M.J., Cox, A.J., Rosone, G.: Lightweight algorithms for constructing and inverting the BWT of string collections. Theor. Comput. Sci. **483**, 134–148 (2013). https://doi.org/10.1016/j.tcs.2012.02.002
25. Belazzougui, D., Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Lempel-ziv decoding in external memory. In: SEA. pp. 63–74. Springer (2016). https://doi.org/10.1007/978-3-319-38851-9_5
26. Bentley, J.L., Sedgewick, R.: Fast algorithms for sorting and searching strings. In: SODA. pp. 360–369. ACM/SIAM (1997)
27. Berkman, O., Schieber, B., Vishkin, U.: Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. J. Algorithms **14**(3), 344–370 (1993). https://doi.org/10.1006/jagm.1993.1018
28. Bingmann, T.: Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools. Ph.D. thesis, Karlsruhe Institute of Technology, Germany (2018). https://doi.org/10.5445/IR/1000085031
29 SPP. Bingmann, T., Axtmann, M., Jöbstl, E., Lamm, S., Nguyen, H.C., Noe, A., Schlag, S., Stumpp, M., Sturm, T., Sanders, P.: Thrill: High-performance algorithmic distributed batch data processing with C++. In: BigData. pp. 172–183. IEEE Computer Society (2016). https://doi.org/10.1109/BigData.2016.7840603
30 SPP. Bingmann, T., Eberle, A., Sanders, P.: Engineering parallel string sorting. Algorithmica **77**(1), 235–286 (2017). https://doi.org/10.1007/s00453-015-0071-1
31. Bingmann, T., Fischer, J., Osipov, V.: Inducing suffix and LCP arrays in external memory. ACM J. Exp. Algorithmics **21**(1), 2.3:1–2.3:27 (2016). https://doi.org/10.1145/2975593

32 SPP. Bingmann, T., Gog, S., Kurpicz, F.: Scalable construction of text indexes with thrill. In: BigData. pp. 634–643. IEEE (2018). https://doi.org/10.1109/BigData.2018.8622171
33. Bingmann, T., Sanders, P.: Parallel string sample sort. In: ESA. pp. 169–180. Springer (2013). https://doi.org/10.1007/978-3-642-40450-4_15
34 SPP. Bingmann, T., Sanders, P., Schimek, M.: Communication-efficient string sorting. In: IPDPS. pp. 137–147. IEEE (2020). https://doi.org/10.1109/IPDPS47924.2020.00024
35. Blacher, M., Giesen, J., Sanders, P., Wassenberg, J.: Vectorized and performance-portable quicksort. CoRR **abs/2205.05982** (2022)
36. Blelloch, G.E., Anderson, D., Dhulipala, L.: Parlaylib - A toolkit for parallel algorithms on shared-memory multicore machines. In: SPAA. pp. 507–509. ACM (2020). https://doi.org/10.1145/3350755.3400254
37. Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J., Zagha, M.: A comparison of sorting algorithms for the connection machine CM-2. Commun. ACM **39**(12es), 273–297 (1996)
38. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. J. Parallel Distributed Comput. **37**(1), 55–69 (1996). https://doi.org/10.1006/jpdc.1996.0107
39. Borkar, S.: Exascale computing - A fact or a fiction? In: IPDPS. p. 3. IEEE Computer Society (2013). https://doi.org/10.1109/IPDPS.2013.121
40. Boucher, C., Gagie, T., Kuhnle, A., Langmead, B., Manzini, G., Mun, T.: Prefix-free parsing for building big bwts. Algorithms Mol. Biol. **14**(1), 13:1–13:15 (2019). https://doi.org/10.1186/s13015-019-0148-5
41. Bramas, B.: A novel hybrid quicksort algorithm vectorized using AVX-512 on intel skylake. Int. J. of Advanced Computer Science and Applications **8**(10) (2017), arXiv:1704.08579
42. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Tech. rep. (1994)
43. Canzar, S., Salzberg, S.L.: Short read mapping: An algorithmic tour. Proceedings of the IEEE **105**(3), 436–458 (2017). https://doi.org/10.1109/JPROC.2015.2455551
44. Chen, C., Schmidt, B.: Constructing large suffix trees on a computational grid. J. Parallel Distributed Comput. **66**(12), 1512–1523 (2006). https://doi.org/10.1016/j.jpdc.2006.08.004
45. Chim, H., Deng, X.: A new suffix tree similarity measure for document clustering. In: WWW. pp. 121–130. ACM (2007). https://doi.org/10.1145/1242572.1242590
46. Claude, F., Navarro, G.: Practical rank/select queries over arbitrary sequences. In: SPIRE. pp. 176–187. Springer (2008). https://doi.org/10.1007/978-3-540-89097-3_18
47. Claude, F., Navarro, G., Pereira, A.O.: The wavelet matrix: An efficient wavelet tree for large alphabets. Inf. Syst. **47**, 15–32 (2015). https://doi.org/10.1016/j.is.2014.06.002
