Parallel Lightweight Wavelet-Tree, Suffix-Array and FM-Index Construction

Bachelor Thesis of

Julian Labeit

At the Department of Informatics
Institute for Theoretical Computer Science

Reviewer:	Prof. Dr.rer.nat. Peter Sanders
Second reviewer:	Prof. Dr. Dorothea Wagner
Advisor:	Prof. Dr.rer.nat. Peter Sanders
Second advisor:	Prof. Guy Blelloch

Duration: June 30, 2015 – September 30, 2015

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

Karlsruhe, September 27, 2015

. . . .

(**Julian Labeit**)

Acknowledgements

This work would not have been possible without Julian Shun and professor Guy Blelloch from Carnegie Mellon University. Both helped me with their advice, conceptionally and on implementation details. They introduced me to the PBBS framework and provided high-end hardware for the experimental evaluation. Additionally, great parts of the parallel code used by the implementations are authored by Julian Shun and Guy Blelloch.

I also want to thank Simon Gog from Karlsruhe Institute of Technology and professor Peter Sanders. Simon Gog contributed to the initial idea and helped with everything regarding the SDSL. Peter Sanders put me in contact with Guy Blelloch and helped me with the initial brainstorming process.

Most of the work concerning suffix-array construction was strongly influenced by Yuta Mori's DivSufSort implementation. I thank him for making his code publicly available at GitHub[1].

Finally, I thank the Baden-Württember Stiftung and interACT for giving me the possibility work on this bachelor's thesis at Carnegie Mellon University.

[1] `https://github.com/y-256/libdivsufsort`

Abstract

We present parallel lightweight algorithms to construct rank and select structures, wavelet trees and suffix arrays in a shared memory setting. In experiments the presented wavelet tree algorithms achieve a speedup of 2-4x over existing parallel algorithms while reducing the asymptotic memory requirement. The suffix array construction algorithm presented is the first parallel algorithm using the induced copying approach. It only uses a small amount of extra space while staying competitive with existing parallel algorithms. Finally, we show how our algorithms can be used to build compressed full-text indexes, such as the FM-index, in parallel.

Contents

1. Introduction

The amount of digitally available information grows rapidly. To process and store this data efficiently new methods are needed. In recent years compressed full-text indexes [NM07] have become more and more popular as they provide an elegant way of compressing data while still supporting queries on it efficiently. The most popular indexes all rely on three basic concepts: succinct rank and select on bitvectors, wavelet trees and suffix arrays. For modern applications, construction algorithms for these data structures are needed that are fast, scalable and memory efficient. The succinct data structure library (SDSL) [Gea14] is a sequential state of the art library. Additionally, in recent years wavelet-tree construction [Shu15] and suffix-array construction [KS03] has been successfully parallelized. However, so far most implementations are not memory efficient. The goal of this work is to develop algorithms which are memory efficient and scale well, so that they are applicable for libraries such as SDSL.

For wavelet-tree construction we adapt the algorithm by Shun [Shu15] to only use $n \log \sigma + o(n)$ bits additional space beyond the input and output while remaining $\mathcal{O}(\log n \log \sigma)$ depth and $\mathcal{O}(n \log \sigma)$ work. In our experiments the changes to the algorithm result in a speedup of 2-4x on the original algorithm. Additionally, we propose a variation of the domain-decomposition algorithm by Fuentes et al. [Fea14] designed for small alphabets. When $\frac{\sigma}{\log \sigma} \in \mathcal{O}(\log n)$ holds for the alphabet size, the algorithm requires same space and time as the variation of Shun's algorithm.

For suffix-array construction there are three main approaches: prefix doubling, recursive algorithms and induced copying [PST07]. Algorithms using prefix-doubling [LS07] and recursive algorithms [KS03], have been parallelized. However, the sequential algorithms that are lightweight and perform best in practice all use induced copying. The problem is that these algorithms use sequential loops with non-trivial data dependencies which are hard to parallelize. In this work we first use parallel rank and select on bitvectors to reduce the memory requirement of the parallel implementation of prefix doubling from the problem based benchmark suite (PBBS) [Sea12]. Then we show how to parallelize induced sorting for byte alphabets in polylogarithmic depth. Finally we combine both techniques to generate a parallel version of the two-stage algorithm [IT99]. In the experimental evaluation we will see that the proposed algorithm uses very little additional space and is one of the fastest parallel suffix array construction algorithms on byte alphabets. Finally, we use our algorithms to construct FM-indexes [FM00] in parallel and make our implementations available as part of the SDSL. In experiments we show that our algorithms scale well on a shared memory machine on real-world inputs.

2. Preliminaries

2.1 Strings

A *string* S is a sequence of characters from a finite ordered set Σ called the *alphabet*. The length of S is denoted as $n = |S|$. The size of the alphabet is denoted as $\sigma = |\Sigma|$ and the alphabet will often be interpreted as the integers $[0, \sigma - 1]$. The i-th character of S is denoted as $S[i]$ (zero based) and $S[i..j]$ denotes the substring from position i to position j (inclusive $S[i]$ and $S[j]$). $S[0..j]$ is called prefix of S and $S[i..(n-1)]$ is called *suffix* of S. By appending two strings A and B we get the concatenation denoted as AB. As every character can be seen as a string of length 1, we use the same notation cS to append a string S to a character c. As the characters of Σ are ordered with $<$, we can define the lexicographical order $<$ of strings. Let X,Y be strings and a,b be characters. Then $aX < bY$, iff $a < b$ or $a = b$ and $X < Y$. The empty string ϵ is defined to be smaller then all other strings.

2.2 Bitvectors

A *bitvector* B is a special string where the alphabet is restricted to zeros and ones, $\Sigma = \{0, 1\}$. Bitvectors can be used for many different applications. For example a bitvector of length n can be used to represent a subset A of some ordered set $B = [0..(n-1)]$. Then $B[i] = 1$ iff $i \in A$. There are three queries we typically want to answer on bitvectors in constant time.

- $B[i]$: Access the i-th element
- $rank_b(B, i)$: Return the number of appearances of bit b in $B[0..(i-1)]$
- $select_b(B, i)$: Return the position of the i-th appearance of bit b in B

Note, if we can answer $rank_1$ queries in constant time, then we can use the relation $rank_0(B, i) = i - rank_1(B, i)$ to answer $rank_0$ queries. However, we cannot use $select_1$ to answer $select_0$ queries and vice versa.

Jacobson [Jac89] first showed that rank could be answered in constant time with a data structure occupying only $o(n)$ bits additional space. Later Munro [Mun96] and Clark [Cla96] introduced the first select structure using only $o(n)$ bits of memory and answering select queries in constant time.

2.3 Wavelet Trees

Wavelet trees can be seen as a data structure to generalize functionality of bitvectors to general alphabets. However, since they first appeared in 2003 [GGV03] they have been applied to all kinds of problems. As described by Navarro [Nav14], wavelet trees can used for point grids, sets of rectangles, strings, permutations, binary relations, graphs, inverted indexes, document retrieval indexes, full-text indexes, XML indexes, and general numeric sequences. In this work we will focus on the construction of wavelet trees. The construction is handled for both byte and integer alphabets which covers all listed applications.

Let S be a string over an alphabet Σ with $|\Sigma| = \sigma$. We assume that $\sigma \leq n$, otherwise we could map each the alphabet to an alphabet with at most n different characters. By using wavelet trees we can answer access, select and rank queries on S in $\Theta(\log \sigma)$ time. We first give the definition of the standard wavelet tree and then go into some interesting variations.

2.3.1 Balanced Binary Shaped

The wavelet tree of the string S over the alphabet $\Sigma = [0, (\sigma-1)]$ can be defined recursively. Let $[a, b] \subseteq [0, (\sigma - 1)]$ be a section of the alphabet. The wavelet tree of S over $[a, b]$ has a root node v. If $a = b$, v is a leaf labeled with a. Otherwise, v has a bitvector B_v defined as:

$$B_v[i] = \begin{cases} 0, \text{if } S[i] \leq (a+b)/2, \\ 1, \text{else} \end{cases} \tag{2.1}$$

Let S_l be the string of all the characters $S[i]$ where $B_v[i] = 0$ and S_r the string of all characters $S[i]$ with $B_v[i] = 1$. The left child v_l of v is the wavelet tree of the string S_l over the alphabet $[a, \lfloor \frac{a+b}{2} \rfloor]$ and the right child v_r of v is the wavelet tree of the string S_r over the alphabet $[\lfloor \frac{a+b}{2} \rfloor + 1, b]$. In order to answer queries on the wavelet tree the bitvector of each node is augmented with rank and select structures. With the algorithms described Claude and Navarro [CN09], we can then answer general access, rank and select queries on the wavelet tree. In practice we concatenate all bitvectors of the nodes in breadth-first left to right order to one large bitvector. Then we only need to build one rank and two select structures on this bitvector and still have the full wavelet tree functionality. For most applications the nodes with pointers to the children are omitted. Only the bitvector with the rank and select structure is needed to traverse the tree.

