Bioinformatics, 32(2), 2016, 173–180
doi: 10.1093/bioinformatics/btv561
Advance Access Publication Date: 30 September 2015
Original Paper

OXFORD

Sequence analysis

smallWig: parallel compression of RNA-seq WIG files

Zhiying Wang[1,*], Tsachy Weissman[1] and Olgica Milenkovic[2]

[1]Department of Electrical Engineering, Stanford University, Stanford, CA 94305, USA and [2]Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

*To whom correspondence should be addressed.

Associate Editor: Ivo Hofacker

Received on November 27, 2014; revised on September 4, 2015; accepted on September 23, 2015

Abstract

Contributions: We developed a new lossless compression method for WIG data, named smallWig, offering the best known compression rates for RNA-seq data and featuring random access functionalities that enable visualization, summary statistics analysis and fast queries from the compressed files. Our approach results in order of magnitude improvements compared with bigWig and ensures compression rates only a fraction of those produced by cWig. The key features of the smallWig algorithm are statistical data analysis and a combination of source coding methods that ensure high flexibility and make the algorithm suitable for different applications. Furthermore, for general-purpose file compression, the compression rate of smallWig approaches the empirical entropy of the tested WIG data. For compression with random query features, smallWig uses a simple block-based compression scheme that introduces only a minor overhead in the compression rate. For archival or storage space-sensitive applications, the method relies on context mixing techniques that lead to further improvements of the compression rate. Implementations of smallWig can be executed in parallel on different sets of chromosomes using multiple processors, thereby enabling desirable scaling for future transcriptome Big Data platforms.

Motivation: The development of next-generation sequencing technologies has led to a dramatic decrease in the cost of DNA/RNA sequencing and expression profiling. RNA-seq has emerged as an important and inexpensive technology that provides information about whole transcriptomes of various species and organisms, as well as different organs and cellular communities. The vast volume of data generated by RNA-seq experiments has significantly increased data storage costs and communication bandwidth requirements. Current compression tools for RNA-seq data such as bigWig and cWig either use general-purpose compressors (gzip) or suboptimal compression schemes that leave significant room for improvement. To substantiate this claim, we performed a statistical analysis of expression data in different transform domains and developed accompanying entropy coding methods that bridge the gap between theoretical and practical WIG file compression rates.

Results: We tested different variants of the smallWig compression algorithm on a number of integer- and real- (floating point) valued RNA-seq WIG files generated by the ENCODE project. The results reveal that, on average, smallWig offers 18-fold compression rate improvements, up to 2.5-fold compression time improvements, and 1.5-fold decompression time improvements when compared with bigWig. On the tested files, the memory usage of the algorithm never exceeded 90 KB. When more elaborate context mixing compressors were used within smallWig, the obtained compression rates were as much as 23 times better than those of bigWig. For smallWig used in the random query mode, which also supports retrieval of the summary statistics, an overhead in the compression rate of roughly 3–17% was

introduced depending on the chosen system parameters. An increase in encoding and decoding time of 30% and 55% represents an additional performance loss caused by enabling random data access. We also implemented smallWig using multi-processor programming. This parallelization feature decreases the encoding delay 2–3.4 times compared with that of a single-processor implementation, with the number of processors used ranging from 2 to 8; in the same parameter regime, the decoding delay decreased 2–5.2 times.

Availability and implementation: The smallWig software can be downloaded from: http://stanford.edu/~zhiyingw/smallWig/smallwig.html, http://publish.illinois.edu/milenkovic/, http://web.stanford.edu/~tsachy/.

Contact: zhiyingw@stanford.edu

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

1 Introduction

Next-generation sequencing technologies have resulted in a dramatic decrease of genomic data sequencing time and cost. As an illustrative example, the HiSeq X machines introduced by Illumina in 2014 enable whole human genome sequencing in less than 15 h and at a cost of only $1000 (http://www.illumina.com/systems/hiseq-x-sequencing-system.ilmn). A suite of other-seq techniques has closely followed this development (for a comprehensive overview, see http://res.illumina.com/documents/products/research_reviews/), including the by now well-documented RNA-seq method. RNA-seq is a shotgun sequencing technique for whole transcriptomes (Marioni *et al.* 2008) used for quantitative and functional genomic studies. In addition to generating sequence-related information, RNA-seq methods also provide dynamic information about gene or functional RNA activities as measured by their expression (abundance) values. This makes RNA-seq techniques indispensable for applications such as mutation discovery, fusion transcript detection and genomic medicine (Wang *et al.* 2009). As a result, the volume of data produced by RNA-seq methods can be foreseen to increase at a much faster rate than Moore's law. It is therefore imperative to develop highly efficient lossless compression methods for RNA-seq data.

The problem of DNA and RNA sequence and expression compression has received much attention in the bioinformatics community. Compression methods for whole genomes include direct sequence compression (e.g. Cao *et al.* 2007; Pinho *et al.* 2011; Tabus and Korodi 2008) and reference-based compression schemes (e.g. Kuruppu *et al.* 2011; Pinho *et al.* 2012; Wang and Zhang 2011). The former class of methods explores properties of genomic sequences such as small alphabet size and large number of repeats. The latter techniques use previously sequenced genomes as references with which to compare the target genome or sequencing reads, leading to dramatic reductions in compressed file sizes. Related similarity-discovery-based schemes are usually applied to a large collection of genomes and they achieve very small per genome compression rates (e.g. Deorowicz *et al.* 2013). Moreover, recent work also includes the compressive genomics paradigm, which allows for direct computation and alignment on compressed data (Loh *et al.* 2012). The aforementioned methods and some information-theoretic techniques to biological data compression were reviewed in (Vinga 2013).