48. Claude, F., Nicholson, P.K., Seco, D.: Space efficient wavelet tree construction. In: SPIRE. pp. 185–196. Springer (2011). https://doi.org/10.1007/978-3-642-24583-1_19
49. Clifford, R.: Distributed suffix trees. J. Discrete Algorithms **3**(2-4), 176–197 (2005). https://doi.org/10.1016/j.jda.2004.08.004
50. Clifford, R., Sergot, M.J.: Distributed and paged suffix trees for large genetic databases. In: CPM. pp. 70–82. Springer (2003). https://doi.org/10.1007/3-540-44888-8_6
51. Consortium, T.C.P.: Computational pan-genomics: status, promises and challenges. Briefings Bioinform. **19**(1), 118–135 (2018). https://doi.org/10.1093/bib/bbw089
52. Crauser, A., Ferragina, P.: A theoretical and experimental study on the construction of suffix arrays in external memory. Algorithmica **32**(1), 1–35 (2002). https://doi.org/10.1007/s00453-001-0051-5

53. Dagum, L., Leonardo, R.: Openmp: An industry-standard api for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998). https://doi.org/10.1109/99.660313
54. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008). https://doi.org/10.1145/1327452.1327492
55. Delcher, A.L., Kasif, S., Fleischmann, R.D., Peterson, J., White, O., Salzberg, S.L.: Alignment of whole genomes. Nucleic Acids Research **27**(11), 2369–2376 (1999). https://doi.org/10.1093/nar/27.11.2369
56. Dementiev, R., Kärkkäinen, J., Mehnert, J., Sanders, P.: Better external memory suffix array construction. ACM J. Exp. Algorithmics **12**, 3.4:1–3.4:24 (2008). https://doi.org/10.1145/1227161.1402296
57. Dementiev, R., Kettner, L., Sanders, P.: STXXL: standard template library for XXL data sets. Softw. Pract. Exp. **38**(6), 589–637 (2008). https://doi.org/10.1002/spe.844
58. Dementiev, R., Sanders, P.: Asynchronous parallel disk sorting. In: SPAA. pp. 138–148. ACM (2003). https://doi.org/10.1145/777412.777435
59. Deo, M., Keely, S.: Parallel suffix array and least common prefix for the GPU. In: PPOPP. pp. 197–206. ACM (2013). https://doi.org/10.1145/2442516.2442536
60. DeWitt, D.J., Naughton, J.F., Schneider, D.A.: Parallel sorting on a shared-nothing architecture using probabilistic splitting. In: PDIS. pp. 280–291. IEEE Computer Society (1991). https://doi.org/10.1109/PDIS.1991.183115
61. Dinklage, P.: Translating between wavelet tree and wavelet matrix construction. In: Stringology. pp. 126–135. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science (2019)
62 SPP. Dinklage, P., Ellert, J., Fischer, J., Köppl, D., Penschuck, M.: Bidirectional text compression in external memory. In: ESA. pp. 41:1–41:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ESA.2019.41
63 SPP. Dinklage, P., Ellert, J., Fischer, J., Kurpicz, F., Löbel, M.: Practical wavelet tree construction. ACM J. Exp. Algorithmics **26** (2021). https://doi.org/10.1145/3457197
64 SPP. Dinklage, P., Fischer, J., Kurpicz, F.: Constructing the wavelet tree and wavelet matrix in distributed memory. In: ALENEX. pp. 214–228. SIAM (2020). https://doi.org/10.1137/1.9781611976007.17
65. Edelkamp, S., Weiß, A.: Blockquicksort: Avoiding branch mispredictions in quicksort. ACM J. Exp. Algorithmics **24**(1), 1.4:1–1.4:22 (2019). https://doi.org/10.1145/3274660
66. Edwards, J.A., Vishkin, U.: Parallel algorithms for burrows-wheeler compression and decompression. Theor. Comput. Sci. **525**, 10–22 (2014). https://doi.org/10.1016/j.tcs.2013.10.009
67. Egidi, L., Louza, F.A., Manzini, G., Telles, G.P.: External memory BWT and LCP computation for sequence collections with applications. Algorithms Mol. Biol. **14**(1), 6:1–6:15 (2019). https://doi.org/10.1186/s13015-019-0140-0
68 SPP. Ellert, J., Fischer, J., Sitchinava, N.: LCP-aware parallel string sorting. In: Euro-Par 2020: Parallel Processing - 26th International Conference on Parallel and Distributed Computing, Warsaw, Poland, August 24-28, 2020, Proceedings. pp. 329–342. Springer (2020). https://doi.org/10.1007/978-3-030-57675-2_21
69 SPP. Ellert, J., Kurpicz, F.: Parallel external memory wavelet tree and wavelet matrix construction. In: SPIRE. pp. 392–406. Springer (2019). https://doi.org/10.1007/978-3-030-32686-9_28
