Efficient Construction of a Complete Index for Pan-Genomics Read Alignment

ALAN KUHNLE,[1,2,**] TAHER MUN,[3,**] CHRISTINA BOUCHER,[2,*] TRAVIS GAGIE,[4,5,*] BEN LANGMEAD,[3,*] and GIOVANNI MANZINI[6,*]

ABSTRACT

Short-read aligners predominantly use the FM-index, which is easily able to index one or a few human genomes. However, it does not scale well to indexing collections of thousands of genomes. Driving this issue are the two chief components of the index: (1) a rank data structure over the Burrows–Wheeler Transform (BWT) of the string that will allow us to find the interval in the string's suffix array (SA), and (2) a sample of the SA that—when used with the rank data structure—allows us to access the SA. The rank data structure can be kept small even for large genomic databases, by run-length compressing the BWT, but until recently there was no means known to keep the SA sample small without greatly slowing down access to the SA. Now that (SODA 2018) has defined an SA sample that takes about the same space as the run-length compressed BWT, we have the design for efficient FM-indexes of genomic databases but are faced with the problem of building them. In 2018, we showed how to build the BWT of large genomic databases efficiently (WABI 2018), but the problem of building the sample efficiently was left open. We compare our approach to state-of-the-art methods for constructing the SA sample, and demonstrate that it is the fastest and most space-efficient method on highly repetitive genomic databases. Lastly, we apply our method for indexing partial and whole human genomes and show that it improves over the FM-index-based Bowtie method with respect to both memory and time and over the hybrid index-based CHIC method with respect to query time and memory required for indexing.

Keywords: Burrows–Wheeler Transform, indexing, *r*-index, pan-genomics.

---

[1]Department of Computer Science, Florida State University, Tallahassee, Florida.
[2]Department of Computer and Information Science and Engineering, University of Florida, Gainesville, Florida.
[3]Department of Computer Science, John Hopkins University, Baltimore, Maryland.
[4]Faculty of Computer Science, Dalhousie University, Halifax, Canada.
[5]School of Computer Science and Telecommunications, Universidad Diego Portales and CeBiB, Santiago, Chile.
[6]Department of Science and Technological Innovation, University of Eastern Piedmont, Alessandria, Italy.
*Equal contribution, ordered alphabetically.
**Equal contribution, ordered alphabetically.

1. INTRODUCTION

THE FM-INDEX, which is a compressed subsequence index based on Burrows–Wheeler Transform (BWT), is the primary data structure of the majority of short-read aligners—including Bowtie (Langmead et al., 2008), BWA (Li and Durbin, 2009), and SOAP2 (Li et al., 2009). These aligners build an FM-index-based data structure of sequences from a given genomic database and then use the index to perform queries that find approximate matches of sequences to the database. While these methods can easily index one or a few human genomes, they do not scale well to thousands of genomes. This is problematic in analysis of the data produced by consortium projects, which routinely have several 1000 genomes.

In this article, we address this need by introducing and implementing an algorithm for efficiently constructing the FM-index, which allows for the FM-index construction to scale to larger sets of genomes. To understand the challenge and solution behind our method, consider the two principal components of the FM-index: first, a rank data structure over the BWT of the string that enables us to find the interval in the string's suffix array (SA) containing pointers to starting positions of occurrences of a given pattern (and to compute how many such occurrences there are); second, a sample of the SA that, when used with the rank data structure, allows us to access the SA (so we can list those starting positions). Searching with an FM-index can be summarized as follows: starting with the empty suffix, for each proper suffix of the given pattern, we use rank queries at the end of the BWT interval containing the characters immediately preceding occurrences of that suffix in the string, to compute the interval containing the characters immediately preceding occurrences of the suffix of length 1 greater; when we have the interval containing the characters immediately preceding occurrences of the whole pattern, we use an SA sample to list the contexts of the corresponding interval in the SA, which are the locations of those occurrences.

Although it is possible to use a compressed implementation of the rank data structure that does not become much slower or larger even for thousands of genomes, the same cannot be said for the SA sample. The product of the size and the access time must be at least linear in the length of the string for the standard SA sample. This implies that the FM-index will become much slower and/or much larger as the number of genomes in the databases grows significantly. This bottleneck has forced researchers to consider variations of FM-indexes adapted for massive genomic data sets, such as the Valenzuela et al. (2018) pan-genomic index or the Garrison et al. (2018) variation graph. Some of these proposals use elements of the FM-index, but all deviate in substantial ways from the description above. Not only does this mean they lack the FM-index's long and successful track record, it also means they usually do not give us the BWT intervals for all the suffixes as we search (whose lengths are the suffixes' frequencies, and thus, a tightening sequence of upper bounds on the whole pattern's frequency), nor even the final interval in the SA (which is an important input in other string processing tasks).

Recently, Gagie et al. (2018) proposed a different approach to SA sampling, which takes space proportional to that of the compressed rank data structure while still allowing reasonable access times. While their result yielded a potentially practical FM-index on massive databases, it did not directly lead to a solution since the problem of how to efficiently construct the BWT and SA sample remained open. In a direction toward fully realizing the theoretical result of Gagie et al. (2018), Boucher et al. (2018) showed how to build the BWT of large genomic databases efficiently. We refer to this construction as *prefix-free parsing*. It takes as input string S and in one pass generates a dictionary and a parse of S with the property that the BWT can be constructed from dictionary and parse using workspace proportional to their total size and in $O(|S|)$ time. Yet, the resulting index of Boucher et al. (2018) lacks the SA sample and therefore does not support locating. This makes this index not directly applicable to many bioinformatic applications, such as sequence alignment.

1.1. Our contributions

In this article, we present a solution for building the FM-index* for very large data sets by showing that we can build the BWT and Gagie et al.'s (2018) SA sample together in roughly the same time and memory needed to construct the BWT alone. We note that this algorithm is also based on prefix-free parsing. Thus, we begin by describing how to construct the BWT from the prefix-free parse, and then, we show that it can be modified to build the SA sample in addition to the BWT in roughly the same time and space. We

---

*With the SA sample of Gagie et al. (2018), this index is termed the *r*-index.

implement this approach, and we refer to the resulting implementation as bigbwt. We compare it to state-of-the-art methods for constructing the SA sample and demonstrate that bigbwt is currently the fastest and most space-efficient method for constructing the SA sample on large genomic databases.

Next, we demonstrate the applicability of our method to short-read alignment. In particular, we compare the memory and time needed by our method to build an index for collections of chromosome 19 with those of Bowtie (Langmead et al., 2008) and CHIC (Valenzuela and Mäkinen, 2017). We also compare the sizes of the resulting indexes as well as the amount of time required to perform several locate queries against the indexes. We find that Bowtie is unable to build indexes for our largest collections (500 or more) because it exhausted memory, whereas our method is able to build indexes up to 2000 chromosome 19s (and likely beyond). At 250 chromosome 19 sequences, our method requires only about 2% of the time and 6% the peak memory of Bowtie's. While CHIC can produce the smallest indexes for smaller sequence collections, this comes at the cost of higher indexing memory footprint and dramatically higher query time. Lastly, we demonstrate that it is possible to index collections of whole human genome assemblies with sublinear scaling as the size of the collection grows.

