Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Manuscript Category

OXFORD

Subject Section

DREAM-Yara: An implementation of an exact read mapper for very large databases

Temesgen Hailemariam Dadi [1,*], Vitor C. Piro [2,4], Andreas Andrusch [3], Enrico Seiler [1], Bernhard Y. Renard [2] and Knut Reinert [1,]

[1] Algorithmic Bioinformatics, Institute for Bioinformatics, FU Berlin, Takustrasse 9, 14195 Berlin, Germany, [2] Bioinformatics Unit (MF1), Robert Koch Institute, 13353 Berlin, Germany, [3] Centre for Biological Threats and Special Pathogens (ZBS1), Robert Koch Institute, 13353 Berlin, Germany and [4] CAPES Foundation, Ministry of Education of Brazil, Brasília - DF, 70040-020 Brazil.

* To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: Mapping-based approaches have become limited in their application to very large sets of references since computing an FM-index for very large databases (e.g. > 10 GB) has become a bottleneck. This affects many analyses that need this index as an essential step for approximate matching of the NGS reads to reference databases. For instance, in metagenomics, the size of the reference sequences has become prohibitive to compute a single full-text index on standard machines. Even on large memory machines, computing the index takes about one day of compute time. As a result, updates of indices are rarely performed. Hence, it is desirable to distribute the indices to solve the index construction and update problem while preserving fast search times.

Results: To solve the index construction and update problem we propose the DREAM (Dynamic seaRchablE pArallel coMpressed index) framework in this paper and provide an implementation. The main contributions are the introduction binning directories using a novel use of Bloom filters. We combine several Bloom filters to form an *interleaved* Bloom filter and use this new data structure to quickly exclude reads for parts of the databases where they cannot match. This allows us to keep the databases in several indices which can be easily rebuild if parts are updated while maintaining a fast search time. The second main contribution lies in the integration of the Yara exact read mapper (Siragusa, 2013) in a distributed version for the DREAM framework.

Contact: temesgen.dadi@fu-berlin.de

Availability: https://gitlab.com/pirovc/dream_yara/

1 Introduction

Within the last ten years modern sequencing technologies have brought a super-exponential growth of sequencing capacities. This has enabled the cheap sequencing of the genomic content of pangenomes (Consortium, 2018), metagenomes, or many individuals of the same species (e.g. the 100,000 genome project) that differ only slightly from each other. Yet, the small individual differences are of interest (i.e. SNPs, or small structural polymorphisms) to elucidate the cause of diseases or reconstruct evolutionary events.

These data sets expose interesting characteristics. They are large, while some large fractions are highly redundant (e.g. 100,000 genome project, or storing different strains of bacteria) and hence amenable to compression techniques (e.g. Rahn *et al.* (2014); Schneeberger *et al.* (2009)). On the other hand, compression usually makes it costly to implement the main operations on the data, namely *finding approximate matches* of (many) queries (approximate in the sense of edit distance).

While a lot of research has focused on indexing such data sets, the resulting solutions lack, in general, the ability to easily *change* the underlying data sets. That means it is costly (a) to change small parts of a sequence, or (b) to add or delete complete sequences while maintaining the ability to support fast approximate string searches. For example, in metagenomics this problem becomes more and more pressing.

© The Author 2018. Published by Oxford University Press. All rights reserved. For permissions, please e-mail: journals.permissions@oup.com

Many metagenomics search tools (e.g. Hauswedell *et al.* (2014); Piro *et al.* (2016)) and read mappers Li and Durbin (2010); Siragusa (2013) use Burrows-Wheeler-Transform (BWT) based FM-indices (also often referred to as compressed suffix arrays (CSA)) Ferragina and Manzini (2000) which have to index about 50 to 200 gigabases. Due to constant database updates changes occur on a daily or weekly basis and thus require a newly constructed index. Recomputing a single index of this size is quite costly in terms of space and time, even if approaches of merging BWTs are used Bauer *et al.* (2011); Sirén (2009). For example, it takes about one day to compute the index for the data set used in our experiments.

On the other hand, the ability for fast approximate searches in such an index is crucial. It is used either directly to find all approximate occurrences of a (short) string, or parts of it in seed-and-extend approaches.

In this work, we address the problem posed by relying on one large index. We propose a data structure framework which can offer *various* solutions for the above areas depending on some key parameters of the input set (size of input, amount of redundancy, importance of rebuilding time vs. search time). We name the framework a DREAM index (Dynamic seaRchablE pArallel coMpressed index) and describe in this work a first working implementation of it which allows to easily update the underlying database and at the same time fast read mapping using a standard approach.

In the following, we describe the general DREAM index framework, followed by a description of our implementation which consists of three major contributions. First, a taxonomy based binning approach for a collection of database sequences (e.g. bacterial genomes), second a novel data structure for quickly distributing reads to bins for mapping that relies on a combination of Bloom filters (Bloom, 1970) and k-mer counting, and lastly a distributed, parallel version of the Yara read mapper (Siragusa, 2013).

