Bioinformatics, 35(2), 2019, 227–234
doi: 10.1093/bioinformatics/bty619
Advance Access Publication Date: 13 July 2018
Original Paper

OXFORD

Sequence analysis

CoMSA: compression of protein multiple sequence alignment files

Sebastian Deorowicz*, Joanna Walczyszyn and Agnieszka Debudaj-Grabysz

Institute of Informatics, Faculty of Automatic Control, Electronics and Computer Science, Silesian University of Technology, Gliwice, PL-44100, Poland

*To whom correspondence should be addressed.

Associate Editor: John Hancock

Received on December 28, 2017; revised on July 3, 2018; editorial decision on July 6, 2018; accepted on July 10, 2018

Abstract

Motivation: Bioinformatics databases grow rapidly and achieve values hardly to imagine a decade ago. Among numerous bioinformatics processes generating hundreds of GB is multiple sequence alignments of protein families. Its largest database, i.e. Pfam, consumes 40–230 GB, depending of the variant. Storage and transfer of such massive data has become a challenge.

Results: We propose a novel compression algorithm, CoMSA, designed especially for aligned data. It is based on a generalization of the positional Burrows–Wheeler transform for non-binary alphabets. CoMSA handles FASTA, as well as Stockholm files. It offers up to six times better compression ratio than other commonly used compressors, i.e. gzip. Performed experiments resulted in an analysis of the influence of a protein family size on the compression ratio.

Availability and implementation: CoMSA is available for free at https://github.com/refresh-bio/comsa and http://sun.aei.polsl.pl/REFRESH/comsa.

Contact: sebastian.deorowicz@polsl.pl

Supplementary material: Supplementary data are available at *Bioinformatics* online.

1 Introduction

In modern times, the sizes of data collected in various fields are huge and are constantly growing. This has a direct impact on the increasing costs of IT infrastructure. As far as bioinformatics is concerned a number of surveys claim that soon the cost related to storage of data will be at least comparable to the costs of generating these data (Deorowicz and Grabowski, 2013; Stephens *et al.*, 2015). A natural solution to this problem is to reconsider the need to store all collected data. One could advocate that a significant part of collected files are rather 'temporary' and can be safely removed after performing the analyses. Unfortunately, this could ruin one of the fundamentals of science, i.e. reproducibility, as many bioinformatics tools work in a non-deterministic manner. That means that when using these tools several times, we would obtain similar results, but not always exactly the same. What is more, the emergence of new bioinformatics software allows revealing undiscovered relationships between collected data and their new biological meaning. Hence, very often the reproducibility is of high importance, so a number of temporary files are stored for a long time in many research centres.

To make storage possible compressing data seems to be a primary solution. Even the 25-years old, but still popular gzip algorithm, treated as the *de facto* standard, allows to reduce many types of data several times. That is why gzip is a common choice in bioinformatics when storing sequencing data, protein sequences, etc. Nevertheless, gzip is a universal compressor, developed mainly to compress textual files with sizes considered to be small from the modern perspective. Besides, being all-purpose, it cannot make use of special types of redundancy existing in the bioinformatic datasets. Therefore, a number of specialized compressors were developed in recent years.

The most attention was devoted to the field of genome sequencing (Bonfield and Mahoney, 2013; Numanagić *et al.*, 2016; Pinho and Pratas, 2014; Roguski and Deorowicz, 2014). The obtained compression ratios are sometimes an order of magnitude better than offered by gzip. Other examples of huge datasets whose storage is

supported by dedicated software are collections of complete genomes of the same species. Various algorithms for reducing their sizes by a few orders of magnitude were proposed (Deorowicz *et al.*, 2013; Li, 2016). In addition, these algorithms may have offered fast queries to the compressed data. Results of ChIP-seq and RNA-seq experiments are another noteworthy example. Solutions capable of compressing them by the factor of at least a few have been proposed (Ravanmehr *et al.*, 2018; Wang *et al.*, 2016) very recently. Finally, taking into account the specificity of the multiple sequence alignment (MSA) of nucleotide sequences, allowed to create compressors that operate definitely more efficiently than general purpose tools (Hanus *et al.*, 2010; Matos *et al.*, 2015).

Storage of protein databases, like Pfam (Finn *et al.*, 2016), is also a challenge. For example, the recent release of Pfam (v. 31.0) contains 16 712 protein families and about 5 billion residues. The uncompressed size of the default variant of the database is about 42 GB. Moreover, its NCBI variant stores more than 22 billion residues and occupies about 228 GB. Even when gzip-compressed, the files are still of extraordinary sizes, i.e. 5.6 and 24 GB, respectively.

A comprehensive survey on existing compression methods used for protein sequences is included in (Hosseini *et al.*, 2016). However, it refers to much smaller databases, that unlike Pfam, do not contain MSAs. The mentioned paper by Pinho and Pratas (Pinho and Pratas, 2014) introduces a tool designed for FASTA and multi-FASTA files containing not only nucleotide but protein sequences as well. An interesting application of compression is presented in (Daniels *et al.*, 2013), where a compression-accelerated search algorithm is proposed. Although it considers protein databases of sizes in the order of gigabytes, it focuses on fast searching. Summarizing, to the best of our knowledge, there has been no breakthrough in the field of compression of present-day protein datasets so far. An interesting alternative could be to use a better universal compressor and the popular 7-zip programme seems to be a reasonable choice. Nevertheless, it consumes a lot of memory for large files and is relatively slow. What is more important, it is still a universal tool, thus it does not use knowledge about properties of these datasets. Therefore, significantly better results should be possible to obtain if we took this information into account and use it during the construction of the algorithm.

