CHICO: A Compressed Hybrid Index for Repetitive Collections

Daniel Valenzuela(✉)

Department of Computer Science,
Helsinki Institute for Information Technology HIIT,
University of Helsinki, Helsinki, Finland
dvalenzu@cs.helsinki.fi

Abstract. Indexing text collections to support pattern matching queries is a fundamental problem in computer science. New challenges keep arising as databases grow, and for repetitive collections, compressed indexes become relevant. To successfully exploit the regularities of repetitive collections different approaches have been proposed. Some of these are Compressed Suffix Array, Lempel-Ziv, and Grammar based indexes.

In this paper, we present an implementation of an hybrid index that combines the effectiveness of Lempel-Ziv factorization with a modular design. This allows to easily substitute some components of the index, such as the Lempel-Ziv factorization algorithm, or the pattern matching machinery.

Our implementation reduces the size up to a 50 % over its predecessor, while improving query times up to a 15 %. Also, it is able to successfully index thousands of genomes in a commodity desktop, and it scales up to multi-terabyte collections, provided there is enough secondary memory. As a byproduct, we developed a parallel version of Relative Lempel-Ziv compression algorithm.

1 Introduction

In 1977 Abraham Lempel and Jacob Ziv developed powerful compression algorithms, namely LZ77 and LZ78 [29,30]. Almost forty years from their conception, they remain central to the data compression community. LZ77 is among the most effective compressors which also offers extremely good decompression speed. Those attributes have made it the algorithm of choice for many compression utilities like zip, gzip, 7zip, lzma, and the GIF image format.

These algorithms are still being actively researched [1,3,10,16], and with the increasing need to handle ever-growing large databases the Lempel-Ziv family of algorithms still has a lot to offer.

Repetitive datasets and the challenges on how to index them are actively researched at least since 2009 [21,22,27]. Canonical examples of such datasets are biological databases, such as the 1000 Genomes projects, UK10K, 1001 plant genomes, etc.

Among the different approaches to index repetitive collections, we will focus in one of the Lempel-Ziv based indexes, the Hybrid Index of Ferrada et al. [7].

© Springer International Publishing Switzerland 2016
A.V. Goldberg and A.S. Kulikov (Eds.): SEA 2016, LNCS 9685, pp. 326–338, 2016.
DOI: 10.1007/978-3-319-38851-9_22

Their approach is very promising, specially to be used in biological databases. Among the features that make the Hybrid Index attractive, is that it offers a framework to answer approximate pattern matching queries. Also, by design it does not stick to any specific (approximate) pattern matching technique. For instance, if the index was built to align genomic reads to a reference, it could use a tool like BWA that is highly specialized for this kind of queries (by taking care of the reverse complements, considering quality scores, specificities of the sequencing technologies, etc.).

In this paper we present an improved version of that index that achieves up to a 50 % reduction in its space usage, while maintaining or improving query times up to a 15 %. We achieve faster building time than other indexes for repetitive collections. For collections in the tens of gigabytes, our index can still be built in a commodity machine. Using a more powerful server, we successfully indexed a collection of 201 GB in about an hour, and a collection of 2.4 TB in about 12 h.

A key element to achieve those results is that our index does not stick to a particular Lempel-Ziv factorization. Instead, it accepts the LZ77 parsing as input, and also a variety of LZ77-like parsings. That allows our index to offer appealing trade-offs between indexing time, index size, and resource usage during construction time. As a byproduct, we developed a parallel Relative Lempel-Ziv parser that can parse 2.4 TB in about 10 h using 24 cores.

The structure of this paper is as follows. In Sect. 2 we briefly discuss previous work. In particular we discuss relevant aspects and variants of the Lempel-Ziv parsing. In Sect. 3 we review the fundamental ideas behind the Hybrid Index of Ferrada et al. In Sect. 4 we present our improvements on the Hybrid Index and we discuss how it compares with the original proposal. In Sect. 5 we discuss the practicalities of our implementation. In Sect. 6 we show the experimental behavior of our index. Finally in Sect. 7 we discuss our findings and future research.

2 Related Work

The idea of exploiting repetitiveness to decrease the space requirement of indexing data structures can be traced back at least to Kärkkäinen and Ukkonen [17].