2 Methods

2.1 The DREAM index framework

In Figure 1 we describe the general DREAM index framework. On the left we show the input to our framework, namely a set of database sequences that needs to be indexed and one or more sets of sequencing reads. As pointed out above, we want to be able to add sequences to and (possibly) delete sequences from the index quickly. At the same time we want to be able to conduct approximate searches of reads in the index.

The DREAM index itself consists of two layers for access and a collection of smaller sub-indices which will contain highly similar (sub)sequences. The *dynamic operation distributor* layer will determine which sub-indices need to be changed to complete the update operations and trigger the respective operations in parallel on the sub-indices. The *approximate search distributor* layer will determine which sub-indices need to be searched, conduct the searches in parallel, and consolidate the results. It is obvious, that this will work better, if we place relatively similar sequences (e.g. bacterial genomes of the same genus) into the same sub-index. Note that this will also benefit compression of those indices. However, we will not address compression deeper within this work.

We will rather use standard FM-indices for the sub-indices since they support fast approximate queries and give a coarse-grained dynamization by simply rebuilding a sub-index if needed. For other implementations of a sub-index, different solutions for the approximate search are possible and not in the scope of this publication.

For the dynamic operation and approximate search layer, we propose various solutions with interesting space/time trade-offs. The distributors are based on an application of the k-mer counting lemma and the use of k-mer dictionaries or a novel type of Bloom filters.

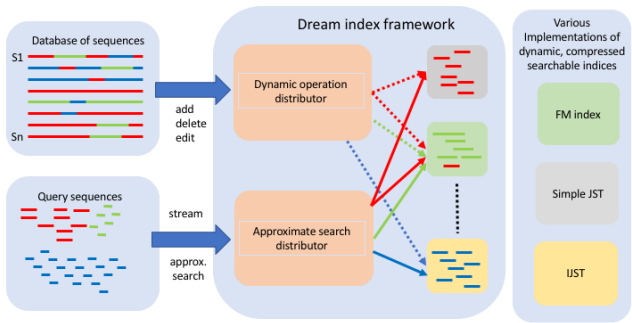

Fig. 1. Sketch of the DREAM index framework. The red sequence piece among the green ones symbolizes that we do not require a perfect partitioning allowing us to use fast methods. The boxes on the right symbolize the potential use of different index implementations. Note that we use in this work solely FM indices.

For the following discussion we assume that we have an overall text length of T, and define the *effective* text length of $T(k)$ as the number of distinct k-mers in T. Further we assume that we have divided the text into b bins in such a way, that a bin B_i contains parts of T with a small effective text length. Hence we denote the set of all k-mers in bin B_i with $T_i(k)$.

2.2 Binning sequences, dynamic updates

In this work we will consider contiguous sequences in the given reference genomes as the smallest unit of sequences that can be clustered into bins. That means we will not split those sequences into smaller parts. The results we present later are based on a metagenomics data sets for which it is relatively easy to compute or obtain a taxonomic tree. For binning, we hence used clusters generated with TaxSBP (https://github.com/pirovc/taxsbp), an implementation of the approximation algorithm for the hierarchically structured bin packing problem (Codenotti *et al.*, 2004) based on the NCBI Taxonomy database (Federhen, 2012). This clustering method is very efficient, given that it uses the "pre-clustered" taxonomic tree information to generate similar sized groups of closely related sequences. Adding and removing sequences is also straightforward once their taxonomic classification is known. Note that in the absence of taxonomic information other, e.g. k-mer based, clustering methods can be used in our framework.

To evaluate how well the used strategy places sequences into bins, we can use the *effective bin ratio* $r_i(k) = T_i(k)/T(k)$ for each bin B_i as measure and the *effective ratio* $r(k) = \sum_{i=1}^{b} r_i(k)$ as overall measure. Our approach will work well, if the effective ratio is near 1. In that case, the likelihood of a false positive verification is low because the bins do not share many k-mers. In 3.2 we will report on $r(k)$ for a variety of bin numbers and values of k.

2.3 Approximate search distributor

For our approximate search distributor we use what we call a *binning dictionary* in conjunction with a well-know k-mer counting lemma. The general general idea of a binning dictionary D is that we will mark for a fixed k-mer in which bin it occurs using a bitvector. Then, when we want to search a pattern p approximately, the following well known Lemma gives a sufficient condition for the pattern to occur in a bin.

Lemma 1. *For a given k and number of errors e, there are $k_p = \mid p \mid -k+1$ many k-mers in p and an approximate occurrence of p in T has to share at least $t = (k_p - k \cdot e)$ k-mers.*

Hence, we can simply look up each k-mer of a pattern in D, retrieve the bitvector marking its occurrences in the bins, and update an array of counters for each bin. If the counter exceeds the threshold for the bin, the pattern will be searched approximately in the bin, otherwise not. This approach is depicted in Figure 2.

