STrieGD: A Sampling Trie Indexed Compression Algorithm for Large-Scale Gene Data

Yanzhen Gao[1,2(✉)], Xiaozhen Bao[1,2], Jing Xing[1], Zheng Wei[1], Jie Ma[1], and Peiheng Zhang[1]

[1] Institute of Computing Technology, Beijing, China
gaoyanzhen@ncic.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China
http://www.ict.ac.cn/

Abstract. The development of next-generation sequencing (NGS) technology presents a considerable challenge for data storage. To address this challenge, a number of compression algorithms have been developed. However, currently used algorithms fail to simultaneously achieve high compression ratio as well as high compression speed. We propose an algorithm STrieGD that is based on a trie index structure for improving the compression speed of FASTQ files. To reduce the size of the trie index structure, our approach adopts a sampling strategy followed by a filtering step using quality scores. Our experiment shows that the compression ratio of our algorithm increased by approx. 50% over GZip, while being nearly equal to that of DSRC. Importantly, the compression speed of the STrieGD is 3 to 6 times faster than GZip and about 55% faster than DSRC. Moreover, with the increase of compressors, the compression ratio remains stable and the compression speed is nearly linear scalable.

Keywords: Sampling trie · FASTQ file · Data compression

1 Introduction

Analysis of large DNA sequencing datasets is extensively applied to a wide range of research areas, including genetic engineering, medical diagnosis, and forensic biology [1]. Importantly, with the development of next-generation sequencing (NGS) technology, the cost of DNA sequencing has decreased considerably. DNA sequencing data has grown rapidly and had gotten to the petabyte scale until 2017 [2], presenting a considerable challenge for data storage, content access and transfer [3]. Compressing DNA data is an effective way to solve these problems.

In addition, DNA data generated by mainstream high-throughput sequencing platforms, including the SOLiD sequencer independently developed by Illumina GA and ABI [4], are generally stored in the FASTQ format [5]. Therefore, compression of FASTQ is important for computational biology. FASTQ files consist

Published by Springer Nature Switzerland AG 2018
F. Zhang et al. (Eds.): NPC 2018, LNCS 11276, pp. 27–38, 2018.
https://doi.org/10.1007/978-3-030-05677-3_3

of records, each record has four lines as shown in Fig. 1: title line, genomic sequence, "+" and quality scores. Genomic sequence is the nucleotide sequence obtained by sequencing, containing only five different kinds of characters. The character which is not identified as A, C, G and T, is expressed as the "N". The quality score is the probability of the character being incorrectly identified, which means that the length of Quality scores is the same as that of the genomic sequence. In addition, the length of the title line is shorter than that of the genomic sequence. Therefore, the genomic sequence occupies one-third or more of the entire file, which means that compressing genomic sequence is important for the FASTQ file.

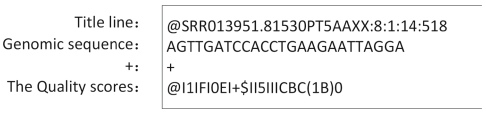

Fig. 1. Format of the FASTQ file

Based on the FASTQ file described above, one can draw a conclusion that compressing four parts of FASTQ data can naturally be processed (almost) independently. Great efforts have been put towards improving compression of gene data with FASTQ format. However, currently used algorithms fail to simultaneously achieve a high compression ratio as well as high compression speed. General compression algorithms do not consider the feature of the FASTQ file, causing a low compression ratio. However, special compression algorithms add judging operations to achieve a high compression ratio, causing a low compression speed. Here, we propose an algorithm STrieGD that is based on trie index structure for improving the compression speed. To reduce the size of the trie index structure, our approach adopts a sampling strategy followed by a filtering step using quality scores, simultaneously aiming at high compression ratio and high compression speed.

The following sections: Sect. 2 describes the related works in compression algorithms. Section 3 describes our algorithms. Section 4 describes the details about the implementation of the distributed compression system. Section 5 presents the evaluation we conducted in the distributed compression system. The last chapter summarizes the paper.