2.3.2 Different Variations

There are two main techniques to compress wavelet trees. Either, one can compress the bitvector and the rank/select structures directly. Or, one changes the shape of the wavelet tree in order to achieve better space usage. Here we will focus on the latter. The standard wavelet tree is a balanced binary tree, at each node the alphabet is exactly split in half. Thus, the wavelet tree has $\log_2 \sigma$ levels and each character is represented by exactly one bit in each level. The wavelet tree then has a total space requirement of $n \log_2 \sigma$ bits. As proposed by Mäkine and Navarro [MN05], we can instead use *Huffman-shaped* wavelet trees. In a Huffman-shaped wavelet tree the leaf representing a character has exactly the depth of the number of bits in the Huffman code of the character. We can easily observe that the Huffman-shaped wavelet tree of S has the same size as the Huffman encoding of S. Thus, the Huffman-shaped wavelet tree of S compresses to the zero order entropy of S while still remaining fully functional.

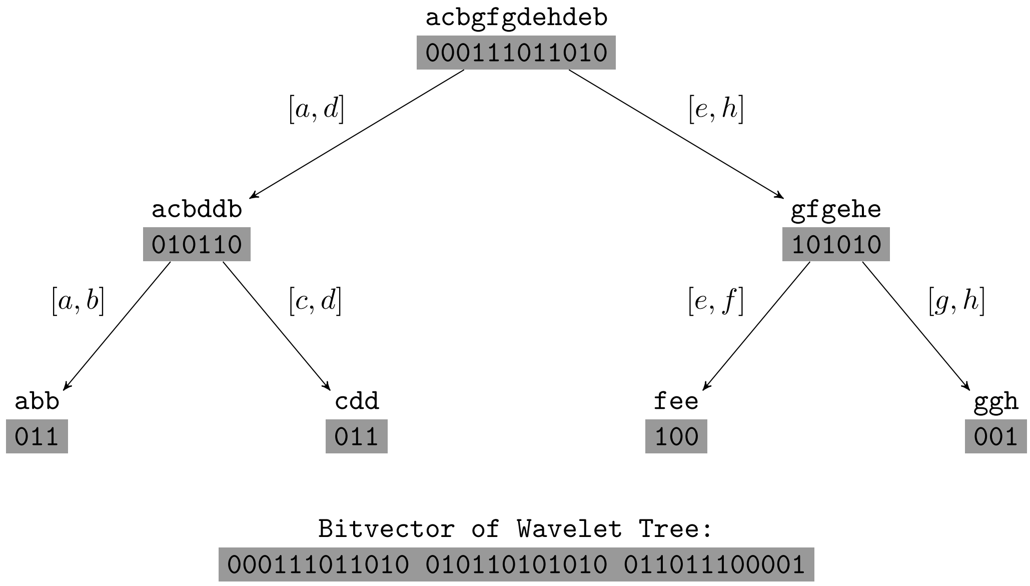

Figure 2.1: Example of a balanced binary wavelet tree over the string acbgfgdehdeb.

1 serialWT (S, $\Sigma = [a, b]$):
2   **if** a = b: **return** leaf labeled with a
3   v := root node
4   v.bitvector := bitvector of size $|S|$
5   **for** i := 0 to $|S| - 1$:
6         $v.bitvector[i] = \begin{cases} 0, \text{if } S[i] \leq (a+b)/2, \\ 1, \text{else} \end{cases}$
7
8     $(S_0,\ S_1)$ = pack(S, $v.bitvector$)
9   v. left_child = serialWT(S_0, $[a, \lfloor \frac{a+b}{2} \rfloor]$)
10   v. right_child = serialWT(S_1, $[\lfloor \frac{a+b}{2} \rfloor + 1, b]$)
11   **return** v

Figure 2.2: Sequential wavelet-tree construction algorithm

2.3.3 Construction Algorithms

For the standard wavelet tree it is easy to devise a $\Theta(n \log \sigma)$ construction algorithm directly from the recursive definition . pack(S,B) in line 8 of figure 2.2 describes the basic method inserting all $S[i]$ with $B[i] = 0$ into S_0 and all $S[i]$ with $B[i] = 1$ into S_1. It is easy to see that the work done in each recursive step is linear in $|S|$. For the recursive calls $|S_0| + |S_1| = |S|$ holds. As we have $\log \sigma$ levels in the resulting tree the total, work sums up to $n \log \sigma$. Note that by changing line 6 of the algorithm it is possible to construct wavelet trees of different shapes. For example we can fill v.bitvector with the corresponding bit of the Huffman code of the characters in S to construct the Huffman shaped wavelet tree.

To keep the descriptions concise, we will mostly leave out the construction of the actual nodes and concentrate on the construction of the bitvector.

2.4 Suffix Arrays

Suffix Arrays, first introduces by Manbers and Myers in 1990[MM93], are a space efficient alternative to suffix trees. Suffix arrays have a wide range of applications in text-indexing, bioinformatics or compression algorithms. Especially in recent years we have seen interesting developments towards compressed full-text indexes. These indexes use less space than

i	T[i]	SA[i]	ISA[i]	BWT	T[SA[i] ... (n-1)]
0	a	4	1	c	aaabcabc
1	a	0	4	c	aabcaaabcabc
2	b	5	7	a	aabcabc
3	c	9	10	c	abc
4	a	1	0	a	abcaaabcabc
5	a	6	2	a	abcabc
6	a	10	5	a	bc
7	b	2	8	a	bcaaabcabc
8	c	7	11	a	bcabc
9	a	11	3	b	c
10	b	3	6	b	caaabcabc
11	c	8	9	b	cabc

Table 2.1: Example of a suffix array over the string aabcaaabcabc.

the actual text, while at the same time replacing it and providing efficient search functionality. Most compressed full-text indexes use suffix arrays as an underlying concept. For a comprehensive study of compressed full-text indexes see [NM07].

2.4.1 Definition

The i-th suffix of a string S of length n is the substring $S[i..(n-1)]$. The *suffix array* (SA) of S is the array with the starting positions of all suffixes of S in lexicographic order. Thus, $S[SA[i-1], n] < S[SA[i], n]$ holds for all $i \in [1, (n-1)]$. We can easily see that SA is a permutation of the integers $[0, (n-1)]$. The inverse permutation of SA is referred to as *inverse suffix array* (ISA).

2.4.2 Construction Algorithms

Suffix Array can be constructed in linear time by first constructing the suffix tree and use a depth-first traversal to build the suffix array. However, this approach is impractical for most applications because of the increased size of suffix trees compared to suffix arrays. Algorithms constructing the suffix array without prior constructing the suffix tree can be divided into three main approaches.

- *Prefix-Doubling* computes in each step the approximated SA and ISA, where the suffixes are only sorted by a prefix of certain length. In the following step the approximated SA and ISA is refined by doubling the prefix length used for comparison. Each refinement step can be done in linear time using the information given by the approximated ISA of the last step. After $\log n$ refinement steps the suffixes are sorted by their complete length, thus resulting in a worst case runtime of $\mathcal{O}(n \log n)$.
- *Recursive* algorithms recursively construct the suffix array SA' for a subset of suffixes of S. Then use SA' to infer the suffix array of the remaining suffixes. In the last step the two suffix arrays are merged into the final suffix array. Usually, the reduction in each recursive step is at least $\frac{2}{3}$ and takes linear time. Thus resulting in linear time algorithms.
- *Induced Copying*, as the recursive approach, uses the fact that a sorted subset of suffixes can be used to induce a sorting of the remaining suffixes. The subset is chosen in such a way, that after sorting the subset, the remaining suffixes can be inserted into the suffix array by scanning the suffix array sequentially. This technique is often referred to as induced sorting. SA-IS algorithm [NZC09] uses induced sorting

to also sort the selected suffixes in linear time. The SA-IS algorithm applies induced copying recursively, resulting in a linear time algorithm. Other, in practice fast, induced copying algorithms use string sorting as a subroutine resulting in $\mathcal{O}(n \log n)$ or even $\mathcal{O}(n^2 \log n)$ worst case run-times.

For further reading about different suffix array construction algorithms we refer to "A Taxonomy of Suffix Array Construction Algorithms" [PST07].

2.4.3 Notable Implementations

As there are many different SA construction algorithms, there are also many different implementations. However, here we want to point out two implementations which are lightweight, fast in practice and are often used for comparison purposes in benchmarks.

- *DivSufSort* by Yuta Mori is an optimized implementation of the two-stage algorithm[IT99]. *DivSufSort* is one of the fastest SACA for byte alphabets. It makes use of the induced copying approach in combination with optimized string sorting routines. This implementation uses constant additional space and has a worst-case runtime of $\mathcal{O}(n \log n)$. For more details see `https://github.com/y-256/libdivsufsort`.
- *SA-IS* is an implementation of the SA-IS algorithm[NZC09] also implemented by Yuta Mori. Remarkable is that the algorithm has linear worst-case runtime, is space efficient and performs well in-practice. `https://sites.google.com/site/yuta256/sais`

2.4.4 DivSufSort

DivSufSort uses the notion of *A-type*, *B-type* and B^**-type* suffixes. In the description of *SA-IS* [NZC09] similar definitions are used only named L, S and LMS types.