In the string processing community, a full-text index that uses space proportional to that of the compressed text is called a *compressed index* [25]. The main strategies are based on the compression of the suffix array, the compression of the Burrows Wheeler transform, or the use of the Lempel-Ziv parsing. For further reading, we refer the reader to a survey covering the theoretical aspects of compressed full-text indexes [25] and to a recent experimental study [12].

2.1 Repetitive Collections

A pioneer work that formally introduced the challenges of indexing repetitive collection and that offered a successful solution was the Run Length Compressed Suffix Array [21,22]. In parallel, in the context of plant genomics, Schneeberger et al. [27] faced similar challenges, and they solved them by using a q-gram index.

Full-Text Indexing. Since the publication of the Run Length Compressed Suffix Array (RLCSA) [21,22], many techniques have been developed to tackle this problem [24]. Some of them are mainly theoretical works [2,6,11], and some have been implemented [4,7,19,26]. However, to the best of our knowledge, the only one that successfully scales to collections in the tens of gigabytes is the RLCSA.

Bioinformatics Tools. Some indexes are specially tailored for biological databases. One of the foundational works handling large repetitive datasets is GenomeMapper [27]. Some recent works, like MuGI [5], also take as an input a *reference* sequence and a file that informs about the variations present in a set of sequences corresponding to individuals. Other tools [23,28] consider the input to be a (multiple) sequence alignment. This assumption, even though natural in some contexts, restricts the usability of those solutions.

2.2 LZ77

Lempel-Ziv is a family of very powerful compression algorithms that achieve compression by using a dictionary to encode substrings of the text. Here we focus on the LZ77 algorithm, in which the dictionary (implicitly) contains all the substrings of the part of the text that has already been processed.

The compression algorithm consists on two phases: *parsing* (also called *factorization*) and *encoding*. Given a text $T[1,N]$, a *LZ77 valid parsing* is a partition of T into z substrings T^i (often called *phrases* or *factors*) such that $T = T^1T^2 \ldots T^z$, and for all $i \in [1,z]$ either there is at least one occurrence of T^i with starting position strictly smaller than $|T^1T^2 \ldots T^{i-1}|$, or T^i is the first occurrence of a single character.

The *encoding* process represents each phrase T^i using a pair (p_i, l_i), where p_i is the position of the previous occurrence of T_i and $l_i = |T^i|$, for phrases that are not single characters. When $T^i = \alpha \in \Sigma$, then it is encoded with the pair $(\alpha, 0)$. We call the latter *literal phrases* and the former *copying phrases*.

Decoding LZ77 compressed text is particularly simple and fast: the pairs (p_i, l_i) are read from left to right, if $l_i = 0$, then p_i is interpreted as a char and it is appended to the output, if $l_i \neq 0$, then l_i characters are copied from the position p_i to $p_i + l_i - 1$ and are appended to the current output.

Greedy Parsing. So far we have defined what is a *LZ77 valid parsing*, but we have not discussed how to compute such a parsing. Indeed, there are different possibilities. It is a common agreement in the literature to reserve the name LZ77 for the case when the parsing is a *greedy parsing*. That is, if we assume that we have already parsed a prefix of T, $T' = T^1T^2 \ldots T^p$ then the phrase T^{p+1} must be the longest substring starting at position $|T^1T^2 \ldots T^p| + 1$ such that there is a previous occurrence of T^{p+1} in T starting at some position smaller than $|T^1T^2 \ldots T^p|$. There are numerous reasons to choose the *greedy parsing*. One of them is that it can be computed in linear time [15]. Another reason is that the parsing it produces is optimal in the number of phrases [8]. Therefore, if the pairs are encoded using any fixed-length encoder, the *greedy parsing* always produces the smaller representation. Moreover, various authors [18,29] proved that greedy parsing achieves asymptotically the (empirical) entropy of the source generating the input string.

3 Hybrid Index

The Hybrid Index of Ferrada et al. [7] extends the ideas of Kärkkäinen and Ukkonen [17] to use the Lempel-Ziv parsing as a basis to capture repetitiveness. This Hybrid Index combines the efficiency of LZ77 with any other index in a way that can support not only exact matches but also approximate matching.