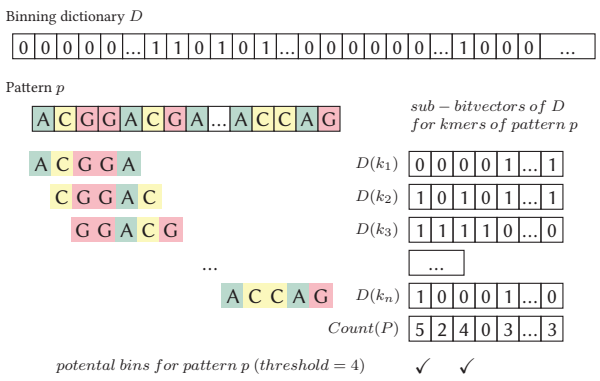

Fig. 2. Binning directory using the k-mer Lemma. For each k-mer k_i generated from a pattern p we extract binning subbitvectors $D(k_i)$ representing the bins containing k-mer k_i. For all set bits in $D(k_i)$ we increment the counter of corresponding bin. Bins with counter greater or equal than the threshold (in this case 4) need to be validated for p.

We discuss here two variants for implementing the binning dictionary, each of which can use either a standard bitvector, or alternatively, a sparse bitvector implementation from the SDSL Gog *et al.* (2014). While a standard bitvector of size n uses n bits, the compressed bitvector of the SDSL (Gog *et al.*, 2014) uses approximately $m \cdot \left(2 + \log \frac{n}{m}\right)$ bits, where m is the number of bits set, and n the length of the bitvector. Note, that while this reduces the space consumption, it increases the access time.

The first alternative is to interpret the k-mers directly as addresses. Hence, this approach is only viable for moderate k (e.g. $k < 16$). For the DNA alphabet, each of the 4^k entries then points to a binning bitvector for the bins (either compressed or uncompressed). For $k = 14$ and $b = 1024$ bins, this would result in a space consumption of $4^14 \cdot 1024 = 2^{38}$ bits which are about 34 Gb.

As a second alternative, we propose a novel idea of combining b standard Bloom filters Bloom (1970) in an *Interleaved Bloom Filter* (IBF). A Bloom filter is simply a bitvector of size n and a set of h hash functions that map a key value, in our case a k-mer, to one of the bit positions. A value is present in the Bloom filter, if all h positions return a 1. Note that a Bloom filter can give a false positive answer. However, if the Bloom filter size is large enough, the probability of a false positive answer is low. A Bloom filter of size n bits with h different hash functions and m elements inserted has a probability for giving a false positive answer of

$$p_{fp} = \left(1 - \left(1 - \frac{1}{n}\right)^{h \cdot m}\right)^h.$$

For this reason we have to allocate sufficient space, such that p_{fp} does not become too large. Still, the problem of using a simple Bloom filter is that it does not point us to the binning bitvectors. To alleviate the problem, we propose to use b Bloom filters (one for each bin) with identical hash functions and then interleave their bitvectors. Putting it differently, this means that we replace each bit in the Bloom filter by a (sub)-bitvector of size b, where the i-th bit "belongs" to the Bloom filter for bin B_i. We call the resulting Bloom filter an *Interleaved* Bloom Filter (IBF). The IBF has a size of $b \cdot n$. When inserting a k-mer from bin B_i into the IBF, we compute all h hash functions which point us to the position of the block where the sub-bitvectors are and then simply set the i-th bit from the respective beginnings. Hence, we effectively interleave b Bloom filters in a way that allows us to retrieve the binning bitvectors for the h hash functions easily. When querying in which bins a k-mer is, we would retrieve the h sub-bitvectors and apply a logical AND to them which results then in the required binning bitvector indicating the membership of the k-mer in the bins. In reality, this approach has a significant advantage in query time as retrieving a (sub)-bitvector is extremely cache-friendly. As pointed out above, the binning bitvectors can be compressed if appropriate (i.e. if the vectors are sparse enough). The procedure is depicted in Figure 2.3.

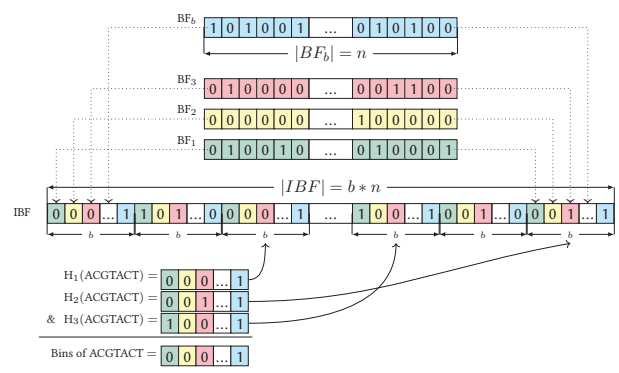

Fig. 3. Example of an IBF. Differently colored Bloom filters of length n for the b bins are shown in the top. The individual Bloom filters are interleaved to make an IBF of size $b \times n$. In the example we retrieve 3 positions for a k-mer (ACGTACT) using 3 different hash functions. The corresponding sub bitvectors are combined with a bitwise & giving us the needed binning bitvector.