1.2. Related work

The development of methods for building the FM-index on large data sets is closely related to the development of short-read aligners for pan-genomics—an area where there is growing interest (Schneeberger et al., 2009; Danek et al., 2014; Gagie and Puglisi, 2015). Here, we briefly describe some previous approaches to this problem and detail their connection to the work in this article. We note that the majority of pan-genomic aligners require building the FM-index for a population of genomes and thus could increase proficiency using the methods described in this article.

GenomeMapper (Schneeberger et al., 2009), the method of Danek et al. (2014), and Generalized Compressed Suffix Array (GCSA) (Sirén et al., 2014) represent the genomes in a population as a graph and then reduce the alignment problem to finding a path within the graph. Hence, these methods require all possible paths to be identified, which is exponential in the worst case. Some of these methods—such as GCSA—use the FM-index to store and query the graph and could capitalize on our approach by building the index in the manner described here. Another set of approaches (Mäkinen et al., 2010; Ferrada et al., 2014; Gagie and Puglisi, 2015; Valenzuela and Mäkinen, 2017) considers the reference pan-genome as the concatenation of individual genomes and exploits redundancy using a compressed index.

The hybrid index (Ferrada et al., 2014) operates on a Lempel-Ziv compression of the reference pan-genome. An input parameter M sets the maximum length of reads that can be aligned. This has a major impact on the final size of the index. For this reason, the hybrid index is suitable mainly for short-read alignment, although there have been recent heuristic modifications to allow for longer alignments (Ferrada et al., 2018). In contrast, the r-index, of which we provide an implementation in this work, has no such length limitation. The most recent implementation of the hybrid index is CHIC (Valenzuela et al., 2018; based on CHICO; Valenzuela, 2016). Although CHIC has support for counting multiple occurrences of a pattern within a genomic database, it is an expensive operation, namely $O(\ell \log \log n)$, where ℓ is the number of occurrences in the databases and n is the length of the database. However, the r-index is capable of counting all occurrences of a pattern of length m in $O(m)$ time up to polylog factors. There are a number of other approaches building off the hybrid index or similar ideas (Wandelt et al., 2013; Danek et al., 2014); for an extended discussion, we refer the reader to the survey of Gagie and Puglisi (2015).

Finally, a third set of approaches (Huang et al., 2013; Maciuca et al., 2016) attempts to encode variants within a single reference genome. BWBBLE by Huang et al. (2013) follows this by supplementing the alphabet to indicate if multiple variants occur at a single location. This approach does not support counting of the number of variants matching a specific alignment; also, it suffers from memory blow-up when larger structural variations occur.

2. BACKGROUND

2.1. BWT and FM indexes

Consider a string S of length n from a totally ordered alphabet Σ, such that the last character of S is lexicographically less than any other character in S. Let F be the list of S's characters sorted lexicographically by the suffixes starting at those characters, and let L be the list of S's characters sorted

lexicographically by the suffixes starting immediately after those characters. The list L is termed the BWT (Burrows and Wheeler, 1994) of S and denoted BWT. If $S[i]$ is in position p in F, then $S[i-1]$ is in position p in L. Moreover, if $S[i]=S[j]$, then $S[i]$ and $S[j]$ have the same relative order in both lists; otherwise, their relative order in F is the same as their lexicographic order. This means that if $S[i]$ is in position p in L then, assuming arrays are indexed from 0 and $\prec$ denotes lexicographic precedence, in F it is in position $j_i = |\{h: S[h] \prec S[i]\}| + |\{h: L[h]=S[i], h \le p\}| - 1$. The mapping $i \mapsto j_i$ is termed the LF mapping. Finally, notice that the last character in S always appears first in L. By repeated application of the LF mapping, we can invert the BWT, that is, recover S from L. Formally, the SA of the string S is an array such that entry i is the starting position in S of the ith largest suffix in lexicographical order. The above definition of the BWT is equivalent to the following:

$$\text{BWT}[i] = S[(\text{SA}[i]-1) \mod n]. \tag{1}$$

The BWT was introduced as an aid to data compression: it moves characters followed by similar contexts together and thus makes many strings encountered in practice locally homogeneous and easily compressible. Ferragina and Manzini (2000) showed how the BWT may be used for *indexing* a string S: given a pattern P of length $m < n$, find the number and location of all occurrences of P within S. If we know the range $\text{BWT}(S)[i..j]$ occupied by characters immediately preceding occurrences of a pattern Q in S, then we can compute the range $\text{BWT}(S)[i'..j']$ occupied by characters immediately preceding occurrences of cQ in S, for any character $c \in \Sigma$, since

$$i' = |\{h : S[h] \prec c\}| + |\{h : S[h]=c, h < i\}|$$
$$j' = |\{h : S[h] \prec c\}| + |\{h : S[h]=c, h \le j\}| - 1.$$

Notice $j' - i' + 1$ is the number of occurrences of cQ in S. The essential components of an FM-index for S are, first, an array storing $|\{h : S[h] \prec c\}|$ for each character c and, second, a rank data structure for BWT that quickly tells us how often any given character occurs up to any given position.[†] To be able to locate the occurrences of patterns in S (in addition to just counting them), the FM-index uses a sampled[‡] SA of S and a bit vector indicating the positions in the BWT of the characters preceding the sampled suffixes.

2.2. Prefix-free parsing

Next, we give an overview of prefix-free parsing, which produces a dictionary $\mathcal{D}$ and a parse $\mathcal{P}$ by sliding a window of fixed width through the input string S and dividing it into variable-length overlapping substrings with delimiting prefixes and suffixes. We refer the reader to Boucher et al. (2018) for the formal proofs and Section 3.1 for the algorithmic details. A rolling hash function identifies when substrings are parsed into elements of a dictionary, which is a set of substrings of S. Intuitively, for a repetitive string, the same dictionary phrases will be encountered frequently.

We now formally define the dictionary $\mathcal{D}$ and parse $\mathcal{P}$. Given a string[§] S of length n, window size $w \in \mathbb{N}$, and modulus $p \in \mathbb{N}$, we construct the dictionary $\mathcal{D}$ of substrings of S and the parse $\mathcal{P}$ as follows; we let f be a hash function on strings of length w, and let $\mathcal{T}$ be the sequence of substrings $W=S[s, s+w-1]$ such that $f(W) \equiv 0 \pmod{p}$ or $W=S[0, w-1]$ or $W=S[n-w+1, n-1]$, ordered by initial position in S; let $\mathcal{T} = (W_1 = S[s_1, s_1+w-1], \ldots, W_k = [s_k, s_k+w-1])$. By construction, the strings

$$S[s_1, s_2+w-1], S[s_2, s_3+w-1], \ldots, S[s_{k-1}, s_k+w-1]$$

form a parsing of S, in which each pair of consecutive strings $S[s_i, s_{i+1}+w-1]$ and $S[s_{i+1}, s_{i+2}+w-1]$ overlaps by exactly w characters. We define $\mathcal{D} = \{S[s_i, s_{i+1}+w-1] : 1 \le i < k\}$; that is, $\mathcal{D}$ consists of the set of the unique substrings s of S such that $|s| > w$ and the first and last w characters of s form consecutive elements in $\mathcal{T}$. If S has many repetitions, we expect that $|\mathcal{D}| \ll k$. With a little abuse of notation, we define the parsing $\mathcal{P}$ as the sequence of lexicographic ranks of substrings in $\mathcal{D}$: $\mathcal{P} = (\text{rank}_{\mathcal{D}}(S[s_i, s_{i+1}+w-1]))_{i=1}^{k-1}$. The parse $\mathcal{P}$ indicates how S may be reconstructed using elements of $\mathcal{D}$. The dictionary $\mathcal{D}$ and parse $\mathcal{P}$ may

---

[†]Given a sequence (string) $S[1, n]$ over an alphabet $\Sigma = \{1, \ldots, \sigma\}$, a character $c \in \Sigma$, and an integer i, $\text{rank}_c(S, i)$ is the number of times that c appears in $S[1, i]$.