Given the text to be indexed T, a LZ77 parsing of T consisting of z phrases, and also the maximum length M of a query pattern and the maximum number of mismatches K, the Hybrid Index is a data structure using space proportional to z and to M that supports approximate string matching queries. That is, it is able to find all the positions i in T such that $ed(T[i,|P|-1], P[1,|P|-1]) \leq K$ for a given query pattern P, where $ed(x,y)$ stands for the edit distance between strings x and y.

Let us adopt the following definitions [7,17]: A *primary occurrence* is an (exact or approximate) occurrence of P in T that spans two or more phrases. A *secondary match* is an (exact or approximate) occurrence of P in T that is entirely contained in one phrase. Kärkkäinen and Ukkonen [17] noted that every secondary match is an exact copy of a previous (secondary or primary) match. Therefore, the pattern matching procedure can be done in two stages: first all the primary occurrences are identified, then, using the structure of the LZ77 parse, all the secondary matches can be discovered.

3.1 Kernelization to Find Primary Occurrences

Conceptually, the kernel string aims to get rid of large repetitions in the input text, and extract only the non-repetitive areas. To do that, it extracts the characters in the neighborhoods of the phrase boundaries, while discarding most of the content of large phrases.

More formally, given the LZ77 parsing of T, the kernel text $\mathcal{K}_{M,K}$ is defined as the concatenation of characters within distance $M+K-1$ from their nearest phrase boundaries. Characters not contiguous in T are separated in $\mathcal{K}_{M,K}$ by $K+1$ copies of a special separator #. It is important to note that for any substring of T with length at most $M+K$ that crosses a phrase boundary in the LZ77 parse of T, there is a corresponding and equal substring in $\mathcal{K}_{M,K}$.

To be able to map the positions from the kernel text to the original text the Hybrid Index uses two sorted lists with the phrase boundaries. L_T stores the phrase boundaries in T and $L_{\mathcal{K}_{M,K}}$ stores the list of phrase boundaries in $\mathcal{K}_{M,K}$.

The kernel text does not need to be stored explicitly; what is required is the ability to query it and, for that reason, the Hybrid Index stores an index $I_{M,K}$

that supports the desired queries on $\mathcal{K}_{M,K}$ (e.g. exact and approximate pattern matching).

By construction of $\mathcal{K}_{M,K}$, it is guaranteed that all the primary matches of P occur in $\mathcal{K}_{M,K}$. However, there are also some secondary matches that may appear on $\mathcal{K}_{M,K}$. When a query pattern P, $|P| \leq M$ is given the first step is to query the index $I_{M,K}$ to find all the matches of P in $\mathcal{K}_{M,K}$. These are all the potential primary occurrences. They are mapped to T using L and L_M. Those matches that do not overlap a phrase boundary are discarded. For queries of length one, special care is taken with matches that corresponds to the first occurrence of the character [7].

3.2 Reporting Secondary Occurrences

The secondary occurrences are reported using *2-sided range reporting* [7]. The idea is that, once the positions of the primary occurrences are known, the parsing information is used to discover the secondary matches. Instead of looking for theoretically optimal data structures to solve this problem, the Hybrid Index proposes a simple and practical way to solve it.

Each phrase (pos, len) of the LZ77 parsing can be expressed as triplets $(x, y) \rightarrow w$ where $(x = pos, y = pos + len)$ is said to be the source, and w is the position in the text that is encoded with such phrase. The sources are sorted by the x coordinate, and the sorted x values are stored in an array $\mathcal{X}$. The corresponding w positions are stored in an array $\mathcal{W}$. The values of the y coordinates are not explicitly stored. However, a position-only Range Maximum Query [9] data structure is stored for the y values.

The 2-sided recursive reporting procedure works as follows: For a given primary occurrence in position pos, the goal is to find all the phrases whose source entirely contains $(pos, pos + |P| - 1)$. To do that the first step is to look for the position of the predecessor of pos in $\mathcal{X}$. The second step is to do a range maximum query of the y values in that range. Even though y is not stored explicitly, it can be easily computed [7]. If that value is smaller than $pos + |P| - 1$ the search stops. If the y value is equal or larger than $pos + |P| - 1$, it corresponds to a phrase that contains the primary position. Then, the procedure recurses on the two intervals that are induced by it. For further details we refer the reader to the Hybrid Index paper [7].