Both approaches for binning directories share the implementation for counting. For this, we allocate a t many counters, each with b entries, where t is the number of threads used. These counters are reused as we process the reads in parallel.

2.4 Estimating the number of verifications

Obviously the choice of parameters has a profound effect on space requirement and speed in this approach. In the next paragraph we will discuss the effect of the choices under some simplifying assumptions.

Assume we want to distribute the approximate searches for a pattern p of length l with a fixed choice of k and e using the k-mer lemma. This also fixes the threshold t and the number of k-mers in a read k_p. We can then argue as follows. It is straightforward to compute the effective bin ratios $r_i(k)$ and we can count the number $U_i(k)$ of unique k-mers in each bin. Hence, a bin B_i has $T_i(k) - U_i(k)$ non-unique k-mers and hence the probability of a false positive k-mer match in bin B_i is about $p_k^i = \frac{T_i(k) - U_i(k)}{T(k)} = r_i(k) - \frac{U_i(k)}{T(k)}$. This holds in case of a k-mer directory, which gives us no false positive answers. In case of a Bloom filter, we can adjust this probability by adding $p_{fp}^i \cdot (1 - p_k^i)$, where p_{fp}^i is the false positive rate for the Bloom filter of bin B_i.

According to the k-mer lemma, t out of k_p many k-mers have to match and hence it holds that the probability for a false positive pattern verification in bin B_i is:

$$p_v^i = 1 - \sum_{x=0}^{t-1} \binom{k_p}{x} p^x \cdot (1-p)^{k_p - x}$$

with $p = p_k^i$ or $p = p_k^i + p_{fp}^i \cdot (1 - p_k^i)$. From this we can compute the overall expected number of false positive verifications for a read as $\sum_{i=1}^{b} p_v^i$. This will serve as a lower bound, since we assume a random distribution of the k-mers. In reality, bins will contain some repeat regions or are split if they are too big. In the result section we will report on the actual number of verifications per bin for simulated reads from an actual binning to see a) whether all reads from the bin are indeed found and b) how many false positive reads are mapped to each bin, that means reads that map within the error bounds but were simulated from another bin.

Note that the value p has also an impact on the density of the bitvectors. The Bloom filter will have more bits set than the k-mer dictionary and hence the compression ratio is worse.

While the IBF is only a part of the binning directory, we anticipate that the IBF will be a very useful data structure for set membership of objects in bins and will see a wide usage, especially for assessing k-mer content. When writing this paper, it was brought to our attention that Bradley et al. independently thought of a similar data structure in Bradley *et al.* (2017), although they do not use them in conjunction with the k-mer Lemma and do not interleave them.

2.5 The Yara read mapper

In this section, we describe Yara (Siragusa, 2013), currently the state-of-the art read mapper of the SeqAn library, an exact read mapping tool that is efficient, easy to use, and produces well-defined and interpretable results. It outperforms previous exact tools like RazerS3 (Weese *et al.*, 2012) by a factor of 200 in speed and even heuristic methods like Bowtie 2 (Langmead and Salzberg, 2012) and BWA (Li and Durbin, 2010) by a factor of two and three, respectively (Siragusa, 2013).

The efficiency of the tool is due to a novel combination of known algorithmic concepts and a solid implementation based on the SeqAn library (Reinert *et al.*, 2017). Yara is based on the notion of alignment strata (best+x mode) which we advocate as an alternative to the conventional all-mapping paradigm, since stratification will always yield at least one hit (within reasonable bounds) and appears to be more practical. To explain, we define the e-stratum $\mathbb{S}_e = \{(i, j, e) : d_E(G_{i\ldots j}, R) = e\}$, which denotes the set of all mapping locations of a read R at edit distance e from the reference genome G. According to the above definition, conventional all-mapping under the edit distance defines the set $\mathbb{S} = \mathbb{S}_0 \cup \mathbb{S}_1 \cup \cdots \cup \mathbb{S}_k$ as relevant mapping locations within an *absolute* error threshold k. Stratified all-mapping refines this definition by considering only mapping locations up to a certain degree s from the *optimal* stratum. That means if the distance of any optimal mapping location for read R is $o = \min\{e \in [0, k - s] : \mathbb{S}_e \neq \emptyset\}$, then stratified all-mapping considers mapping locations $\mathbb{S} = \mathbb{S}_o \cup \cdots \cup \mathbb{S}_{o+s}$ within a *relative* suboptimality error threshold s to be relevant.

Filtration by strata. Yara significantly improves the runtime of stratified all-mapping over conventional all-mapping. Obviously, the most straightforward way to achieve stratified all-mapping consists of performing conventional all-mapping and subsequently filtering out any irrelevant mapping location (For instance, RazerS 3 implements this method). Another naïve method consists of performing up to k rounds of conventional all-mapping, using filtration schemes with thresholds incrementing from 0 to $o + s$. It is easy to see that also this method performs redundant computation: the total work for any read that maps at distance greater than or equal to k corresponds to the sum of all k filtration schemes.

