Compressed Suffix Arrays for Massive Data

Jouni Sirén[⋆]

Department of Computer Science, University of Helsinki, Finland
jltsiren@cs.helsinki.fi

Abstract. We present a fast space-efficient algorithm for constructing compressed suffix arrays (CSA). The algorithm requires $O(n \log n)$ time in the worst case, and only $O(n)$ bits of extra space in addition to the CSA. As the basic step, we describe an algorithm for merging two CSAs. We show that the construction algorithm can be parallelized in a symmetric multiprocessor system, and discuss the possibility of a distributed implementation. We also describe a parallel implementation of the algorithm, capable of indexing several gigabytes per hour.

1 Introduction

Self-indexing [23] is a new approach for storing sequence data. The main idea is to combine the data and its index in a compressed structure, which provides random access to the data and supports various pattern matching queries. Some of the most relevant self-indexes are the *compressed suffix array (CSA)* [12] and the *FM-index* [8], both offering suffix array-like functionality.

With the explosive growth of sequential data in many applications such as genome browsers, version control systems, and online document collections, good search capabilities are becoming more and more important every day. This trend is making the self-indexes, combining small size with full-text searching, a promising approach for indexing large and massive data sets.

Obviously we need efficient practical algorithms for constructing these self-indexes, if we want them to truly live up to their promises. Unfortunately all the experiments reported so far have been performed with data sets at most a few gigabytes in size [24, 7, 14, 5, 17, 16], telling that the construction algorithms have trouble scaling up for massive data sets.

The typical way to construct a compressed self-index has been to use a regular suffix array construction algorithm [24]. While these algorithms are fast, they must store the data and the suffix array in main memory, making the memory requirements many times the size of the data. This is a major problem, especially with highly repetitive collections [26, 21], where the final index can be more than a hundred of times smaller than the suffix array.

Other alternatives have been to use secondary memory suffix array construction algorithms [4, 5], dynamic indexes [3, 19, 20, 11, 25], or algorithms for constructing the compressed index directly [15, 22, 13]. While these algorithms are

[⋆] Funded by the Academy of Finland under grant 119815. Part of the work was done while visiting NICTA Neville Roach Laboratory.

often memory efficient, they are also slow. Experiments have reported throughputs in the order of 100 kilobytes/second, which is more than an order of magnitude slower than the regular suffix array construction algorithms, and clearly too slow for data sets of tens of gigabytes or more.

The most promising algorithms are the distributed suffix array construction algorithm by Kulla et al. [17] and the space-efficient Burrows-Wheeler transform construction algorithm by Kärkkäinen [16]. Still, we must store either the suffix array in distributed memory or the entire data set in local memory, making both of the algorithms unsatisfactory for highly compressible data sets.

In this paper, we present a fast and space-efficient algorithm for direct CSA construction. The algorithm is related to the incremental suffix array construction algorithm by Gonnet et al. [10], as well as to the incremental CSA construction algorithm by Hon et al. [13]. Alternatively our algorithm can be thought of as replacing a dynamic CSA with a static structure and batch updates.

We start by some basic definitions in Sect. 2. Section 3 describes an algorithm for merging two compressed suffix arrays. Section 4 builds upon it, describing a parallelizable incremental CSA construction algorithm. The details of our implementation of the algorithm are discussed in Sect. 5. In Sect. 6, we validate the effectiveness of our algorithm experimentally. Finally, we discuss the possibility of a distributed implementation in Sect. 7.

2 Background Information

A *string* $S = S_{1,n} = s_1 s_2 \cdots s_n$ is a *sequence* of *symbols* (characters, letters). Each symbol is an element of an *alphabet* $\Sigma = \{1, 2, \ldots, \sigma\}$. A *substring* of S is written as $S_{i,j} = s_i \cdots s_j$. A substring of type $S_{1,j}$ is called a *prefix*, while a substring of type $S_{i,n}$ is called a *suffix*. We often assume that the string is an array, and refer to its symbols as $S[i]$, and to its substrings as $S[i, j]$. A *text* string $T = T_{1,n}$ is a sequence terminated by $t_n = \$ \notin \Sigma$ smaller than any symbol in Σ. The *lexicographic order* "$<$" among strings is defined in the usual way.