[‡]*Sampled* means that only some fractions of entries of the suffix array are stored.

[§]For technical reasons, the string S must terminate with w copies of lexicographically least $ symbol.

be constructed in one pass over S in $O(n + |\mathcal{D}| \log|\mathcal{D}|)$ time if the hash function f can be computed in constant time.

2.3. r-*index locating*

Policriti and Prezza (2018) showed that if we have stored SA[k] for each value k such that BWT[k] is the beginning or end of a run (i.e., a maximal nonempty unary substring) in BWT, and we know both the range BWT[$i..j$] occupied by characters immediately preceding occurrences of a pattern Q in S and the starting position of one of those occurrences of Q, then when we compute the range BWT[$i'..j'$] occupied by characters immediately preceding occurrences of cQ in S, we can also compute the starting position of one of those occurrences of cQ. Bannai et al. (2018) then showed that even if we have stored only SA[k] for each value k such that BWT[k] is the beginning of a run, then as long as we know $SA[i]$, we can compute $SA[i']$.

Gagie et al. (2018) showed that if we have stored in a predecessor data structure SA[k] for each value k such that BWT[k] is the beginning of a run in BWT, with $\phi^{-1}(SA[k]) = SA[k+1]$ stored as satellite data, then given SA[h] we can compute SA[$h+1$] in $O(\log \log n)$ time as $SA[h+1] = \phi^{-1}(pred(SA[h])) + SA[h] - pred(SA[h])$, where $pred(\cdot)$ is a query to the predecessor data structure. Combined with Bannai et al.'s (2018) result, this means that while finding the range $BWT[i..j]$ occupied by characters immediately preceding occurrences of a pattern Q, we can also find $SA[i]$ and then report $SA[i+1..j]$ in $O((j-i)\log \log n)$-time; that is, $O(\log \log n)$-time per occurrence.

Gagie et al. (2018) gave the name r-index to the index resulting from combining a rank data structure over the run-length-compressed BWT with their SA sample, and Bannai et al. (2018) used the same name for their index. Since our index is an implementation of theirs, we keep this name; on the contrary, we do not apply it to indexes based on run-length-compressed BWTs that have standard SA samples or no SA samples at all.

3. METHODS

Here we describe our algorithm for building the SA or the sampled SA from the prefix-free parse of an input string S, which is used to build the r-index. We first review the algorithm from Boucher et al. (2018) for building the BWT of S from the prefix-free parse. Next, we show how to modify this construction to compute the SA or the sampled SA along with the BWT.

3.1. *Construction of BWT from prefix-free parse*

We assume we are given a prefix-free parse of $S[1..n]$ with window size w consisting of a dictionary $\mathcal{D}$ and a parse $\mathcal{P}$. We represent the dictionary as a string $\mathcal{D}[1..\ell] = t_1 \# t_2 \# \cdots t_{d-1} \# t_d \#$ where t_i's are the dictionary phrases in lexicographic order and $\#$ is a unique separator. We assume we have computed the SA of $\mathcal{D}$, denoted by $SA_{\mathcal{D}}[1..\ell]$ in the following, and the BWT of $\mathcal{P}$, denoted $BWT_{\mathcal{P}}$, and the array Occ[$1, d$] such that Occ[i] stores the number of occurrences of the dictionary phrase t_i in the parse. These preliminary computations take $O(|\mathcal{D}| + |\mathcal{P}|)$ time.

By the properties of the prefix-free parsing, each suffix of S is prefixed by *exactly one* suffix α of a dictionary phrase t_j with $|\alpha| > w$. We call α_i the *representative prefix* of the suffix $S[i..n]$. From the uniqueness of the representative prefix, we can partition S's suffix array $SA[1..n]$ into k ranges

$$[b_1, e_1], \quad [b_2, e_2], \quad [b_3, e_3], \quad \ldots, \quad [b_k, e_k]$$

with $b_1 = 1$, $b_i = e_{i-1} + 1$ for $i = 2, \ldots, k$, and $e_k = n$, such that for $i = 1, \ldots, k$ all suffixes

$$S[SA[b_i]..n], \quad S[SA[b_i+1]..n], \quad \ldots, \quad S[SA[e_i]..n]$$

have the same representative prefix α_i. By construction $\alpha_1 \prec \alpha_2 \prec \cdots \prec \alpha_k$.

By construction, any suffix $\mathcal{D}[i..\ell]$ of the dictionary $\mathcal{D}$ is also prefixed by the suffix of a dictionary phrase. For $j = 1, \ldots, \ell$, let β_j denote the longest prefix of $\mathcal{D}[SA_{\mathcal{D}}[j]..\ell]$, which is the suffix of a phrase (i.e., $\mathcal{D}[SA_{\mathcal{D}}[j] + |\beta_j|] = \#$). By construction, the strings β_j's are lexicographically sorted $\beta_1 \prec \beta_2 \prec \cdots \prec \beta_\ell$. Clearly, if we compute $\beta_1, \ldots, \beta_\ell$ and discard those such that $|\beta_j| \leq w$, the remaining β_j's will coincide with

the representative prefixes α_i·s. Since both β_j·s and α_i·s are lexicographically sorted, this procedure will generate the representative prefixes in the order $\alpha_1, \alpha_2, \ldots, \alpha_k$. We note that more than one β_j can be equal to some α_i since different dictionary phrases can have the same suffix.

We scan $SA_{\mathcal{D}}[1..\ell]$, compute $\beta_1, \ldots \beta_\ell$, and use these strings to find the representative prefixes. As soon as we generate an α_i, we compute and output the portion $BWT[b_i, e_i]$ corresponding to the range $[b_i, e_i]$ associated with α_i. To implement the above strategy, assume there are exactly k entries in $SA_{\mathcal{D}}[1..\ell]$ prefixed by α_i. This means that there are k distinct dictionary phrases $t_{i1}, t_{i2}, \ldots, t_{ik}$ that end with α_i. Hence, the range $[b_i, e_i]$ contains $z_i = e_i - b_i + 1 = \sum_{h=1}^{k} \text{Occ}[i_h]$ elements. To compute $BWT[b_i, e_i]$ we need to (1) find the symbol immediately preceding each occurrence of α_i in S, and (2) find the lexicographic ordering of S's suffixes prefixed by α_i. We consider the latter problem first.