Let T be a text of size n. A suffix $T[i..(n-1)]$ is of type A if $T[(i+1)..(n-1)] < T[i..(n-1)]$ and of type B otherwise. B^* suffixes are all B-type suffixes that are followed by an A-type suffix. The suffix only containing the last character of T is defined to be of type A. The string between two starting positions of B^*-type suffixes is referred to as B^* substring.

DivSufSort uses an optimized implementation of multikey quicksort to sort all B^* substrings. Then, a reduced text is formed by concatenating the ranks of all the B^* substrings in text order. By solving the reduced problem with an optimized suffix sorting routine, the final positions of the B^* suffixes are calculated. Then, induced sorting is used to insert the remaining B-type suffixes into the suffix array. Finally, with a second traversal of the suffix array, the A-type suffixes are put into the suffix array. For more details on induced sorting see the description of the SA-IS algorithm [NZC09].

2.5 FM-Index

The FM-index is a compressed full-text index. It was invented by Paolo Ferragina and Giovanni Manzin [FM00]. The FM-index uses the *Burrows-Wheeler transform* (BWT) to answer *count* and *locate* queries on strings.

The *Burrows-Wheeler transform* of a string S is a permutation of the input characters of S. Using the suffix array of S we can define the Burrows-Wheeler transform with $BWT[i] := S[SA[i] - 1]$ for $i \in \{0, \cdots, n-1\}$ and $S[-1] = S[n-1]$. For a brief example of the Burrows-Wheeler transform see table 2.1 .

The *count* query on a string S returns the number of occurrences of a pattern P in S. The *locate* query additionally returns the positions of the occurrences of P in S.

To answer count and locate queries the FM-index has a data structure answering $rank_\alpha(BWT, i)$ for all $\alpha \in \Sigma$ efficiently. For a pattern P, the so called backward-search, see figure 2.3, gives us an interval $[s, e]$ for which $SA[s..e]$ holds all occurrences of P in S. If the FM-

```
backwardSearch(P):
    i = |P| − 1
    e = 0
    s = n − 1
    while (s ≤ e and i ≤ 0 ):
        s = rank_P[i](BWT, n − 1) + rank_P[i](BWT, s − 1) + 1
        e = rank_P[i](BWT, n − 1) + rank_P[i](BWT, e)
        i − −
    return [s, e]
```

Figure 2.3: Finding all occurrences $SA[s, e]$ of a pattern P with backward-search.

index also needs to support locate queries, then samples of the suffix array are stored. In this work we use wavelet trees to store the Burrows-Wheeler transform and to support the rank queries in $\mathcal{O}(\log \sigma)$.

2.6 Parallel Computation Model

Our focus lies on shared memory machines, even though some algorithms described in this work can be applied to a distributed memory setting. The computation model we choose is the PRAM (Parallel Random Access Machine) model allowing concurrent reads and special atomic write operations.

For complexity analysis of our algorithms we use the work-depth model. *Work W* is the number of total operations required by the algorithm and *depth D* is the number of time steps required. Brent's scheduling theorem [JaJ92] states that if we have P processors available we can bound the running time by $\mathcal{O}(\frac{W}{P} + D)$.

A parallel algorithm is *work efficient* if the work matches with the time complexity of the best sequential algorithm. The goal of this work is to design parallel algorithms which are work efficient while achieving polylogarithmic depth.

2.7 Parallel Primitives

In the described algorithms we will make use of the parallel primitives *prefix sum*, *filter* and *pack*.

2.7.1 Prefix Sum

Prefix sum takes an array A of n elements of type T, an associative binary operator $\oplus : T \times T \to T$ and an identity element 0 with $0 \oplus x = x$ for all x. Prefix sum calculates the array $\{0, 0 \oplus A[0], 0 \oplus A[0] \oplus A[1], \cdots, 0 \oplus A[0] \oplus A[1] \cdots \oplus A[n-2]\}$ and returns $0 \oplus A[0] \oplus A[1] \oplus \cdots A[n-1]$. We mostly use prefix sum in such a way that it writes the output to the input array. Prefix sums can be implemented with $\mathcal{O}(n)$ work and $\mathcal{O}(\log n)$ depth [JaJ92].

2.7.2 Filter

Filter takes an array A of n elements of type T and a predicate function $f : T \to \{0, 1\}$, which can be evaluated in constant time. Filter returns all elements $A[i]$ with $f(A[i]) = 1$. In pseudocode filter sometimes has three arguments were the first is the input array, the second is the output array and the third is the predicate function. Using prefix sums, filter can be implemented in $\mathcal{O}(n)$ work and $\mathcal{O}(\log n)$ depth [JaJ92].

2.7.3 Pack

For our purposes pack is a primitive taking an array $A[0..(n-1)]$ of elements and a function $f_A : \{0,..,(n-1)\} \to \{0,1\}$ which can be evaluated in constant time. As a result pack returns two arrays A_0 and A_1. A_k holds all elements $A[i]$ where $f_A(i) = k$. The relative order between the elements in A_0 and A_1 remain unchanged.

As an example pack$([1,3,6,5,3,2,7], i \mapsto A[i] \mod 2)$ *returns* $[6,2])$ *and* $[1,3,5,7]$.

The algorithm described in figure 2.4 calculates pack in $\mathcal{O}(n)$ work and $\mathcal{O}(\log n)$ depth. The pack method uses additional n bits of space for the array S.

```
pack (A, f_A):
    S := Array of n/log n  integers
    block {0,..,(n-1)} into groups of size log n
    parfor S[b] := sum over all f_A(i) of group b
    perform exclusive prefix sum over S with total sum m
    A_0,A_1 := Arrays of size m-n and m
    parfor b := 0 to n/log n:
        pack block b serial to correct positions in A_0 and A_1 using S[b]
    return (A_0, A_1)
```

Figure 2.4: Parallel-pack algorithm computing A_0 and A_1 where A_k holds all elements $A[i]$ with $f_A(i) = k$.

3. Related Work

3.1 Parallel Rank and Select Structures on Bitvectors

Recently Shun[Shu15] described how the construction of succinct rank and select structures can be done in parallel. Additionally, there are different practical implementations of rank and select structures [GP14, Vig08, Cla96]. Especially the numbers of cache misses play an important role for the performance of these structures on modern machines. So far we know, at the time, there is no implementation constructing one of the mentioned structures in parallel. Even though some of the proposed rank and select structures are very straight forward to implement in parallel.

3.2 Parallel Wavelet-Tree Construction

Fuentes et al. [Fea14] introduced two different algorithms to build wavelet trees in parallel. Both algorithms perform $\mathcal{O}(n \log n)$ work and $\mathcal{O}(n)$ span. However, two very different concepts are applied. The first algorithm uses the observation that on the i-th level, the node index in which a character s is represented in, can be calculated by simply taking the first i bits of the binary representation of s (this only works with the standard balanced wavelet tree). Thus, the levels can be constructed independently. In the described algorithm the l-th level is build by scanning the input sequentially and using the first l bits of each character to assign the character to the corresponding node of the level. Then, with a second scan, the bitvectors of the nodes can be calculated. The second algorithm splits the input string S into p parts and then builds a wavelet tree on each of the parts. In the final step the p wavelet trees are merged by concatenating the bitmaps of each level sequentially. The second approach is referred to as domain decomposition because of the initial decomposition of the input string.

Recently, Shun introduced two algorithms with polylogarithmic depth [Shu15]. The first algorithm, we will refer to it as *levelWT*, builds the wavelet tree in a top-down level by level approach. Each level is build in parallel by first writing the bitvectors of all nodes of the level in parallel. Then the characters of the input string are reordered using prefix sums over the bitvectors of the nodes. This approach basically performs a parallel binary radix sort starting with the most significant bit.

The second algorithm again uses the before mentioned observation to build the levels of the wavelet tree independently. Additional parallelism is introduced by using a parallel integer sorting algorithm to order the elements by their top bits. The second algorithm is referred to as *sortWT*.

Both *levelWT* and *sortWT* achieve very good speedups on shared memory machines and there are implementations available in the problem based benchmark suite [Sea12]. So far, *levelWT* is in practise the fastest parallel implementation we are aware of [Shu15].

3.3 Parallel Suffix-Array Construction

Since suffix arrays have been introduced by Manber and Myers [MM93], many different suffix-array construction algorithms (SACA) have been published. Some of the most important ones are the difference cover algorithm (DC3)[KS03] and the induced sorting algorithm (SA-IS) [NZC09]. DC3 was one of the first linear time SACAs. It can be parallelized well in all kinds of computational models. For example there are numerous parallel implementations for distributed memory [KS07] or GPUs available [Osi12, DK13, WBO15].

SA-IS is a lightweight linear work algorithm and one of the fastest SACAs in practice. However, it is hard to parallelize, as induced sorting consists out of multiple sequential scans with non-trivial data dependencies.

For some compressed SA applications only the burrows wheeler transformation is needed (BWT), so there has been significant work on constructing the BWT in parallel [HT13, Lea15]. These algorithms first construction the BWT over parts of the input and then merge these parts in parallel. Many bioinformatics applications use compressed SA, thus there are frameworks with parallel SACA implementations especially optimized for DNA input [Lea13, Hea09]. For example, PASQUAL [Lea13] has a fast implementation using a combination of prefix doubling and string sorting algorithms.