2 Related Research

Genomic sequence occupies one-third or more of FASTQ file. It is redundancy, which dues to the simple structure, great depth of sequencing and large similarity between the same species [6]. How to take full advantage of the peculiar redundancy of genomic sequence is the key to improve compression ratio and compression speed. In recent years, scholars had done in-depth research on the

characteristics of genes data and proposed various compression algorithms for the FASTQ file.

G-SQZ algorithm [7] constructs the unit <bases, Quality scores> and adopts the Huffman algorithm to compress. G-SQZ is too simple. The compression ratio and speed are only slightly better than GZip.

The DSRC algorithm [8] moves the character "N" to the quality stream and uses the LZ algorithm [9] to compress the remainder. For the quality scores, the DSRC algorithm records the place of "#" that means the character "N" appears in the genomic sequence and uses RLE algorithm to compress the characters that are repeated with a continuously high rate. However, it achieves a high compression ratio but low compression speed.

KungFQ [10] stores a single bit flag and up to three base calls or a run length for repetitions longer than four bases. The bit flag is necessary to discriminate between these two cases. The quality scores are directly compressed with RLE [11]. This method achieves a high compression ratio, but low compression speed. Moreover, KungFQ wastes space on encoding "N".

LFQC algorithm [2] splits the sequences into non-overlapping $l-mers$ with an empirically decided $l-value$ and counts the frequency of distinct quality scores in each $l-mer$. Assume that the quality score q_i has a frequency of f_i in $l-mer$. LFQC picks the quality score q_i with the largest frequency f_i in $L_j \forall_j$. L_j goes to the q_j^{th} bucket. $l-mers$ where none of the symbols showed a majority of occurrences go to a special bucket called generic bucket B_G. The file is also compressed using Huffman Encoding. The encoding method of genomic sequence is similar to that of the quality score. LFQC is able to achieve a high compression ratio. However, the process of separating buckets slows down the overall compression speed.

Using GZip as a benchmark (compression ratio and compression speed are all set to 1), the compression ratio and compression speed of various compression algorithms are shown in Fig. 2. We found that the compression speed is inversely proportional to compression ratio, which means that to further improve the compression ratio requires more CPU time and memory space. Therefore, it is important to maintain a balance between compression ratio and compression speed in the compressing process.

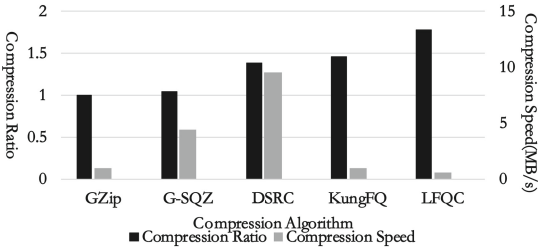

Fig. 2. Comparison of FASTQ file compression algorithms

3 A Trie Index Structure Based Compression Algorithm

Fragments of genomic sequence are highly repeatable in a FASTQ file. How to fully replace repeatable fragments is the key to improve the effect of compression. The most ideal method is to store the repeatable fragment only once. Therefore, we need an index structure to index the repeatable fragments. The index structure is better to support to quickly query and insert data. However, all of the existing algorithms adopted hash table to index data, which needs to traverse all strings before searching. Therefore, to improve the compression speed, we adopt a trie structure to index strings.

3.1 Trie Index Structure

Trie is a tree structure, which only saves the same prefix once. The first step of compressing involves searching a fixed-length sub-string in the trie index structure. If the same fragment is found, the position and length of the matching fragment are recorded. Otherwise, the sub-string will be added into the trie index structure. Query and insert contribute most of the overhead among all the operations. Although the time complexity of the hash table and trie both are O(n), only a trie is able to avoid the collision and support partial matching, thus reducing unnecessary string comparing.

Trie index structure is able to achieve partial matching, which is different from the hash table. If we search string "TCCTA" in the trie shown in Fig. 3, we will obtain the best matching sub-string with the length of four. This matching process reduces the unnecessary character comparison by trie index structure. If we search string "TTACG" in the same trie shown in Fig. 3, we will fail to match the best sub-string on the third character "A" of the string. It is not necessary to match the other characters, which is helpful to query efficiently.