3.1.1. Computing the lexicographic ordering of suffixes.

For $j = 1, \ldots, z_i$, consider the jth occurrence of α_i in S and let i_j denote the position in the parsing of S of the phrase ending with the jth occurrence of α_i. In other words, $\mathcal{P}[i_j]$ is a dictionary phrase ending with α_i and $i_1 < i_2 < \cdots < i_{zi}$. By the properties of $BWT_{\mathcal{P}}$, the lexicographic ordering of S's suffixes prefixed by α_i coincides with the ordering of the symbols $\mathcal{P}[i_j]$ in $BWT_{\mathcal{P}}$. In other words, $\mathcal{P}[i_j]$ precedes $\mathcal{P}[i_h]$ in $BWT_{\mathcal{P}}$ if and only if S's suffix prefixed by the jth occurrence of α_i is lexicographically smaller than S's suffix prefixed by the hth occurrence of α_i.

We could determine the desired lexicographic ordering by scanning $BWT_{\mathcal{P}}$ and noticing which entries coincide with one of the dictionary phrases $t_{i1}, \ldots, t_{ik}$ that end with α_i, but this would clearly be inefficient. Instead, for each dictionary phrase t_i, we maintain an array IL_i of length $\text{Occ}[i]$ containing the indexes j such that $BWT_{\mathcal{P}}[j] = i$. These sorts of ''inverted lists'' are computed at the beginning of the algorithm and replace the $BWT_{\mathcal{P}}$, which can be discarded.

3.1.2. Finding the symbol preceding α_i.

Given a representative prefix α_i from $SA_{\mathcal{D}}$, we retrieve the indexes $i_1, \ldots, i_k$ of the dictionary phrases $t_{i1}, \ldots, t_{ik}$ that end with α_i. Then, we retrieve the inverted lists $IL_{i_1}, \ldots IL_{i_k}$ and we merge them, obtaining the list of the z_i positions $y_1 < y_2 < \cdots < y_{zi}$ such that $BWT_P[y_j]$ is a dictionary phrase ending with α_i. Such a list implicitly provides the lexicographic order of S's suffixes starting with α_i.

To compute the BWT, we need to retrieve the symbols preceding such occurrences of α_i. If α_i *is not* a dictionary phrase, then α_i is a proper suffix of the phrases $t_{i1}, \ldots, t_{ik}$ and the symbols preceding α_i in S are those preceding α_i in $t_{i1}, \ldots, t_{ik}$ that we can retrieve from $\mathcal{D}[1..\ell]$ and $SA_{\mathcal{D}}[1..\ell]$. If α_i *coincides* with a dictionary phrase t_j, then it cannot be a suffix of another phrase. Hence, the symbols preceding α_i in S are those preceding t_j in S that we store at the beginning of the algorithm in an auxiliary array PR_j along with the inverted list IL_j.

3.2. Construction of SA and SA sample along with the BWT

We now show how to modify the above algorithm so that, along with BWT, it computes the full SA of S or the sampled SA consisting of the values $SA[s_1], \ldots, SA[s_r]$ and $SA[e_1], \ldots, SA[e_r]$, where r is the number of maximal nonempty runs in BWT and s_i and e_i are the starting and ending positions in BWT of the ith such run, respectively. Note that if we compute the sampled SA, the actual output will consist of r start-run pairs $\langle s_i, SA[s_i] \rangle$ and r end-run pairs $\langle e_i, SA[e_i] \rangle$ since the SA values alone are not enough for the construction of the r-index.

We solve both problems using the following strategy. Simultaneously to each entry $BWT[j]$, we compute the corresponding entry $SA[j]$. Then, if we need the sampled SA, we compare $BWT[j-1]$ and $BWT[j]$ and if they differ, we output the pair $\langle j-1, SA[j-1] \rangle$ among the end-runs and the pair $\langle j, SA[j] \rangle$ among the start-runs. To compute the SA entries, we only need d additional arrays $EP_1, \ldots EP_d$ (one for each dictionary phrase), where $|EP_i| = |IL_i| = \text{Occ}[i]$, and $EP_i[j]$ contains the ending position in S of the dictionary phrase, which is in position $IL_i[j]$ of $BWT_{\mathcal{P}}$.

Recall that in the above algorithm for each occurrence of a representative prefix α_i, we compute the indexes $i_1, \ldots, i_k$ of the dictionary phrases $t_{i1}, \ldots, t_{ik}$ that end with α_i. Then, we use the lists $IL_{i_1}, \ldots, IL_{i_k}$ to retrieve the positions of all the occurrences of $t_{i1}, \ldots, t_{ik}$ in $BWT_{\mathcal{P}}$, thus establishing the relative lexicographic order of the occurrences of the dictionary phrases ending with α_i. To compute the corresponding SA entries, we need the starting position in S of each occurrence of α_i. Since the ending position in S of the phrase with relative lexicographic rank $IL_{i_h}[j]$ is $EP_{i_h}[j]$, the corresponding SA entry is $EP_{i_h}[j] - |\alpha_i| + 1$.

Hence, along with each BWT entry, we obtain the corresponding SA entry, which is saved to the output file if the full SA is needed, or further processed as described above if we need the sampled SA.

4. TIME AND MEMORY USAGE FOR SA AND SA SAMPLE CONSTRUCTION

We compare the running time and memory usage of bigbwt with the following methods, which represent the current state-of-the-art.

4.1. bwt2sa

Once the BWT has been computed, the SA or SA sample may be computed by applying the *LF* mapping to invert the BWT and the application of Equation (1). Therefore, as a baseline, we use bigbwt to construct the BWT only, as in Boucher et al. (2018); we use bigbwt since it seems best suited to the inputs we consider. Next, we load the BWT as a Huffman-compressed string with access, rank, and select support to compute the *LF* mapping. We step backward through the BWT and compute the entries of the SA in nonconsecutive order. Finally, these entries are sorted in external memory to produce the SA or SA sample. This method may be parallelized when the input consists of multiple strings by stepping backward from the end of each string in parallel.

4.2. pSAscan

A second baseline is to compute the SA directly from the input; for this computation, we use the external-memory algorithm pSAscan (Kärkkäinen et al., 2015), with available memory set to the memory required by bigbwt on the specific input; with the ratio of memory to input size obtained from bigbwt, pSAscan is the current state-of-the-art method to compute the SA. Once pSAscan has computed the full SA, the SA sample may be constructed by loading the input text T into memory, streaming the SA from the disk, and the application of Equation (1) to detect run boundaries. We denote this method of computing the SA sample by pSAscan+.

We compared the performance of all the methods on two data sets: (1) *Salmonella* genomes obtained from GenomeTrakr (Stevens et al., 2017); and (2) chromosome 19 haplotypes derived from the 1000 Genomes (1KG) Project phase 3 data (Auton et al., 2015). The *Salmonella* strains were downloaded from NCBI (NCBI BioProject PRJNA183844) and preprocessed by assembling each individual sample with IDBA-UD (Peng et al., 2012) and counting k-mers ($k = 32$) using KMC (Deorowicz et al., 2015). We modified IDBA by setting kMaxShortSequence to 1024 per public advice from the author to accommodate the longer paired end reads that modern sequencers produce. We sorted the full set of samples by the size of their k-mer counts and selected 1000 samples about the median. This avoids exceptionally short assemblies, which may be due to low read coverage, and exceptionally long assemblies, which may be due to contamination.