4. Parallel Wavelet-Tree Construction

We first describe how to parallelize the construction of rank and select structures on bitvectors. Then two algorithms are introduced to build wavelet trees in parallel.

The first rank and select structures for bitvectors were introduced by Jacobson[Jac89]. Since then many different variations were described. Here we focus on two in practice fast data structures which already have sequential implementations in the SDSL. Especially a cache friendly query pattern is in practice important for good performance.

4.1 Parallel Rank on Bitvectors

There are numerous different implementations of rank support structures on bitvectors. One of the simplest, and in practice fastest, is the broadword implementation suggested by Vigna [Vig08]. The corresponding SDSL implementation is called *rank_support_v*. The answers to the rank queries are precomputed and stored in a first and second level lookup table. The tables are packed into memory such that in most cases at most one cache miss is needed to answer a rank query. This method uses 25% additional space to the bitvector.

The construction algorithm for this data structure can trivially be parallelized by using prefix sums. By temporary storing the values for the prefix sums in the output array no additional memory is needed.

4.2 Parallel Select on Bitvectors

For select support queries we parallelized the SDSL implementation of the structure proposed by Clark [Cla96]. In the SDSL the implementation is referred to as *select_support_mcl*. First, the range of possible queries is blocked into large blocks. Then, a case distinction is made. If a block spans more than $\log^4 n$ bits (where n is the size of the bitvector), then the block is a *long* block. Otherwise, the block is called *short* block. For long blocks simply the answers to all queries are stored explicitly. For short blocks the answer to every 64th query is stored. In practice this structure only uses small amount of additional space. For a more precise analysis see [Cla96] and the SDSL documentation.

Using multiple parallel for-loops with prefix sums the blocks can be categorized into long and short blocks. After categorizing the blocks, they can be initialized independently. As small blocks are only of logarithmic size, they can be initialized sequentially while staying in logarithmic depth. Long blocks are again initialized by multiple parallel for-loops and prefix sums.

Using the parallel rank and select structure implementations, we can now parallelize wavelet-tree construction.

4.3 Domain Decomposition Algorithm

```
1  ddWT (S, Σ, P):
2      decompose S into P strings S_P
3      parfor p := 0 to P−1:
4          (B_p, Nodes[p]) = serialWT(S_p)
5      offsets := array of size 2 · |Σ| · P
6      parfor n := 0 to 2 · |Σ| − 1:
7          parfor p := 0 to p−1:
8              offsets [n · P + p] = Nodes[p][n].length
9      perform prefixsum on offsets
10     parfor p := 0 to p−1
11         parfor n := 0 to 2 · |Σ| − 1:
12             // destination: bitvector and offset, source: bitvector and offset, number of bits to
                      copy
13             copy(B, offsets [n · P + p], B_p, Nodes[p][n].start, Nodes[p][n].length)
14     return B
```

Figure 4.1: Domain decomposition algorithm for parallel wavelet-tree construction.

The domain decomposition algorithm described in figure 4.1 is an modified version of the algorithm described by Fuentes et al. [Fea14]. The first part of the algorithm (line 2-4) remains unchanged. Here the string is split into P chunks for which P WTs are built. We observe that merging the WTs essentially is a reordering of the bitvectors. Initially, the bitvectors of the P WTs are stored consecutevly in the bitmaps B_p. We reorder the lengths of the bitvectors of the nodes into their final order (line 6-8). After calculating the prefix sum (line 9), `offsets` stores the starting positions of all nodes in the resulting bitvector in increasing order. The result bitvector B can now be written in parallel.

Through the modification the algorithm has $\mathcal{O}(\log P\sigma + \frac{n \log \sigma}{P})$ depth and $\mathcal{O}(P\sigma + n \log \sigma)$ work. With $P = \frac{n}{\log n}$ we get $\mathcal{O}(\log n \log \sigma)$ depth and $\mathcal{O}(\frac{n\sigma}{\log n} + n \log \sigma)$ work. For small alphabet sizes with $\frac{\sigma}{\log \sigma} \in \mathcal{O}(\log n)$ *ddWT*, has $\mathcal{O}(n \log \sigma)$ work and the memory requirement beyond the input and the output is $n \log \sigma$ bits for the bitmaps of the wavelet trees generated in the first part of the algorithm. Additionally, $\mathcal{O}(\sigma \log n)$ bits per processor are needed to store the node offsets of the wavelet trees. *ddWT* can also be adapted to construct different shapes of wavelet trees by simply adapting the *serialWT* algorithm accordingly.

Note, if implemented carefully, all writes to bitvectors by a single thread in the *ddWT* algorithm are consequtive. This makes *ddWT* very cache efficient. Additionally, it can make sense to slightly change the merging step to avoid multiple threads from writing to the same word. One way to do so is to assign each thread a block $B[start..end]$ of the output bitvector. Then, for example with a binary search on the offset array, each thread can determine the wavelet tree index n and the node index p on which the first bit to copy lies. Then each thread writes to $B[start..end]$ just as described in the pseudocode.

4.4 Recursive Algorithm

The levelWT algorithm proposed by Shun [Shu15] uses prefix sums over the bitvectors of a level of the WT as subroutine. As a result the algorithm has a memory requirement of at least $n \log n$ bits. We can reduce the memory requirement by using the parallel pack operation. Additionally, the recursive approach reduces the memory overhead by implicitly passing the node boundaries as described in figure 4.2. Especially when dealing with large alphabets, this additionally saves memory. This algorithm has $\mathcal{O}(\log n \log \sigma)$

```
1  recursiveWT (S, Σ = [a, b]):
2      if a = b: return leaf labeled with a
3      v := root node
4      v.bitvector := bitvector of size |S|
5      parfor i := 0 to |S| − 1:
6          v.bitvector[i] = { 0, if S[i] ≤ (a + b)/2,
                              1, else
7
8      (S_0, S_1) = parallelPack(S, v.bitvector[i])
9      v.left_child = recursiveWT (S_0, [a, ⌊(a+b)/2⌋]) // async call
10     v.right_child = recursiveWT (S_1, [⌊(a+b)/2⌋ + 1, b]) // async call
11     return v
```

Figure 4.2: Recursive construction algorithm for parallel wavelet-tree construction.

depth and $\mathcal{O}(n \log \sigma)$ work, matching that of the original levelWT. However, it only needs additional $n \log \sigma$ bits for the output of the parallel pack routines. Thus we reduce the memory consumption from at least $n \log n$ to $n(1 + \log \sigma)$ bits of additional space, plus $\mathcal{O}(\log n \log \sigma)$ bits stack space per processor.

Note both *ddWT* and *recursiveWT* can be easily adapted to build wavelet trees of different shapes by changing line 6 in *serialWT* or in *recursiveWT*. We provide implementations of *recursiveWT* building different-shaped wavelet trees in the parallel branch of the SDSL.

5. Parallel Suffix-Array Construction

In this chapter we describe a new parallel suffix-array construction algorithm. So far all parallel SACAs either use the recursive approach [KS03] or use prefix doubling[LS07]. Both approaches can be parallelized well because their most time consuming step relies on integer sorting. We first describe a simple parallel prefix doubling algorithm which in practice needs $n(2 + \log n) + o(n)$ bits of additional memory. Then we introduce a parallel algorithm which uses induced copying and in practice uses $2n + o(n)$ bits beyond input and output.

5.1 Parallel Range

One simple, but in practice very scalable, suffix-array construction algorithm uses the prefix doubling approach. *ParallelRange* from PBBS is a parallel version of the algorithm described by Larsson and Sadakane [LS07]. The algorithm, described in figure 5.1, starts

```
parallelRange (T,n):
    SA, ISA := integer arrays of size n
    parfor i := 0 to n−1:
        ISA[i] = T[i]
        SA[i] = i
    ranges = {(0,n−1)}
    offset := 1
    while ranges not empty:
        nranges = {}
        parfor (s,e) in ranges:
            sort SA[s..e] by the values at ISA[SA[s..e]+offset]
        parfor (s,e) in ranges:
            scan SA[s..e] in parallel update ISA and add equal ranges to nranges
        ranges = nranges
        offset = min(1,offset)
        offset *= 2
    return SA
```

Figure 5.1: Parallel range algorithm for parallel suffix-array construction.

with an approximate SA and ISA, in which the suffixes are only sorted by their first character. The idea is to refine SA and ISA in each round. In each refinement step groups of suffixes with the same ISA value are sorted. These groups are referred to as *range*. In the following round suffixes can be sorted by twice as many characters by using the approximated ISA for comparison. More precisely in step d the suffix i is sorted by $ISA[SA[i]+2^d]$. After at most $\log n$ rounds all suffixes have a distinct ISA value and thus have been sorted correctly.

In the PBBS implementation of *parallelRange* the data structure used to store and update the ranges is two simple array of integer pairs. However, this approach uses at least $2n \log n$ bits additional space. To save space we can tag the start and end of each range with a bitflag in a bitvector. By using the parallel rank and select structures we can then efficiently navigate through all ranges. This approach then only needs $2n + o(n)$ bits managing the ranges. Additionally, parallelRange uses $n \log n$ bits space for the ISA array plus the space needed for the sorting routine. The space needed for the sorting routine depends on the size of the ranges that are sorted. In practice this space requirement can be reduced drastically by presorting the suffixes directly by their first characters into buckets and initializing ranges accordingly.