We call a set $\mathcal{C}$ of texts $T^1, T^2, \ldots, T^r$ a *collection*. The collection can be represented as a string $\mathcal{T} = T^1 T^2 \cdots T^r$. We denote the length of each text T^i as n_i, and the total length of the collection as $|\mathcal{C}| = |\mathcal{T}| = n$. Lexicographic order among such strings is defined in the usual way, except that each of the end markers \$ is considered a different symbol, so that every suffix of every string will be unique in the collection. If $\mathcal{T}[i] = \mathcal{T}[j] = \$$ and $i < j$, we define $\mathcal{T}[i] < \mathcal{T}[j]$. We informally call a collection *highly repetitive*, if most of its texts are highly similar to some other text in the collection. Examples of highly repetitive collections include individual genomes and different versions of a document.

The *suffix array* $SA[1, n]$ of a string S is an array of pointers to the suffixes of S in lexicographic order. As an abstract data type, a suffix array is any data structure providing similar functionality as the concrete suffix array. This can be defined by the following operations: (a) *count* the number of occurrences of a *pattern* in the string; (b) *locate* these occurrences (or more generally retrieve a suffix array value); and (c) *display* any substring of S.

The compressed suffix arrays discussed in this paper support these operations. Their compression is based on the *Burrows-Wheeler transform (BWT)* [2], a permutation created by sorting *cyclical strings*. The cyclical strings corresponding to string S are all strings of the form $CS_i = S_{i,n}S_{1,i-1}$, including $CS_1 = S$. The BWT is a sequence L such that $L[i] = CS_j[n]$, where CS_j is the ith cyclical string in lexicographic order. If S is a text or a collection, sorting cyclical strings is the same as sorting suffixes, as the first end marker encountered will end the comparison between two cyclical strings. In that case the BWT can be defined as $L[i] = S[SA[i] - 1]$, where $S[0] = S[n]$.

The Burrows-Wheeler transform is reversible. The reverse transform is based on a function called *LF-mapping* [2, 8] that is also used extensively in compressed self-indexes. The mapping is usually described by using an array $C[1, \sigma]$ such that $C[c]$ is the number of characters in $\{\$, 1, 2, \dots, c-1\}$ occurring in the collection. With this array and the sequence L, we can define LF-mapping as $LF(i) = C[L[i]] + rank_{L[i]}(L, i)$, where $rank_c(L, i)$ is the number of occurrences of character c in the prefix $L[1, i]$. We leave $LF(i)$ undefined when $L[i] = \$$, as each \$ is actually a different character. This is not a problem, as LF-mapping is not used for these positions in CSA operations.

LF-mapping and its inverse function Ψ [12] form the backbone of many compressed self-indexes. As $SA[LF(i)] = SA[i] - 1$ [8] and hence $SA[\Psi(i)] = SA[i] + 1$, we can use these functions to move the suffix array position backward and forward in the sequence. Both of the functions can be efficiently implemented by adding some extra information to a compressed representation of the BWT. Standard techniques [23] to support suffix array operations by using these functions include *backward searching* [8] for *count*, and adding a sample of suffix array values for *locate* and *display*.

The regular BWT is based on the cyclical strings of a single string. In this paper, we generalize the transform by allowing multiple strings, each of which can be a concatenation of several texts [9]. This makes it easier to merge the BWTs of two collections. We call the way the texts of a collection $\mathcal{A}$ are concatenated to form strings the *structure* of $\mathcal{A}$. Collection $\mathcal{B}$ contains the structure of $\mathcal{A}$, if $\mathcal{A} \subseteq \mathcal{B}$ and the texts of $\mathcal{A}$ are concatenated to form the same strings in the structures of $\mathcal{A}$ and $\mathcal{B}$.