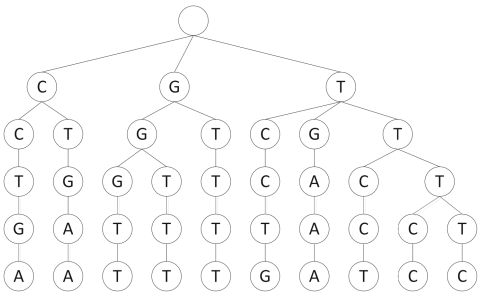

Fig. 3. Trie constructed by the string "GGGTTTTCCTGAAA" with the sub-string's length 5.

Trie is a typical space-time trade-off data structure, which means trie have to consume more memory to achieve efficient query. As the scale of data grows,

if the hardware cannot provide enough memory, a query will be less efficient as data will exchange frequently between memory and disk. If the trie index structure only occupies limited memory, the subsequent string will not be added to the index structure, thus decreasing the successful matching rate. In order to reduce excessive memory occupied by the trie, we propose two optimization strategies.

3.2 Optimization of Trie Tree

In order to describe the characteristics of the genomic sequence, we propose two concepts: String coverage calculated by formulas (1), SubString coverage calculated by formulas (2), shown in Fig. 4. As the length of the string grows, the SubString coverage drops from 50% to 27% and the String coverage increases to 82%, indicating that the repeated substring is relatively concentrated.

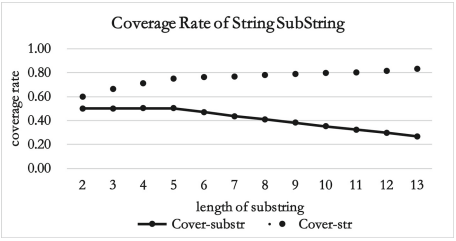

Fig. 4. String coverage and substring coverage.

M_{length} indicates the number of substring types whose length is $length$, N_{M_i} indicates the number of substrings M_i, $Sum\,(M_{length})$ indicates the number of substrings that are generated in length, $Cover_{str}$ indicates the coverage of the string and $Cover_{SubStr}$ indicates the coverage of the substring.

$$Cover_{SubStr} = \frac{\sum_{i=1}^{M_{length}} a}{M_{length}}, N_{M_i} > \frac{sum(M_{length})}{M_{length}} \tag{1}$$

$$Cover_{Str} = \frac{\sum_{i=1}^{M_{length}} N_{M_i}}{sum(M_{length})}, N_{M_i} > \frac{sum(M_{length})}{M_{length}} \tag{2}$$

Therefore, it is not necessary to store all the strings to obtain a higher compression ratio in the trie. However, how to choose the right string to save and how many strings to save are problems.

Sampling. Sampling is mainly used to reduce the scale of referenced objects to a certain size that is covered by the processing system. We still take the string in Fig. 3 as an example. Several strings are inserted into the trie structure

when the sampling rate is 1/3. Figure 5(a) shows substrings and Fig. 5(b) shows the trie. The occupied space greatly reduces. However, the sampling rate has a great influence on matching. If the sampling rate is too high, the problem of excessive memory space will still exist. If the sampling rate is too low, the matching will often fail, causing the compression ratio to decrease. Therefore, when we select the sampling rate, we need to consider occupied memory space and the compression ratio.

The trie is a perfect structure for a partial matching. For the Trie structure in Fig. 5, we obtain the best matching sub-string with the length of 4 to compress the string of "TCCTA". This matching reduces unnecessary character comparisons as much as possible and achieves efficiently query.

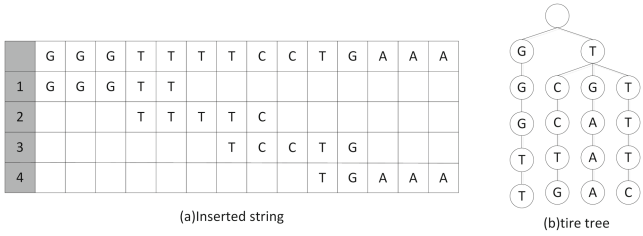

Fig. 5. String coverage and substring coverage.

However, not all substrings are inserted into the trie structure, causing a problem in the matching process. For the string "GTTTT", the matching length is one (matching to the insert string one) and the matching length is too short. If we ignore the first character "G" and starts to match from the second character "T", we will obtain a matching length of four (matching to the insert string 2). In the actual process, the normal matching will be done first. If the substring is not completely matched, the first character will be ignored. Then compare the two ways and select a longer matching length. This process is called "lazy match".