If we use a sorting algorithm with $\mathcal{O}(n^\epsilon)$ depth and $\mathcal{O}(n)$ work, *parallelRange* has $\mathcal{O}(n^\epsilon \log n)$ depth and $\mathcal{O}(n \log n)$ work (for $\epsilon > 0$).

5.2 Parallel DivSufSort

Using *parallelRange* we can parallelize the *DivSufSort* implementation by Mori of the two-stage algorithm [IT99]. In this section we will first give an overview over our parallel DivSufSort algorithm. Then we will explain in detail how to parallelize induced sorting, the key concept used by most state-of-the-art suffix-array construction tools.

5.2.1 Overview

First we give a general overview over the algorithm in figure 5.2. In the first step the

```
parallelDivSufSort(T,n):
    categorize  A, B and BStar suffixes
    sort BStar substrings to get reduces SA problem
    use parallelRange to solve reduced SA problem
    induce positions of B−type suffixes
    induce positions of A−type suffixes
```

Figure 5.2: Overview over the parallel DivSufSort algorithm.

suffixes are categorized into A, B and B^* suffixes. A suffix $T[i..(n-1)]$ is of type A if $T[(i+1)..(n-1)] < T[i..(n-1)]$ and of type B otherwise. B^* suffixes are all B-type suffixes that are followed by an A-type suffix. This step can be parallelized using two parallel loops and prefix sums in $\mathcal{O}(\log n)$ depth and linear work.

The second step is string sorting all the B^* substrings lexicographically. B^* substrings are all strings formed by the characters between two consecutive B^* suffixes. Then each B^* substring can be replaced by its rank among the B^* substrings, forming a reduced text. Note that there are very efficient parallel string sorting algorithms available [BES14]. Our implementation, however, only parallelizes an initial bucket sort and uses the sequential multikey quicksort for the resulting buckets. This is sufficient for our purpose.

The third step is to construct the suffix array of the reduced text. As the text size has reduced by at least half, the unused part of the SA array can be used for the ISA array, thus *parallelRange* can be applied with only $n + o(n)$ bits additional space, plus the space needed for the sorting routine.

In the final step the sorting of the B^* suffixes is used to induce the sorting of the remaining suffixes. This step is non-trivial to parallelize. We introduce a polylogarithmic depth algorithm for induced sorting for constant alphabet size.

5.2.2 Parallelizing Induced Sorting

Induced sorting is a concept used in most of the fastest suffix-array construction algorithms like *SA-IS* or *DivSufSort*. *DivSufSort* uses induces sorting in the last two steps to induce positions of the A and B-type suffixes from the positions of the B^* suffixes. The sequential algorithm shown in figure 5.3 consists of two sequential loops, one sorting the B suffixes and one sorting the A suffixes. The order in which the suffix array is traversed is crucial

```
1  bucketA := Starting positions of the A buckets
2  bucketB := Ending position of the B buckets
3  for i := n−1 to 0:
4      if SA[i] has been initialized and SA[i]−1 is B−type suffix:
5          SA[bucketB[T[SA[i]−1]]] = SA[i]−1
6          bucketB[T[SA[i]−1]]−−
7  for i := 0 to n−1:
8      if SA[i]−1 is A−type suffix:
9          SA[bucketA[T[SA[i]−1]]] = SA[i]−1
10         bucketA[T[SA[i]−1]]++
```

Figure 5.3: Induced sorting all A and B-type suffixes by using the already sorted B^*-type suffixes.

to guarantee a correct sorting. B-type suffixes are defined in such a way that if a B-type suffix is directly preceded (in text order) by another B-type suffix, then the preceding suffix has to be lexicographical smaller. This is the key observation used here. For A-type suffixes the observation is analogous. For the full proof that this algorithm induces the correct suffix array we refer the reader to the SA-IS paper [NZC09]. At first sight induced sorting is inherent sequential. Now we describe how to parallelize the induced sorting of the B-type suffixes. Sorting the A-type suffixes can be done analogously.

i	0	1	2	3	4	5	6	7	8	9	10	11
T[i]	a	a	b	c	a	a	a	b	c	a	b	c
type	B	B	B^*	A	B	B	B	B^*	A	B	B^*	A
T[SA[i]]	a	a	a	a	a	a	b	b	b	c	c	c
SA[i]	4	0	5	9	1	6	10	2	7	11	3	8

Table 5.1: Example of induced sorting B^* suffixes

In line 5 we say that SA position $bucketB[T[SA[i]-1]$ is being initialized by position i. To perform the iteration i of the scan independently from the other iterations, we need to know the value of $SA[i]$ and of $bucketB[T[SA[i]-1]$. Assuming an interval of SA values have already been initialized. The scan over this interval can then be executed in parallel using prefix sums to calculate the number of writes to buckets before a distinct position.

For the sake of simplicity let us first only consider special inputs. Assuming that consecutive characters are always different in the input string, then the invariant in line 5 $T[SA[i]-1] < T[SA[i]]$ holds. Hence, no B-type suffix is initialized by a suffix in the same bucket. Thus, once the loop has been executed for all B-type suffixes with lexicographical larger first characters than α, all B-type suffixes starting with character α have been initialized. This gives us a way to parallelize induced sorting for the special case where there are no repetitions in the input string with depth $\mathcal{O}(\sigma^2 \log n)$. The algorithm is described in figure 5.4. The depth is dominated by calculating the σ prefix sums in each

```
1  bucketB := Ending position of the B buckets
2  for α :=  σ−1 to 0:
3     [s,e] := interval  in SA of all  suffixes  starting  with α
4     bucketSums:= Arrays to count number of suffixes put into buckets
5     parfor i := e to s:
6        if SA[i] has been initialized  and SA[i]−1 is B−type suffix:
7           bucketSums[T[SA[i]−1]][i]++
8     parfor α := 0 to σ−1:
9        perform prefix sum on bucketSums[α]
10    parfor i := e to s:
11       if SA[i] has been initialized  and SA[i]−1 is B−type suffix:
12          b := T[SA[i]−1]
13          SA[bucketB[b] − bucketSums[b][i]] = SA[i]−1
```

Figure 5.4: Parallel induced sorting all B-type suffixes for inputs with no repetitions of characters.

iteration. Each prefix sum can be calculated in $\mathcal{O}(\log n)$ depth resulting in a total depth of $\mathcal{O}(\sigma^2 \log n)$. By executing the parallel for-loops in blocks of size $\sigma \log n$ sequentially, the overall depth remains $\mathcal{O}(\sigma^2 \log n)$. However, by using blocks of size $\sigma \log n$ only the prefix sums over every $\sigma \log n$ bucketSums entry is needed. However, then only the prefix sums over every $\sigma \log n$ bucketSums entry is needed, thus reducing the overall work to $\mathcal{O}(n)$.

5.2.3 Dealing with Repetitions in the Input

In general inputs there are of course repetitions of characters. The first approach we present in this section leads to an overall algorithm with depth $\mathcal{O}(\sigma \log n \sum_{\alpha \in \Sigma} R_\alpha)$, where R_α is the longest run (consecutive repetitions of the same character) of the character α in T. With an additional technique presented afterwards, the depth can be brought down to $\mathcal{O}(\sigma^2 \log n + \sigma \log^2 n)$.

For general texts the invariant $T[SA[i]-1] < T[SA[i]]$ is relaxed to $T[SA[i]-1] \leq T[SA[i]]$ in line 5 of the induces sorting algorithm. This in turn means that even after executing the loop for all B-type suffixes with lexicographical larger first character than α, not all SA values in $[s, e]$ have been initialized. More precisely, all B-type suffixes with multiple repetitions of α have not been initialized. We observe that the B-type suffixes that begin with multiple repetitions of α are lexicographical smaller than those with only a single α. Thus $[s, e]$ can be devided into two contiguous parts $[s, e' - 1]$ and $[e', e]$ where all SA values in $[e', e]$ are already initialized and all values $[s, e' - 1]$ still need to be initialized. The algorithm descibed in figure 5.5 initializes in the i-th iteration of the while loop all suffixes which have $(i + 1)$ repetitions of α. At most R_α iterations of the while loop in line 3 are needed until all B-type suffix starting with α are initialized. Calculating k and the filter primitive have depth $\log n$. In total the algorithm then has $\mathcal{O}(\sigma \log n \sum_{\alpha \in \Sigma} R_\alpha)$ depth.

The depth can be reduced by first initializing all suffixes in $[s, e]$ which have 2^k $(k \in \mathbb{N})$ repetitions of the character α. This can again be done with at most $\log n$ executions of the filter primitive. Then the suffixes which have p repetitions of α can be initialized by filtering the suffixes with at least $2^{\lfloor \log_2 p \rfloor}$ repetitions. Note that to use the filter primitive in linear work, the number of repetitions of the character α have to be pre calculated. A suffix with p repetitions of α is part in at most $2p$ filter calls. This ensures that in total only constant work per initialized SA value is spent.

```
[s,e] := interval of SA values that already are initialized
α := first character of all the suffixes in SA[s,e]
while [s,e] not empty:
    m := |{i ∈ [s,e]|T[SA[i] − 1] = α}|
    filter (SA[s,e], SA[s−m−1,s−1], T[SA[i]−1] = α)
    s := s−m−1
    e := s−1
    parfor i := s to e:
        SA[i]−−
```

Figure 5.5: Parallel induced sorting all B-type suffixes for general inputs.