Next, we downloaded and preprocessed a collection of chromosome 19 haplotypes from 1KG Project. Chromosome 19 is 58 million base pairs in length and makes up around 1.9% of the total human genome sequence. Each sequence was derived using the bcftools consensus tool to combine the haplotype-specific (maternal or paternal) variant calls for an individual in the 1KG project with the chr19 sequence in the GRCH37 human reference, producing an FASTA record per sequence. All DNA characters besides A, C, G, Tm and N were removed from the sequences before construction.

We performed all experiments in this section on a machine with Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80 GHz and 324 GB RAM. We measured running time and peak memory footprint using/usr/bin/time −v, with peak memory footprint captured by the Maximum resident set size (kbytes) field and running time by the User Time and System Time field.

We witnessed that the running time of each method to construct the full SA is shown in Figure 1a–c. On both the *Salmonella* and chr19 data sets, bigbwt ran the fastest, often by more than an order of magnitude. In Figure 1d, we show the peak memory usage of bigbwt as a function of input size. Empirically, the peak memory usage was sublinear in input size, especially on the chr19 data, which exhibited a high degree of repetition. Despite the higher diversity of the *Salmonella* genomes, bigbwt remained space-efficient and the fastest method for construction of the full SA. Furthermore, we found qualitatively similar results for construction of the SA sample, shown in Figure 2. Similar to the results on full SA construction, bigbwt outperformed both baseline methods and exhibited sublinear memory scaling on both types of databases.
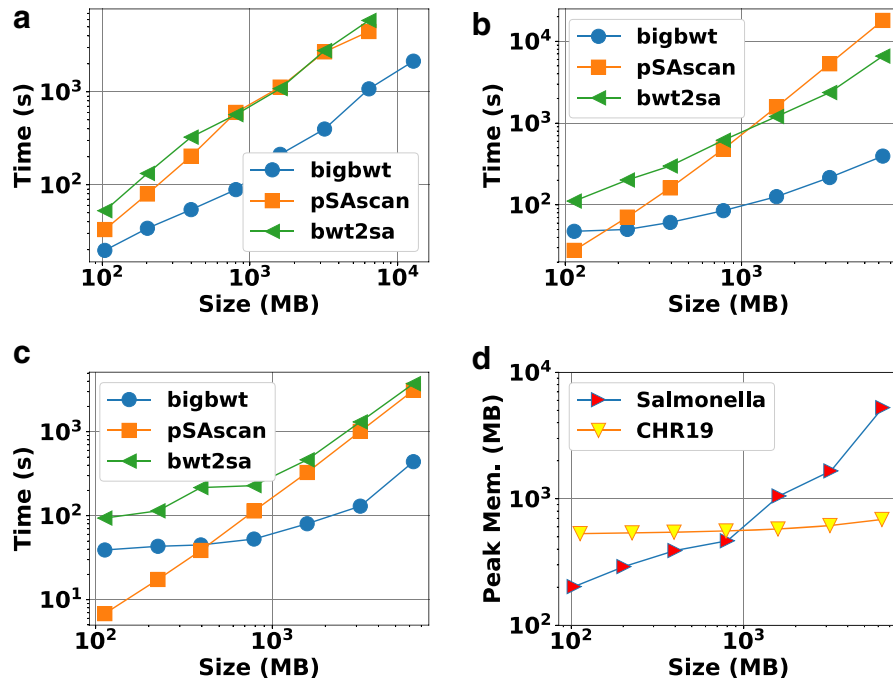
FIG. 1. Runtime and peak memory usage for construction of full SA. **(a)** *Salmonella*, 1 thread. **(b)** chr19, 1 thread. **(c)** chr19, 16 threads. **(d)** Peak memory, bigbwt. SA, suffix array.

5. COMPARISON WITH BOWTIE AND CHIC

We studied how *r*-index scales to repetitive texts consisting of many similar genomic sequences, comparing it with Bowtie (version 1.2.2) (Langmead et al., 2008), a traditional FM-index-based aligner, and CHIC (Valenzuela and Mäkinen, 2017), a Hybrid Index that uses LZ compression to scale to repetitive texts. We measured indexing memory footprint, indexing time, index size, and locate query time.
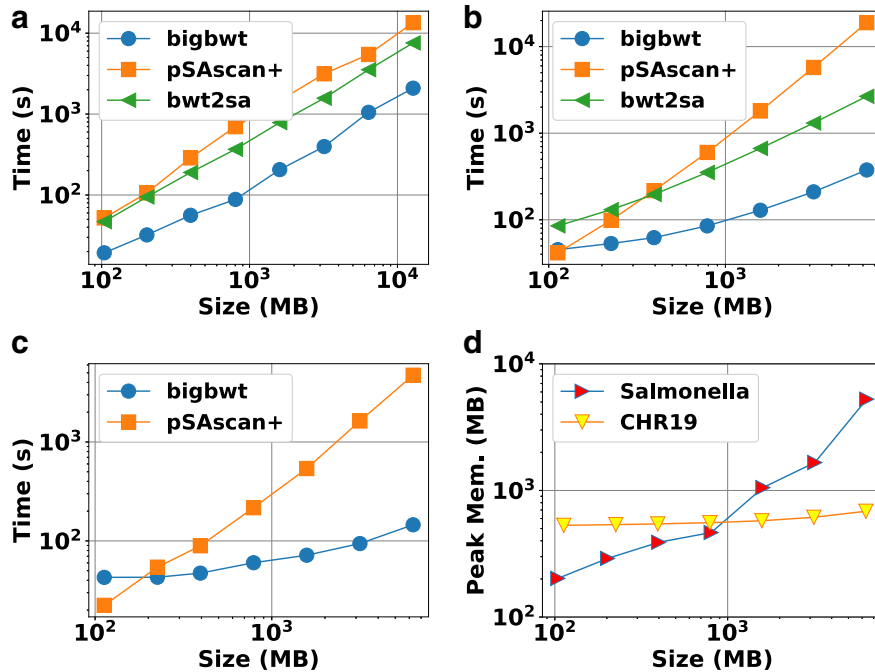
FIG. 2. Runtime and peak memory usage for construction of SA sample. **(a)** *Salmonella*, 1 thread. **(b)** chr19, 1 thread. **(c)** chr19, 16 threads. **(d)** Peak memory, bigbwt.

We ran Bowtie with the −v 0 and –norc options; −v 0 disables approximate matching, while –norc causes Bowtie (like *r*-index) to perform the locate query with respect to the query sequence only and not its reverse complement.

CHIC parses the text with an LZ-like compression algorithm, storing the resulting phrases in a kernel string that can be indexed and aligned to with a standard aligner. Kernel-string alignments are transformed back to the original text coordinates using range-finding data structures. CHIC parameters include the following: which LZ parsing algorithm to use, the text prefix length from which phrases in the parse can be sourced (if a relative LZ algorithm is specified), the kernel-string indexing method, and the maximum length of the query patterns. We used the RLZ parsing method and the FM-index method for indexing the kernel string in all our experiments. For the prefix length, we tried both 10% and 30% of the text length. For the maximum query length, we tried both 100 and 250 bp, these being realistic second-generation sequencing read lengths. We refer to each parameter combination as ''CHIC_Xp_Yb,'' where X is the prefix-length percentage and Y is the maximum query length.