In this article, we propose a specialized compression algorithm designed to decrease a size of databases of MSA of protein families. Our tool, CoMSA, can compress both FASTA files containing a single protein family data as well as collections of proteins in Stockholm format (Finn *et al.*, 2016) (used in Pfam database). The proposed algorithm offers several times better compression ratios than gzip (and significantly better than 7-zip) for large protein families. It is also much faster in compression than the examined counterparts and has moderate memory requirements, roughly similar to the input dataset (or a single family for Stockholm files) size.

2 Materials and methods

2.1 Background

The heart of gzip and 7-zip compressors is a well-known Ziv–Lempel family of algorithms (Storer and Szymanski, 1982; Ziv and Lempel, 1977). Its key idea is to look for exact copies of some parts of a file being compressed. When a copy is sufficiently long (usually at least three or four symbols), then it is beneficial to store the information about its location and its length instead of storing its symbols literary. Roughly speaking, both gzip and 7-zip perform so-called LZ-factoring in which the input sequence of symbols is transformed into a sequence of tuples that can be of two types: matches (describing a repetition of a fragment of already processed part of the file) and literals (single symbols stored when no sufficiently long match was found). This sequence of tuples is then coded using one of the entropy coders like Huffman coder (Huffman, 1952) or range coder (Salomon and Motta, 2010). In case of MSA files, looking for copies of the fragments of a text means looking for identical fragments of sequences of various proteins. In this way long runs of gaps, that supplement alignments, are stored as copies of some other 'lines' of MSA files. Such a strategy leads to up to 15-fold compression in case of gzip and up to 50-fold for 7-zip, when compressing large protein families with many gaps.

When considering MSA files several drawbacks of the mentioned approach can be shown. The first one results from the specificity of an index structure, which is necessary to search for matches in the part of the file that has been just processed. Such index structures (like suffix trees, hash tables) are usually several times larger than the indexed part of the data. The second disadvantage results from the dependence between the size of the indexed part and the number of positions that should be examined when looking for the best match. As a consequence, the operating time increases together with the growth of the indexed part. To preserve speed, LZ-compressors have to make a trade-off, e.g. restrict to examining only some fraction of potential matches. What is more, defining what 'the best match' means is not clear. Coding of the longest possible match not always leads to the highest compression ratios. It happens that choosing a shorter match (or even a literal) at a current position leads to a longer match at the following position, which in consequence improves the compression ratio. Besides, in case of MSA files, the sequences are aligned along the columns and it is unusual to find a long match starting at different column than the current one. That is why, most of potential candidates for matches could be omitted during analyses. Finally, this knowledge (inaccessible to universal compressors) could be used to encode the found matches cheaper, i.e. using the smaller number of bits.

In the universal compression field there are also different approaches than LZ-based. The two most known families are based on the Burrows–Wheeler transform (BWT) and prediction by partial matching (PPM) (Cleary and Witten, 1984). The popular bzip2 programme, implementing Fenwick's variant (Fenwick, 1996) of the BWT-based compressor, is a representative of the former family. The BWT (ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-124.ps.zip) of the input sequence is performed at its first stage. As a result, long parts of the transformed sequence contain only a few different symbols (or even a single symbol). Such a temporary sequence is then further changed using a move-to-front transformation (Bentley *et al.*, 1986) and Huffman encoded (Huffman, 1952). In the PPM algorithms the statistics of symbol occurrences in various contexts (formed by preceding symbols) are collected. They are used to estimate the probability of appearance of each symbol in every single place. Based on this estimation short codewords are assigned to more probable symbols and longer ones to less probable, with the use of an entropy coder. An interested reader is referred to one of textbooks on data compression, for more detailed discussion and examples of the LZ-based, BWT-based and PPM-based algorithms (Salomon and Motta, 2010).

2.2 General idea of the proposed algorithm

To overcome the problems with indexing of huge files and make use of the alignment property of the MSA files we decided not to follow the obvious strategy of implementing the LZ-based algorithms adopted for MSA data. Rather than that, we focussed on the recently proposed, positional Burrows–Wheeler transform (PBWT) (Durbin, 2014). Its name reflects that it was motivated by the

```
procedure gPBWT(S)
Input: S — ordered collection of n sequences each of length ℓ
Output: P — ordered collection of n sequences each of length ℓ,
        where P is gPBWT(S)
1  prev_ordering ← {1, 2, ..., n}
2  for j ← ℓ downto 1 do
   {Find histogram of symbols in current column}
3      for c ← 0 to σ − 1 do  H[c] ← 0
4      for i ← 1 to n do  H[S_i^j] ← H[S_i^j] + 1
   {Find cumulative histogram}
5      H*[0] ← 0
6      for c ← 1 to σ − 1 do  H*[c] ← H*[c − 1] + H[c − 1]
   {Determine new ordering}
7      for i ← 1 to n do
8          s ← S^j_{prev_ordering[i]}
9          H*[s] ← H*[s] + 1
10         p ← H*[s]
11         cur_ordering[p] ← prev_ordering[i]
   {Permute current column}
12     for i ← 1 to n do  P_i^j ← S^j_{prev_ordering[i]}
   {Update ordering for next column}
13     prev_ordering ← cur_ordering
```

```
procedure rev-gPBWT(S)
Input: P — ordered collection of n sequences each of length ℓ,
       where P is gPBWT(S)
Output: S — ordered collection of n sequences each of length ℓ
1  prev_ordering ← {1, 2, ..., n}
2  for j ← ℓ downto 1 do
   {Permute current column}
3      for i ← 1 to n do  S^j_{prev_ordering[i]} ← P_i^j
   {Find histogram of symbols in current column}
4      for c ← 0 to σ − 1 do  H[c] ← 0
5      for i ← 1 to n do  H[S_i^j] ← H[S_i^j] + 1
   {Find cumulative histogram}
6      H*[0] ← 0
7      for c ← 1 to σ − 1 do  H*[c] ← H*[c − 1] + H[c − 1]
   {Determine new ordering}
8      for i ← 1 to n do
9          s ← S^j_{prev_ordering[i]}
10         H*[s] ← H*[s] + 1
11         p ← H*[s]
12         cur_ordering[p] ← prev_ordering[i]
   {Update ordering for next column}
13     prev_ordering ← cur_ordering
```