Thus the B-type suffixes starting with a character α can actually be initialized in $\mathcal{O}(\log^2 n)$ depth and constant work per suffix. In total this results in a depth of $\mathcal{O}(\sigma^2 \log n + \sigma \log^2 n)$ and linear work.

For constant alphabet size this results in a polylogarithmic-depth and linear-work parallelization of the induced sorting approach used by *DivSufSort* or by the *SA-IS* algorithm. Note that in practice $\sum_{\alpha \in \Sigma} R_\alpha$ typically is small. Our fastest implementation actually uses the simpler $\mathcal{O}(\sigma \log n \sum_{\alpha \in \Sigma} R_\alpha)$ depth algorithm.

With the $\mathcal{O}(\sigma^2 \log n + \sigma \log^2 n)$ depth and linear-work parallelization of induced sorting the first iteration of the *SA-IS* algorithm can be parallelized. As σ may grow our parallelization cannot be applied recursively. However, it can be used to reduce any SA problem with constant σ to a SA problem with half the size and general σ in linear work and polylogarithmic depth.

6. Parallel FM-Index Construction

In this chapter we show how the algorithms introduced in the previous chapters can be used to construct FM-Indexes in parallel.

In the experimental evaluation of this work our implementation of *recursiveWT* performs best under all other parallel wavelet-tree construction implementations. For suffix-array construction our implementation of *parallelDivSufSort* performs best. Thus we choose *recursiveWT*, *parallelDivSufSort* and our parallel implementations for rank/select support on bitvectors for parallel FM-index construction. Figure 6.1 gives an overview over the parallel FM-index construction. First the suffix array of the input is constructed using

```
fm_index (S,n):
    SA := parallelDivSufSort(S,n)
    parfor i := 0 to n−1:
        BWT[i] := S[SA[i]−1]
    WT := recursiveWT(BWT,n)
    selectSupport0(WT)
    selectSupport1(WT)
    rankSupport(WT)
```

Figure 6.1: Parallel FM-index construction.

parallelDivSufSort. Then, using the suffix array, the Burrows-Wheeler transform is constructed. Then we use *recursiveWT* to construct the wavelet tree of the Burrows-Wheeler transform. Note, for convenience we define $S[-1] = S[n-1]$. In the final steps the bitvector of the wavelet tree is augmented with two select structures (one to support $select_0$ and one to support $select_1$ queries) and one rank structure.

To support locate queries efficiently with the FM-index often a sampling of the suffix array is stored. We added our parallel algorithms to a parallel version of the SDSL so they can be easily adapted for different purposes. For example the wavelet-tree shape can be changed to balanced, Huffman shaped or Hu-Tucker shaped. Additionally, in the future the implementation of *parallelDivSufSort* can also be used to construct the Burrows-Wheeler transform directly without first constructing the suffix array. This will most likely be faster than first constructing the suffix array and then constructing the Burrows-Wheeler transform.

7. Experimental Evaluation

7.1 Setup

7.1.1 Hardware

All experiments are run on a 64-core NUMA AMD machine which was generously provided by the research group of Guy Blelloch at Carnegie Mellon University. The machine has 4 $\times$ 2.4 GHz 16-core AMD Opteron(tm) 6278 processors with each 64GiB of RAM. In total the machine thus has 64 physical cores and 256GiB RAM. The memory access from a CPU to it's local memory bank is typically faster than access to the other memory banks. This effect is called non-uniform memory access (NUMA). Our implementations are unaware of the NUMA architecture. However, we use a so called interleaved allocation policy. This means that memory is allocated in a stripped fashion over the different memory banks. In practice this NUMA policy showed good results when using 64 threads.

7.1.2 Compiler

The code is compiled with the GCC 4.8.0 cilkplus branch with the -O2 flag, as we use CilkPlus to express parallelism. Most of the code can also be compiled with OpenMP. However compiling it with CilkPlus showed better results on our machine.

7.1.3 Input Files

Two sets of input files were used for the benchmarks. For byte alphabets the Pizza&Chili corpus `http://pizzachili.dcc.uchile.cl` was used. Note that for the experiments the 1GiB version of the file "english" was used. The bigger version would have required the 64bit version of *DivSufSort* which would make comparing timings and memory usage more complicated. For integer alphabets randomly chosen integer sequences with alphabet sizes 2^k were used. All integer sequences consists out of 100 million non-negative integers. The benchmarks of the rank and select structures were run on the bitvectors of the balanced wavelet trees of the corresponding input files.

7.1.4 Methodology

To minimize variance the code of each benchmark was run multiple times and the fastest time was recorded. Reading the input and allocating memory were excluded from the reported times as good as possible. Memory consumption was measured with rusage from the GNU C library. We reported the maximum used memory over all runs of the code on a specific input file. Memory usage includes the space needed for the input and output. All algorithms are non-destructive regarding the input.

7.2 Results

7.2.1 Select/Rank on Bitvectors

The parallelized rank structure implementation scales well to 64 cores, as can be seen in Table 7.1. The parallel implementation uses two passes over the bitvector, while the sequential implementation only uses one pass. This explains why the parallel implementation, run with a single thread, is almost twice as slow as the sequential implementation. For some reason the parallel implementation of the select structure, run on a single thread, is around 3 times faster than the sequential implementation. Presumably, pipelining or cache effects could be the reason for this speedup. However, the parallel select implementation does not scale to more than 36 threads as can be seen in Figure 7.1. This is probably due to cache invalidation caused by the memory layout of the resulting select structure. In future work maybe better speedups can be achieved by first writing to buffers and then reorganising the memory layout.

	rank			serRank	select			serSelect
Input	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1
sources	0.93	0.028	33.0	0.5	2.5	0.22	11.0	4.42
pitches	0.24	0.0077	31.0	0.134	0.67	0.061	11.0	1.1
proteins	3.4	0.09	37.0	1.81	13.0	0.88	14.0	48.2
dna	0.88	0.027	33.0	0.493	3.9	0.3	13.0	24.4
english	4.9	0.13	36.0	2.69	13.0	1.2	12.0	27.7
dblp.xml	1.1	0.034	34.0	0.628	3.4	0.3	11.0	7.09

Table 7.1: Running times (in seconds) sequential, parallel and self-relative speedup of rank and select structure construction algorithms on 64 cores.

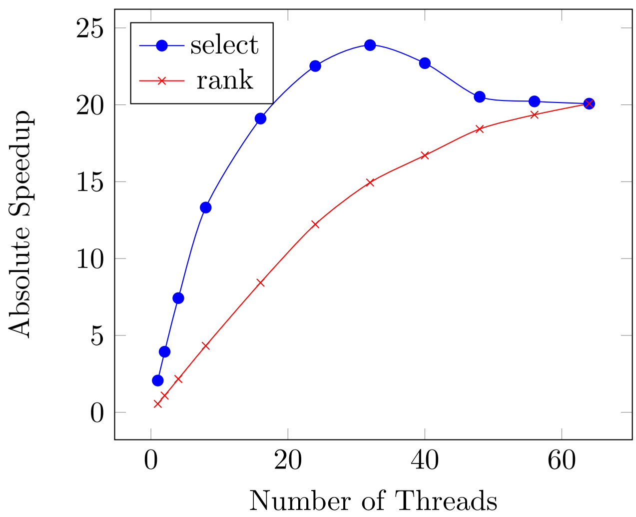

Figure 7.1: Speedup over serial construction algorithm for rank and select as function of number of threads on input file english

7.2.2 Parallel Wavelet-Tree Construction

In Table 7.2 we can see that the proposed memory efficient algorithms *recursiveWT* and *ddWT* outperform the *levelWT* implementation on all byte alphabet inputs and all but one integer alphabet input. Especially *recursiveWT* achieves very high speedups on all inputs. Shun's implementation of *levelWT* only achieves self-relative speedups of around $\times 20$. One reason could be the higher memory usage of *levelWT*. Shun uses smaller input files and a machine with more modern and thus faster memory system in his benchmarks [Shu15]. Hence, there were probably less issues with memory bandwidth in his benchmarks. If run on a single thread, the *recursiveWT* implementation is only slightly slower than the fastest sequential implementation. This leads to outstanding absolute speedups as shown in Figure 7.2.

	levelWT			recursiveWT			ddWT			serWT
Input	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1
sources	19.0	0.88	21.0	12.0	**0.26**	47.0	16.0	0.45	35.0	9.21
pitches	4.7	0.23	20.0	2.8	**0.068**	41.0	3.7	0.1	35.0	2.17
proteins	72.0	2.9	25.0	56.0	**1.1**	51.0	69.0	1.7	40.0	38.6
dna	16.0	0.72	22.0	12.0	**0.24**	49.0	15.0	0.47	33.0	8.65
english	99.0	4.4	22.0	65.0	**1.3**	49.0	110.0	2.2	49.0	49.5
dblp.xml	24.0	1.1	23.0	16.0	**0.34**	48.0	20.0	0.57	35.0	12.1
rnd-2^{8}	10.0	0.58	17.0	9.9	**0.27**	37.0	13.0	0.45	28.0	6.37
rnd-2^{10}	13.0	0.74	17.0	13.0	**0.33**	37.0	16.0	0.53	29.0	7.97
rnd-2^{12}	14.0	0.84	17.0	15.0	**0.38**	40.0	19.0	0.61	30.0	9.62
rnd-2^{14}	17.0	0.92	18.0	18.0	**0.43**	41.0	22.0	0.71	31.0	11.2
rnd-2^{16}	20.0	1.0	19.0	20.0	**0.48**	43.0	25.0	0.81	31.0	12.9
rnd-2^{18}	22.0	1.1	20.0	23.0	**0.53**	44.0	28.0	0.99	29.0	14.5
rnd-2^{20}	24.0	1.2	21.0	27.0	**0.61**	44.0	32.0	1.4	22.0	16.1

Table 7.2: Running times (in seconds) sequential, parallel and self-relative speedup of wavelet-tree construction algorithms on 64 cores.