The position of a character $T^i[j] \neq \$$ in the BWT is determined by the cyclical string of the string containing it starting at $T^i[j+1]$. As the first end marker encountered ends any comparison, we only need the suffix $T^i[j+1, n_i]$ to determine the position, as with the regular BWT. The position of the end marker $T^i[n_i]$ is determined by the text $T^{i'}$ following T^i in the cyclical strings. As each text is used to determine the position of exactly one end marker in the BWT, the structure of a collection does not affect its BWT.

3 Merging Compressed Suffix Arrays

Consider the collection $\{T^1, T^2\}$, where $T^1 = ababbaa\$$ and $T^2 = abbaa\$$. The BWTs of the texts are $aab\$bbaa$ and $aab\$ba$, respectively. Figure 1 shows a generalized Burrows-Wheeler transform of the collection, where the characters of text T^2 are marked with 1-bits in bit vector I.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\mathcal{T}$	a	$b$	a	$b$	$b$	a	a	\$	a	$b$	$b$	a	a	\$
SA	8	14	7	13	6	12	1	3	9	5	11	2	4	10
I	0	1	0	1	0	1	0	0	1	0	1	0	0	1
L	a	a	a	a	$b$	$b$	\$	$b$	\$	$b$	$b$	a	a	a
Ψ_a	1	1	1	1	0	0	0	0	0	0	0	1	1	1
Ψ_b	0	0	0	0	1	1	0	1	0	1	1	0	0	0

Fig. 1. A generalized Burrows-Wheeler transform L and the suffix array SA of collection $\{T^1, T^2\}$. Note that T^1 and T^2 are separate strings in the structure of the collection. One can get the regular BWT by changing which \$ belongs to which text.

We see that the marked characters form the BWT of T^2, while the other characters form the BWT of T^1. This is true in general as well. Assume we have two collections $\mathcal{A}$ and $\mathcal{B}$, where $\mathcal{B}$ contains the structure of $\mathcal{A}$. As the position of each character of $\mathcal{A}$ is determined by the same cyclical string in the BWTs of $\mathcal{A}$ and $\mathcal{B}$, the BWT of $\mathcal{A}$ is a *subsequence* of the BWT of $\mathcal{B}$.

Now let us turn our attention to the bit vectors Ψ_c marking the occurrences of character $c \in \Sigma$ in L. These vectors completely describe the BWT of the collection. We can perform the $rank_c(L, i)$ used in LF-mapping as $rank_1(\Psi_c, i)$. We can also compute $\Psi(j)$ as $select_1(\Psi_c, j - C[c])$, where $C[c] < j \leq C[c+1]$ and $select_1(\Psi_c, i)$ returns the position of the ith 1-bit in Ψ_c. Hence we can implement a self-index by compressing the bit vectors Ψ_c.

In fact, that is exactly what compressed suffix arrays [12], based on the function Ψ, already do. As the values of Ψ form an increasing sequence in the region of the suffix array corresponding to a character c, any representation of that part of Ψ is also a representation of the bit vector Ψ_c.

This gives us an idea for an algorithm to merge two compressed suffix arrays. If we have the CSAs of collections $\mathcal{C}_1$ and $\mathcal{C}_2$ and the bit vector I, we can use them to build a CSA for the combined collection. For each $c \in \Sigma$, we simply take the Ψ_c vectors of the two CSAs and merge them. Vector I is used to indicate how to interleave the bits from the two vectors. Sampled suffix array positions can be merged in a similar manner.

Let $n_i = |\mathcal{C}_i|$ and $n = n_1 + n_2$. With a suitable representation of the bit vectors (as in Sect. 5), we can merge the CSAs in-place in $O(|\mathrm{CSA}| + n_2\sigma)$ time, where $|\mathrm{CSA}|$ is the size of the resulting CSA. The $O(n_2\sigma)$ part comes from the fact that we have to scan the bit vector I once for every pair of vectors merged. This is not very efficient for large alphabets.

In such situations, it is better to merge the BWTs instead of the bit vectors. We can read the BWT of collection $\mathcal{C}_i$ from its CSA in $O(n_i)$ time by using a buffer of $\Omega(\sigma)$ characters. If we decompress both of the BWTs simultaneously, merge the buffers, and write the results immediately to the combined CSA, we can perform the merge in $O(n)$ time and $O(\sigma \log n)$ extra space for the buffer and bookkeeping.