For every base pair in the genome, an RNA-seq WIG file contains an integer or floating-point expression value. Human transcriptome WIG files may contain hundreds of millions of expression values, which amounts to GB of storage space (e.g. one of the subsequently analyzed WIG files randomly chosen from the ENCODE (Encode Project Consortium 2004) project has a size of 5 GB). WIG files are usually compressed by bigWig (Kent *et al.* 2010), which basically performs gzip compression on straightforwardly preprocessed data. Unfortunately, the bigWig format does not appear to offer significant data volume reductions and about 10% of the tracks from the UCSC ENCODE hg19 browser in bigWig format take up 31% in storage space (Hoang and Sung 2014). Recently, another compression suite, termed cWig (Hoang and Sung 2014), was implemented as an alternative to bigWig. The cWig method outperforms bigWig in terms of compression rate, and random query time, although it still relies on suboptimal compression techniques such as Elias delta and gamma coding (Salomon 2007).

This work focuses on transform and arithmetic compression methods for expression data in the WIG format. Since WIG files capture expressions of correlated RNA sequence blocks, modeling these values as independent and identically distributed random variables is inadequate for the purpose of compression. Hence, we first perform a statistical analysis of expression values to explore their dependencies/correlations and then proceed to devise a new suite of compression algorithms for WIG files. Since the WIG format is not limited to RNA-seq data, our compression methods are also suitable for other types of dense data, or quantitative measurements, such as GC content values, probability scores, proteomic measurements and metabolomic information. The main *analytic and algorithmic contributions* of our work are as follows:

i. Devising a new combination of run length and delta encoding that allows for representing the expression data in highly compact form. As part of this procedure, we identified runs of locations with the same expression value and then computed the differences of adjacent run values. The resulting transformed sequences are referred to as *run difference sequences* and specialized statistical analysis of difference sequences constitutes an important step towards identifying near-optimal compression strategies.
ii. Analyzing the probability distributions of the difference sequences and inferring mixture Markov models for the data. As part of this step, we estimated information-theoretic quantities, such as the (conditional) entropy, to guide us in our design and evaluation process. More precisely, we first fitted power-law distributions to the empirical probability distributions of the difference sequences. Second, we showed that strong correlations exist between adjacent run differences, while there exists only a relatively small correlation between the sequences of run length differences and that of the corresponding run expression differences. These findings provide a strong basis for performing separate compression of the run length and the expression information.
iii. Developing arithmetic encoders for compression of the difference sequences, including options such as basic arithmetic coding and context-mixing coding based on the work in Mahoney (2002). In this step, we were guided by the results of the

statistical analysis and performed alphabet size reduction in the difference sequences and subsequent run length and run expression compression. With this step, we were able to achieve 17-fold improvements in the compression rate when compared with bigWig: as an illustration, a typical WIG file of size 5 GB was compressed to roughly 64–69 MB, depending on the user-defined operational mode; in comparison, traditional gzip and the bigWig (Kent *et al.* 2010) compressors produced files of sizes 1.1 GB and 1.2 GB, respectively.

Our new compression algorithm follows the standard requirements for expression data representation/visualization by allowing random access features via data blocking and separate block compression. It also encodes data summary statistics, akin to bigWig data formats. Furthermore, smallWig has two implementation modes, one of which runs on a single processor and another which uses multiple processors in parallel. The parallelized version of the algorithm offers significant savings in computational time, with identical rate performance as the serial version.

The remainder of the article is organized as follows. Section 2 provides the idea behind our sequence transformations and coding methods. Section 3 contains our statistical analysis. A detailed description of the smallWig algorithm is provided in Section 4. Compression results and a comparative study of compression methods is given in Section 5. A discussion of our findings and concluding remarks are given in Sections 6 and 7, respectively.

2 Methods: sequence transformations

We start our analysis by introducing WIG data transforms that allow for efficient run length encoding and by explaining how to use difference values in subsequent compression steps. To illustrate some of the concepts behind our analysis, we make use of a WIG file from the ENCODE project (Encode Project Consortium 2004) pertaining to RNASeq cell line GM12878, for which the RNA fraction is Long PolyA+ and the compartment is Nucleus.

Throughout, we use capital letters to represent sequences, and lower case letters to represent elements in sequences. We write $[a] = \{1, 2, \ldots, a\}$, for any positive integer $a \in \mathbb{N}^+$, and $[a, b] = \{a, a+1, \ldots, b\}$, for two positive integers $a \leq b$.