4 CHICO: Beyond Greedy LZ77

When we described the LZ77 algorithm in Sect. 2.2, we first used a general description of the LZ parsing. We noted that in the description of the Hybrid Index the parsing strategy is not specified, and indeed, it does not affect the working of the index. Therefore, the index works the same way with any *LZ77 valid parsing* where the phrases are either single characters (*literal phrases*, or pairs (pos, len) representing a reference to a previously occurring substring (*copying phrases*).

The greedy parsing is usually the chosen scheme because it produces the minimum number of phrases. This produces the smallest possible output if a plain encoding is used. If other encoding schemes are allowed, it has been proven that the greedy parsing does not imply the smallest output anymore. Moreover, different parsing algorithms exists that provide bit-optimality for a wide variety of encoders [8].

4.1 Reducing the Number of Phrases

It is useful to note that all the phrases shorter than $2M$ are entirely contained in the kernel text. Occurrences that are entirely contained in such phrases are found using the index of the kernel string $I_{M,K}$. Then they are discarded, just to be rediscovered later by the recursive reporting procedure.

To avoid this redundancy we modify the parsing in a way that phrases smaller than $2M$ are avoided. First we need to accept that *literal phrases* (those pairs $(\alpha, 0)$ such that $\alpha \in \Sigma$) can be used not only to represent characters but also to hold longer strings $(s, 0)$, $s \in \Sigma^*$.

To actually reduce the number of phrases in the recursive data structure we designed a **phrase merging** procedure. The phrase merging procedures receives a LZ77 parsing of a text, and produces a parsing with less phrases, using *literal phrases* longer than one character. The procedure works as follows. It reads the phrases in order, and when it finds a *copying phrase* (pos, len) such that $len < 2M$, it is transformed into a *literal phrase* $(T[pos, pos + len - 1], 0)$. That is, a *literal phrase* that decodes to the same string. If two *literal phrases* $(s_1, 0)$ and $(s_2, 0)$ are consecutive, they are merged into $(s_1 \circ s_2, 0)$ where $\circ$ denotes string concatenation. It is clear that the output parsing decodes to the same text as the input parsing. Moreover, the output parsing produce the same kernel text $\mathcal{K}_{M,K}$ as the input parsing.

Because the number of phrases after the phrase merging procedure is strictly smaller, the space needed for the recursive reporting data structure also decreases. In addition, the search space for the recursive reporting queries shrinks.

4.2 Finding the Occurrences

We extend the definition of primary occurrences as follows: A *primary occurrence* is an (exact or approximate) occurrence of P in T that either crosses one or more phrase boundary, or it lies entirely within a *literal phrase*. To ensure that every occurrence is reported once and only once we store a bitmap $F[1..z]$ such that $F[i] = 1$ if the i-th phrase is a *literal phrase* and $F[i] = 0$ otherwise.

Hence, when a query pattern P is processed, first $I_{M,K}$ is used to find potential primary occurrence. Then the only occurrences to be discarded are those that lie entirely within *copying phrases*. Using this approach, there is also no need to handle a special case for queries of length one.

4.3 RLZ: A Faster Building Algorithm

A classical way to reduce the parsing time of the LZ77 algorithm is to use a smaller dictionary. For instance, the *sliding window* approach constrains the parsing algorithm to look for matches only in the ω last positions of the processed text. This approach is used in the popular gzip program.

A recent approach that presents a different modification to the LZ77 algorithm is the Relative Lempel-Ziv algorithm (RLZ) [20]. Here, the dictionary is not a prefix of the text. Instead, the dictionary is a different text that is provided separately. We note that while the sliding window approach still generates a *LZ77 valid parsing*, the RLZ algorithm does not. Therefore, the former one is a valid input for the Hybrid Index, while the second is not.

To make the RLZ algorithm compatible with the Hybrid Index, a natural way would be to prepend the reference to the input text. In that way, the first phrase would be an exact copy of the reference. We mark this phrase as a *literal phrase* to ensure all its contents goes to the kernel text. One caveat of this approach is that if the reference is compressible we would not exploit it. An alternative way to modify the RLZ algorithm without having this trouble is as follows: We parse the input text using the traditional RLZ algorithm, to obtain a parse $\mathcal{P}_T$. Then, we parse the reference using the LZ77 greedy parsing, to obtain a parse $\mathcal{P}_R$. It is easy to see that $\mathcal{P}_R \circ \mathcal{P}_T$ is a LZ77 *valid parsing* of $R \circ T$.