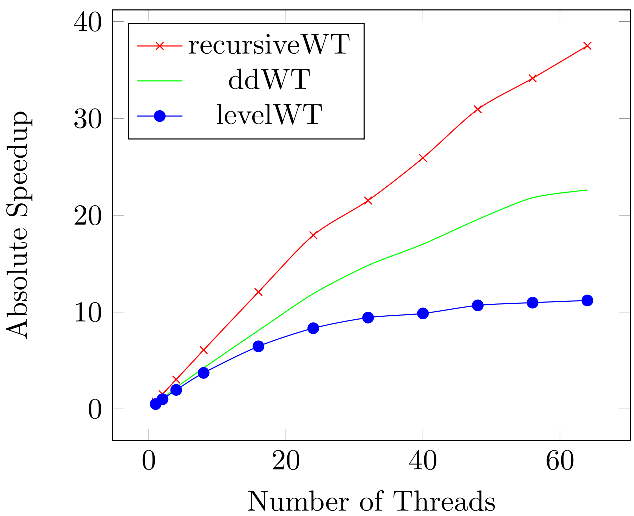

Figure 7.2: Speedup over serialWT as function of number of threads on input file english

7.2.3 Parallel Suffix-Array Construction

In our experiments the proposed *parallelDivSufSort*(Dss) algorithm performs best on all but the dna input file, as shown in Table 7.3. The used implementation probably scales badly on very small alphabets, as the dna file, because only the initial bucket sort of the string sorting is parallelized. In future work this issue can be fixed by using a parallelized string sorting algorithm. On the larger input files, as proteins and english, all algorithms achieve good self-relative speedups. As depicted in Figure 7.3 *parallelDivSufSort* achieves substantial speedup even compared to the sequential version of *DivSufSort*. In Figure 7.3 all implementations only partially scale past 24 threads. We account this to the complex memory access patterns of the algorithms and the low memory bandwidth of the used machine. In Table 7.4 the maximal memory consumption of the implementations during the construction is stated. *ParallelDivSufSort* uses multiple times less memory than the other parallel implementations and slightly more memory than the original *DivSufSort*. For example on the bigger input files, proteins and english, *parallelDivSufSort* uses less than 5% more space than the sequential in-place algorithm.

Input	KS			Range			Dss			serDss	serKS
	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1	T_1
sources	220	11	19	190	7.9	24	99	**4.8**	21	28	180
pitches	42	2.5	16	50	2.1	24	23	**1.4**	17	6.2	36
proteins	1900	65	29	1800	49	37	1100	**34**	32	300	1900
dna	480	20	24	280	**10**	27	190	13	15	80	440
english	1900	62	30	2700	78	35	1300	**41**	32	230	1800
dblp.xml	310	15	21	210	11	20	130	**7.0**	19	42	250

Table 7.3: Running times (in seconds) sequential, parallel and self-relative speedup of suffix-array construction algorithms on 64 cores.

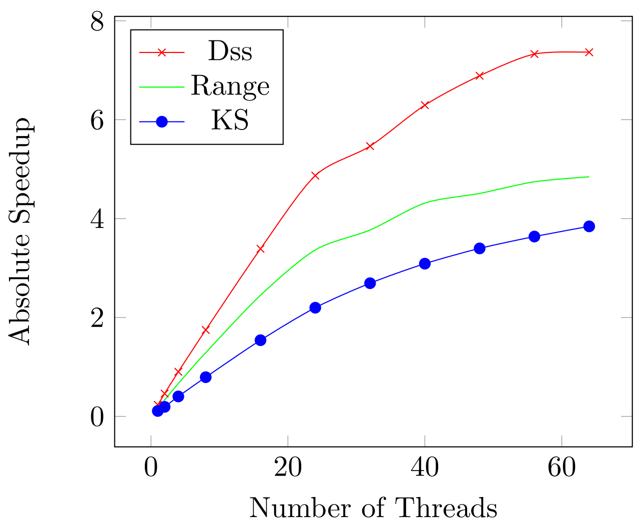

Figure 7.3: Speedup over DivSufSort as function of number of threads on input file english

7.2.4 Parallel FM-Index Construction

In the final experiments we show how the presented algorithms are used in the parallel version of the SDSL. We use the parallel rank/select structures, *parallelDivSufSort* and *recursiveWT* to construct the FM-index for the input files. In opposition to the experiments on wavelet-tree construction, here the Huffman-shaped wavelet tree is constructed. In Table 7.5 we see similar timings as in the suffix-array construction benchmarks. This is not surprising because the suffix-array construction is the most timing consuming step of the FM-index construction. Figure 7.4 shows the absolute speedup on different inputs

Input	KS	Range	Dss	serDss	serKS
sources	21.4	28.5	5.2	5.0	32.7
pitches	21.3	28.1	5.7	5.0	32.2
proteins	20.1	26.3	5.1	5.0	30.1
dna	21.4	27.8	5.5	5.0	32.8
english	21.5	28.9	5.2	5.0	32.9
dblp.xml	21.5	28.7	5.6	5.0	31.9

Table 7.4: Memory consumption (in byte per input character) of suffix-array construction algorithms on 64 cores.

in direct comparison with the sequential SDSL. Table 7.6 shows that the memory consumption of the parallel implementation is around one byte per input character higher than of its sequential counterpart. However, in this measurement the suffix array, the Burrows-Wheeler transform and the wavelet tree of the FM-index is included. To save memory most applications do not keep all the components in-memory throughout the construction. Typically, the suffix-array construction poses the memory bottleneck.

Input	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1
sources	180.0	7.5	24.0	45.2
pitches	42.0	2.0	21.0	10.7
proteins	1600.0	49.0	33.0	415
dna	290.0	17.0	17.0	114
english	1700.0	53.0	32.0	362
dblp.xml	240.0	10.0	24.0	72.1

Table 7.5: Running times (in seconds) sequential, parallel and self-relative speedup of FM-index construction algorithms on 64 cores.

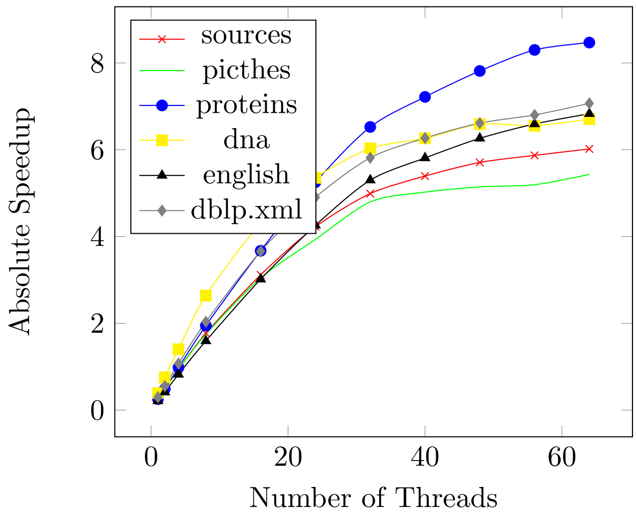

Figure 7.4: Speedup over SDSL implementation of FM-index construction as function of number of threads on input file english

Input	fm-index	fm-index-ser
sources	8.0	6.9
pitches	7.9	6.8
proteins	7.6	6.6
dna	7.2	6.1
english	7.7	6.6
dblp.xml	8.0	6.8

Table 7.6: Memory consumption (in byte per input character) of FM-index construction algorithms on 64 cores.

8. Future Work

There are numerous open questions and issues we have not addressed due to time constraints. Here we list possible implementation improvements, additional applications of the techniques presented and general ideas for future research in this field.

8.1 Implementation Improvements

As we have seen in the experimental evaluation, the parallel select implementation does not scale optimally to many cores. We assume this is due to cache invalidation as all threads write to rather small output arrays in parallel. This can probably be improved by changing the memory access pattern of the algorithm, for example by using intermediate buffers to write to. Additionally, on small alphabets the proposed parallel suffix-array construction algorithm has slight scaling issues. This is most likely due to the lack of parallelism in the string sorting implementation. There are two possible solutions to solve this issue. First, one can use a parallel string sorting implementation, for example as engineered by Bingmann, Eberle and Sanders [BES14]. Another solution is to use induced sorting, as used in the SA-IS algorithm [NZC09]. By design induced sorting does not use additional space and can be parallelized just like described in this work.