5.1. Indexing chromosome 19s

We performed our experiments on collections of one or more haplotypes of chromosome 19. These haplotypes were obtained from the 1KG Project in the manner described in the previous section. We used 10 collections of chromosome 19 haplotypes, containing 1, 2, 10, 30, 50, 100, 250, 500, and 1000, 1250, 1500, and 2000 sequences, respectively. Each collection is a superset of the previous. Again, all DNA characters besides A, C, G, T, and N were removed from the sequences before construction. All experiments in this section were run on an Intel Xeon system with an E5-2680 v3 CPU clocked at 2.50 GHz and 512 GB memory. We measured running time and peak memory footprint as described in the previous section.

First, we constructed *r*-index, Bowtie, and CHIC indexes for successively larger chromosome 19 collections (Fig. 3a, b). *r*-Index uses the least indexing memory for collections of 10 chromosomes and larger. At 250 chr19s, the *r*-index procedure takes about 2% of the time and 6% of the peak memory of Bowtie's procedure.

While CHIC's peak memory is also much higher than *r*-index's at 10 sequences and above, CHIC tends to construct indexes faster, especially when using a prefix length of 10% of the text. At 2000 sequences, CHIC_10p_100b takes about 64% of the time, but 920% of the memory of *r*-index. Bowtie is drastically slower to index than either CHIC or *r*-index, especially for larger collections. Due to memory exhaustion, Bowtie fails to index collections of more than 250 sequences and 2 of the CHIC modes (those using a 30% prefix) fail for collections of more than 1500 sequences.

Next, we compared the disk footprint of the index files produced by all three methods (Fig. 3c). *r*-Index currently stores only the forward strand of the sequence. Bowtie, on the contrary, stores both the forward sequence and its reverse as needed by its double-indexing heuristic (Langmead et al., 2008). Since the heuristic is relevant only for approximate matching, we omit the reverse sequence in these size comparisons. We also omit the 2-bit encoding of the original text (in the *.3.ebwt and *.4.ebwt files) as these too are used only for approximate matching. Specifically, the Bowtie index size was calculated by adding the sizes of the forward *.1.ebwt and *.2.ebwt files, which contain the BWT, SA sample, and auxiliary data structures for the forward sequence. CHIC stores the forward strand of the kernel string, along with range-finding data structures. These consist of the files ending with *.P512_GC4_kernel_text.MAN.kernel_index (the kernel index), and *.book_keeping, *.is_literal, *.limits, *.limits_kernel, *.ptr, *.rmq, *.sparse_sample_limits_kernel, *.sparseX, *.variables and *.x (the range-finding data structures).

An *r*-index is considerably larger than a Bowtie or CHIC index for smaller collections. However, it grows at a slower rate than any of the other indexes, becoming smaller than Bowtie at 30 sequences and smaller than CHIC_10p_250b at 1500 sequences. The *r*-index incurs more overhead for smaller collections because SA sample density depends on the ratio n/r. When the collection is small, n/r is small leading to a denser SA sample than the 1-in-32 rate used by Bowtie. The CHIC index stays small by indexing only the kernel string, which is smaller than the text. Like Bowtie, the FM-index kernel samples a constant fraction of SA elements. Finally, CHIC's range-query data structures are typically smaller than the kernel index. At 250 sequences, the *r*-index takes 6% the space of the Bowtie index and 509% the space of the CHIC_30p_100b index (the smallest CHIC index at this point). At 1500 sequences, the CHIC_30p_100b index takes 45% the space of *r*-index.
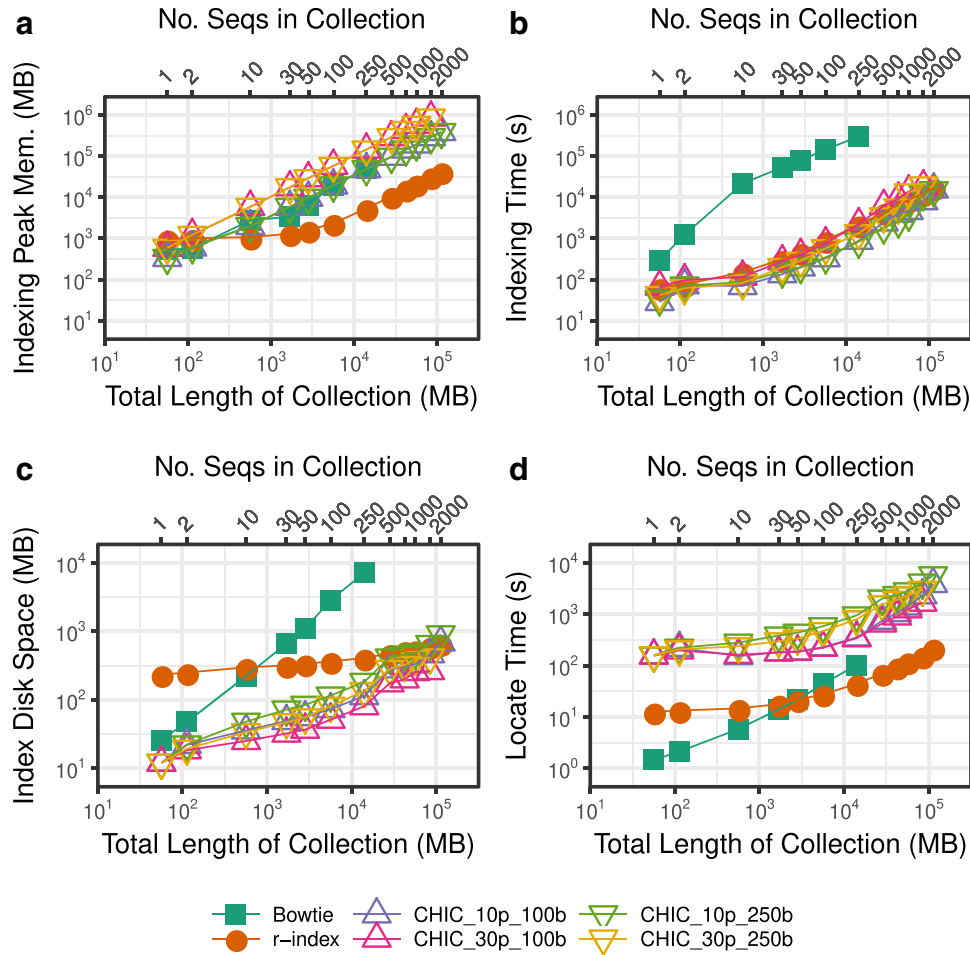
FIG. 3. Scalability of r-index, Bowtie, and CHIC (RLZ compressed, FM-index kernel) against chr19 haplotype collection size and total sequence length (megabases) with respect to index construction time (seconds) **(a)**, index construction peak memory (megabytes) **(b)**, index disk space (megabytes) **(c)**, and locate time (seconds) of 100,000 one hundred base pair queries **(d)**. Four different CHIC indexes were used, using different combinations of prefix size and maximum query length, each labeled as *CHIC_*(prefix size)*p_*(max query length).