The remaining question is, how to construct the bit vector I, denoting the ranks of every suffix of $\mathcal{B}$ in the combined suffix array. As the rank of a suffix is the sum of its ranks among the suffixes of $\mathcal{A}$ and $\mathcal{B}$ [13], we get the following algorithm for merging two CSAs:

2. **Search** for the ranks of the suffixes of $\mathcal{B}$ among the suffixes of $\mathcal{A}$ by backward searching [13]. Store the ranks in an integer array in any order.
3. **Sort** the array. Increment the values by their positions in the array (by the ranks of the suffixes of $\mathcal{B}$ among themselves) to get I.
4. **Merge** the BWTs of the two CSAs.

Searching takes $O(n_2 t_\Psi)$ time, where t_Ψ is the cost of one access to Ψ. We are not aware of any upper bounds better than $t_\Psi = O(\log n_1)$ (as in Sect. 5 with a logarithmic value for B) for CSAs that allow efficient merging. Array I requires $O(n_2 \log n)$ bits of space, and sorting it takes $O(n_2 \log n_2)$ time. Hence the entire algorithm takes $O(|\mathrm{CSA}| + n_2(\sigma + \log n))$ or $O(n + n_2 \log n)$ time, and works in $|\mathrm{CSA}| + O(n_2 \log n)$ or $|\mathrm{CSA}| + O((n_2 + \sigma) \log n)$ bits of space, respectively, with regular and BWT-based merging.

A similar algorithm can be used to remove sequences from the collection. We search for the positions of the suffixes to be removed, marking them on a bit vector I. Then we scan the bit vectors Ψ_c, removing bits as indicated by I.

4 CSA Construction

The algorithm for merging two compressed suffix arrays can be used as a building block for a CSA construction algorithm. The basic idea is to divide the collection into smaller ones, each of which can be indexed in limited memory, build CSAs for the parts, and merge the resulting partial indexes by using the algorithm in the previous section. For each part of the input, we first execute the *build* phase:

1. **Build** a CSA for the current input collection.

Then we merge the resulting partial index to the existing CSA by executing the *search*, *sort*, and *merge* phases.

Assume a collection of size n has been split into p parts of size n/p. Then, with any $O(n \log n)$ time and space suffix array construction algorithm, the *build* phases take a total of $O(n \log(n/p))$ time and require $|\mathrm{CSA}| + O((n/p) \log(n/p))$ bits of space. By using BWT-based merging, the other three phases require $O(pn + n \log n)$ time and $|\mathrm{CSA}| + O((n/p + \sigma) \log n)$ bits of space. If we assume $p = \Theta(\log n)$ and $\sigma = O(n/p)$, we get an algorithm requiring $O(n \log n)$ time and $|\mathrm{CSA}| + O(n)$ bits of space.

The algorithm can be parallelized with the following modifications:

1. **Build.** We can either build indexes for multiple input collections in parallel, increasing memory usage, or use a parallel suffix array construction algorithm such as [17].
2. **Search.** The ranks of the suffixes of a text are independent from the other texts in the input collection. Hence we can perform the search for multiple texts in parallel. If there are too few texts to distribute the searches evenly, we can try to split a search into multiple smaller ones. Assume we are searching for the ranks of the suffixes of text T backwards from position $T[j]$. If we find a substring $T[i, j]$ with no occurrences in the index, we can start reporting the ranks, as the symbols after $T[j]$ do not affect them.
3. **Sort.** Use a parallel sorting algorithm.
4. **Merge.** Multiple bit vector pairs can be merged in parallel. If there are more processors than bit vectors, work can be divided by splitting the vectors into multiple parts.