The WIG files of interest comprise two sequences:

- *The Location Sequence* $A = (a_1, a_2, \ldots, a_M)$, where M denotes the length of the sequence. Sequence A contains chromosomal positions (or *locations*) satisfying $a_i \in \mathbb{N}^+$ for all $i \in [M]$, and $a_i < a_{i+1}$ for all $i \in [M-1]$. This sequence typically contains consecutive indices of the base pairs, for which $a_{i+1} = a_i + 1$, except for skipped locations with expression value equal to zero, for which $a_{i+1} > a_i + 1$.
- *The Expression Sequence* $B = (b_1, b_2, \ldots, b_M)$. The sequence B contains expression values $b_i \in \mathbb{R}$. The b_i's indicate the number of RNA transcripts that include location a_i. Note that the sequences A and B have the same length.

A *run* is defined as a sequence of consecutive locations with identical expression value. The number of locations in the run is called the *run length*, and the corresponding expression value is called the *run expression*. Note that if for some integers $i \leq j$, one has $a_{t+1} = a_t + 1$ for all $t \in [i, j-1]$, and $b_i = b_{i+1} = \ldots = b_j$, then the sequence corresponds to a run of length $j - i + 1$ with run expression equal to b_i. On the other hand, if for some integer i there exist skipped locations, i.e. locations for which $a_{i+1} > a_i + 1$, and $b_i, b_{i+1} \neq 0$, then the run is of length $a_{i+1} - a_i - 1$ with corresponding run expression value equal to 0. Thus, there is a 1-1 mapping from the sequences A and B to the run length and run expression sequences described below:

- *The Run length Sequence* $C = (c_1, c_2, \ldots, c_N)$, $c_i \in \mathbb{N}^+, i \in [N]$, a sequence of run lengths that describes the runs of consecutive locations with identical expression values. Alternatively, one may define the sequence by stating that for locations confined to $[\sum_{t=1}^{i} c_t + 1, \sum_{t=1}^{i+1} c_t]$, the expression values are identical. Here, N denotes the number of runs. If N_0 denotes the number of skipped runs of value 0 in a WIG file, then $\sum_{i=1}^{N} c_i = M + N_0$.
- *The Run Expression Sequence* $D = (d_1, d_2, \ldots, d_N)$, $d_i \in \mathbb{R}$, $i \in [N]$, a sequence of expression values that corresponds to the runs. More precisely, the sequence specifies that the ith run has expression value d_i, for all $i \in [N]$.

For our running example, the original WIG sequences were of length $N = 3.2e+8$, while the run and difference sequences were of length $M = 3.9e+7$. Note that a similar transform is used in the bigBed format, which provides a more succinct representation of sparse WIG data (we will revisit the Bed format in the results section). Since adjacent runs tend to have similar lengths and expressions, the differences between consecutive runs may lead to further compaction of WIG information. To describe the difference sequences, let $c_0 = 0, d_0 = 0$.

- *The Run length Difference Sequence*, $X = (x_1, x_2, \ldots, x_N)$, is defined by $x_i = c_i - c_{i-1}$, so that $x_i \in \mathbb{Z}$ for all $i \in [N]$.
- *Run Expression Difference Sequence*, $Y = (y_1, y_2, \ldots, y_N)$, is defined by $y_i = d_i - d_{i-1}$, so that $y_i \in \mathbb{R}$ for all $i \in [N]$.

Here, we also point out that a related run length transformation has been investigated in BedGraph format, while the idea of run length and difference-value transformations has been studied in (Hoang and Sung 2014). In the latter work, the authors also demonstrated the potential benefit of using these transformations on WIG data.

3 Methods: statistical data analysis

In what follows, we describe how to fit empirical probability mass functions and compute empirical entropies for the run length and run expression difference sequences X, Y. Moreover, we study higher order dependencies of adjacent elements within the sequences X and Y, as well as dependencies between the sequences X and Y.

3.1 Fitting empirical probability mass functions

We computed the empirical frequencies for the run expression and run length difference sequences. Both sequences roughly follow a power-law distribution, with probability density function

$$p(x) \simeq \frac{\alpha - 1}{\beta} \left(\frac{x}{\beta}\right)^{-\alpha},$$

which can consequently be used to approximate the empirical probability mass function [similar to the method in Pavlichin *et al.* (2013)]. Therefore, we can parameterize the two empirical distributions with only two parameters, α and β. Goodness of fit may be estimated via the standard Kolmogorov-Smirnov statistics or some other means. For a more detailed analysis of the empirical probability distributions, the reader is referred to the Supplementary Material.

3.2 Empirical entropy computation

Next, we propose three correlation models for the difference sequences X, Y and estimate the entropy of their underlying distributions.

A detailed description of these models may be found in the Supplementary Material.

Let Z, W be discrete random variables with alphabet $\mathcal{Z}, \mathcal{W}$, respectively. The Shannon entropy of Z is defined as

$$H(Z) = -\sum_{z\in\mathcal{Z}} P_Z(z)\log_2 P_Z(z).$$

Similarly, the conditional entropy of Z given W is defined as

$$H(Z|W) = -\sum_{w\in\mathcal{W}} P_W(w)\sum_{z\in\mathcal{Z}} P_{Z|W}(z|w)\log_2 P_{Z|W}(z|w).$$

In what follows, we assume that X, Y are two random sequences of length N of the form $(X_1, X_2, \ldots, X_N)$ and $(Y_1, Y_2, \ldots, Y_N)$.

Independent run difference model. Assume that the sequences X, Y are independent and that the elements of X (Y) are independent and identically distributed. The entropy in this case reads as $H_I = H(X_1) + H(Y_1)$.