The following stratified all-mapping method, implemented in Yara, guarantees not to perform more work than conventional all-mapping. Indeed, the key idea is to simply reduce any filtration scheme full-sensitive within distance k to be full-sensitive within distance $o + s$.

Adaptive filtration. Yara applies an adaptive filtration scheme *per read* because, under any fixed filtration scheme, the number of verifications per read is not uniform: within a typical human genome resequencing, most reads produce very few verifications and are easily mappable, while few other reads are problematic and often not even confidently mappable to one single location. Consequently, any fixed filtration scheme turns out to be too weak for some reads yet too strong for others, thus suboptimal in practice. An adaptive filtration scheme per read improves filtration efficiency by optimizing the ratio between filtration speed and specificity. Yara thus automatically chooses an adaptive filtration scheme per read, without requiring manual parameterization by the user. Adaptive filtration works as follows. Yara initially applies filtration with exact seeds to all reads. The tool counts the number of verifications to be performed for each read, thus decides if it is worth proceeding with the verification phase or alternatively applying a stronger filtration scheme. This decision depends on fine-tuned internal verification thresholds. Under standard Illumina setups, exact seeds provide efficient filtration for up to 70–80 % of the reads; on the remaining reads, a filtration scheme using 1– or 2–approximate seeds works better. Thus, Yara starts with the quickest filtration scheme and becomes more specific whenever it pays off to do so.

Greedy verification. Yara implements a simple greedy strategy to minimize the number of verifications necessary to find all relevant stratified mapping locations. As candidate locations can be verified in any order, Yara chooses an ordering of the seeds that minimizes the expected number of verifications. The tool first finds all seeds and ranks them by number of candidate locations produced. Then it processes all candidate locations, from the least to the most frequent seed, until it explores s strata from the first non-empty one. The verification is done by a banded version of Myers Bit-vector algorithm.

Paired-end protocol. Paired-end and mate-pair protocols are the sequencing protocols of choice of Illumina instruments. As reads are sequenced in pairs from the two ends of the same DNA fragment, they are expected to map closely in the reference genome. The added information of an expected DNA insert size allows to disambiguate the original location of read pairs more confidently than in the single-end protocol. Nonetheless, the lack of any proper pair of mapping locations signals a potential structural variation, a long indel or an inversion. In the paired-end or mate-pair workflows, Yara maps paired reads independently, exactly as in the single-end workflow, and then reports all relevant mapping locations per read.

Yara is already in use by several groups, it was used for improved metagenomics classification Dadi *et al.* (2017) and was tested favorably by Břinda et al. (Binda *et al.*, 2016). In addition it scales well using many threads. For those reasons we choose it as the first implementation within the DREAM framework and call the resulting tool DREAM-Yara.

In DREAM-Yara, we load sequencing reads in batches and identify which bins contain potential mapping locations for individual reads. This is done using the binning directories. As a result, we get b different subsets out of the loaded reads representing the different bins. A read belongs to a subset R_{bi} if and only if it or its reverse complement or its mate pair shares enough number of k-mers with bin B_i. After that, we map each subset of reads against the index of the corresponding bin. All the mapping results from all subsets are collected, we consolidate the mapping for the whole set of reads considering the mapping qualities for each read. Finally, the mapping result is written to a single SAM/BAM file.

3 Evaluation

In this section we report about our computational experiments using DREAM-Yara evaluating the needed update times for a realistic, metagenomics data set as well as the time to map real Illumina reads.

In addition we simulated some reads form the individual bins to evaluate the sensitivity of the tool.

To evaluate the DREAM-Yara database indexing and updating we use a set of Archaeal and Bacterial complete genome sequences retrieved from NCBI's RefSeq database (Haft *et al.*, 2017) dating from 2017-09-26. This dataset comprises 15,250 sequences representing 2,991 species, summing up to a total of 31.34 Gbp, a database size for which it takes significant time to compute a single index. An updated set from the same database was selected based on the updated *Escherichia Coli* sequences from 2017-12-19. This update set sums up to 0.23 Gbp with one removed sequence and 155 new sequences, thus a typical set of sequences for which we want to update our index.

3.1 Time and memory consumption for real reads

In our first evaluation the DREAM-Yara mapping was evaluated with a synthetic set of reads consisted of 64 laboratory-mixed microbial genomic DNAs. The sample provided by Shakya et al. (Shakya *et al.*, 2013) contains reads based on know organisms, providing a broad diversity among bacteria and archaea at different taxonomic levels as well as a wide range of genetic variation. One million reads were randomly selected from our 64-organism read set to evaluate mapping performance. We used a computer with 32 (Intel(R) Xeon(R) CPU E5-2650 v3 2.30GHz) processors and 130GB of memory.

DREAM-Yara was compared against BWA (Li and Durbin, 2010) and standard Yara (Siragusa, 2013) each using one large index to see how much slower DREAM-Yara would be for using a IBF binning filter and many indices. We also ran DREAM-Yara without the search distributor (called distributed Yara) to assess the time needed for a trivial distribution.