Fig. 1. Pseudocodes of the generalized PBWT algorithm (left) and its reverse (right)

classical BWT. Nevertheless, it was designed to transform aligned bit vectors to allow faster queries for genotype data. Later, Li used the PBWT to develop a specialized compressor of genotype datasets (Li, 2016). One of the assets of the PBWT is its memory frugality as no large index structures (required not only by LZ-based algorithms, but also by classical BWT-based and PPM-based ones) are necessary. Instead, the original PBWT processes rather short bit vectors (of values 0 and 1) one by one.

The PBWT was defined for bit vectors, but fortunately it can be simply generalized to larger alphabets. In the next section we propose such a generalization. Then, we adopt some transforms known from the BWT-based compressors to obtain the novel algorithm for the MSA data.

2.3 Positional Burrows–Wheeler transform for non-binary alphabets

Let Σ be an alphabet of symbols $\{0, 1, \ldots, \sigma - 1\}$. Let S be an ordered collection of equal-length sequences $\{S^1, S^2, \ldots, S^n\}$. The length of each sequence is denoted by ℓ and for each valid i: $S^i = s^i_1 s^i_2 \ldots s^i_\ell$. Each s^i_j is a symbol from the alphabet Σ. A substring of S^i from jth to kth symbols is defined as $S^i_{j,k} = s^i_j s^i_{j+1} \ldots s^i_k$. A special type of a substring is a suffix of a sequence: $S^i_{j,\ell} = s^i_j s^i_{j+1} \ldots s^i_\ell$. The S can be seen as a matrix of n rows and ℓ columns. For simplicity of presentation it is convenient to define also a jth column of S as $S_j = s^1_j s^2_j \ldots s^n_j$. Finally, by $\mathrm{sort}_j(S)$ we denote the collection S lexicographically sorted according to the suffixes starting at $(j+1)$th symbol.

The *generalized positional Burrows–Wheeler transform* (gPBWT) changes S into P, where P is defined in the following way. The last column of P is equal to the last column of S, i.e. $P_\ell = S_\ell$. To obtain P_j we sort the sequences of S according to their suffixes starting from the $(j+1)$th symbol and pick the jth column, i.e. $P_j = (\mathrm{sort}_j(S))_j$.

The original PBWT by Durbin transforms bit vectors with the prefix-sorting order. We decided to present the gPBWT algorithm in the suffix-sorting order just to be more similar to the original BWT definition. It is worth to mention that both orderings for PBWT and gPBWT are equivalent in the sense that it is enough to reverse the input sequences to switch between the variants.

What is important from the performance point of view, the $\mathrm{sort}_j(S)$ can be easily obtained from $\mathrm{sort}_{j+1}(S)$. Moreover, in practice it suffices to store the ordering of indices of the S sequences and no coping of the complete sequences of S is made. The pseudocodes of algorithms for determination of the gPBWT and its reverse are presented in Figure 1. In the gPWBT construction, for the jth column, we calculate the histogram H of symbols at the jth column. Then we compute the cumulative statistics H^*, where $H^*[c]$ stores the information about the total number of symbols lexicographically smaller than c in the jth column. In the next step, we make use of the ordering of suffixes obtained when processing $(j+1)$th column and H^* to obtain the ordering according to the suffixes starting from the jth column. Finally, we permute the symbols of the jth column of S to obtain the jth symbol of P using the ordering according to suffixes of S starting from the $(j+1)$th symbol. The reverse gPBWT algorithm performs essentially the same steps, but some of them in the opposite order.

Let us focus now on the time and space complexity of the gPWBT computation. For each column we have to initialize array H and calculate array H^* which is made in $O(\sigma)$ time. Determination of the new ordering, as well as performing the permutation takes $O(n)$ time. Since there are ℓ columns, the total time complexity of both the forward as well as the reverse gPBWT algorithms is $O(\max(n, \sigma), \ell)$. In case of sufficiently large collections S, i.e. when the number of sequences n is not smaller than the alphabet size σ, the time complexity reduces to $O(n\ell)$, which is linear in terms of the total number of symbols in the input dataset S. Since the only data that must be stored in memory consists of the previous and current orderings and the statistics H and H^*, the space complexity of the algorithm is just $O(\max(n, \sigma))$, which for sufficiently large collections is just $O(n)$.