5 Implementation

We implemented the index in C++, relying on the Succinct Data Structure Library 2.0.3 (SDSL) [12] for most succinct data structures, such as the RMQ [9] data structure on the Y array.

We encoded the phrase boundaries L_T, $L_{K,M}$ and the x-values of the sources in the $\mathcal{X}$ array using SDSL elias delta codes implementation. Following the ideas of the original Hybrid Index we did not implement specialized data structure for predecessor queries on arrays $\mathcal{X}$ and $L_{K,M}$. Instead, we sampled these arrays and perform binary searches to find the predecessors.

For the index of the kernel text $I_{K,M}$ we used a SDSL implementation of the FM-Index. As that FM-Index does not support approximate searches natively, we decided to exclude those queries in our experimental study.

For further details, the source code of our implementation is publicly available at https://www.cs.helsinki.fi/u/dvalenzu/software/.

5.1 LZ77

To offer different trade-offs we included alternative modules for LZ77 factorization, all of them specialized in repetitive collections. The default construction algorithm of the index uses **LZscan** [14], an in-memory algorithm that computes the LZ77 greedy parsing.

To handle larger inputs, we also included an external memory algorithm called **EM-LZscan** [16]. An important feature is that it allows the user to specify the amount of memory that the algorithm is allowed to use.

5.2 Relative Lempel-Ziv

The third parser that we included is our own version of Relative Lempel-Ziv **RLZ**. Our implementation is based on the Relative Lempel-Ziv of Hoobin et al. [13]. This algorithm uses the suffix array of the reference and then it uses it to compute the parsing of the input text.

Our implementation differs in two aspects with the original algorithm. The first (explained in Sect. 4.3) is that the reference itself is also represented using LZ77. To compute the LZ77 parsing of the reference we use the KKP algorithm [15]. The second difference is that instead of using an arbitrary reference, we restrict ourselves to use a prefix of the input text. In that way there is no need to modify the input text by prepending the reference (see Sect. 4.3).

5.3 Parallel Relative Lempel-Ziv

We implemented a fourth parser, **PRLZ**, which is a parallel version of RLZ. This is a simple yet effective parallelization of our RLZ implementation. The first step of the algorithm is to build the suffix array of the reference. This is done sequentially. Then, instead of processing the text sequentially, we split it into chunks and process them in parallel. Each chunk is assigned to a different thread and the thread computes the RLZ parsing of its assigned chunk using the reference. This is easily implemented using OpenMP. Using a moderate number of chunks (e.g. the number of available processors) we expect similar compression ratios to those achieved by a sequential RLZ implementation.

6 Experimental Results

We used collections of different kinds and sizes to evaluate our index in practice. In the first round of experiments we used some repetitive collections from the pizzachilli repetitive corpus[1].

`Einstein`: All versions of Wikipedia page of Albert Einstein up to November 10, 2006. Total size is 446 MB.
`Cere`: 37 sequences of Saccharomyces Cerevisiae. Total size is 440 MB.
`Para`: 36 sequences of Saccharomyces Paradoxus. Total size is 410 MB.
`Influenza`: 78, 041 sequences of *Haemophilus Influenzae*. Total size is 148 MB.
`Coreutils`: Source code from all 5.x versions of the *coreutils* package. Total size is 196 MB.

We extracted 1000 random patterns of size 50 and 100 from each collection to evaluate the indexes. In addition, we generated two large collections using data from the 1000 genomes project:

$\texttt{CHR}_{21}$: 2000 versions of Human Chromosome 21. Total size 90 GB.

[1] http://pizzachili.dcc.uchile.cl/.

$\texttt{CHR}_{14}$: 2000 versions of Human Chromosome 14. Total size 201 GB.
$\texttt{CHR}_{1...5}$: 2000 versions of Human Chromosomes 1, 2, 3, 4 and 5. Total size 2.4 TB.

To demonstrate the efficiency of our index, we ran most of our experiments in a commodity computer. This machine has 16 GB of RAM and 500 GB of hard drive. The operative system is Ubuntu 14.04.3. The code was compiled using gcc 4.8.4 with full optimization.