Markov run difference model. Again we assume that the sequences X, Y are independent, but let X (Y) be a first order Markov sequence. We write the entropy under this model as $H_M = H(X_2|X_1) + H(Y_2|Y_1)$.

Paired sequence model. We assume the sequence of pairs $((X_1, Y_1), (X_2, Y_2), \ldots, (X_N, Y_N))$ is a first order Markov sequence. Thus, the entropy formula reads as $H_P = H(X_2, Y_2|X_1, Y_1)$.

In the Supplementary Material, we list the entropies of all the aforementioned models and for 14 Wig and 10 different BedGraph files, computed using the information theoretic tools described in (Jiao *et al.* 2015). As one may see, the entropy is 19–56% smaller for the Markov model than the independent run difference model. In most cases, this reduction in entropy may be attributed to dependencies in the run length differences. In other words, run length values are more likely to be affected by adjacent run length values. On the other hand, considering dependencies between the run length differences and run expression differences only reduces the entropy by about 5–16%. As a result, the most effective compression strategy appears to be separate compression of the difference sequences X and Y.

Because of the large variations in the run expression and run length difference values, computing and storing all conditional probabilities (about 10^{10} such entries) under the Markov model requires very large memory. Hence, we first focus on a compression algorithm for the independent model and then discuss our generalized compression scheme based on context mixing, which requires specialized means for overcoming the memory overflow problem.

4 Compression algorithms

We start by describing our basic compression algorithms based on arithmetic coding and then show how to enable random queries within the given algorithmic coding framework. In addition, we describe how to reduce the compressed file sizes even further via context-mixing methods. We conclude this section by introducing parallelization techniques for the proposed algorithms. Diagrams of our compression and decompression architectures are given in the Supplementary Material.

To compress the RNA-seq expressions, we used two individual arithmetic encoders and decoders for the difference sequences X, Y defined in Section 2. We observe that since expression values can be real valued, any errors in computing the expression differences may cause error propagation during decompression. As a result, the expression difference alphabet has to be stored as well, with a precision large enough to allow for correct decompression.

Arithmetic compression (Rissanen and Langdon 1979) is an entropy coding method that converts the entire input sequence into a range of values (interval) determined by its cumulative frequency. On length-n sequences in $\mathcal{Z}^n$, one defines a total order $W \prec Z$ by requiring that W precedes Z lexicographically. For a sequence Z, its code word is the binary representation of a real number between $\sum_{W:W\prec Z} P(W)$ and $\sum_{W:W\prec Z} P(W) + P(Z)$. For sequences with independent and identically distributed entries, arithmetic coding is an entropy-approaching compression scheme, given that the distribution of the sequence is known.

To ensure small computational complexity, we used arithmetic compression algorithms with range encoding (Martin 1979) and some techniques from the package rangemapper by Polar (http://ezcodesample.com/reanatomy.html?Source=To+article+and+source+code). Our implementation is similar to the original version of arithmetic coding, except that the underlying probabilities are represented with binary sequences of fixed length, which allows for more efficient computations. Moreover, encoding/decoding may be performed in a streaming fashion. A buffer is used to store the "unresolved range" depending on the yet unobserved part of the sequence. In our implementation, the precision of the buffer was limited to 30 bits so as to control the number of operations performed. Unlike range encoding, in which the calculations are performed base 256, we used base 2 so as to achieve the best compression rate.

To facilitate random queries during decompression, we divided the difference sequences into blocks of fixed length. The length—subsequently termed *block size*—can be chosen by the user, to allow for desired trade-offs between compression rate and query time. Since the compressed sequences have lengths that vary from block to block, we also store the address of each block. Moreover, to quickly obtain the original sequences from the difference sequences, we also store the (start location, expression, run length) triple (a_i, b_i, c_i) for every starting element in a block. For this purpose, we implemented a simple binary search procedure originating from the start location to identify the blocks corresponding to a random query.

For fast visualization of the WIG data and summary statistics analysis, we stored an additional *summary vector* for every block. The summary vector contains the following six values: (i) the minimum expression value in the block; (ii) the maximum expression value in the block; (iii) the mean value of the block; (iv) the standard deviation of the block; (v) the number of locations covered in the block and (vi) the total number of locations within the block. If a random region is queried, the aggregated summary vector for the queried region is computed as follows. First, all blocks that are completely included in the queried region are identified, and their summary information is computed from the summary vectors of the blocks; then, the starting and ending blocks partially contained within the query region are retrieved and their summary information is computed directly from the symbols in the blocks. The complexity of the statistics query is linear in the number of queried blocks and in the block size.

To explore dependencies among elements in the run sequences, we used the context-mixing algorithm implemented as part of the lpaq1 package (Mahoney 2002). To illustrate the idea behind context mixing, we focus on context-tree weighting (Willems *et al.* 1995). In this model, we assume that every element $x_t \in \{0, 1\}$ in a sequence is generated based on a suffix set $\mathcal{S}$, which can be represented by a degree-two tree of depth not more than D. Here, D is a parameter that indicates the dependency between symbols at a

certain distance in the sequence and consequently determines the memory requirements of the algorithm. The root of the tree is indexed by the empty string, while the left (or right) edge of every node represents a 1 (or 0), and every node corresponds to the string associated with its path to the root. Each suffix/leaf s is associated with a parameter θ_s, which equals the probability of the next source symbol x_{t+1} being equal to 1 conditioned on the suffix of the semi-infinite sequence $\ldots x_{t-2}x_{t-1}x_t$ beings. Let $\theta_s(x_1^t) = P(x_{t+1} = 1 | x_{t-|s|+1}^t = s)$ denotes the conditional probability of a symbol given the preceding $|s|$ symbols being equal to s, where we denote by x_i^j the string $x_i, x_{i+1}, \ldots, x_j$, for $i \leq j$.