2.4 Novel compression algorithm

The compression algorithm we propose in the article is able to handle both FASTA and Stockholm files. FASTA files contain a single (aligned) protein family whereas Stockholm files can be seen as a

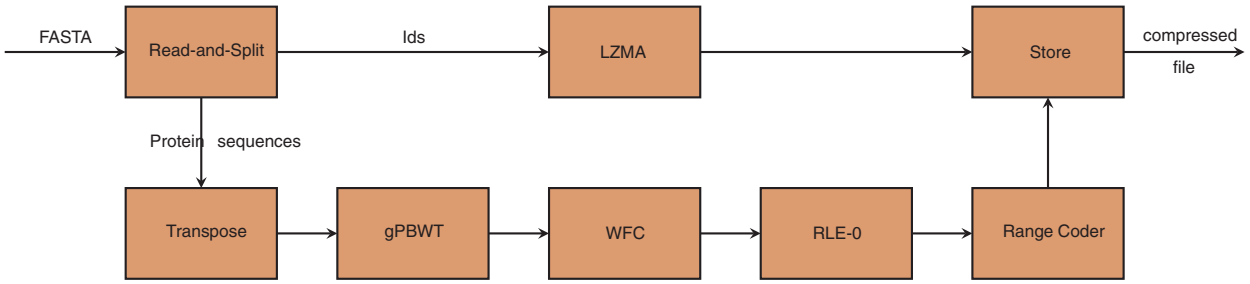

Fig. 2. Scheme of the proposed compression algorithm

concatenation of alignments of many protein families supplemented by some metadata. For simplicity we start with the description of the variant for FASTA files.

The general scheme of the proposed compression algorithm is presented in Figure 2, while the scheme of the decompression algorithm is given as Supplementary Figure S1. In the first stage, the input FASTA file is read into the memory (Read-and-Split block). The ids of the sequences are concatenated into a single string to be transferred to the LZMA block where they are compressed using the LZMA algorithm (used for example in the 7-zip programme). The raw protein sequences are treated as rows of a matrix S. They are transferred to the Transpose block for the transposition of the matrix, which allows to represent the columns of the matrix S as rows of the matrix S^{T}. The columns of the matrix S (now rows of S^{T}) are transferred to the gPBWT block from the last to the first one. In this block they are transformed by the gPBWT algorithm described in the previous subsection.

Each permuted row of S^{T} is then transferred separately to the WFC (*weighted frequency count* transform) block. This block implements the WFC (Deorowicz, 2002). The WFC transform changes the sequence of symbols over any alphabet into a sequence of integers. Roughly speaking, for each position of the sequence which is transformed, WFC predicts a rank for each alphabet symbol (the smaller the rank, the more frequently the alphabet symbol appeared in the former part of the sequence). Then the current symbol is replaced by its rank. The output of this block is a sequence of integers, that correspond to the symbols in the rows of the matrix S^{T}. The integers are usually small and majority of them (sometimes even up to 90%) are zeroes.

In the next stage the zero-run-length-encoding transform (RLE) (https://www.cs.auckland.ac.nz/~peter-f/FTPfiles/TechRep130.ps), RLE-0 block, replaces the repetitions of zeroes using a simple coding scheme: 0 is replaced by 0_a, $00 \rightarrow 0_b$, $000 \rightarrow 0_a0_a$, $0000 \rightarrow 0_a0_b$, $00000 \rightarrow 0_b0_a$, $000000 \rightarrow 0_b0_b$, $0000000 \rightarrow 0_a0_a0_a$, etc. This reduces the length of sequences noticeably.

In the next stage, each sequence is entropy coded. For this purpose we use a *range coder* (Salomon and Motta, 2010) (RangeCoder block). This coder assigns short codewords to frequent symbols and longer to the rare ones, which results in significant reduction of the space necessary to represent the input sequence. Since the frequency of symbols in the input of this block could differ by a few orders of magnitude (which causes some problems to entropy coders), we employ a simple modelling. For each symbol we initially encode a flag indicating whether it is 0_a, 0_b, 1, or something larger. In most cases, we are ready here, but when the symbol belongs to the last group (it is larger than 1), we proceed in the following way. We encode the group id of the symbol, where the available groups are: {2, 3}, $\{4,\ldots,7\}$, $\{8,\ldots,15\}$, $\{16,\ldots,31\}$, $\{32,\ldots,63\}$. Then we encode the value of the symbol withing a group. The maximal value of the integer (63) is determined by the number of allowed symbols in the input sequences, which include lower- and upper-case letters, and a few special values, i.e. '-', '.', '*'.

To improve the compression ratio even more, the symbols within a group are encoded in contexts, i.e. the probability of appearance of each symbol (necessary for range coder) is estimated taking into account a short-time history. For example, the context for the flags is defined by up to five recently encoded flags. The contexts for group ids and symbols inside groups are composed from up to three recently encoded group ids. Finally the compressed sequences (rows of S^{T} being columns of S) and the LZMA-compressed ids are collected in a single output file (Store block).

The compression of Stockholm files containing alignments of many protein families is similar. The file is processed in parts where each part contains a single family. At the beginning the family data are split into metadata and alignment data. The metadata are LZMA compressed while the alignment data are compressed using the algorithm for the FASTA files.

Our algorithm was implemented in the C++14 programming language. To make use of the multi-core architecture of modern CPUs, it is implemented using the C++ native threads. The main thread is responsible for Read-and-Split and Store stages, as well as it controls the execution. The stages Transpose, gPBWT, RLE-0 and RangeCoder are made by their own threads. As WFC is the most time consuming, it is executed by up to four separate threads (each of them independently processes some subset of columns). The software was compiled using GCC 6.2 with -O3 optimization enabled.