70. Fagerberg, R., Pagh, A., Pagh, R.: External string sorting: Faster and cache-oblivious. In: STACS. pp. 68–79. Springer (2006). https://doi.org/10.1007/11672142_4
71. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. ACM **47**(6), 987–1011 (2000). https://doi.org/10.1145/355541.355547

72. Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. Algorithmica **63**(3), 707–730 (2012). https://doi.org/10.1007/s00453-011-9535-0
73. Ferragina, P., Giancarlo, R., Manzini, G.: The myriad virtues of wavelet trees. Inf. Comput. **207**(8), 849–866 (2009). https://doi.org/10.1016/j.ic.2008.12.010
74. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: FOCS. pp. 390–398. IEEE Computer Society (2000). https://doi.org/10.1109/SFCS.2000.892127
75. Fischer, J., Heun, V.: A new succinct representation of rmq-information and improvements in the enhanced suffix array. In: ESCAPE. pp. 459–470. Springer (2007). https://doi.org/10.1007/978-3-540-74450-4_41
76. Fischer, J., I, T., Köppl, D., Sadakane, K.: Lempel-ziv factorization powered by space efficient suffix trees. Algorithmica **80**(7), 2048–2081 (2018). https://doi.org/10.1007/s00453-017-0333-1
77 SPP. Fischer, J., Kurpicz, F.: Dismantling divsufsort. In: Stringology. pp. 62–76. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague (2017)
78 SPP. Fischer, J., Kurpicz, F.: Lightweight distributed suffix array construction. In: ALENEX. pp. 27–38. SIAM (2019). https://doi.org/10.1137/1.9781611975499.3
79 SPP. Fischer, J., Kurpicz, F., Löbel, M.: Simple, fast and lightweight parallel wavelet tree construction. In: ALENEX. pp. 9–20. SIAM (2018). https://doi.org/10.1137/1.9781611975055.2
80 SPP. Fischer, J., Kurpicz, F., Sanders, P.: Engineering a distributed full-text index. In: ALENEX. pp. 120–134. SIAM (2017). https://doi.org/10.1137/1.9781611974768.10
81. Flick, P., Aluru, S.: Parallel distributed memory construction of suffix and longest common prefix arrays. In: SC. pp. 16:1–16:10. ACM (2015). https://doi.org/10.1145/2807591.2807609
82. Flick, P., Aluru, S.: Parallel construction of suffix trees and the all-nearest-smaller-values problem. In: IPDPS. pp. 12–21. IEEE Computer Society (2017). https://doi.org/10.1109/IPDPS.2017.62
83. Flick, P., Aluru, S.: Distributed enhanced suffix arrays: efficient algorithms for construction and querying. In: SC. pp. 72:1–72:17. ACM (2019). https://doi.org/10.1145/3295500.3356211
84. da Fonseca, P.G.S., da Silva, I.B.F.: Online construction of wavelet trees. In: SEA. pp. 16:1–16:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.SEA.2017.16
85. Fuentes-Sepúlveda, J., Elejalde, E., Ferres, L., Seco, D.: Parallel construction of wavelet trees on multicore architectures. Knowl. Inf. Syst. **51**(3), 1043–1066 (2017). https://doi.org/10.1007/s10115-016-1000-6
86. Fuentes-Sepúlveda, J., Navarro, G., Nekrich, Y.: Parallel computation of the burrows wheeler transform in compact space. Theor. Comput. Sci. **812**, 123–136 (2020). https://doi.org/10.1016/j.tcs.2019.09.030
87. Furtak, T., Amaral, J.N., Niewiadomski, R.: Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms. In: SPAA. pp. 348–357. ACM (2007). https://doi.org/10.1145/1248377.1248436
88. Futamura, N., Aluru, S., Kurtz, S.: Parallel suffix sorting. Electrical Engineering and Computer Science **64** (2001)
89. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L.,

Woodall, T.S.: Open MPI: goals, concept, and design of a next generation MPI implementation. In: PVM/MPI. pp. 97–104. Springer (2004). https://doi.org/10.1007/978-3-540-30218-6_19
90. Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y., Puglisi, S.J.: Lz77-based self-indexing with faster pattern matching. In: LATIN. pp. 731–742. Springer (2014). https://doi.org/10.1007/978-3-642-54423-1_63
91. Gagie, T., Navarro, G., Prezza, N.: Fully functional suffix trees and optimal text searching in bwt-runs bounded space. J. ACM **67**(1), 2:1–2:54 (2020). https://doi.org/10.1145/3375890
92. Garrison, E., Sirén, J., Novak, A.M., Hickey, G., Eizenga, J.M., Dawson, E.T., Jones, W., Garg, S., Markello, C., Lin, M.F., Paten, B., Durbin, R.: Variation graph toolkit improves read mapping by representing genetic variation in the reference. Nature Biotechnology **36**(9), 875–879 (2018). https://doi.org/10.1038/nbt.4227