The probability θ_s (called the *model parameter*) is usually not known but can be estimated using the Krichevsky–Trofimov (KT) method (Krichevsky and Trofimov 1981). Moreover, the actual tree generating the sequence (called the *model*) is also unknown. The context-tree weighting algorithm takes a weighted sum over all tree models with depth not exceeding D. The redundancy introduced by the lack of knowledge of both the parameter and the model is bounded, and context-tree weighting is optimal since it achieves the lower bound of redundancy derived in Rissanen (1984). More details regarding this method are provided in the Supplementary Material.

The context-mixing algorithm lpaq1 (Mahoney 2002) that we used in our implementation predicts the next bit based on the previous six bytes, as well as the last matching context. Hash tables are built to store the history and the context. During compression, the algorithm adaptively updates the probability distribution of the next bit based on its current prediction and uses arithmetic coding with time varying probability values. Since adaptive schemes perform poorly for short sequence lengths, the context-mixing scheme is only recommended in the one-block compression mode which does not allow random query. As a result, context-mixing compression should be used for archival storage.

To speed up compression/decompression, we also implemented a parallel scheme for arithmetic coders with random query. The scheme partitions the original sequences based on its chromosome index and compresses each substring on a separate processor. Details of this implementation and its performance are discussed in the Results Section.

5 Results

We tested our compression algorithm on 14 integer-valued WIG files with sizes ranging from 1.5 to 5.3 GB and on 10 integer and real-valued BedGraph files. All Wig files contain human transcriptome RNA-seq data from the ENCODE hg19 browser. Since smallWig is designed for WIG files, here we mainly focus on the 14-file set. A more detailed report on the performance of smallWig on both file sets can be found in Supplementary Material.

We measured the performance of smallWig and other existing algorithms through the:

- *Compression rate (compression ratio)*, the compressed file size divided by the original file size.
- *Running time* of: (i) the encoding, (ii) the decoding and (iii) the random query process.

Figure 1 shows the compression rates achieved by various variants of smallWig, compared with the rates of gzip, bigWig and cWig through BedGraph. The depicted entropy is under the independent run difference model. With arithmetic coding, our algorithm offers 18-fold rate improvements compared with bigWig. In fact, the compressed file size of our running example is only 1/80 of the original WIG file. Furthermore, the compression rate is only 1.6% larger than the empirical entropy and may be attributed to storing the empirical probabilities. With context-mixing, one can further improve the compression rate to 23 times compared with bigWig. For compression with random queries, smallWig offers 17-fold rate improvements compared with bigWig.

According to the report in Hoang and Sung (2014), the compression rate of the state-of-the art cWig method is about 3.1 times better than that of bigWig. However, we found that one can obtain an even better rate by first converting a WIG file into a BedGraph file and then converting the BedGraph file to cWig with some simple additional processing (BedGraph files are compact representations of WIG files that fundamentally rely on run length coding). Our sequential WIG-BedGraph-cWig pipeline performs about 8.5 times better than bigWig. The newly introduced smallWig method still performs twice as well as the proposed modification of cWig. For databases containing TB/PB of WIG files, a 2-fold reduction in file sizes may lead to exceptionally important storage cost savings.

In Figure 2, we present the running time of smallWig encoding/decoding schemes, as well as those of gzip, bigWig and cWig. With arithmetic coding, smallWig has a 2.5 times smaller encoding and 1.5 times smaller decoding time compared with that of bigWig. Arithmetic coding with random query has 1.9 times smaller encoding time than bigWig. Context-mixing algorithms are computationally intensive compared with arithmetic coding and require significantly longer running time.

To compare the effect of different block sizes used for random query on compression rate and encoding/decoding time, we refer the reader to Figure 3. In the experiments, the block sizes ranged from 512 to 4096. To enable random query, we introduced a 3–17% overhead in compression rate and a 30% and 55% overhead in encoding and decoding time, respectively.

Table 1 lists the random query time. Note that the start positions (and for long queries the end positions) of the queries were generated uniformly at random among all allowed chromosomal locations for every chromosome. For short queries, the query length was fixed to 1000, so that one query falls within a single block; in this case, the query time corresponds to the time needed to retrieve the corresponding block. One can see that smallWig is comparable in performance to bigWig for short queries and runs about three times faster for long queries. It is also comparable to cWig for both types