Filtering by Quality Scores. The quality score is the probability of the base being incorrectly identified. It is known that if a sequence's quality score is too low, it indicates that the accuracy of the sequence obtained by sequencing is low, meaning that the sequence is next to impossible to be matched in the future. Therefore, the quality score is used to decide whether the string deserves to be inserted into the trie index structure. Strings with low quality score will be filtered out, which ensures high speed.

4 Implementation of Distributed Compression System

4.1 Compression of Quality Portion and Identification Portion

Each identification field of the record is highly similar. Therefore, we divide the identification field into four fields according to the feature of each field.

1. The data remains unchanged in different record. (Field 1)
2. Integer values vary monotonically over consecutive records. (Field 2)
3. Integer values vary in a certain range. (Field 3)
4. The data does not belong to any of the above-mentioned types. (Field 4)

StrieGD stores Field 1 only once and uses RLE algorithm to encode Field 2. In addition, StrieGD stores Field2 with a minimum of bits and stores Field 4 without compressing.

4.2 Compression of Quality Portion

Since the quality scores range from 33 to 126, it is possible to restore the character of "N" according to its Quality scores during the decompression process. Therefore, we add the score 128 representing "N" of sequence portion to the quality scores, achieving to delete the character of"N" from sequence portion. Although the length of the quality scores is equal to that of the sequence portion, quality scores contain much more variety of characters than sequence portion, causing that to improve the compression performance of the quality score is more difficult. Therefore, we did not take much effort to improve the compression performance of the quality score. Our STrieGD adopts the RLE algorithm to encode characters with high repeatable and Huffman algorithm to encode others. STrieGD stores a single bit flag to discriminate between these two cases.

4.3 Implementation of Distributed Compression System

It is impractical to support compressing a large volume of DNA files for a single server. To compress large-scale genetic data, we designed and implemented a distributed compression system, Dic-DNA. Dic-DNA includes client, server and compressor shown in Fig. 6.

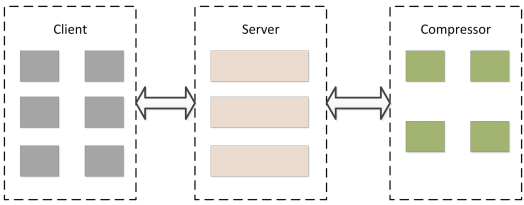

Fig. 6. String coverage and substring coverage.

In the distributed system, the client asks to write (compress), read (decompress), search, and delete genomic sequence. In addition, the client will send the genomic sequence to the server if the server allows the client to write.

The server plays a bridge role, connecting the client and the Compressor and maintains a request queue to receive requests. The server extracts the request from the queue and selects the appropriate processing according to the type of request. In addition, requests of compression and decompression are forwarded to the compressed node. The server maintains a file-block map that stores file-block mapping information, including block offsets, target compression nodes and other useful information. The server make it possible to compress and decompress the same file in different clients.

The compressor compresses and decompresses the files. Each compressor employs individual block-location to map information, thus the distributed system is more scalable.

5 Evaluation

The distributed system includes eight clients, four servers and eight compressors. Each node runs on 64-bit CentOS 6.3 operating system with 16-core 2.00 GHz Intel(R) Xeon(R) CPU and memory 16 G. The test data is from the NCBI, ENA and other sites. The size of files ranges from 3 GB to 15 GB and the length of each sequence is between 45 and 120.

5.1 Performance of Compressing Single FASTQ File

In order to verify whether our optimization strategy is effective, we evaluated the compression speed and compression ratio in different sampling rates and different thresholds of quality scores.

Firstly, Fig. 7 shows compression speed and compression ratio at different sampling rates. The compression ratio is the highest when the sampling rate is 1. However, the compression speed is very slow, only 1 MB/s or so, due to that the size of the trie structure is quite large. With the decrease of the sampling rate, the compression ratio gradually decreases but without great fluctuation, because the repeated fragments are relatively concentrated. In addition, with the decrease of the sampling rate, the compression speed increases. When the sampling rate is 1/8, the compression speed reaches the maximum value. As the sampling rate further decreases, both the compression speed and the compression ratio begin to decrease quickly. The lower sampling rate, the more sub-string adopts Huffman encoding, affecting the compression speed and compression ratio. Therefore, the data shows that our sampling strategy considerably improves the compression speed and simultaneously obtain a high compression ratio.