93. Gilchrist, J., Cuhadar, A.: Parallel lossless data compression based on the burrows-wheeler transform. In: AINA. pp. 877–884. IEEE Computer Society (2007). https://doi.org/10.1109/AINA.2007.109
94. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: SEA. pp. 326–337. Springer (2014). https://doi.org/10.1007/978-3-319-07959-2_28
95. Gonnet, G.H., Baeza-Yates, R.A., Snider, T.: New indices for text: Pat trees and pat arrays. In: Information Retrieval: Data Structures & Algorithms, pp. 66–82. Prentice-Hall (1992)
96. Goodrich, M.T.: Communication-efficient parallel sorting. SIAM J. Comput. **29**(2), 416–432 (1999). https://doi.org/10.1137/S0097539795294141
97. Goto, K.: Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. In: Stringology. pp. 111–125. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science (2019)
98. Gropp, W., Lusk, E.L., Doss, N.E., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Comput. **22**(6), 789–828 (1996). https://doi.org/10.1016/0167-8191(96)00024-5
99. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: SODA. pp. 841–850. ACM/SIAM (2003)
100. Grossi, R., Vitter, J.S., Xu, B.: Wavelet trees: From theory to practice. In: CCP. pp. 210–221. IEEE Computer Society (2011). https://doi.org/10.1109/CCP.2011.16
101. Ha, L.K., Krüger, J.H., Silva, C.T.: Fast four-way parallel radix sorting on gpus. Comput. Graph. Forum **28**(8), 2368–2378 (2009). https://doi.org/10.1111/j.1467-8659.2009.01542.x
102. Hagerup, T.: Optimal parallel string algorithms: sorting, merging and computing the minimum. In: STOC. pp. 382–391. ACM (1994). https://doi.org/10.1145/195058.195202
103. Han, L.B., Wu, Y., Nong, G.: Succinct suffix sorting in external memory. Inf. Process. Manag. **58**(1), 102378 (2021). https://doi.org/10.1016/j.ipm.2020.102378
104. Hayashi, S., Taura, K.: Parallel and memory-efficient burrows-wheeler transform. In: BigData. pp. 43–50. IEEE Computer Society (2013). https://doi.org/10.1109/BigData.2013.6691757
105. He, X., Huang, C.: Communication efficient BSP algorithm for all nearest smaller values problem. J. Parallel Distributed Comput. **61**(10), 1425–1438 (2001). https://doi.org/10.1006/jpdc.2001.1741
106. Helman, D.R., Bader, D.A., JáJá, J.: A randomized parallel sorting algorithm with an experimental study. J. Parallel Distributed Comput. **52**(1), 1–23 (1998). https://doi.org/10.1006/jpdc.1998.1462

107. Hoare, C.A.R.: Quicksort. Comput. J. **5**(1), 10–15 (1962). https://doi.org/10.1093/comjnl/5.1.10
108. Hou, K., Wang, H., Feng, W.: A framework for the automatic vectorization of parallel sort on x86-based processors. IEEE Trans. Parallel Distrib. Syst. **29**(5), 958–972 (2018). https://doi.org/10.1109/TPDS.2018.2789903
109. Huang, B., Gao, J., Li, X.: An empirically optimized radix sort for GPU. In: ISPA. pp. 234–241. IEEE Computer Society (2009). https://doi.org/10.1109/ISPA.2009.89
110. Inoue, H., Moriyama, T., Komatsu, H., Nakatani, T.: A high-performance sorting algorithm for multicore single-instruction multiple-data processors. Softw. Pract. Exp. **42**(6), 753–777 (2012). https://doi.org/10.1002/spe.1102
111. Itoh, H., Tanaka, H.: An efficient method for in memory construction of suffix arrays. In: SPIRE/CRIWG. pp. 81–88. IEEE Computer Society (1999). https://doi.org/10.1109/SPIRE.1999.796581
112. JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley (1992)
113. JáJá, J., Ryu, K.W., Vishkin, U.: Sorting strings and constructing digital search trees in parallel. Theor. Comput. Sci. **154**(2), 225–245 (1996). https://doi.org/10.1016/0304-3975(94)00263-0
114. Kaneta, Y.: Fast wavelet tree construction in practice. In: SPIRE. pp. 218–232. Springer (2018). https://doi.org/10.1007/978-3-030-00479-8_18
115. Kärkkäinen, J., Kempa, D.: Faster external memory LCP array construction. In: ESA. pp. 61:1–61:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.ESA.2016.61
116. Kärkkäinen, J., Kempa, D.: LCP array construction in external memory. ACM J. Exp. Algorithmics **21**(1), 1.7:1–1.7:22 (2016). https://doi.org/10.1145/2851491