5 Implementation

We have implemented a sequential version of the algorithm, as well as a parallel version for *symmetric multiprocessor (SMP)* systems.[1] The implementation is written in in C++. The input is assumed to be divided into a number of files, each of them consisting of concatenated C-style 0-terminated strings. Each string is considered a separate text, with the trailing 0 interpreted as an end marker. The *build* phase is executed for all input files in the beginning of the construction to save memory. The resulting partial indexes as well as unused parts of the input are stored in secondary memory until needed.

We use two kinds of bit vectors in the implementation: gap encoded and run-length encoded. In gap encoding, the vector is encoded as a sequence of integers denoting the distances between the successive 1-bits, while in run-length encoding each run of 1s is encoded as the gap after the previous run followed by the length of the run. In both cases, δ codes [6] are used to encode the integers.

The compressed bit vectors are divided into blocks of B bytes. For each block, we sample the first 1-bit in the block, writing down its rank and position in the vector. Each sample takes $2 \log u$ bits, where u is the length of the vector. By using these samples, we can determine, which block to decompress to answer bit vector operations such as *rank* and *select*.

As a binary search among the samples is quite slow, we speed up the search by constructing secondary indexes for *rank* and *select* when the vector is loaded into memory. Both indexes consist of about $b/5$ integers of $\log b$ bits, where b is the number of blocks in the vector. For *rank*, the ith value is the number of the the block storing the first 1-bit at or after position $i \cdot 5u/b$. For *select*, the jth value is similarly the number of the block storing the 1-bit of rank $j \cdot 5n_o/b$, where n_o is the number of 1s in the vector. By using these indexes, we can limit the search to a (typically) small number of samples.

[1] The implementation is available at http://www.cs.helsinki.fi/group/suds/rlcsa/.

Instead of a compressed bit vector, we use a simpler structure as the indicator vector I in the *merge* phase. This structure is just an array of native 32-bit or 64-bit integers in increasing order, each of them indicating a 1-bit in the vector.

Our implementation of CSA is based on the Run-Length Compressed Suffix Array [26, 21]. We use a run-length encoded bit vector to represent each Ψ_c. This makes the index most suitable for highly repetitive collections, while some compression is lost on other types of collections.

Suffix array samples are marked in a gap encoded bit vector and stored as $\log(n/d)$-bit integers, where n is the size of the collection and d is the sample rate. Inverse suffix array samples used in *display* are constructed when the index is loaded, and stored as another array of $\log(n/d)$-bit integers. The end points of all sequences in the collection are marked in a gap encoded bit vector E.

The implementation supports multiple parallel queries. Each thread using the CSA maintains separate state information, while large arrays, such as samples and bit vector blocks, are shared between the threads. Large queries are not automatically split into smaller ones, but must be parallelized manually.

Locate queries are optimized for retrieving multiple occurrences simultaneously [21]. This greatly reduces the required number of accesses to Ψ and suffix array samples on highly repetitive collections.

We use the suffix array construction algorithm by Larsson and Sadakane [18] in the *build* phase because of its robustness with highly repetitive collections. The algorithm supports large alphabets, making it possible to use a different character value for each \$ in the collection. By limiting the size of the input files to less than 2 gigabytes, we can build the CSA for a file of size n_i in about $8n_i$ bytes. We build the indexes for multiple files in parallel, making this phase the most memory intensive one in the algorithm.

When the partial indexes have been built, we take one of them as the initial index, and begin merging the other indexes with it one at a time. We distribute the sequences in the input file dynamically between the threads, and report the ranks of the suffixes as either 32-bit or 64-bit integers. When all threads have finished searching, we sort the resulting array, and increment each value by its position in the array to get the bit vector I used in merging.

We merge the bit vectors instead of the BWTs in our implementation. Suffix array samples, bit vector E, and each of the bit vectors Ψ_c are merged as separate subtasks that are dynamically allocated to available threads. Large subtasks are not divided into smaller ones, which can be a problem with small alphabets, or when merging a large number of suffix array samples. In-place merging is not implemented, doubling the memory usage of the bit vectors being merged.