8.2 Additional Applications of the Techniques Presented

Even though this work focuses on shared memory architecture, some algorithms presented can be adapted to a distributed memory setting. For example the *ddWT* construction algorithm for wavelet trees can easily be implemented for distributed memory machines. If the input already is distributed among all nodes, then only the prefix sums and one all-to-all communication step is needed. Also, it may be beneficial to implement a hybrid approach combining *ddWT* and *recursiveWT*. For example if the algorithm is run on multiple distributed nodes where each node itself is a shared memory machine with multiple CPUs. First, *ddWT* can be used to decompose the input and distribute it to the nodes. Then, with *recursiveWT* each node builds a local wavelet tree on part of the input. Finally, *ddWT* again is used to combine the local wavelet trees to a final result.

For many applications only the Burrows-Wheeler transform (BWT) is needed. There are algorithms which directly compute the BWT from the text using less space than needed to store the suffix array. Again the fastest algorithms in practice use induced sorting. For example the algorithm by Okanohara and Sadakane [OS09] uses induced sorting. As with the *SA-IS* algorithm, the techniques described in this work can be applied if the alphabet size is small enough.

8.3 General Ideas

This work is only a first step towards utilizing modern multi-core architecture in the field of compressed indexes and succinct data structures. There are still many different rank and select structures on bitvectors with different properties with no parallel implementation available. For example compressed representations designed for sparse bitvectors. Even though our parallel implementation for wavelet trees can be used for all kind of different alphabet sizes and different shapes of wavelet trees, there are still variations of wavelet trees not covered. For example multiary wavelet trees or wavelet matrices were not covered. Additionally, no experiments were made on how the shape of the wavelet tree effects the performance of the proposed algorithms. Concerning parallel suffix array-construction, it would be a major breakthrough if induced sorting could be parallelized also for non-constant alphabet size. Then the lightweight linear work *SA-IS* algorithm could fully be parallelized. However, it remains uncertain if induced sorting, for non-constant alphabet size, can be parallelized in linear work and polylogarithmic depth.

9. Conclusion

In this work, we show how to parallelize a number of basic construction algorithms needed for compressed full-text indexes. We cover rank and select structures on bitvectors, wavelet trees and suffix arrays. Additionally, we show how to use the parallelized algorithms to construct FM-indexes in parallel. We implement all algorithms presented in this work, evaluate them experimentally, and make the implementations available to the public as part of the Problem Based Benchmark Suits (PBBS) and the Succinct Data Structure Library (SDSL).

We implement parallel construction algorithms for two rank and select structures used in practice as part of the SDSL. Our implementations scale well and thus are further used in our wavelet-tree and suffix-array implementations.

We reduce the memory requirement of the so far fastest parallel wavelet-tree construction algorithm from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n \log \sigma)$ bits. Two algorithms are introduced, *ddWT* and *recursiveWT*. *ddWT* is a variation of the domain decomposition algorithm proposed by Fuentes [Fea14]. We show how to adapt the domain decomposition approach to achieve polylogarithmic depth for small alphabets. In our experiments we show that our memory efficient algorithms both perform very well on real world inputs. Especially *recursiveWT* turns out to be very applicable in practice, as it is up to 4 times faster than existing parallel algorithms and can be used for all alphabet sizes. We provide an implementation of *recursiveWT* for different-shaped wavelet trees as part of the SDSL.

As suffix arrays are the basis of most compressed indexes, parallel lightweight contruction algorithms are needed. We propose *parallelDivSufSort*, which is a parallel version of the two-stage algorithm using induced sorting. As part of *parallelDivSufSort*, we show how to parallelize induced sorting for byte alphabets in polylogarithmic depth. Additionally, in our implementation the sequential tandem repeat sort is replaced with a parallel prefix doubling algorithm. We show how to implement prefix doubling fast and memory efficient, using the parallel select structure on bitvectors. With experiments on real world datasets we show that *parallelDivSufSort* is lightweight and fast. More precisely, on large input files our *parallelDivSufSort* implementation only uses around 5% more memory, while achieving speedups of up to 8 fold, compared to the highly optimized sequential counterpart.

All in all, the parallel implementations provided through this work can be used to build compressed indexes as the FM-Index fully in parallel. All algorithms and implementations are designed to be memory efficient. Our work shows that a focus on memory efficiency can also improve the performance of algorithms.

Bibliography

[BES14] T. Bingmann, A. Eberle, and P. Sanders, "Engineering parallel string sorting," *arXiv preprint arXiv:1403.2056*, 2014.

[Cla96] D. Clark, "Compact pat trees," Ph.D. dissertation, PhD thesis, University of Waterloo, 1996.

[CN09] F. Claude and G. Navarro, "Practical rank/select queries over arbitrary sequences," in *SPIRE*, 2009.

[DK13] M. Deo and S. Keely, "Parallel suffix array and least common prefix for the GPU," in *Symposium on Principles of Parallel Programming*, 2013, pp. 197–206.

[Fea14] J. Fuentes et al., "Efficient wavelet tree construction and querying for multicore architectures," in *SEA*, 2014.

[FM00] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Foundations of Computer Science*, 2000, pp. 390–398.

[Gea14] S. Gog et al., "From theory to practice: Plug and play with succinct data structures," in *SEA*, 2014.

[GGV03] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms.* Society for Industrial and Applied Mathematics, 2003, pp. 841–850.

[GP14] S. Gog and M. Petri, "Optimized succinct data structures for massive data," *Software: Practice and Experience*, vol. 44, no. 11, pp. 1287–1314, 2014.

[Hea09] R. Homann et al., "mkesa: enhanced suffix array construction tool," *Bioinformatics*, 2009.

[HT13] S. Hayashi and K. Taura, "Parallel and memory-efficient burrows-wheeler transform," in *BDA*, 2013.

[IT99] H. Itoh and H. Tanaka, "An efficient method for in memory construction of suffix arrays," in *SPIRE*, 1999.

[Jac89] G. Jacobson, "Space-efficient static trees and graphs," in *Foundations of Computer Science, 1989., 30th Annual Symposium on.* IEEE, 1989, pp. 549–554.

[JaJ92] J. JaJa, *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[KS03] J. Kärkkäinen and P. Sanders, "Simple linear work suffix array construction," in *Automata, Languages and Programming*, 2003, pp. 943–955.

[KS07] F. Kulla and P. Sanders, "Scalable parallel suffix array construction," *Parallel Computing*, vol. 33, no. 9, pp. 605–612, 2007.

[Lea13] X. Liu et al., “Pasqual: Parallel techniques for next generation genome sequence assembly,” *TPDS*, 2013.

[Lea15] Y. Liu et al., “Parallel and space-efficient construction of burrows-wheeler transform and suffix array for big genome data,” *TCBB*, 2015.

[LS07] N. J. Larsson and K. Sadakane, “Faster suffix sorting,” *Theor. Comput. Sci.*, 2007.

[MM93] U. Manber and G. Myers, “Suffix arrays: a new method for on-line string searches,” *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[MN05] V. Mäkinen and G. Navarro, “Succinct suffix arrays based on run-length encoding,” in *Combinatorial Pattern Matching.* Springer, 2005, pp. 45–56.

[Mun96] J. I. Munro, “Tables,” in *Foundations of Software Technology and Theoretical Computer Science.* Springer, 1996, pp. 37–42.

[Nav14] G. Navarro, “Wavelet trees for all,” *Journal of Discrete Algorithms*, vol. 25, pp. 2–20, 2014.

[NM07] G. Navarro and V. Mäkinen, “Compressed full-text indexes,” *ACM Computing Surveys (CSUR)*, vol. 39, no. 1, p. 2, 2007.

[NZC09] G. Nong, S. Zhang, and W. H. Chan, “Linear suffix array construction by almost pure induced-sorting,” in *DCC*, 2009.

[OS09] D. Okanohara and K. Sadakane, “A linear-time burrows-wheeler transform using induced sorting,” in *String Processing and Information Retrieval.* Springer, 2009, pp. 90–101.

[Osi12] V. Osipov, “Parallel suffix array construction for shared memory architectures,” in *SPIRE*, 2012.

[PST07] S. J. Puglisi, W. F. Smyth, and A. H. Turpin, “A taxonomy of suffix array construction algorithms,” *ACM Computing Surveys (CSUR)*, vol. 39, no. 2, p. 4, 2007.

[Sea12] J. Shun et al., “Brief announcement: the problem based benchmark suite,” in *SPAA*, 2012.

[Shu15] J. Shun, “Parallel wavelet tree construction,” in *DCC*, 2015.

[Vig08] S. Vigna, “Broadword implementation of rank/select queries,” in *Experimental Algorithms*, 2008, pp. 154–168.

[WBO15] L. Wang, S. Baxter, and J. D. Owens, “Fast parallel suffix array on the GPU,” *Euro-Par*, 2015.