117. Kärkkäinen, J., Kempa, D.: Engineering a lightweight external memory suffix array construction algorithm. Mathematics in Computer Science **11**(2), 137–149 (2017). https://doi.org/10.1007/s11786-016-0281-1
118. Kärkkäinen, J., Kempa, D.: Engineering external memory LCP array construction: Parallel, in-place and large alphabet. In: SEA. pp. 17:1–17:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.SEA.2017.17
119. Kärkkäinen, J., Kempa, D.: Better external memory LCP array construction. ACM J. Exp. Algorithmics **24**(1), 1.3:1–1.3:27 (2019). https://doi.org/10.1145/3297723
120. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Lempel-ziv parsing in external memory. In: DCC. pp. 153–162. IEEE (2014). https://doi.org/10.1109/DCC.2014.78
121. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Parallel external memory suffix sorting. In: CPM. pp. 329–342. Springer (2015). https://doi.org/10.1007/978-3-319-19929-0_28
122. Kärkkäinen, J., Kempa, D., Puglisi, S.J., Zhukova, B.: Engineering external memory induced suffix sorting. In: ALENEX. pp. 98–108. SIAM (2017). https://doi.org/10.1137/1.9781611974768.8
123. Kärkkäinen, J., Rantala, T.: Engineering radix sort for strings. In: SPIRE. pp. 3–14. Springer (2008). https://doi.org/10.1007/978-3-540-89097-3_3
124. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53**(6), 918–936 (2006). https://doi.org/10.1145/1217856.1217858
125. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: CPM. pp. 181–192. Springer (2001). https://doi.org/10.1007/3-540-48194-X_17
126. Kempa, D., Kociumaka, T.: String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In: STOC. pp. 756–767. ACM (2019). https://doi.org/10.1145/3313276.3316368

127. Kitajima, J.P., Navarro, G.: A fast distributed suffix array generation algorithm. In: SPIRE/CRIWG. pp. 97–105. IEEE Computer Society (1999). https://doi.org/10.1109/SPIRE.1999.796583
128. Kitajima, J.P., Ribeiro-Neto, B., Ziviani, N.: Network and memory analysis in distributed parallel generation of pat arrays. In: BSCA. pp. 192–202 (1996)
129. Knuth, D.E.: The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley (1973)
130. Kuhnle, A., Mun, T., Boucher, C., Gagie, T., Langmead, B., Manzini, G.: Efficient construction of a complete index for pan-genomics read alignment. In: RECOMB. pp. 158–173. Springer (2019). https://doi.org/10.1007/978-3-030-17083-7_10
131. Kulla, F., Sanders, P.: Scalable parallel suffix array construction. Parallel Comput. **33**(9), 605–612 (2007). https://doi.org/10.1016/j.parco.2007.06.004
132. Labeit, J., Shun, J., Blelloch, G.E.: Parallel lightweight wavelet tree, suffix array and fm-index construction. J. Discrete Algorithms **43**, 2–17 (2017). https://doi.org/10.1016/j.jda.2017.04.001
133. Lan, Y., Mohamed, M.A.: Parallel quicksort in hypercubes. In: SAC. pp. 740–746. ACM (1992). https://doi.org/10.1145/130069.130085
134. Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with bowtie 2. Nature methods **9**(4), 357 (2012). https://doi.org/10.1038/nmeth.1923
135. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.L.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biology **10**(3), R25 (2009). https://doi.org/10.1186/gb-2009-10-3-r25
136. Lao, B., Nong, G., Chan, W.H., Pan, Y.: Fast induced sorting suffixes on a multicore machine. J. Supercomput. **74**(7), 3468–3485 (2018). https://doi.org/10.1007/s11227-018-2395-5
137. Lao, B., Nong, G., Chan, W.H., Xie, J.Y.: Fast in-place suffix sorting on a multicore computer. IEEE Trans. Computers **67**(12), 1737–1749 (2018). https://doi.org/10.1109/TC.2018.2842050
138. Lee, S., Jeon, M., Kim, D., Sohn, A.: Partitioned parallel radix sort. J. Parallel Distributed Comput. **62**(4), 656–668 (2002). https://doi.org/10.1006/jpdc.2001.1808
139. Li, H.: Fast construction of fm-index for long sequence reads. Bioinform. **30**(22), 3274–3275 (2014). https://doi.org/10.1093/bioinformatics/btu541
140. Li, H., Durbin, R.: Fast and accurate short read alignment with burrows-wheeler transform. Bioinform. **25**(14), 1754–1760 (2009). https://doi.org/10.1093/bioinformatics/btp324
141. Li, Z., Li, J., Huo, H.: Optimal in-place suffix sorting. In: SPIRE. pp. 268–284. Springer (2018). https://doi.org/10.1007/978-3-030-00479-8_22