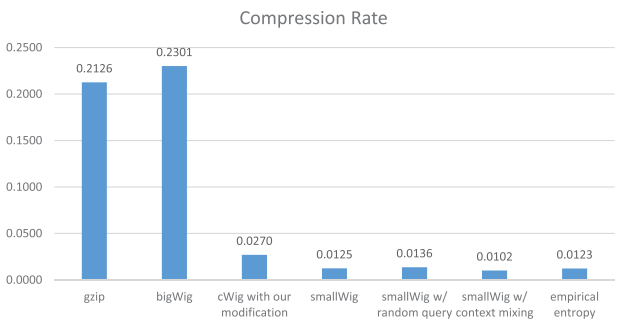

Fig. 1. Compression rates achieved by gzip, bigWig (Kent *et al.* 2010), cWig (Hoang and Sung 2014) through BedGraph and smallWig methods, which encompass arithmetic coding, arithmetic coding on blocks of size 1024 and context-mixing algorithms using lpaq1 (Mahoney 2002). To test cWig, we constructed our own WIG-BedGraph-cWig pipeline. All presented results are averaged over 14 sample files taken from ENCODE hg19. A more detailed table is included in the Supplementary Material

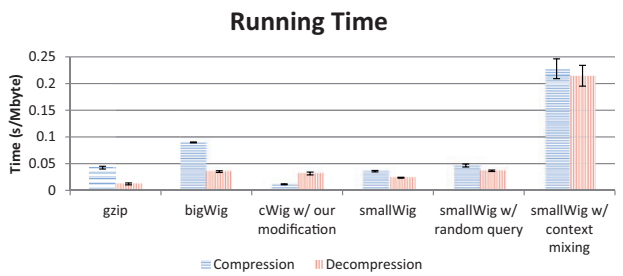

Fig. 2. Encoding and decoding time of gzip, bigWig and smallWig algorithms using arithmetic coding, arithmetic coding on blocks of size 1024 and our context-mixing algorithm. The encoding/decoding time is expressed in seconds per MB of the original WIG file. All the results were averaged over 14 sample files from ENCODE hg19. The error bars indicate the standard deviation. A more detailed table is included in the Supplementary Material

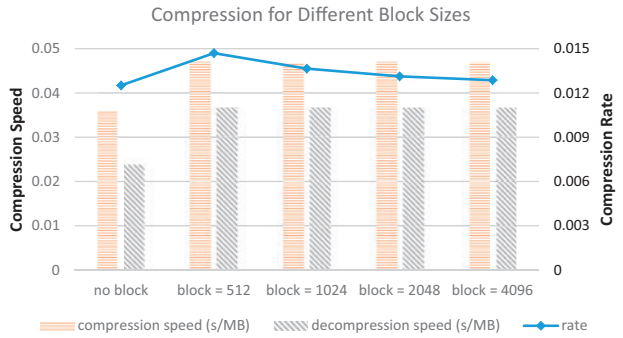

Fig. 3. Compression rate, encoding time and decoding time given different block sizes fed to arithmetic encoders. The label "no block" indicates that the whole sequence is compressed as a single block. The encoding/decoding time is expressed in seconds per MB in the original file. The y-label is for both the rate and the speed (s/MB). All the results are averaged over 14 sample files from ENCODE hg19

Table 1. Comparison of smallWig, bigWig and cWig with respect to random query time

Query Type	Measure	bigWig	cWig	smallWig
Long query	Average (s/bp)	1.99E-7	5.20E-8	5.87E-8
	std	4.74E-6	6.48E-8	6.08E-7
Short query	Average (s)	0.0565	0.0574	0.0711
	std	0.0515	0.1360	0.0176

We list the average query time in seconds per queried location for long queries, and the average query time in seconds for short queries, together with the corresponding standard deviations, over all 14 WIG files and 240 queries on each file.

of queries. Moreover, to facilitate visualization, in the random query functions, smallWig outputs the *exact* summary information together with the queried location-expression pairs. On the other hand, the bigWig summary function only outputs information corresponding to the overlapped blocks but not to that of the exact queried region.

We observe that for all the tested files, smallWig with arithmetic coding had a relatively small memory usage, as listed in Table 2. In particular, during most of the compression tests, the memory usage was less than 10 KB. With different user-defined parameters, smallWig with context mixing had higher and more variable memory usage, ranging from 90 KB to 1200 MB. We note here that since gzip does not offer random access and summary information, its memory usage is smaller than that of the other algorithms.

Table 2. Maximum memory usage during encoding and decoding in bytes for gzip, bigWig, cWig, smallWig and smallWig with context mixing on the tested files

	gzip	bigWig	cWig	smallWig	smallWig cntx
enc.	664	2315M	1935M	89K	90K–1200M
dec.	932	24K	45K	19K	90K–1200M

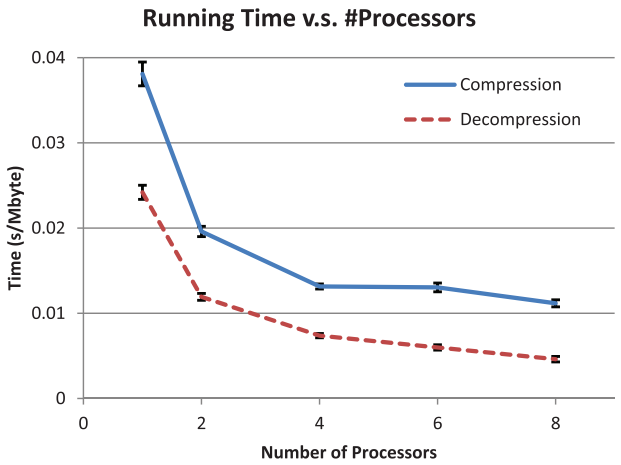

Fig. 4. Compression time versus the number of processors used in smallWig. The time is expressed in seconds per MB of the original WIG file. The block size is 512, and the results are averaged over 14 sample files from ENCODE hg19. Error bars indicate the standard deviation