3 Results

The test platform was equipped with two Intel Xeon E5-2670 v3 CPUs (clocked at 2.3 GHz, 24 physical cores in total) and 128 GiB of RAM. In the experiments we limited the number of threads used by the compressors to four. As the test datasets we picked the default Pfam v. 31.0 collection of protein sequences (Finn *et al.*, 2016). The Stockholm file with the whole collection has size of 41.6 GB and contains 16 479 protein families. To examine the scalability of our software by the growing sizes of collections, we evaluated also the larger variants of Pfam database: Uniprot and NCBI. They are two and five times larger, respectively, than the default collection.

Currently, gzip is the most commonly used compressor for MSA data, so it was an obvious choice as the benchmark. Alas, gzip is a single-threaded application, so for a fair comparison we used its multi-threaded variant, i.e. pigz. To the best of our knowledge, there are no specialized compressors for protein MSA files, which are able to handle the Pfam datasets in the original form. The problem for existing FASTA compressors is the mix of lower- and upper-case letters in the Pfam files. Moreover, FASTA compressors are often designed just for the DNA alphabet. Therefore, we examined one

Table 1. Compression results for complete collections of Pfam database

Dataset	No. of Families	No. of sequences	File size	pigz			7-zip			CoMSA		
				size	c-time	d-time	size	c-time	d-time	size	c-time	d-time
Stockholm files												
Pfam-A	16 479	31 051 470	41.6	5.60	4723	**227**	2.38	6130	379	**1.75**	**2766**	874
Pfam-Uniprot	16 712	84 689 547	82.3	8.73	9504	**404**	3.30	13 968	632	**2.63**	**3600**	1904
Pfam-NCBI	16 712	177 952 603	227.7	26.93	**24 413**	**1185**	10.35	41 329	1811	**8.82**	28 486	4723
FASTA files												
Pfam-A	16 479	31 051 470	31.9	4.83	4495	**132**	2.13	9590	831	**1.55**	**1400**	1072
Pfam-Uniprot	16 712	84 689 547	81.2	8.49	9225	**459**	3.32	22 331	1233	**2.64**	**4173**	2249
Pfam-NCBI	16 712	177 952 603	179.4	19.46	22 962	**948**	6.55	48 163	1977	**4.97**	**7412**	4495

Notes: The top rows show the compression for a single Stockholm file for each collection. The bottom rows are for separate FASTA files (single file for each family) without metadata. File sizes are in GBs and times are in seconds. 'c-time' means compression time and 'd-time' means decompression time. Bold font means the best results.

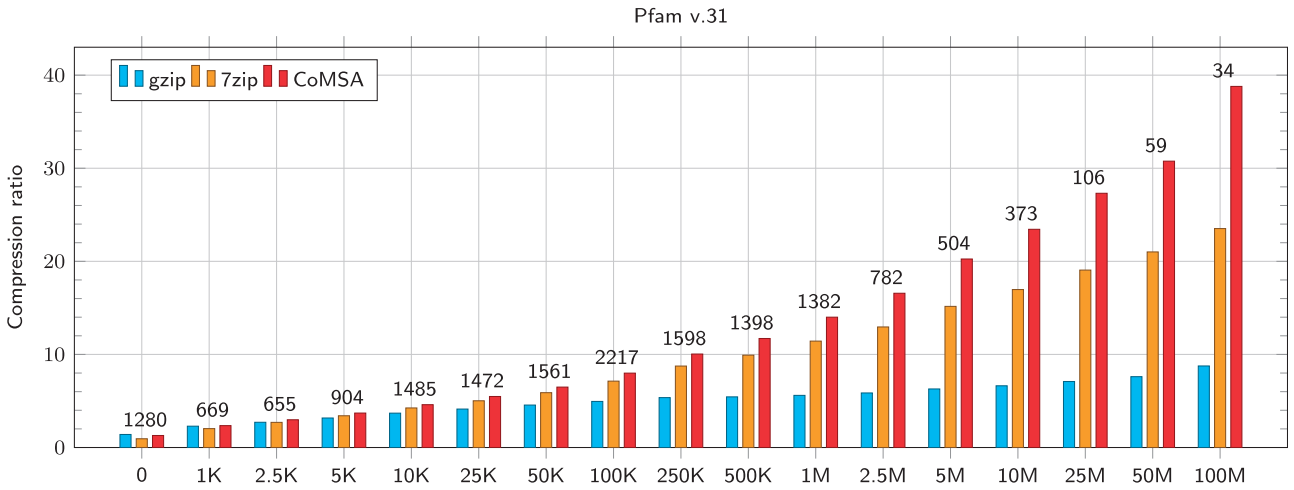

Fig. 3. Compression ratios for subsets of Pfam collection. The sets are of given size (in bytes), e.g. the 100 K set contains all files with sizes in range (100 KB; 250 KB). The number above the bars shows the number of families in a subset

more universal tool, namely 7-zip. It is quite often used when the better compression ratio is required, and the slower compression and decompression speeds could be accepted.

CoMSA was designed for both separate FASTA files (each file containing one protein family data) and collections stored in Stockholm files. Initially we evaluated the compression of the whole Stockholm files. The results given in Table 1 show that our compressor achieves the compression ratio (defined as the size of the original file divided by the size of the compressed file, so the higher the value, the better) about 24, which is 3.5 times better than the ratio offered by pigz. In real numbers, the default Pfam collection can be stored in as little as 1.74 GB (compared to 5.6 GB for pigz). CoMSA compression ratio is also 37% better than that of 7-zip. Similar observations can be made for Uniprot and NCBI variants of Pfam database.