Table 1 shows time and memory consumption for indexing, updating and mapping for DREAM-Yara, BWA, standard Yara and distributed Yara. DREAM-Yara outperforms the other methods in nearly all measurements. Distributed Yara (without bloom filter) has the best time in indexing and updating, but its implementation is not practical considering the high mapping time since all reads have to be mapped against all distributed indices which takes almost $10h$.

Indexing more than 30Gb of reference sequences took around 55 minutes with 1024 bins, while BWA required more than 19 hours and standard Yara more than 28 hours. Hence DREAM-Yara achieves a speed-up of $20x$ and $31x$, respectively. Memory consumption was lower for DREAM-Yara, using 26% less memory (11.8Gb) when compared against BWA, where the peak memory consumption of DREAM-Yara is from the use of the binning directory and not for the indices.

DREAM-Yara has also the advantage of having b many small indices which makes rebuilding indices less time-consuming. In case of changes in reference-set, we have to rebuild only the indices of the bins that the change has affected, whereas BWA and standard Yara need to rebuild the complete index for any change in the reference set. To demonstrate the advantage of this, we considered all the reference changes under the species *E. Coli* between 2017-09-26 and 2017-12-19. This accounted for 155 new sequences and one removed sequence which affected 42, 15 and 4 bins out of 1024, 256 and 64 bins respectively. These are a small portion of the complete set and rebuilding them took 6.5 minutes for 1024 bins, 9.7 minutes for 256 bins and 17.8 minutes for 64 bins. This is a significant improvement considering the amount of time needed if we had one big index which is about a day.

When we consider the time and space requirements for mapping, DREAM-Yara needed 40% less memory when compared against BWA and standard Yara in this task, being also faster than BWA for any bin size. Standard Yara is faster for mapping, even though with a higher memory consumption. Clustering times with TaxSBP are negligible for the evaluated data, having the only cost of reading/writing the input file into separated files for each bin.

Hence, in this evaluation DREAM-Yara removes the bottleneck of index reconstruction successfully while remaining in speed and memory consumption competitive to standard read mappers and being a factor of about 130 times faster than the trivial distribution.

3.2 Evaluation of distributed mapping using simulated reads

In this section we want to evaluate whether the DREAM framework affects the sensitivity of the read mapper. For this, we sampled one reference out of each bin and simulated a total of $100,000$ reads from the sampled reference using the standard Illumina error profile of Mason2 (Holtgrewe, 2010). We then mapped the reads against the complete reference using BWA, Standard Yara, and DREAM-Yara. The mappers were set to report *all* alignments with up to an edit distance of 3 in order to make them comparable.

A read mapper should at least map each of the simulated reads back to the bin it was simulated from. Of course, it can also map it to another bin within the error bounds, for example if our clustering split a large bin into two smaller bins. Since we know from which bin a read was simulated, we can count the number of primary (from the bin the read is from) and secondary mappings (other bins it can be aligned to) and compare it to the number of verifications allowed by our binning filter. For the binning filter to be effective for a bin, we would expect the number of verifications and the number of actually mapped reads to be close. The secondary mappings can of course occur, because our taxonomic clustering is not perfect and we do not expect the effective ratio to be 1.

We start by giving the effective ratio for different k for the dataset described above and for 64, 256 and 1024 bins.

Table 2 shows that the effective text ratio grows moderately with increasing number of bins, indicating that the taxonomic clustering works fairly well, albeit not perfectly. As we can see from Table 3 the actual mapping time is growing similarly to the effective ratio for $b = 64, 256$ and 1024. Another observation is that for $k = 14$, the effective ratio is quite high which excludes the use of k-mer binning directories. Instead an IBF was used for this data set. Since the update time is clearly faster for 1024 bins, we decided to do the analysis with simulated reads for $b = 1024$ and with $k = 19$, since this was slightly faster than $k = 20$.

The results of the sensitivity evaluation are shown in Table 3.

Out of 100K reads, DREAM-Yara is able to map *all* reads back to the original bins, resulting in a sensitivity of 100%. Standard Yara and BWA fall short of reporting 4 and 24 of those alignments, respectively. BWA also misses some of the secondary mappings. It computes about 26.1M alignments while standard Yara finds 2.8% (+743K) and DREAM-Yara finds 2.9% (+754K) more secondary alignments. We also counted the number of verifications triggered by DREAM-Yara. Our filter based on binning directories with IBF generated a total of $1,453,686$ verifications out of which 81.7% lead to at least one unique alignment (primary or secondary). This shows that the filter is highly effective. In addition, Table 1 shows that filtering takes only $\approx 16\%$ of the overall time for mapping reads.

3.3 Speed and space consumption of binning directories

Binning directories use either a k-mer table for moderate k or an IBF to retrieve the binning bitvectors. The IBF has to compute h hash functions, retrieve h sub-bitvectors and compute a bitwise `AND` while a k-mer table can directly retrieve the binning bitvector.