In our first round of experiments we compared CHICO with the original Hybrid Index on some collections of the pizzachilli corpus. As the size of those collections is moderate, we ran our index using the in-memory version of LZscan. We also compared with the LZ77 Index of Kreft and Navarro [19], and with RLCSA [21,22], with sampling parameters 128 and 1024. To illustrate how small can be the *RLCSA*, we also show the space usage of the *RLCSA* without the samples. This version of the RLCSA can only count the number of occurrences but cannot find them.

The results are presented in Tables 1 and 2. First we observe that the construction time of our index dominates every other alternative. This is possible as we use a highly engineered LZ parsing algorithm [14]. As expected, our index is consistently smaller and faster than the original Hybrid Index. Also it is competitive with other alternatives for repetitive collections.

6.1 Larger Collections

The next experiment considered the 90 GB collection $\texttt{CHR}_{21}$. None of the competitors were able to index the collection in the machine we were using. As the input text greatly exceeded the available RAM, we studied different parsing strategies using external memory.

Table 1. Construction time in seconds and size of the resulting index in bytes per character for moderate-sized collections.

	HI		CHICO		LZ77-Index		$RLCSA_{128}$		$RLCSA_{1024}$		$RLCSA_{min}$
	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size	Size
Influenza	204.1	0.0795	30.1	0.0572	40.8	0.0458	64.3	0.0959	63.9	0.0462	0.039
Coreutils	393.1	0.1136	32.9	0.0508	49.9	0.0792	139.2	0.1234	134.1	0.0737	0.066
Einstein	98.9	0.0033	63.5	0.0019	95.3	0.0019	389.0	0.0613	347.5	0.0097	0.002
Para	1065.7	0.0991	33.8	0.0577	157.3	0.0539	232.0	0.1598	217.3	0.1082	0.100
Cere	620.3	0.0767	42.6	0.0517	175.1	0.0376	264.9	0.1366	268.0	0.0850	0.077

Table 2. Time in milliseconds to find all the occurrences of a query pattern of length 50 and 100. Times were computed as average of 1000 query patterns randomly extracted from the collection.

	HI		CHICO		LZ77-Index		$RLCSA_{128}$		$RLCSA_{1024}$	
	$\|P\| = 50$	$\|P\| = 100$	$\|P\| = 50$	$\|P\| = 100$	$\|P\| = 50$	$\|P\| = 100$	$\|P\| = 50$	$\|P\| = 100$	$\|P\| = 50$	$\|P\| = 100$
Influenza	43.51	7.63	42.39	7.20	20.01	48.72	2.54	1.01	57.21	25.16
Coreutils	28.17	2.24	26.92	1.79	10.43	20.07	0.86	0.14	16.08	0.78
Einstein	18.65	11.41	16.43	9.45	23.03	34.90	3.28	2.55	30.94	22.66
Para	2.56	1.55	2.38	1.34	14.37	32.31	0.14	0.15	1.54	1.23
Cere	3.07	1.80	2.88	1.58	13.81	33.31	0.15	0.17	1.95	1.68

Table 3. Different parsing algorithms to index collection CHR$_{21}$, 90 GB. The first row shows the results for EM-LZ, which computes the LZ77 greedy parsing. The next rows show the results for the RLZ parsing using prefixes of size 500 MB and 1 GB. The size of the index is expressed in bytes per char, the building times are expressed in minutes, and the query times are expressed in milliseconds. Query times were computed as average of 1000 query patterns randomly extracted from the collection.

	Size(bpc)	Build time(min)	Query time (ms)	
			$\lvert P\rvert = 50$	$\lvert P\rvert = 70$
EM-LZ	0.00126	4647	18.16	14.90
RLZ$_{0.5\,\text{GB}}$	0.0060	143	55.28	46.67
RLZ$_{1\,\text{GB}}$	0.0047	65	50.30	40.72

The first setting tested was using the EM-LZscan algorithm to compute the greedy parsing. We ran it allowing a maximum usage of 10 GB of RAM. We also tried the RLZ parser using as reference prefixes of sizes 500 MB and 1 GB. For each of the resulting indexes, we measure the query time as the average over 1000 query patterns. The results are presented in Table 3.