Regarding compression times, CoMSA is significantly faster than pigz and 7-zip in compression but slower in decompression. In real numbers, we were able to compress the 42 GB Pfam database in less than 45 min and decompress it in <15 min, which should be acceptable for typical scenarios.

The evaluated compressors differ significantly in their architecture, so it would be interesting to take a closer look at how they perform for protein families of different size. Therefore, we extracted all protein families and stored them in separate FASTA files. The summary of sizes and (de)compression times is given in Table 1. More details can be found in Figure 3. We grouped the FASTA files according to their sizes into disjoint subsets. The smallest size for each subset is presented at the horizontal axis, e.g. the value of 100 K means that the bars above refer to files of sizes between 100 (including) and 250 KB. The height of a single bar represents the average compression ratio obtained for a given subset, while the number above the bar reflects cardinality of the subset. The largest file in this experiment was slightly smaller than 1 GB. As one can see, for all size ranges (except for the smallest one containing files below 1 KB) the best compression ratio was obtained by CoMSA. More importantly, the ratio rapidly increases with the size of the file. A similar trend is observed for pigz and 7-zip, but their compression-ratio growth is slower.

The FASTA file size is only one of possible indicators related to MSA. We investigated also the influence of the number of sequences and the alignment length. Figure 4 shows the compression ratios for all families included in Pfam-A v. 31.0. Each subplot depicts 16 479 points, where each point represents a compression ratio for a single family. The left subplots show the advantage of CoMSA over 7-zip (defined as the size of 7-zip-compressed file divided by the size of CoMSA-compressed file), while the right subplots show the advantage of CoMSA over pigz. Since the number of points in the plots is

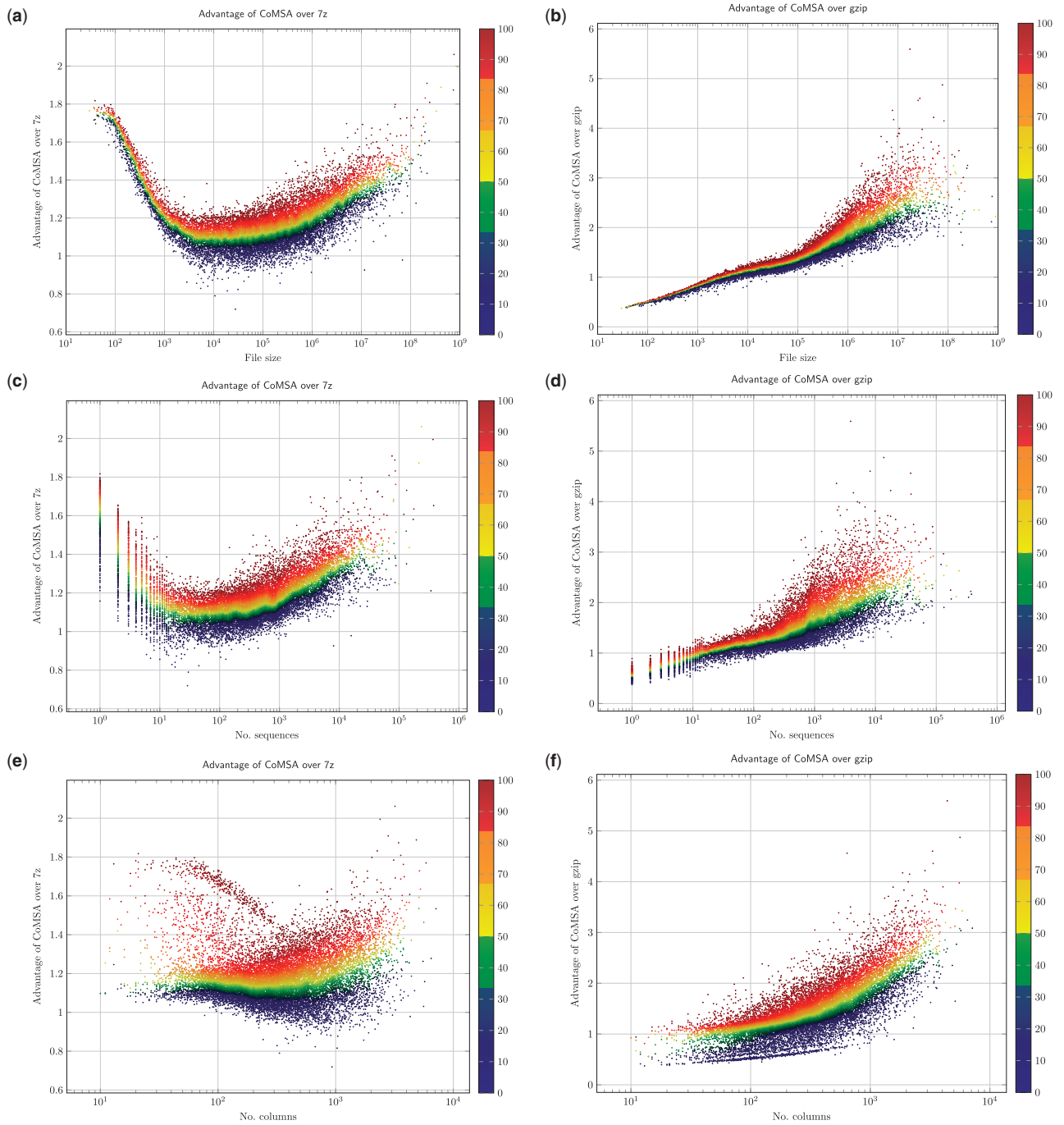

Fig. 4. Comparison of the advantage of CoMSA over 7-zip (left column) and pigz (right column) compressors. The analyses are for various: file sizes (subfigures **a** and **b**), number of sequences (subfigures **c** and **d**) and alignment length (subfigures **e** and **f**). Each point represents one of 16 479 files. The point colours represent the percentile in the neighbourhood (defined as 10% difference in the size, sequence number and alignment length, respectively). File sizes are in bytes