In both approaches we have to traverse the binning bitvector of size b and increment counters for each bit set to 1. To speed up this crucial step we used the `lzcount` intrinsic operation which counts the number of leading

	Indexing		Updating		Mapping	
	Time (sec)	Memory (Mb)	Time (sec)	Memory (Mb)	Time (sec)	Memory (Mb)
BWA	69071	44869	N/A*	N/A*	121	54813
Standard Yara	103758	87555	N/A*	N/A*	27	54826
DREAM-Yara (64 bins)	6373 (5521, 852)	33083 (15518, 33083)	1068 (522, 546)	32823 (8749, 32823)	77 (11, 66)	34638
DREAM-Yara (256 bins)	5503 (4695, 808)	33064 (10895, 33064)	583 (344, 239)	32957 (10506, 32957)	84 (14, 70)	34181
DREAM-Yara (1024 bins)	3345 (2685, 770)	33048 (7922, 33048)	390 (166, 224)	32994 (8546, 32994)	89 (14, 75)	34406
Dist. Yara (1024 bins)	2685	7922	166	8546	11662	1652

Table 1. Time and memory consumption for index building, updating and read mapping. DREAM-Yara indexing and updating time values stand for summed wall time and memory for peak memory. The values between parenthesis stand for building the FM-indices and the Bloom filter, respectively. DREAM-Yara was executed in 8 threads for indexing and updating while BWA and standard Yara do not support multi-threading. Mapping was performed before updating the index with 16 threads for all tools. Here the values between parenthesis stand for the filter time and actual mapping time. Values are rounded. N/A *: Updating databases is Not Available and full indexing is necessary.

b,k	12	13	14	15	16	17	18	19	20
64	56.6	44.9	27.2	12.0	4.7	2.2	1.5	1.2	1.1
256	180.1	109.8	49.1	17.1	5.8	2.6	1.7	1.4	1.2
1024	511.0	234.7	82.2	24.7	7.7	3.3	2.1	1.7	1.5

Table 2. Effective ratio for $64, 256$ and 1024 bins and different values of k for the metagenomics data set. Values are rounded.

zeros in a 64 bit word. This accelerated the bin counting step by a factor of almost 2 compared to the individual checking of each bit (data not shown). The `lzcount` intrinsic is part of the BMI CPU extensions and available on all modern processors. Interestingly, the AVX512 SIMD extensions offer a vectorized `lzcount` which will speed up this step further, once the AVX512 set is available on standard computers (Intel's Skylake processor already supports the instruction set). As pointed out above, we used an IBF for our analysis, since the effective ratio is too large for k-mer binning directories. Overall the filtering time for $b = 1024$ using IBF took a total of 14 seconds. Hence it contributes not much to the overall mapping time.

4 Discussion

We hope that our work in this manuscript has an impact in various research directions. First, we think that DREAM-Yara will serve the community as a very practical, exact read mapper for Illumina reads and large data bases. Second, we think that the DREAM framework will trigger further work in this area. The most crucial point in the DREAM framework is the question whether a distribution of reads can be done fast enough in comparison to the time needed for mapping the reads to one or a handful of indices. We showed that the combination of binning directories and the k-mer lemma answers this question positively.

We think it will be interesting to investigate in the future for which parameters the k-mer directory implementation and for which the IBF implementation have more advantages. For large, basically arbitrary k, we presented with the IBF a very viable way to implement binning directories. We also conducted experiments with compressed bitvectors for the binning bitvectors (data not shown) which showed that it is possible to further reduce the space consumption at the expense of run time. For example, if we would have only every 256th bit set to 1 in the used bitvectors, we can compress them by a factor of about 21 while being only 5 times slower in access. In our computational experiments the bitvectors were not sparse enough to gain a significant amount of compression.

Another interesting area to extent this work will be how to bin the sequences. How many bins are optimal for a given set? Will we benefit from allowing to further subdivide the genomic contigs? In this work we opted for a taxonomic based clustering for being a straight forward implementation with an imprtant use case. However, a clustering based on sequence similarity would be beneficial for computing the bins in DREAM-Yara. The taxonomy classification is not purely sequence-based and by definition can have distant related groups with highly similar sequences or closed related groups with low sequence similarity. A sequence-based approach would provide a lower effective ratio, meaning that reads would be potentially mapped to fewer bins, speeding up the mapping procedure. Important to note is that the evenness of the bin sizes makes a difference for our Bloom filter implementation and it is a important aspect to be considered when computing the binning. This comes from the fact that the IBF has b interleaved Bloom filters of the *same* size. Hence the largest bin will have the highest false positive rate for a k-mer, which would result in worse filtering.

Finally, it will be interesting to investigate whether other indices can take advantage of the binning. While FM-indices support fast approximate searching, they do not support the compression of similar sequences. However, the bins in the DREAM index will contain by construction very

	BWA	Standard Yara	DREAM-Yara
Primary alignments	99,976	99,996	100,000
Secondary alignments	26,128,281	26,871,083	26,882,268
Unique secondary alignments	1,084,480	1,087,838	1,088,449

Table 3. Evaluation of BWA, Yara and DREAM-Yara. The first row shows how many of the 100K reads were mapped to the original genome by the mappers (primary alignments). The second row reflects the number of secondary alignments. The third row shows unique secondary alignments, that is the number of read/bin pairs, disregarding multiple alignments in a bin.

similar sequences. Hence we anticipate that compressed indices together with the IBF based search distributor will result in very practical, space efficient and fast read mappers for very large, repetitive data bases.