6 Experiments

We tested the performance of our new algorithms experimentally. The experiments were performed on a 16-core SMP system running Ubuntu Linux. The system had 128 gigabytes of memory and four quad-core Intel Xeon X7350 processors running at 2.93 GHz. All programs were compiled with GCC version

4.2.4. OpenMP was used for parallelization. MCSTL[2] was used to parallelize `std::sort`, as the GCC version in use did not support libstdc++ parallel mode.

Three data sets were used to test our construction algorithms: genome, enwiki, and fiwiki. Genome is the human reference genome (NCBI build 34), with 25 sequences as individual files for a total of 2.88 gigabytes. Enwiki and fiwiki are larger text collections downloaded from Wikipedia.[3] Enwiki contains a dump of the current versions of all English language Wikipedia articles (as of 2009-03-13), while fiwiki is a highly repetitive collection containing all Finnish language Wikipedia articles with their full version histories (as of 2009-01-22).

The Wikipedia data sets were in XML format, and had to be preprocessed before indexing. In the enwiki collection, we considered the lines between tags `<page>` and `</page>` as one sequence. In fiwiki, each sequence was contained between tags `<revision>` and `</revision>`. The extracted sequences were written into 500-megabyte input files. In this final form, enwiki contains 16080833 sequences in 85 files for a total of 41.48 gigabytes, while fiwiki contains 5849111 sequences in 87 files for a total of 42.03 gigabytes.

We tested our construction algorithm on the three data sets. The sequential implementation was used on the smaller genome data set, while the larger enwiki and fiwiki collections were indexed using the parallel implementation. Index parameters were mostly set to default ones. We used 32-byte block size on the run-length encoded Ψ_c vectors, and 16-byte block size on the gap encoded vectors. Suffix array sample rate was set to 64 on genome and enwiki data sets, and to 512 on the highly repetitive fiwiki data set. With these parameters, the final index sizes for genome, enwiki, and fiwiki were 2.18 GB, 17.37 GB, and 2.13 GB, respectively. Table 1 summarizes the construction.

Table 1. Results for index construction. The construction times are in hours, and the peak memory usage is in gigabytes. Throughput is measured in megabytes / second to make comparisons with earlier results easier.

			Construction Times					
Collection	**Threads**	**Memory**	**Build**	**Search**	**Sort**	**Merge**	**Total**	**MB/s**
genome	1	2.9	0.75	0.86	0.08	0.74	2.43	0.34
enwiki	8	36–37	3.25	1.88	0.37	3.42	9.00	1.31
	16	64	2.97	1.17	0.37	3.35	7.92	1.49
fiwiki	8	32	5.33	1.75	0.36	2.16	9.60	1.24
	16	64	5.01	1.22	0.38	1.99	8.62	1.39

We were able to index the human genome in about 145 minutes using less than 3 gigabytes of memory. Even considering the improvements in processor speeds and cache sizes, this is clearly better than the 24 hours and 3.6 gigabytes

[2] http://algo2.iti.uni-karlsruhe.de/singler/mcstl/

[3] http://download.wikipedia.org/

on a 1.7 GHz Pentium 4 system reported by Hon et al. [14]. By using in-place merging, we should be able to reduce our memory consumption by the final size of the largest bit vector (almost 500 megabytes). Further memory would be saved by replacing the run-length encoded bit vectors with gap encoded ones.

On the the enwiki and fiwiki collections, there was no significant speedup from 8 to 16 threads. Only the *search* phase that involves relatively complex operations on small pieces of data shows major improvement. This behavior is probably caused by cache and memory bus issues in the other phases that process large amounts of data sequentially. Another thing to note is that while merging the small indexes of the highly repetitive fiwiki collection was fast, building the partial indexes for it was much slower than for the enwiki collection.

We also tested our implementation by performing a large number of *count*, *locate*, and *display* queries using 1, 8, and 16 threads. We generated a set of random patterns for *count* and *locate* queries for each of the three collections. On the genome data set, this was 1000 patterns of length 10, with about 15.15 million total occurrences. We modified one pattern with over 2 million occurrences, as it dominated the query times in *locate*. For enwiki and fiwiki, we generated 40 random patterns of length 15 per input file, for a total of 3400 and 3480 patterns, respectively. Patterns with more than 10^5 occurrences were ignored in locate, making the total number of reported occurrences 16.89 million and 14.07 million, respectively.