large, for clarity we assigned a colour to each of them. The colour represents the rank of the advantage (over 7-zip or pigz) in group of files of comparable size—subfigures a and b (or comparable number of sequences—subfigures c and d; or comparable alignment length—subfigures e and f). By comparable we mean 10% neighbourhood. For example, to assign a colour to some file of size 1 MB (Fig. 4a) we investigated all files up to 10% smaller and 10% larger. For each file in this group, we calculated advantage of CoMSA over 7-zip. Then we ranked the results and calculated the percentile. The colour was assigned according to the percentile as shown in the bar on the right side of each plot.

As one can observe, all medians (yellow colour) are always above 1.0 for 7-zip and there is a growing tendency in the advantage for very small and very large files. Similar observations can be made, when the size of the family is defined as the number of sequences (Fig. 4c). However, for growing alignment length (Fig. 4e) the trend is less clear. The maximal advantage over pigz is much larger in real numbers (up to 7.5, compared to 2.1 for 7-zip). For tiny files (smaller than

Table 2. Compression results for modified FASTA files (upper-case symbols) from the default Pfam collections

File size	pigz	MFCompress	7-zip	CoMSA
41.6	4.66	4.55	2.11	1.55

Note: All sizes are in GBs.

300 bytes), the median of the advantage is below 1.0, which means that pigz performs better. Nevertheless, compressing of such small files is considered to be useless, as the gains are negligible.

As the workflow in our compressor comprises a few separate stages, we investigated the impact of these stages as well as their combinations on compression ratios. The detailed results are presented in Supplementary Table S2. The results show that although individual stages alone, especially Transpose and gPBWT, allow noticeable compression, only the set of all four stages gives a significant reduction in the file sizes.

For completeness, we tested not only arbitrary chosen compression modes, but also different levels of compression offered by pigz as well as 7-zip. The detailed results are given in the Supplementary Table S1. The fastest mode of pigz runs almost 20 times faster than the reference-9 mode, nevertheless it offers the compression ratio that is less by 30.7% than that of the reference mode. On the other hand, in its best compression mode (-11 option) pigz runs almost 27 times slower than in the reference mode, but the gain in compression ratio is only 8.8%. The differences in decompression times are almost negligible. In case of 7-zip the reference mode is -mx9 mode. The turn into the fastest mode accelerates the compression 13 times, but the decompression slows down 1.6 times. However, the more important is the decrease in compression ratio, which is reduced of 49%.

It is also noteworthy to say that the largest single protein family from Pfam-NCBI (PF07690.15) consumes about 4.36 GB of space. CoMSA was able to reduce this to 81.7 MB (150.2 MB for 7-zip and 543.5MB for pigz). The CoMSA compression and decompression times were 120 and 95 s, respectively (791 and 33 s for 7-zip; 778 and 13 s for pigz). As the modern algorithms for MSA determination, like MAFFT (Katoh and Standley, 2013), Clustal Omega (Sievers *et al.*, 2011), PASTA (Mirarab *et al.*, 2015), FAMSA (Deorowicz *et al.*, 2016), to name a few, are able to process families containing more than 100 000 sequences in a few hours (or even less) at modern workstations, and the resulting alignments sometimes consume tens of GB, the ability of CoMSA to significantly reduce the necessary space in a very-short time is remarkable.

CoMSA is also able to compress several Stockholm files into a single archive. Each single family is compressed separately, therefore the size of the archive is equivalent to the accumulated size of separately compressed Stockholm files. The difference of a few bytes can appear due to encoding some header/footer information in each file.

Finally, CoMSA gives a possibility to decompress a single protein family from a compressed Stockholm file. To this end the compressed archive is composed of a chunk, each containing single family data. In the file footer, the position in the compressed file is stored together with the family id. Thus, to extract a single family we just browse the small description of archive contents to localize the requested family and then decompress this family. The decompression time in this case is approximately the same as decompression time of the Stockholm file containing just this single family. On average it is <0.1 s for the default Pfam file. Extraction from a gzip file is impossible and to allow it for 7-zip we have to modify the input file. Namely, we split the Stockholm file into 16 479 small Stockholm files (each containing a single family). Then, we compressed all of them into a single archive. For such compressed archive it is possible to extract the requested file by its name. Unfortunately, the extraction time strongly depends on the position of the requested file in the archive and varied between a fraction of a second and 35 s.

In the final test we converted all lower-case symbols in FASTA files to the upper-case form to enable the comparison with MFCompress (Pinho and Pratas, 2014), which is probably the best FASTA compressor able to handle not only DNA alphabet already published. For comparison we run also CoMSA, pigz and 7-zip on this modified dataset. The results given in Table 2 show that also in this test CoMSA is clearly the leader.