142. Lippert, R.: Space-efficient whole genome comparisons with Burrows Wheeler transforms. J. Comput. Biol. **12**(4), 407–415 (2005). https://doi.org/10.1089/cmb.2005.12.407
143. Liu, Y., Hankeln, T., Schmidt, B.: Parallel and space-efficient construction of burrows-wheeler transform and suffix array for big genome data. IEEE/ACM Trans. Comput. Biology Bioinform. **13**(3), 592–598 (2016). https://doi.org/10.1109/TCBB.2015.2430314
144. Louza, F.A., Gog, S., Telles, G.P.: Induced Suffix Sorting. Springer (2020). https://doi.org/10.1007/978-3-030-55108-7_3
145. Louza, F.A., Telles, G.P., Hoffmann, S., de Aguiar Ciferri, C.D.: Generalized enhanced suffix array construction in external memory. Algorithms Mol. Biol. **12**(1), 26:1–26:16 (2017). https://doi.org/10.1186/s13015-017-0117-9
146. Mahmoud, H.M.: Sorting: A Distribution Theory. John Wiley & Sons (2000)
147. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theor. Comput. Sci. **387**(3), 332–347 (2007). https://doi.org/10.1016/j.tcs.2007.07.013

148. Mäkinen, V., Navarro, G., Sadakane, K.: Advantages of backward searching - efficient secondary memory and distributed implementation of compressed suffix arrays. In: ISAAC. pp. 681–692. Springer (2004). https://doi.org/10.1007/978-3-540-30551-4_59
149. Makris, C.: Wavelet trees: A survey. Comput. Sci. Inf. Syst. **9**(2), 585–625 (2012). https://doi.org/10.2298/CSIS110606004M
150. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993). https://doi.org/10.1137/0222058
151. Marchet, C., Boucher, C., Puglisi, S.J., Medvedev, P., Salson, M., Chikhi, R.: Data structures based on k-mers for querying large collections of sequencing datasets. bioRxiv (2020). https://doi.org/10.1101/866756
152. McIlroy, P.M., Bostic, K., McIlroy, M.D.: Engineering radix sort. Comput. Syst. **6**(1), 5–27 (1993)
153. Menon, R.K., Bhat, G.P., Schatz, M.C.: Rapid parallel genome indexing with mapreduce. In: MapReduce. p. 51–58 (2011). https://doi.org/10.1145/1996092.1996104
154. Merrill, D., Grimshaw, A.S.: High performance and scalable radix sorting: a case study of implementing dynamic parallelism for GPU computing. Parallel Process. Lett. **21**(2), 245–272 (2011). https://doi.org/10.1142/S0129626411000187
155. Metwally, A.A., Kandil, A.H., Abouelhoda, M.: Distributed suffix array construction algorithms: Comparison of two algorithms. In: CIBEC. pp. 27–30. IEEE (2016). https://doi.org/10.1109/CIBEC.2016.7836092
156. Morrison, D.R.: PATRICIA - practical algorithm to retrieve information coded in alphanumeric. J. ACM **15**(4), 514–534 (1968). https://doi.org/10.1145/321479.321481
157. Munro, J.I., Navarro, G., Nekrich, Y.: Space-efficient construction of compressed indexes in deterministic linear time. In: SODA. pp. 408–424. SIAM (2017). https://doi.org/10.1137/1.9781611974782.26
158. Munro, J.I., Nekrich, Y., Vitter, J.S.: Fast construction of wavelet trees. Theor. Comput. Sci. **638**, 91–97 (2016). https://doi.org/10.1016/j.tcs.2015.11.011
159. Musser, D.R.: Introspective sorting and selection algorithms. Softw. Pract. Exp. **27**(8), 983–993 (1997). https://doi.org/10.1002/(SICI)1097-024X(199708)27:8¡983::AID-SPE117¿3.0.CO;2-%23
160. Navarro, G.: Wavelet trees for all. J. Discrete Algorithms **25**, 2–20 (2014). https://doi.org/10.1016/j.jda.2013.07.004
161. Navarro, G., Kitajima, J.P., Ribeiro-Neto, B.A., Ziviani, N.: Distributed generation of suffix arrays. In: CPM. pp. 102–115. Springer (1997). https://doi.org/10.1007/3-540-63220-4_54
162. Ng, W., Kakehi, K.: Merging string sequences by longest common prefixes. IPSJ Digital Courier **4**, 69–78 (2008)
163. Nong, G., Chan, W.H., Hu, S.Q., Wu, Y.: Induced sorting suffixes in external memory. ACM Trans. Inf. Syst. **33**(3), 12:1–12:15 (2015). https://doi.org/10.1145/2699665
164. Nong, G., Zhang, S., Chan, W.H.: Two efficient algorithms for linear time suffix array construction. IEEE Trans. Computers **60**(10), 1471–1484 (2011). https://doi.org/10.1109/TC.2010.188
165. Obeya, O., Kahssay, E., Fan, E., Shun, J.: Theoretically-efficient and practical parallel in-place radix sorting. In: SPAA. pp. 213–224. ACM (2019). https://doi.org/10.1145/3323165.3323198