Table 3 shows the impact of choosing different parsing algorithms to build the index. We can see different trade-offs between building time and the resulting index: EM-LZ generates the greedy parsing and is indeed the one that generates the smaller index: 0.00126 bytes per character. The building time to achieve that is more than 70 h. On the other hand, the RLZ parser is able to compute the index in about 10 h. As expected, using RLZ instead of the *greedy parsing* results in a larger index. However, the compression ratio is still good enough and the resulting index fits comfortably in RAM.

Table 4. Results using RLZ and PRLZ to parse CHR$_{14}$, a 201 GB collection. Query times were computed as average of 1000 query patterns randomly extracted from the collection.

	Size (bpc)	Build time (min)		Query Time (ms)	
		LZ parsing	Others	P=50	P=70
RLZ$_{1\,\text{GB}}$	0.00656	308	51	225.43	178.47
PRLZ$_{1\,\text{GB}}$	0.00658	22	55	224.04	181.21

Table 5. Results using PRLZ to parse CHR$_{1\ldots5}$, a collection of 2.4 TB of data. Query times were computed as average of 1000 query patterns randomly extracted from the collection.

	Size (bpc)	Build time (min)		Query Time	
		LZ parsing	Others	P=50 (ms)	P=70 (ms)
PRLZ$_{10\,\text{GB}}$	0.000233	600	191	61.20	36.38

The next test was on $\texttt{CHR}_{14}$, a 201 GB collection that we could no longer process in the same commodity computer. We indexed it in a large server, equipped with 48 cores, 12 TB of hard disk, and 1.5 TB of RAM. We compared the RLZ and PRLZ parsers. The results are shown in Table 4.

We can see that the parallelization had almost no impact in the size of the index, but that the indexing time decreased considerably. In the parallel version about 20 min where spent parsing the input, and 55 min building the index. In all the previous settings, the building time was largely dominated by the parsing procedure.

Finally, to demonstrate the scalability of our approach, we indexed $\texttt{CHR}_{1...5}$, a 2.4 TB collection. For this collection we only run the fastest parsing, and the results can be seen in Table 5.

7 Conclusions

We have presented an improved version of the Hybrid Index of Ferrada et al., that achieves up to a 50 % reduction in its space usage, while also improving the query times. By using state of the art Lempel-Ziv parsing algorithms we achieved different trade-offs between building time and space usage: When the collections size is moderate, we could compare to available implementations, and ours achieved the fastest building time. For collections in the tens of gigabytes, our index can still be built in a commodity machine. Finally, we developed a parallel Relative Lempel-Ziv parser to be run in a more powerful machine. In that setting, we indexed a 201 GB collection in about an hour and a 2.4 TB collection in about 12 h.

Some of our parsing schemes worked effectively in the genomic collections, because a prefix of the collection is a natural reference for the RLZ algorithm. For future developments, we will study alternatives such as artificial references [13], so that the index can be equally effective in different contexts.

We also plan to build a version specialized for read alignment. To that end, it is not enough to replace the kernel index by an approximate pattern matching index: Read aligners must consider different factors, such as base quality scores, reverse complements, among other aspects that are relevant to manage genomic data.

Acknowledgments. Many thanks to Travis Gagie, Simon Puglisi, Veli Mäkinen, Dominik Kempa and Juha Kärkkäinen for insightful discussions. The author is funded by Academy of Finland grant 284598 (CoECGR).

References

1. Al-Hafeedh, A., Crochemore, M., Ilie, L., Kopylova, E., Smyth, W.F., Tischler, G., Yusufu, M.: A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. ACM Comput. Surv. (CSUR) **45**(1), 5 (2012)