In Figure 4, we show the running time of parallel multiprocessor compression methods. The encoding time is decreased by 2–3.4 times as the number of processors increases from 2 to 8. Furthermore, the decoding time is decreased by 2–5.2 times. The time does not decrease linearly since we used a uniform sequence partition procedure for individual chromosomes, and chromosomes have largely different lengths. Moreover, after every step in the algorithm (e.g. sequence transformation, empirical probability computation, arithmetic coding), some components of the pipeline have to pause until all processors have finished their computations and their information is aggregated.

We also tested smallWig on 10 WIG files that were generated from BedGraph files including integer-valued as well as floating-point-valued expressions. The average compression rates are shown in Figure 5. Note that BedGraph already takes into account the run length transformations and hence the compression rate improvements for these files are not as large as those for WIG files. For integer-valued files, smallWig is 5 and 1.8 times more efficient than bigWig and cWig, respectively. For floating point-valued files, smallWig is 4.3 and 1.9 times more efficient than bigWig and cWig, respectively. More details about these tests can be found in Supplementary Material.

6 Discussion

In what follows, we describe the differences in compression strategies used by various methods and attempt to intuitively explain the improved performance of smallWig compared with cWig and bigWig.

All three algorithms—bigWig, cWig and smallWig—use run length encoding. Both cWig and smallWig use delta encoding. Moreover, all three algorithms use blocks of a certain size for random query purposes: bigWig and cWig only operate with fixed

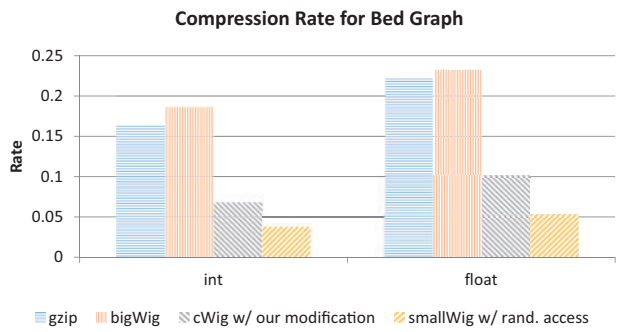

Fig. 5. Compression rates for BedGraph files. The bars on the left correspond to compression rates for integer-valued files, and the bars on the right correspond to compression rates for floating-point-valued file formats. Since the tested files have a large variance on their sizes, we take the sum of the compressed file sizes divided by the sum of the original BedGraph file sizes as our average rate

blocks of size 512, while smallWig allows for variable sizes. Furthermore, smallWig also allows for flexible indexing density, with indexing used to record the chromosomal locations that correspond to the blocks. Compared with bigWig and cWig, smallWig algorithms use a fairly simple data structure for indexing data and block addresses. As a combined result of these properties, for all the files compressed during testing, smallWig had a comparable running time for random query to that of bigWig.

To compress the run sequences, bigWig uses gzip [based on LZ77 (Ziv and Lempel 1977)] on each block. As already pointed out, gzip is a universal source coding scheme that does not rely on prior knowledge about the probability distributions. It approaches the entropy rate if the source is stationary and ergodic and as the sequence length goes to infinity. Since the alphabet sizes of the sequences are fairly large (a few thousand to several tens of thousand) but the block sizes are only 512, gzip offers somewhat poor performance. On the other hand, cWig uses Huffman codes for frequent values, and Elias delta codes for less frequent values. Both codes perform symbol-by-symbol encoding.

Assume that the data source is producing independent and identically distributed outputs with probability mass function $p(\cdot)$. Since the code word of a symbol x must be represented by a binary sequence, say of length $\ell(x)$, the individual symbol redundancy $r(x) = \ell(x) - \log_2 \frac{1}{p(x)}$ is a real number in $[0, \infty)$. Even for Huffman encoding (i.e. the optimal prefix encoding), the expected per-symbol redundancy may be large enough to create "visible" rate losses. There exist a number of results on the upper and lower bounds of the expected redundancy $r = E_X(r(X))$ for a random variable X. For example, Gallager (1978) showed an upper bound based on the largest symbol probability; Capocelli and De Santis (1991) bounded the redundancy both from above and below based on the largest and the smallest symbol probability and Mohajer *et al.* (2012) showed a tight upper and lower bound based on one known symbol probability p. In particular, in the latter case, the redundancy is lower bounded by $r \geq mp - \mathcal{H}(p) - (1-p)\log_2(1-2^{-m})$, where $m > 0$ is either $\lceil -\log_2 p \rceil or \lfloor -\log_2 p \rfloor$, depending on which value minimizes the overall expression. Here, $\mathcal{H}$ denotes the binary Shannon entropy function: $\mathcal{H}(p) = -p\log_2 p - (1-p)\log_2(1-p)$. For our running example, the run expression difference $x = -1$ has the largest probability, $p(-1) = 0.3374$, which leads to the corresponding redundancy of Huffman coding $r \geq 0.0275$. For a given distribution and a given symbol-by-symbol codebook which may not be optimal, there exists a non-negative and non-negligible coding redundancy r; on blocks of length 512, the overall redundancy equals $512r$, which is at least 14 bits per block for Huffman codes. If a different suboptimal code or unmatched Huffman code is used, this redundancy may be even larger. At the same time, arithmetic coding only causes a redundancy up to 2 bits per block if the probability distribution is known. As a result, smallWig files are significantly smaller than cWig files. Furthermore, smallWig is flexible in terms of the block size and enables context-mixing as well as parallel processing.