We then compared the speed of the locate query for the *r*-index, Bowtie, and Compressed Hybrid Index of (Repetitive) Collections (Aligner) [CHIC]. We extracted 100,000 one hundred-character substrings from the chr19 collection of size 1, which is also contained in all of the larger collections. We queried these against each of the indexes constructed. We aimed to measure the speed of locating *all* occurrences of each pattern, because in repetitive indexes, the number of occurrences for one pattern is on average the number of sequences in the collection, but it could also exceed that number due to multimapping within a sequence. Since the source of the substrings is present in all the collections, every query will match at least once. As seen in Figure 3d, the *r*-index locate time is faster than that of Bowtie after 50 sequences, and it is consistently at least 10×faster than any of the CHIC modes.

6. INDEXING WHOLE HUMAN GENOMES

Lastly, we used *r*-index to index many human genomes at once. We repeated our measurements for successively larger collections of (concatenated) genomes. Thus, we first evaluated a series of haplotypes extracted from the 1KG Project (Auton et al., 2015) phase 3 callset (1KG). These collections ranged from 1 up to 10 genomes. As the first genome, we selected the GRCh37 reference itself. For the remaining nine, we used bcftools consensus to insert single nucleotide variations (SNVs) and other variants called by the 1KG Project for a single haplotype into the GRCh37 reference.

Second, we evaluated a series of whole human genome assemblies from six different long-read assembly projects (''LRA''). We selected GRCh37 reference as the first genome, so that the first data point would coincide with that of the previous series. We then added long-read assemblies from a Chinese genome assembly project (Shi et al., 2016), a Korean genome assembly project (Seo et al., 2016), a project to assemble the well-studied NA12878 individual (Jain et al., 2018), a hydatidiform mole (known as CHM1) assembly project (Steinberg et al., 2014), and the Celera human genome project (Levy et al., 2007). Compared with the series with only 1KG Project individuals, this series allowed us to measure scaling while capturing a wider range of genetic variations between humans. This is important since *de novo* human assembly projects regularly produce assemblies that differ from the human genome reference by megabases of sequence (12 megabases in the case of the Chinese assembly; Shi et al., 2016), likely due to prevalent but hard-to-profile large-scale structural variation. Such variation was not comprehensively profiled in the 1KG Project, which relied on short reads.

The 1KG and LRA series were evaluated twice, once on the forward genome sequences and once on both the forward- and reverse-complement sequences. This accounts for the fact that different *de novo* assemblies make different decisions about how to orient contigs. The *r*-index method achieves compression only with respect to the forward-oriented haplotypes of the sequences indexed. That is, if two contigs are reverse complements of each other but otherwise identical, the *r*-index achieves less compression than if their orientations matched. A more practical approach would be to index both forward- and reverse-complement sequences, as Bowtie 2 (Langmead and Salzberg, 2012) and BWA (Li, 2013) do.

We measured the peak memory footprint when indexing these collections (Fig. 4). We ran these experiments on an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20 GHz system with 256 GB memory. Memory footprints for LRA grew more quickly than those for 1KG. This was expected due to the greater genetic diversity captured in the assemblies. This may also be due, in part, to the presence of sequencing errors in the long-read assembles; long-read technologies are more prone to indel errors than short-read technologies, for example, and some may survive in the assemblies. Also as expected, memory footprints for the LRA series that included both forward- and reverse-complement sequences grew more slowly than when just the forward sequence was included. This is due to sequences that differ only (or primarily) in their orientation between assemblies. All series exhibit sublinear trends, highlighting the efficacy of *r*-index compression even when indexing genetically diverse whole-genome assemblies. Indexing the forward- and reverse-complement strands of 10 1KG individuals took about 6 hours and 20 minutes and the final index size was 36 GB.

We also measured lengths and *n/r* ratios for each collection of whole genomes (Table 1). Consistent with the memory-scaling results, we see that the *n/r* ratios are somewhat lower for the LRA series than for the 1KG series, likely due to greater genetic diversity in the assemblies.
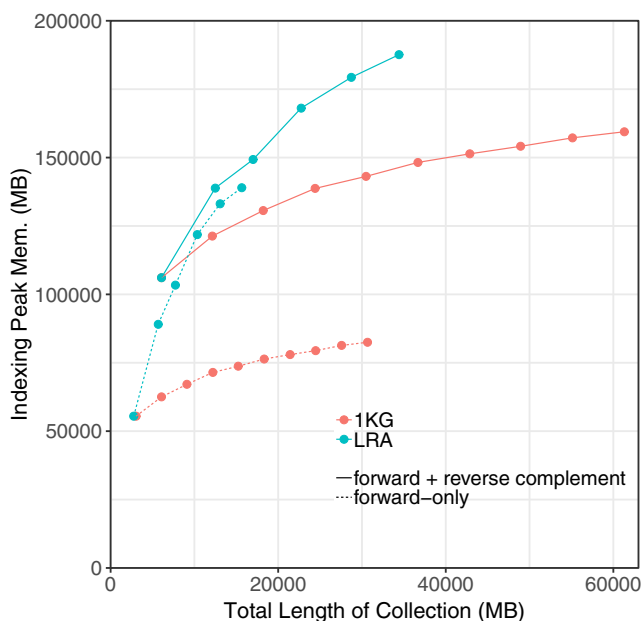
FIG. 4. Peak index-building memory for *r*-index when indexing successively larger collections of 1KG individuals and whole-genome long-read assemblies (LRA). 1KG, 1000 genomes.

Table 1. Sequence Length and *n/r* Statistic with Respect to Number of Whole Genomes for the First 6 Collections in the 1000 Genomes and Long-Read Assembly Series

| No. of genomes | Sequence | | | |
| | Length (MB) | | n/r | |
	1KG	LRA	1KG	LRA
1	6072	6072	1.86	1.86
2	12,144	12,484	3.70	3.58
3	18,217	17,006	5.38	4.83
4	24,408	22,739	7.13	6.25
5	30,480	28,732	8.87	7.80
6	36,671	34,420	10.63	9.28

1KG, 1000 genomes; LRA, long-read assembly.

7. CONCLUSIONS AND FUTURE WORK

We give an algorithm for building the SA and SA sample from the prefix-free parse of an input string *S*, which fully completes the practical challenge of building the index proposed by Gagie et al. (2018). This leads to a mechanism for building a complete index of large databases—which is the linchpin in developing practical means for pan-genomics short-read alignment. We apply our method for indexing partial and whole human genomes, and show that it scales better than Bowtie with respect to both memory and time. This allows for an index to be constructed for large collections of chromosome 19s (500 or more); a task that is out of reach of Bowtie, causing it to exhaust memory even with a budget of 512 GB. Our method produces indexes in a smaller memory footprint than a Hybrid Index-based method (CHIC; Valenzuela and Mäkinen, 2017) while providing much faster locate time.

Although this work opens the door to indexing large collections of genomes, it also highlights problems needing further investigation. A major question is how this work can be adapted to work on large sets of sequence reads. This problem not only requires the construction of the *r*-index but also adapting and incorporating efficient means (Bannai et al., 2018) to update the index as new data sets become available. Moreover, the use of many reference sequences complicates the task of a read aligner performing approximate matching. In the future, it will be important to explore both techniques, such as *r*-index, that can facilitate the seed-finding phase of approximate matching, and also techniques—perhaps such as those proposed in entropy-scaling search (Yu et al., 2015)—that can facilitate the gapped extension phase.