Display queries consist of 10000 random prefixes of at most 10000 characters each. The total size of the extracted prefixes was 95.37 megabytes for genome, 17.64 megabytes for enwiki, and 38.90 megabytes for fiwiki. Table 2 shows the average query times. The results are mostly comparable with those in [21].

Table 2. Query times on the three data sets with a different number of (T)hreads. *(C)ount* and *(D)isplay* times are in microseconds / character, while *(L)ocate* times are in microseconds / occurrence.

	genome			**enwiki**			**fiwiki**		
T	**C**	**L**	**D**	**C**	**L**	**D**	**C**	**L**	**D**
1	1.532	34.155	0.611	2.272	10.603	0.992	1.704	37.081	0.829
8	0.238	4.816	0.082	0.337	1.475	0.137	0.262	5.163	0.116
16	0.211	2.914	0.044	0.221	0.899	0.080	0.177	4.091	0.070

The time required for one random access to the CSA is similar in all three collections. We got a significant improvement from 8 to 16 threads for the same reasons, as in the *search* phase of index construction. *Locate* performance was similar on genome and fiwiki with different sample rates, because of the optimizations for retrieving multiple occurrences. Enwiki was significantly faster, as it benefited both from the low sample rate and the optimizations.

We could not directly compare the performance of our algorithm to other similar algorithms. Of the few known implementations, the one by Hon et al. [14]

is not generally available. While Kärkkäinen's space-efficient BWT construction algorithm [16] is available, we could not compile it in a 64-bit environment. Finally, the dynamic *FM-index* by Gerlach [9] is outperformed by Kärkkäinen's algorithm in BWT construction, making comparisons to it redundant.

With this in mind, we compared our sequential algorithm to Kärkkäinen's algorithm on BWT construction. The comparison was performed on a 2.66 GHz Intel Core 2 Duo E6750 desktop system with 4 GB of memory (3.2 GB visible to OS). We downloaded the 1.10 GB protein sequence collection from the Pizza & Chili Corpus [7], and split it into 5, 10, and 20 parts for our algorithm. We used parameter values $v = 128$ (default), 1024, and 4096 for Kärkkäinen's algorithm. The results can be seen in Table 3. While Kärkkäinen's algorithm was faster with default parameters, our algorithm performed better with limited memory. We also achieved a reasonable speed while using less memory than the input size, which is impossible with Kärkkäinen's algorithm.

Table 3. Construction times for Burrows-Wheeler transform. The BWT is not included in the memory consumption, as both implementations write it directly to disk.

	Our Algorithm			**Kärkkäinen's**		
	5	**10**	**20**	**128**	**1024**	**4096**
Time (minutes)	35	41	46	29	46	68
Memory (GB)	1.86	1.29	1.02	2.00	1.45	1.28

7 Discussion

We have presented a parallel algorithm for constructing compressed suffix arrays, and demonstrated its practical effectiveness by indexing tens of gigabytes with a throughput of about 4–5 gigabytes / hour. When the collection is highly repetitive, this can be done in memory available on today's high-end desktop systems, except for the *build* phase of the algorithm. Hence if we distribute the building of partial indexes to multiple systems, it should be feasible to index collections of hundreds of gigabytes in size with the current implementation.

We actually considered indexing the German language Wikipedia with full version history – a 933 GB highly repetitive collection. The plan was to use two older SMP systems (both with 8 cores and 32 GB of RAM) to index 10-gigabyte parts, and to merge the partial indexes on the larger system. Extrapolating from the results with the Finnish language Wikipedia, this should have taken about four days. However, due to the need for exclusive access to the systems, the experiment had to be postponed.

This naturally leads to the question, whether a true distributed implementation of the algorithm is possible. The answer seems to be yes. In addition to the *build* phase, *sort* and *merge* phases are also relatively easy to distribute. Sorting is one of the fundamental operations in distributed computing, with many efficient practical solutions, as is made evident by the Sort Benchmark.[4] On the other hand, merging can easily be split into as many independent tasks as necessary, making its distribution straightforward.