4 Discussion

The Pfam database is a growing collection of protein families. The raw data for its default variant occupy about 42 GB, which causes problems with both storage and data transfer. Currently, the most popular method to reduce the file sizes is the commonly known, universal compressor—gzip programme. When applied to the basic Pfam dataset it allows to reduce the space to about 5.6 GB. From the practical point of view, application of gzip still leads to important gains, but we can be almost sure that the size of the discussed collection will be growing in the near future. Even now, the largest variant of Pfam v. 31.0 consumes ∼230 GB for raw data and ∼30 GB, when gzip-compressed.

In this article, we proposed a novel compression algorithm for MSAs files. The algorithm works in a few stages. A generalization of the positional Burrows–Wheeler transform (PBWT) for non-binary alphabets is performed as the first stage. Then, transforms adopted from the literature are determined, namely weighted frequency counter transform (WFC) followed by RLE-0. Finally, the modelling stage for the entropy coder is executed.

The experiments made for three variants of the largest existing protein family database, Pfam, showed that CoMSA offers on average more than three times better compression ratio than pigz (i.e. parallel gzip) for the whole collection of families. The advantage over 7-zip is smaller, but still significant. What is important, CoMSA advantage over pigz and 7-zip grows for growing family size, which is promising, as in the future the protein families will be definitely larger. Even today, it seems that the very good compression ratio and good compression and decompression speeds allow to consider CoMSA as a replacement for gzip for those that suffers from huge size of MSA collections.

Acknowledgements

The work was supported by Polish National Science Centre under the project DEC-2015/17/B/ST6/01890 and performed using the infrastructure supported by POIG.02.03.01-24-099/13 grant: 'GeCONiI—Upper Silesian Center for Computational Science and Engineering'.

Conflict of Interest: none declared.

References

Bentley,J.L. *et al.* (1986) A locally adaptive data compression scheme. *Commun. ACM*, **29**, 320–330.

Bonfield,J.K. and Mahoney,M.V. (2013) Compression of FASTQ and SAM format sequencing data. *PLoS One*, **8**, e59190.

Cleary,J. and Witten,I. (1984) Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.*, **32**, 396–402.

Daniels,N.M. *et al*. (2013) Compressive genomics for protein databases. *Bioinformatics*, **29**, i283–i290.

Deorowicz,S. (2002) Second step algorithms in the Burrows–Wheeler compression algorithm. *Software Pract. Exper.*, **32**, 99–111.

Deorowicz,S. *et al*. (2016) FAMSA: fast and accurate miltiple sequence alignment of huge protein families. *Sci. Rep.*, **6**, 33964.

Deorowicz,S. *et al*. (2013) Genome compression: a novel approach for large collections. *Bioinformatics*, **29**, 2572–2578.

Deorowicz,S. and Grabowski,S. (2013) Data compression for sequencing data. *Algorithms Mol. Biol.*, **8**, 25.

Durbin,R. (2014) Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT). *Bioinformatics*, **30**, 1266–1272.

Fenwick,P. (1996) The Burrows–Wheeler transform for block sorting text compression: principles and Improvements. *Comput. J.*, **39**, 731–740.

Finn,R.D. *et al*. (2016) The Pfam protein families database: towards a more sustainable future. *Nucleic Acids Res.*, **44**, D279–D285.

Hanus,P. *et al*. (2010) Compression of Whole Genome Alignments. *IEEE Trans. Inf. Theory*, **56**, 696–705.

Hosseini,M. *et al*. (2016) A Survey on Data Compression Methods for Biological Sequences. *Information*, **7**, 56.

Huffman,D. (1952) A method for the construction of minimum-redundancy codes. *Proc. IRE*, **40**, 1098–1101.

Katoh,K. and Standley,D.M. (2013) MAFFT multiple sequence alignment software version 7: improvements in performance and usability. *Mol. Biol. Evol.*, **30**, 772–780.

Li,H. (2016) BGT: efficient and flexible genotype query across many samples. *Bioinformatics*, **32**, 590–592.

Matos,L. *et al*. (2015) MAFCO: a compression tool for MAF files. *PLoS One*, **10**, e0116082.

Mirarab,S. *et al*. (2015) PASTA: ultra-large multiple sequence alignment for nucleotide and amino-acid sequences. *J. Comput. Biol.*, **22**, 377–386.

Numanagić,I. *et al*. (2016) Comparison of high-throughput sequencing data compression tools. *Nat. Methods*, **13**, 1005–1008.

Pinho,A.J. and Pratas,D. (2014) MFCompress: a compression tool for FASTA and multi-FASTA data. *Bioinformatics*, **30**, 117–118.

Ravanmehr,V. *et al*. (2018) ChIPWig: a random access-enabling lossless and lossy compression method for ChIP-seq data. *Bioinformatics*, **34**, 911–919.

Roguski,L. and Deorowicz,S. (2014) DSRC 2: industry-oriented compression of FASTQ files. *Bioinformatics*, **30**, 2213–2215.

Salomon,D. and Motta,G. (2010) *Handbook of Data Compression*. Springer, London.

Sievers,F. *et al*. (2011) Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega. *Mol. Syst. Biol.*, **7**, 539.

Stephens,Z.D. *et al*. (2015) Big Data: astronomical or Genomical? *PLoS Biol.*, **13**, e1002195.

Wang,Z. *et al*. (2016) smallWig: parallel compression of RNA-seq WIG files. *Bioinformatics*, **32**, 173–180.

Storer,J.A. and Szymanski,T.G. (1982) Data compression via textual substitution. *J. ACM*, **29**, 928–951.

Ziv,J. and Lempel,A. (1977) A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, **23**, 337–343.