ACKNOWLEDGMENTS

AUTHORS' CONTRIBUTIONS

T.G. and G.M. conceptualized the idea and developed the algorithmic contributions of this work. G.M. and A.K. implemented the construction of the *r*-index and conducted experiments. A.K. and C.B. assisted and oversaw the experiments and implementation. B.L. and T.M. assisted in implementing the construction of the *r*-index and design, and executed the comparisons with Bowtie and CHIC. All authors contributed to the writing of this article.

AUTHOR DISCLOSURE STATEMENT

The authors declare they have no conflicting financial interests.

FUNDING INFORMATION

REFERENCES

Auton, A., Brooks, L.D., Durbin, R.M. et al. 2015. A global reference for human genetic variation. *Nature* 526, 68–74.

Bannai, H., Gagie T., and Tomohiro, I. 2018. Online LZ77 parsing and matching statistics with RLBWTs, 7:1–7:12. In *Proceedings of the 29th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 105. Schloss Dagstuhl, Qingdao, China.

Boucher C., Gagie, T., Kuhnle, A., et al. 2018. Prefix-free parsing for building big BWTs, 2:1–2:16. In *Proceedings of 18th International Workshop on Algorithms in Bioinformatics (WABI),* 113. Schloss Dagstuhl, Helsinki, Finland.

Burrows, M., and Wheeler, D.J. 1994. A block sorting lossless data compression algorithm. Technical Report 124. Digital Equipment Corporation.

Danek, A., Deorowicz, S., and Grabowski, S. 2014. Indexes of large genome collections on a PC. *PLoS One* 9, e109384.

Deorowicz, S., Kokot, M., Grabowski, S., et al. 2015. KMC 2: Fast and resource-frugal k-mer counting. *Bioinformatics* 31, 1569–1576.

Ferrada, H., Gagie, T., Hirvola, T., et al. 2014. Hybrid indexes for repetitive datasets. *Philos. Trans. R. Soc. A Math. Phys. Eng. Sci*. 372, 1–9.

Ferrada, H., Kempa, D., and Puglisi, S.J. 2018. Hybrid indexing revisited. In *Proceedings of the 20th Algorithm Engineering and Experiments (ALENEX)*, 1–8. SIAM, New Orleans, LA, USA.

Ferragina, P., and Manzini, G. 2000. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, 390–398. IEEE, Redondo Beach, CA, USA.

Gagie, T., Navarro, G., and Prezza, N. 2018. Optimal-time text indexing in bwt-runs bounded space. In *Proceedings of the 29th Annual Symposium on Discrete Algorithms (SODA)*, 1459–1477. SIAM, New Orleans, LA, USA.

Gagie, T., and Puglisi, S.J. 2015. Searching and Indexing Genomic Databases via Kernelization. *Front. Bioeng. Biotechnol*. 3, 10–13.

Garrison, E., Siren, J., Novak, A.M., et al. 2018. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nat. Biotechnol.* 36, 875–879.

Huang, L., Popic, V., and Batzoglou, S. 2013. Short read alignment with populations of genomes. *Bioinformatics* 29, i361–i370.

Jain, M., Koren, S., Miga, K.H., et al. 2018. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nat. Biotechnol.* 36, 338–345, 04.

Kärkkäinen, J., Kempa, D., and Puglisi, S.J. 2015. Parallel external memory suffix sorting. In *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 329–342. Springer, Ischia Island, Italy.

Langmead, B., and Salzberg, S.L. 2012. Fast gapped-read alignment with Bowtie 2. *Nat. Methods* 9, 357.

Langmead, B., Trapnell, C., Pop, M., et al. 2008. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.* 10, R25.

Levy, S. Sutton, G., Ng, P.C., et al. 2007. The diploid genome sequence of an individual human. *PLoS Biol.* 5, e254.

Li, H. 2013. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv* arXiv:1303.3997.

Li, H., and Durbin, R.M. 2009. Fast and accurate short read alignment with Burrows-Wheeler Transform. *Bioinformatics* 25, 1754–60.

Li, R., Yu, C., Li, Y., et al. 2009. SOAP2: An improved ultrafast tool for short read alignment. *Bioinformatics* 25, 1966–1967.

Maciuca, S., del Ojo Elias, C., McVean, G., et al. 2016. A natural encoding of genetic variation in a Burrows-Wheeler transform to enable mapping and genome inference. In *Proceedings of the 16th Annual Workshop on Algorithms in Bioinformatics (WABI)*, 222–233. Springer, Aarhus, Denmark.

Mäkinen, V., Navarro, G., Sirén, J., et al. 2010. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.* 17, 281–308.

Peng, Y., Leung, H.C.M., Yiu, S.M., et al. 2012. IDBA-UD: A de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth. *Bioinformatics* 28, 1420–1428.

Policriti, A., and Prezza, N. 2018. LZ77 computation based on the run-length encoded BWT. *Algorithmica* 80, 1986–2011.

Schneeberger, K., Hagmann, J., Ossowski, S., et al. 2009. Simultaneous alignment of short reads against multiple genomes. *Genome Biol.* 10, R98.

Seo, J.S., Rhie, A., Kim, J., et al. 2016. De novo assembly and phasing of a Korean human genome. *Nature* 538, 243–247.

Shi, L., Guo, Y., Dong, C., et al. 2016. Long-read sequencing and de novo assembly of a Chinese genome. *Nat. Commun*. 7, 12065.

Sirén, J., Välimäki, N., and Mäkinen, V. 2014. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Trans. Comput. Biol. Bioinform*. 11, 375–388.

Steinberg, K.M., Schneider, V.A., Graves-Lindsay, T.A., et al. 2014. Single haplotype assembly of the human genome from a hydatidiform mole. *Genome Res.* 24, 2066–2076.

Stevens, E.L., Timme, R., Brown, E.W., et al. 2017. The public health impact of a publically available, environmental database of microbial genomes. *Front. Microbiol*. 8, 808.

Valenzuela, D. 2016. CHICO: A compressed hybrid index for repetitive collections. In *15th International Symposium on Experimental Algorithms,* 326–338. Springer, St. Petersburg, Russia.

Valenzuela, D., and Mäkinen, V. 2017. CHIC: A short read aligner for pan-genomic references. Technical Report, BioRxiv. Cold Spring Harbor Laboratory, Cold Spring Harbor, NY, USA. DOI: 10.1101/178129.

Valenzuela, D. Norri, T., Valimaki, N., et al. 2018. Towards pan-genome read alignment to improve variation calling. *BMC Genomics* 19(Suppl 2), 87.

Wandelt, S., Starlinger, J., Bux, M., et al. 2013. RCSI: Scalable similarity search in thousand(s) of genomes. *Proc. VLDB Endow*. 6, 1534–1545.

Yu, Y.W., Daniels, N.M., Danko, D.C., et al. 2015. Entropy-scaling search of massive biological data. *Cell Syst*. 1, 130–140.

Address correspondence to:
Taher Mun, PhD Candidate
Department of Computer Science
John Hopkins University
3400 North Charles Street
Baltimore, MD 21218-2682

E-mail: tmun1@jhu.edu