166. Ohlebusch, E.: Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction. Oldenbusch Verlag (2013)
167. Ohlebusch, E., Beller, T., Abouelhoda, M.I.: Computing the burrows-wheeler transform of a string and its reverse in parallel. J. Discrete Algorithms **25**, 21–33 (2014). https://doi.org/10.1016/j.jda.2013.06.002

168. Ohlebusch, E., Gog, S., Kügel, A.: Computing matching statistics and maximal exact matches on compressed full-text indexes. In: SPIRE. pp. 347–358. Springer (2010). https://doi.org/10.1007/978-3-642-16321-0_36
169. Osipov, V.: Parallel suffix array construction for shared memory architectures. In: SPIRE. pp. 379–384. Springer (2012). https://doi.org/10.1007/978-3-642-34109-0_40
170. Patel, R.A., Zhang, Y., Mak, J., Davidson, A., Owens, J.D.: Parallel lossless data compression on the gpu. In: InPar. pp. 1–9. IEEE Computer Society (2012). https://doi.org/10.1109/InPar.2012.6339599
171. Pockrandt, C.: Approximate String Matching: Improving Data Structures and Algorithms. Ph.D. thesis, Free University of Berlin, Dahlem, Germany (2019). https://doi.org/10.17169/refubium-2185
172. Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Comput. Surv. **39**(2), 4 (2007). https://doi.org/10.1145/1242471.1242472
173. Rahman, N., Raman, R.: Adapting radix sort to the memory hierarchy. ACM J. Exp. Algorithmics **6**, 7 (2001). https://doi.org/10.1145/945394.945401
174. Reinders, J.: Intel threading building blocks - outfitting C++ for multi-core processor parallelism. O'Reilly (2007)
175. Reinert, K., Dadi, T.H., Ehrhardt, M., Hauswedell, H., Mehringer, S., Rahn, R., Kim, J., Pockrandt, C., Winkler, J., Siragusa, E., Urgese, G., Weese, D.: The seqan c++ template library for efficient sequence analysis: A resource for programmers. Journal of Biotechnology **261**, 157–168 (2017). https://doi.org/10.1016/j.jbiotec.2017.07.017
176. Sadakane, K.: Compressed text databases with efficient query algorithms based on the compressed suffix array. In: ISAAC. pp. 410–421. Springer (2000). https://doi.org/10.1007/3-540-40996-3_35
177. Sadakane, K.: Succinct representations of lcp information and improvements in the compressed suffix arrays. In: SODA. pp. 225–232. ACM/SIAM (2002)
178. Sanders, P., Hansch, T.: Efficient massively parallel quicksort. In: IRREGULAR. pp. 13–24. Springer (1997). https://doi.org/10.1007/3-540-63138-0_2
179. Sanders, P., Schlag, S., Müller, I.: Communication efficient algorithms for fundamental big data problems. In: BigData. pp. 15–23. IEEE Computer Society (2013). https://doi.org/10.1109/BigData.2013.6691549
180. Sanders, P., Winkel, S.: Super scalar sample sort. In: ESA. pp. 784–796. Springer (2004). https://doi.org/10.1007/978-3-540-30140-0_69
181. Satish, N., Harris, M.J., Garland, M.: Designing efficient sorting algorithms for manycore gpus. In: IPDPS. pp. 1–10. IEEE (2009). https://doi.org/10.1109/IPDPS.2009.5161005
182. Shepherdson, J.C., Sturgis, H.E.: Computability of recursive functions. J. ACM **10**(2), 217–255 (1963). https://doi.org/10.1145/321160.321170
183. Shun, J.: Fast parallel computation of longest common prefixes. In: SC. pp. 387–398. IEEE Computer Society (2014). https://doi.org/10.1109/SC.2014.37
184. Shun, J.: Parallel wavelet tree construction. In: DCC. pp. 63–72. IEEE (2015). https://doi.org/10.1109/DCC.2015.7
185. Shun, J.: Improved parallel construction of wavelet trees and rank/select structures. Inf. Comput. **273**, 104516 (2020). https://doi.org/10.1016/j.ic.2020.104516
186. Shun, J., Blelloch, G.E., Fineman, J.T., Gibbons, P.B., Kyrola, A., Simhadri, H.V., Tangwongsan, K.: Brief announcement: the problem based benchmark suite. In: SPAA. pp. 68–70. ACM (2012). https://doi.org/10.1145/2312005.2312018
187. Simpson, J.T., Durbin, R.: Efficient construction of an assembly string graph using the FM-index. Bioinform. **26**(12), 367–373 (2010). https://doi.org/10.1093/bioinformatics/btq217

188. Singler, J., Sanders, P., Putze, F.: MCSTL: the multi-core standard template library. In: Euro-Par. pp. 682–694. Springer (2007). https://doi.org/10.1007/978-3-540-74466-5_72