Search phase is the hardest one to distribute, as solving it will probably require a distributed CSA. As long as the CSA fits into the memory of a single node, things are easy. We can just have a copy of the CSA in each node, and distribute the sequences between the nodes. When the index grows larger, we must either store it in secondary memory, or distribute it among the nodes (or both in case of very large collections).

Using secondary memory yields a major performance loss, as we need one random access to the CSA for each character inserted. While a sequential search can process more than 1 MB/s, hard disks allow at most a few hundred random accesses per second. Although modern solid-state drives are much faster, allowing tens of thousands of random accesses per second, they are still about 30–40 times slower than the CPU. With one solid-state drive, one might get a 100 megabytes / hour throughput, so with many drives reasonable speeds could be attained.

Storing the CSA in distributed memory creates different performance problems. Network latency becomes the main factor in sequential search speed, as nodes must communicate with each other to access different parts of the CSA. On the other hand, large bandwidth makes it possible to search for many sequences in parallel, alleviating the problem. If many queries directed to the same node are grouped into one packet, a 5–10 MB/s ($\log n$ bits / character) data stream should be enough for one CPU core.

With this algorithm, distributed construction of CSAs seems feasible for multi-terabyte collections. Much of the work can even be performed on a production system, as new data arrives. Significant resources are only required for the final merging of the indexes. The real question is, can the algorithm be extended for the other structures required for suffix tree functionality [1]. If the answer is positive, it could make compressed suffix arrays the data structure of choice for many applications, such as large-scale analysis of genome data.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal on Discrete Algorithms*, 2(1):53–86, 2004.
2. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
3. H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2):21, 2007.
4. A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.
5. R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *Journal of Experimental Algorithmics*, 12:3.4, 2008.
6. P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[4] http://www.hpl.hp.com/hosted/sortbenchmark/

7. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithms*, 13:11, 2008.
8. P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.
9. W. Gerlach. Dynamic FM-index for a collection of texts with application to space-efficient construction of the compressed suffix array. Master's thesis, Bielefeld University, 2007.
10. G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In *Information retrieval: data structures and algorithms*, pages 66–82. Prentice-Hall, 1992.
11. R. González and G. Navarro. Improved dynamic rank-select entropy-bound structures. In *LATIN 2008*, volume 4957 of *LNCS*, pages 374–386. Springer-Verlag, 2008.
12. R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.
13. W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1):23–36, 2007.
14. W.-K. Hon, T.-W. Lam, W.-K. Sung, W.-L. Tse, C.-K. Wong, and S.-M. Yiu. Practical aspects of compressed suffix arrays and FM-index in searching DNA sequences. In *ALENEX 2004*, pages 31–38. SIAM, 2004.
15. W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM Journal on Computing*, 38(6):2162–2178, 2009.
16. J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007.
17. F. Kulla and P. Sanders. Scalable parallel suffix array construction. *Parallel Computing*, 33(9):605–612, 2007.
18. N. J. Larsson and K. Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.
19. S. Lee and K. Park. Dynamic rank-select structures with applications to run-length encoded texts. In *CPM 2007*, volume 4580 of *LNCS*, pages 95–106. Springer-Verlag, 2007.
20. V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):32, 2008.
21. V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of individual genomes. In *RECOMB 2009*, volume 5541 of *LNCS*, pages 121–137. Springer-Verlag, 2009.
22. J. C. Na and K. Park. Alphabet-independent linear-time construction of compressed suffix arrays using o(nlogn)-bit working space. *Theoretical Computer Science*, 385(1-3):127–136, 2007.
23. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2, 2007.
24. S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):4, 2007.
25. M. Salson, T. Lecroq, M. Léonard, and L. Mouchard. Dynamic extended suffix arrays. Accepted to Journal of Discrete Algorithms.
26. J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *SPIRE 2008*, volume 5280 of *LNCS*, pages 164–175. Springer-Verlag, 2008.