5 Conclusion

In conclusion, we presented the novel DREAM framework for distributed read mapping which can support fast updates of the sub-indices as well as compression by design. We implemented within this framework DREAM-Yara, a distributed version of the Yara (Siragusa, 2013) read mapper. The main contribution lies in our implementation of the dynamic search distributor by using binning directories with interleaved Bloom filters together with the k-mer counting lemma and an in-memory distributed version of Yara. We showed that the resulting read mapper is very competitive in terms of speed, it is faster than BWA and only slightly slower than the original Yara, which both use one large FM-index, but more than 130 times faster than a trivial distribution. We also showed that DREAM-Yara can conduct a typical batch update for a metagenomic data set in about 6 minutes, whereas the rebuilding of the BWA or Yara index takes about a day. DREAM-Yara is part of the SeqAn library for efficient data types and algorithms.

Acknowledgements

We acknowledge Mikaël Salson for his ideas in defining the DREAM framework jointly with Knut Reinert.

Funding

This work was supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Ciência sem Fronteiras (BEX 13472/13-5) to VCP, the InfectControl 2020 Project (TFP-TV4), and the BMG Project "Metagenome Analysis Tool" (2515NIK043). The authors also acknowledge the support of the de.NBI network for bioinformatics infrastructure, the Intel SeqAn IPCC and the IMPRS for Scientific Computing and Computational Biology.

References

Bauer, M. J. *et al.* (2011). Lightweight bwt construction for very large string collections. In R. Giancarlo and G. Manzini, editors, *Combinatorial Pattern Matching*, pages 219–231, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, **13**(7), 422–426.

Bradley, P. *et al.* (2017). Real-time search of all bacterial and viral genomic data. *bioRxiv*.

Binda, K. *et al.* (2016). Rnf: a general framework to evaluate ngs read mappers. *Bioinformatics*, **32**(1), 136–139.

Codenotti, B. *et al.* (2004). Approximation algorithms for a hierarchically structured bin packing problem. *Information Processing Letters*, **89**(5), 215–221.

Consortium, T. C. P.-G. (2018). Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, **19**(1), 118–135.

Dadi, T. H. *et al.* (2017). SLIMM: species level identification of microorganisms from metagenomes. *PeerJ*, **5**(3), e3138.

Federhen, S. (2012). The NCBI Taxonomy database. *Nucleic Acids Research*, **40**(D1), D136–D143.

Ferragina, P. and Manzini, G. (2000). Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398.

Gog, S. *et al.* (2014). From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337.

Haft, D. H. *et al.* (2017). RefSeq: an update on prokaryotic genome annotation and curation. *Nucleic Acids Research*, (November), 1–10.

Hauswedell, H. *et al.* (2014). Lambda: the local aligner for massive biological data. *Bioinformatics*, **30**(17), i349–i355.

Holtgrewe, M. (2010). Mason a read simulator for second generation sequencing data. *Technical Report FU Berlin*.

Langmead, B. and Salzberg, S. L. (2012). Fast gapped-read alignment with Bowtie 2. *Nat Methods*, **9**(4), 357–359.

Li, H. and Durbin, R. (2010). Fast and accurate long-read alignment with burrowswheeler transform. *Bioinformatics*, **26**(5), 589–595.

Piro, V. C. *et al.* (2016). DUDes: a top-down taxonomic profiler for metagenomics. *Bioinformatics*, **32**(15), 2272–2280.

Rahn, R. *et al.* (2014). Journaled string tree-a scalable data structure for analyzing thousands of similar genomes on your laptop. *Bioinformatics (Oxford, England)*, page btu438.

Reinert, K. *et al.* (2017). The SeqAn C++ template library for efficient sequence analysis: A resource for programmers. *Journal of Biotechnology*.

Schneeberger, K. *et al.* (2009). Simultaneous alignment of short reads against multiple genomes. *Genome biology*, **10**(9), R98.

Shakya, M. *et al.* (2013). Comparative metagenomic and rRNA microbial diversity characterization using archaeal and bacterial synthetic communities. *Environmental microbiology*, **15**(6), 1882–99.

Siragusa, E. (2013). *Approximate string matching for high-throughput sequencing*. Ph.D. thesis, Freie Universität Berlin.

Sirén, J. (2009). Compressed suffix arrays for massive data. In J. Karlgren, J. Tarhio, and H. Hyyrö, editors, *String Processing and Information Retrieval*, pages 63–74, Berlin, Heidelberg. Springer Berlin Heidelberg.

Weese, D. *et al.* (2012). Razers 3: Faster, fully sensitive read mapping. *Bioinformatics*, **28**(20), 2592–2599.