Secondly, we evaluated compression speed and compression ratio at different threshold values shown in Fig. 8. Only the sequence, whose the average value of quality scores reaches the threshold, was inserted into the Trie. With the increase of the threshold, the compression speed increases, because a number of

strings with the lower quality score than the threshold are filtered out. When the threshold value is 62, the compression speed reaches the maximum. As the threshold further increase, less and less strings are inserted into the trie structure, causing that many strings are encoded with Huffman and compression speed and compression ratio decrease. Therefore, the data shows that our filter strategy considerably improves the compression speed and simultaneously obtain a high compression ratio.

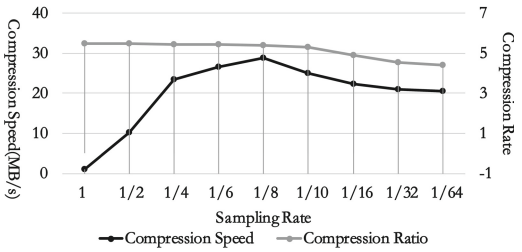

Fig. 7. Effects of different sampling rates trie index structure for compression speed and compression ratio.

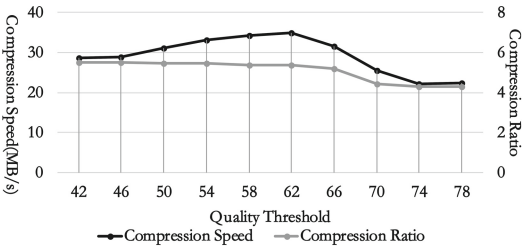

Fig. 8. Trie indexing structures of different effects on the quality scores threshold speed and compression ratio values.

Moreover, in order to compare our STrieGD with other compression algorithms, we evaluated the compression speed and compression ratio of four algorithms: GZip, Bzip2, DSRC and STrieGD. The compression speeds and compression ratios of two files (SRR608881 and ERR217195) are respectively shown in Figs. 9 and 10.

Compared to other compression algorithms, the compression speeds of both test files in STrieGD are the highest and reach 40 MB/s or more shown in Fig. 9. However, the compression speeds of two general algorithms (GZip, Bzip2) are both below 10 MB/s and the compression speeds of the DSRC algorithm are below 30 MB/s. Therefore, our STrieGD achieves a high compression speed. In addition, we found that the compression speed fluctuates greatly in different

files, due to that the levels of file redundancy are different. Moreover, we found that the compression ratio of our STrieGD is 50% higher than that of GZip, 18% higher than that of Bzip2 and nearly equal to that of DSRC shown in Fig. 10. Therefore, our STrieGD is able to achieve high compression speed and high compression ratio.

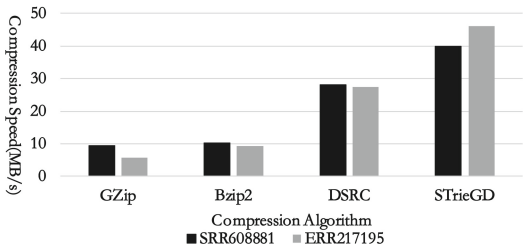

Fig. 9. FASTQ file compression speed comparison stand-alone case.

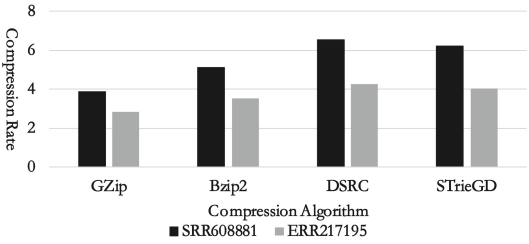

Fig. 10. FASTQ file compression ratio vs. stand-alone case.

5.2 Performance of System

In order to test the scalability of our STrieGD, we evaluated the compression speed and compression ratio at 1–8 different compressors. The testing environment includes eight clients, four servers with four threads. As shown in Fig. 11, with the number of compressors increases, the compression ratio linearly grows and the compression ratio is stable, which shows that the distributed system has a good scalability.