189. Sinha, R., Zobel, J.: Efficient trie-based sorting of large sets of strings. In: ACSC. pp. 11–18. Australian Computer Society (2003)
190. Sirén, J.: Indexing variation graphs. In: ALENEX. pp. 13–27. SIAM (2017). https://doi.org/10.1137/1.9781611974768.2
191. Sirén, J., Garrison, E., Novak, A.M., Paten, B., Durbin, R.: Haplotype-aware graph indexes. Bioinform. **36**(2), 400–407 (2020). https://doi.org/10.1093/bioinformatics/btz575
192. Sohn, A., Kodama, Y.: Load balanced parallel radix sort. In: ICS. pp. 305–312. ACM (1998). https://doi.org/10.1145/277830.277903
193. Solomonik, E., Kalé, L.V.: Highly scalable parallel sorting. In: IPDPS. pp. 1–12. IEEE (2010). https://doi.org/10.1109/IPDPS.2010.5470406
194. Stehle, E., Jacobsen, H.: A memory bandwidth-efficient hybrid radix sort on gpus. In: SIGMOD Conference. pp. 417–432. ACM (2017). https://doi.org/10.1145/3035918.3064043
195. Sun, W., Ma, Z.: Parallel lexicographic names construction with CUDA. In: ICPADS. pp. 913–918. IEEE Computer Society (2009). https://doi.org/10.1109/ICPADS.2009.31
196. Sundar, H., Malhotra, D., Biros, G.: Hyksort: a new variant of hypercube quicksort on distributed memory architectures. In: ICS. pp. 293–302. ACM (2013). https://doi.org/10.1145/2464996.2465442
197. Tischler, G.: On wavelet tree construction. In: CPM. pp. 208–218. Springer (2011). https://doi.org/10.1007/978-3-642-21458-5_19
198. Trinidad, J.F.M., Cumplido-Parra, R., Uribe, C.F.: An fpga-based parallel sorting architecture for the burrows wheeler transform. In: ReConFig. IEEE Computer Society (2005). https://doi.org/10.1109/RECONFIG.2005.9
199. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3), 249–260 (1995). https://doi.org/10.1007/BF01206331
200. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990). https://doi.org/10.1145/79173.79181
201. Wang, H., Peng, S., Lu, Y., Wu, C., Wen, J., Liu, J., Zhu, X.: BWTCP: A parallel method for constructing BWT in large collection of genomic reads. In: ISC. pp. 171–178. Springer (2015). https://doi.org/10.1007/978-3-319-20119-1_13
202. Wang, L., Baxter, S., Owens, J.D.: Fast parallel skew and prefix-doubling suffix array construction on the GPU. Concurr. Comput. Pract. Exp. **28**(12), 3466–3484 (2016). https://doi.org/10.1002/cpe.3867
203. Wassenberg, J., Sanders, P.: Engineering a multi-core radix sort. In: Euro-Par (2). pp. 160–169. Springer (2011). https://doi.org/10.1007/978-3-642-23397-5_16
204. Weiner, P.: Linear pattern matching algorithms. In: SWAT (FOCS). pp. 1–11. IEEE Computer Society (1973). https://doi.org/10.1109/SWAT.1973.13
205. Wild, S., Nebel, M.E., Neininger, R.: Average case and distributional analysis of dual-pivot quicksort. ACM Trans. Algorithms **11**(3), 22:1–22:42 (2015). https://doi.org/10.1145/2629340
206. Wu, Y., Lao, B., Ma, X., Nong, G.: An improved algorithm for building suffix array in external memory. In: PAAP. pp. 320–330. Springer (2019). https://doi.org/10.1007/978-981-15-2767-8_29
207. Xiaochen, T., Rocki, K., Suda, R.: Register level sort algorithm on multi-core SIMD processors. In: IA3@SC. pp. 9:1–9:8. ACM (2013). https://doi.org/10.1145/2535753.2535762
208. Xie, J.Y., Lao, B., Nong, G.: In-place suffix sorting on a multicore computer with better design. In: PAAP. pp. 331–342. Springer (2019). https://doi.org/10.1007/978-981-15-2767-8_30

209. Yin, Z., Zhang, T., Müller, A., Liu, H., Wei, Y., Schmidt, B., Liu, W.: Efficient parallel sort on avx-512-based multi-core and many-core architectures. In: HPCC/SmartCity/DSS. pp. 168–176. IEEE (2019). https://doi.org/10.1109/HPCC/SmartCity/DSS.2019.00038
210. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: HotCloud. USENIX Association (2010)
211. Zhou, D., Andersen, D.G., Kaminsky, M.: Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In: SEA. pp. 151–163. Springer (2013). https://doi.org/10.1007/978-3-642-38527-8_15
212. Zhu, G., Guo, C., Lu, L., Huang, Z., Yuan, C., Gu, R., Huang, Y.: DGST: efficient and scalable suffix tree construction on distributed data-parallel platforms. Parallel Comput. **87**, 87–102 (2019). https://doi.org/10.1016/j.parco.2019.06.002
213. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977). https://doi.org/10.1109/TIT.1977.1055714