7 Conclusions

We studied compression methods for RNA-seq expression data. We proposed a new algorithm, termed smallWig, which achieves a compression ratio that is at least one order of magnitude better than currently used algorithms. At the same time, the algorithm also improves the running time and flexibility of random access. The presented results included detailed performance evaluations of smallWig in the standard, random access, context mixing and parallel operation mode.

Acknowledgements

The authors thank the editor and the anonymous reviewers for their comments and suggestions.

Funding

The work is partially supported by the Center for Science of Information (CSoI), funded under grant agreement CCF-0939370, and by NIH BD2K 1U01CA198943-01.

Conflict of Interest: none declared.

References

Cao,M.D. *et al.* (2007) A simple statistical algorithm for biological sequence compression. In: Data Compression Conference, 2007. *DCC'07*. IEEE, pp. 43–52.

Capocelli,R. and De Santis,A. (1991) New bounds on the redundancy of Huffman codes. *IEEE Trans. Inf. Theory*, **37**, 1095–1104.

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

Encode Project Consortium. (2004) The ENCODE (ENCyclopedia of DNA elements) project. *Science*, **306**, 636–640.

Gallager,R.G. (1978) Variations on a theme by Huffman. *IEEE Trans. Inf. Theory*, **24**, 668–674.

Hoang,D.H. and Sung,W.-K. (2014) Cwig: compressed representation of wiggle/bedgraph format. *Bioinformatics*, **30**, 2543–2550.

Jiao,J. *et al.* (2015) Minimax estimation of functionals of discrete distributions. *IEEE Trans. Inf. Theory*, **61**, 2835–2885.

Kent,W.J. *et al.* (2010) Bigwig and bigbed: enabling browsing of large distributed datasets. *Bioinformatics*, **26**, 2204–2207.

Krichevsky,R. and Trofimov,V. (1981) The performance of universal encoding. *IEEE Trans. Inf. Theory*, **27**, 199–207.

Kuruppu,S. *et al.* (2011) Optimized relative Lempel-Ziv compression of genomes. In: Proceedings of the Thirty-Fourth Australasian Computer Science Conference-Volume 113. Australian Computer Society, Inc., pp. 91–98.

Loh,P.-R. *et al.* (2012) Compressive genomics. *Nat. Biotechnol.*, **30**, 627–630.

Mahoney,M.V. (2002) The paq1 data compression program. *Draft, Jan*, **20**.

Marioni,J.C. *et al.* (2008) RNA-seq: an assessment of technical reproducibility and comparison with gene expression arrays. *Genome Res.*, **18**, 1509–1517.

Martin,G.N.N. (1979) Range encoding: an algorithm for removing redundancy from a digitised message. In: *Proceedings Institution of Electronic and Radio Engineers International Conference on Video and Data Recording*. Institution of Electronic and Radio Engineers.

Mohajer,S. *et al.* (2012) Tight bounds on the redundancy of Huffman codes. *IEEE Trans. Inf. Theory*, **58**, 6737–6746.

Pavlichin,D.S. *et al.* (2013) The human genome contracts again. *Bioinformatics*, **29**, 2199–2202.

Pinho,A.J. *et al.* (2011) On the representability of complete genomes by multiple competing finite-context (Markov) models. *PLoS One*, **6**, e21588.

Pinho,A.J. *et al.* (2012) GReEn: a tool for efficient compression of genome resequencing data. *Nucleic Acids Res.*, **40**, e27.

Rissanen,J. (1984) Universal coding, information, prediction, and estimation. *IEEE Trans. Inf. Theory*, **30**, 629–636.

Rissanen,J. and Langdon,G.G. Jr. (1979) Arithmetic coding. *IBM J. Res. Dev.*, **23**, 149–162.

Salomon,D. (2007) *Variable-Length Codes for Data Compression*. **Vol. 140**. Springer Science & Business Media.

Tabus,I. and Korodi,G. (2008) Genome compression using normalized maximum likelihood models for constrained Markov sources. In: *IEEE Information Theory Workshop, 2008*. ITW'08. *IEEE, pp. 261–265*.

Vinga,S. (2013) Information theory applications for biological sequence analysis. *Brief. Bioinform*, **15**, 376–389.

Wang,C. and Zhang,D. (2011) A novel compression tool for efficient storage of genome resequencing data. *Nucleic Acids Res.*, **39**, e45.

Wang,Z. *et al.* (2009) RNA-seq: a revolutionary tool for transcriptomics. *Nat. Rev. Genet.*, **10**, 57–63.

Willems,F.M. *et al.* (1995) The context-tree weighting method: basic properties. *IEEE Trans. Inf. Theory*, **41**, 653–664.

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