Moreover, we test the bandwidth of the system with the number of compressors from one to eight. As shown in Fig. 12, the system bandwidth is about 200 MB/s when the compressed node is 1, because the system spent many sources in compressing, causing the actual disk write rate in the compressor is much lower than the rate of data received. With the number of node increasing, the system's bandwidth linearly grows. Therefore, our data shows that our STrieGD is highly scalable.

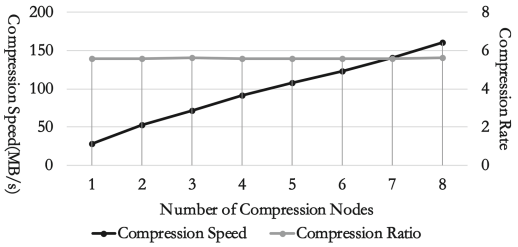

Fig. 11. FASTQ file compression ratio vs. stand-alone case.

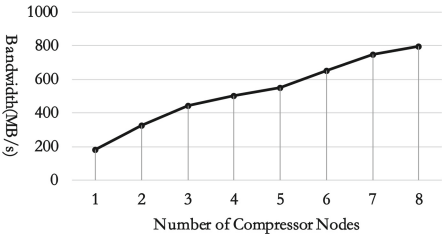

Fig. 12. FASTQ file compression ratio vs. stand-alone case.

6 Conclusions

The advance of NGS produces huge volume of data, presenting a big challenge for gene data storage. To address this challenge, we proposed a sampling trie indexed compression algorithm to compress FASTQ files. It adopts tried indexed structure to accelerate compression speed, and employ a sampling strategy to reduce the size of tried index structure to support large scale gene data. Through evaluation on our distributed compression system, the results show that STrieGD is able to gain a high compression ratio as well as the highest compression speed compared with other related works. With its features of high compression speed and high compression ratio, STrieGD is able to be used on the filed of online processing for gene data.

Acknowledgments. We thank the anonymous reviewers for their insightful comments. We thank Xueqi Li for providing data sets and Torsten Juelich for his helpful advices in writing. We would also like to thank Hougui Liu and Huajie Zheng for the implementation of our deduplication systems. This work was supported by the National Natural Science Foundation of China (Grant No. 601502454).

References

1. Clinton, R.D.: The Selfish Gene. Oxford University Press, Oxford (2006)
2. Nicolae, M., Pathak, S., Rajasekaran, S.: LFQC: a lossless compression algorithm for FASTQ files. Bioinformatics **31**(20), 3276–3281 (2015)

3. Roguski, Ł., Ribeca, P.: CARGO: effective format-free compressed storage of genomic information. Nucleic Acids Res. **44**(12), 114 (2016)
4. Stuart, M.B.: Sequencing-by-synthesis: explaining the illumina sequencing technology. BitesizeBio (2012). https://bitesizebio.com/13546/sequencing-by-synthesis-explaining-the-illumina-sequencing-technology/
5. Cock, P.J., Fields, C.J., Goto, N., Heuer, M.L., Rice, P.M.: The sanger FASTQ file format for sequences with quality scores, and the solexa/illumina FASTQ variants. Nucleic Acids Res. **38**(6), 1767–1771 (2010)
6. WIKIPEDIA. Genetic testing (2017). https://en.wikipedia.org/wiki/Genetic_testing
7. Waibhav, T., James, L., Suh, E.: G-SQZ: compact encoding of genomic sequence and Quality scores. Bioinformatics **26**(17), 2192–2194 (2010)
8. Deorowicz, S., Grabowski, S.: Compression of DNA sequence reads in FASTQ format. Bioinformatics **27**(6), 860–862 (2011)
9. Ziv, J., Lempel, A., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977)
10. Grassi, E., Gregorio, F.D., Molineris, I.: KungFQ: a simple and powerful approach to compress FASTQ files. IEEE/ACM Trans. Comput. Biol. Bioinform. **9**(6), 1837–1842 (2012)
11. Golomb, S.W.: Run-length encodings. IEEE Trans Inf. Theory **12**(3), 399–401 (1966)