2. Belazzougui, D., Cunial, F., Gagie, T., Prezza, N., Raffinot, M.: Composite repetition-aware data structures. In: Cicalese, F., Porat, E., Vaccaro, U. (eds.) CPM 2015. LNCS, vol. 9133, pp. 26–39. Springer, Heidelberg (2015)
3. Belazzougui, D., Puglisi, S.J.: Range predecessor and Lempel-Ziv parsing. In: Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms. SIAM (2016) (to appear)
4. Claude, F., Fariña, A., Martínez-Prieto, M., Navarro, G.: Indexes for highly repetitive document collections. In: Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM), pp. 463–468. ACM (2011)
5. Danek, A., Deorowicz, S., Grabowski, S.: Indexing large genome collections on a PC. PLoS ONE **9**(10), e109384 (2014)
6. Do, H.H., Jansson, J., Sadakane, K., Sung, W.K.: Fast relative Lempel-Ziv self-index for similar sequences. Theor. Comput. Sci. **532**, 14–30 (2014)
7. Ferrada, H., Gagie, T., Hirvola, T., Puglisi, S.J.: Hybrid indexes for repetitive datasets. Philos. Trans. R. Soc. A **372**, 20130137 (2014)
8. Ferragina, P., Nitto, I., Venturini, R.: On the bit-complexity of Lempel-Ziv compression. In: Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 768–777. Society for Industrial and Applied Mathematics (2009)
9. Fischer, J.: Optimal succinctness for range minimum queries. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 158–169. Springer, Heidelberg (2010)
10. Fischer, J., Gagie, T., Gawrychowski, P., Kociumaka, T.: Approximating LZ77 via small-space multiple-pattern matching. In: Bansal, N., Finocchi, I. (eds.) Algorithms - ESA 2015. LNCS, vol. 9294, pp. 533–544. Springer, Heidelberg (2015)
11. Gagie, T., Puglisi, S.J.: Searching and indexing genomic databases via kernelization. Front. Bioeng. Biotechnol. **3**(12) (2015)
12. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: plug and play with succinct data structures. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 326–337. Springer, Heidelberg (2014)
13. Hoobin, C., Puglisi, S.J., Zobel, J.: Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. Proc. VLDB Endow. **5**(3), 265–273 (2011)
14. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Lightweight Lempel-Ziv parsing. In: Bonifaci, V., Demetrescu, C., Marchetti-Spaccamela, A. (eds.) SEA 2013. LNCS, vol. 7933, pp. 139–150. Springer, Heidelberg (2013)
15. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Linear time Lempel-Ziv factorization: simple, fast, small. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 189–200. Springer, Heidelberg (2013)
16. Karkkainen, J., Kempa, D., Puglisi, S.J.: Lempel-Ziv parsing in external memory. In: Data Compression Conference (DCC), pp. 153–162. IEEE (2014)
17. Kärkkäinen, J., Ukkonen, E.: Lempel-Ziv parsing and sublinear-size index structures for string matching. In: Proceedings of the 3rd South American Workshop on String Processing (WSP 1996). Citeseer (1996)
18. Kosaraju, S.R., Manzini, G.: Compression of low entropy strings with Lempel-Ziv algorithms. SIAM J. Comput. **29**(3), 893–911 (2000)
19. Kreft, S., Navarro, G.: On compressing and indexing repetitive sequences. Theor. Comput. Sci. **483**, 115–133 (2013)
20. Kuruppu, S., Puglisi, S.J., Zobel, J.: Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 201–206. Springer, Heidelberg (2010)

21. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of individual genomes. In: Batzoglou, S. (ed.) RECOMB 2009. LNCS, vol. 5541, pp. 121–137. Springer, Heidelberg (2009)
22. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. J. Comput. Biol. **17**(3), 281–308 (2010)
23. Na, J.C., Park, H., Crochemore, M., Holub, J., Iliopoulos, C.S., Mouchard, L., Park, K.: Suffix tree of alignment: an efficient index for similar data. In: Lecroq, T., Mouchard, L. (eds.) IWOCA 2013. LNCS, vol. 8288, pp. 337–348. Springer, Heidelberg (2013)
24. Navarro, G.: Indexing highly repetitive collections. In: Arumugam, S., Smyth, W.F. (eds.) IWOCA 2012. LNCS, vol. 7643, pp. 274–279. Springer, Heidelberg (2012)
25. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. **39**(1), article 2 (2007)
26. Navarro, G., Ordóñez, A.: Faster compressed suffix trees for repetitive collections. ACM J. Exp. Alg. **21**(1), article 1.8 (2016)
27. Schneeberger, K., Hagmann, J., Ossowski, S., Warthmann, N., Gesing, S., Kohlbacher, O., Weigel, D.: Simultaneous alignment of short reads against multiple genomes. Genome Biol. **10**, R98 (2009)
28. Sirén, J., Välimäki, N., Mäkinen, V.: Indexing graphs for path queries with applications in genome research. IEEE/ACM Trans. Comput. Biol. Bioinf. **11**(2), 375–388 (2014)
29. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977)
30. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Trans. Inf. Theory **24**(5), 530–536 (1978)