Compression with unified and accessible byte blocks to enhance management and analyses of UKBB-scale genotypes

Miaoxin Li (limiaoxin@mail.sysu.edu.cn)
Sun Yat-sen University https://orcid.org/0000-0002-4733-0109
Liubin Zhang
Sun Yat-sen University https://orcid.org/0000-0003-2509-4333
Yangyang Yuan
Sun Yat-sen University
Wenjie Peng
Sun Yat-sen University
Bin Tang
Sun Yat-sen University
Mulin Jun Li
Tianjin Medical University
Allen Gui
University of Hong Kong

Brief Communication

Keywords:

Posted Date: October 21st, 2021

DOI: https://doi.org/10.21203/rs.3.rs-944936/v1

Compression with unified and accessible byte blocks to enhance management and analyses of UKBB-scale genotypes

Liubin Zhang, Yangyang Yuan, Wenjie Peng, Bin Tang, Mulin Jun Li, Hongsheng Gui, Miaoxin Li

Abstract

Whole-genome sequencing projects of millions of persons contain enormous genotypes, entailing a huge memory burden and time overhead during computation. Here, we introduce Genotype Blocking Compressor (GBC), a method for rapidly compressing large-scale genotypes into a fast-accessible and highly parallelizable format. We demonstrate that GBC has a competitive compression ratio to help save storage space. Furthermore, GBC is the fastest method to access and manage compressed large-scale genotype files (sorting, merging, splitting, etc.). Our results indicate that GBC can help resolve the fundamental problem of time- and space-consuming computation with large-scale genotypes, and conventional analysis would be substantially enhanced if integrated with GBC to access genotypes. Therefore, GBC's advanced data structure and algorithms will accelerate future population-based biomedical research involving big genomics data.

Main

With dramatically decrease in sequencing cost and advances in precision medicine, whole genome-wide genotypes for millions of persons will be routinely used in basic research or clinical practices in the near future. The Variant Call Format (VCF)[1] is a widely-used framework to store the aggregate information for genome sequencing projects, and it has yet become the standard format for genetic and genomic studies. However, the easy-to-read text format of VCF files occupies much storage space and is not specifically designed for large-scale genotype data. Besides, the random access of large-scale genotype data will impose a huge computational burden during downstream analysis. Therefore, a more efficient genotype compression algorithm along with high-performance supporting functions (e.g., random access and splitting) is urgently needed for large human genetics and genomics projects, e.g., the UK Biobank (UKBB)[2].

So far, multiple effective compressors have already been proposed to compress genotypes in VCF files, including PBWT[3], BGT[4], GQT[5] and GTC[6]. Unfortunately, there are still common limitations of the above methods. First, these methods used lossy compression for genotypes of variants with multiple alleles. Second, their aggressive compression strategies are not efficient for large-scale projects, which may cause a huge memory burden and a slow processing speed. Third, they do not have further efficient management and advanced algorithms for compressed data, making it necessary to decompress the entire data for subsequent management and analysis. Finally, their data structures are not well-designed for highly parallel analyses. To systematically address these issues, we propose an efficient method named Genotype Blocking Compressor (GBC) for rapidly compressing, accessing and managing large-scale genotype data.

GBC was designed to rapidly compress large-scale genotypes into a fast-accessible and highly parallelizable format. To achieve this goal, we made three major technique innovations. First, GBC stores compressed genotypes of all variants into independent blocks in a single file with a compact and unified format (Fig.S1a). An independent block contains genotype data and summary data of variants on the same chromosome, in which the variant number in a block is balanced with the sample size given a maximal array size (2GB). The block summary includes chromosomes, coordinates, alleles of variants in the block. A tree structure containing chromosome, block start and block end positions are designed for fast localization and merging the blocks. Thus, data retrieval and editing in a large GTB file can be quickly accomplished only in involved blocks without decompressing the entire file. Based on the independent blocks, we designed a cyclic locking mechanism (CLM) for parallel decompression without losing original variant orders.

Second, in each independent block, GBC has a unified byte-based encoding strategy of genotype (BEG, Fig.S1b) to denote genotypes of variants with various alleles flexibly. BEG encodes a genotype into a byte at first. For a biallelic variant, the genotypes (i.e., the missing, 0|0, 0|1, 1|1, and 1|0) are encoded as 0, 1, 2, 3 and 4 respective. For multiallelic variants ($n \leq 15$), the non-missing genotype $a \mid b$ are encoded as $(\max\{a, b\})^2 + \max\{a - b, 0\} + a + 1$. BEG solves the problem of loss compression for multiallelic variants. Because most genotype compression algorithms (e.g., BGT[4]) used a bit-based encoding strategy, they had to split a multiallelic variant into multiple biallelic variants and encoded each biallelic variant separately. To save more space, we further combine 3(phased) or 4(unphased) consecutive BEG codes of biallelic variants into one byte, which maximized the available range of a single byte (hence, the shorthand MBEG). We show that the MBEG can save 2/3~3/4 extra space from BEG arrays. In each block with MBEG, the address of original genotype of a subject at a variant can be calculated for quick retrieval.

Third, the BEG and MBEG array in a block are sorted by an approximate minimum discrepancy ordering (AMDO) algorithm, which benefits compression. AMDO considers both the allele frequencies and genotype distribution pattern to sort variants precisely for more efficient compression. AMDO only has an $o(nm)$ time complexity in the fast sorting process, significantly improving the compression ratio up to 14.3%. Based on the AMDO, GBC has a nearly linear time complexity concerning the number of variants and subjects. For example, it takes a similar time to compress 100,000 variants for 10 million subjects or 10 million for 100,000 subjects. In contrast, alternative methods (e.g., GTC[6]) used algorithms of $o(nm^2)$ time complexity to sort variants. The encoded and sorted genotypes in each block can be efficiently compressed by any conventional compression algorithms (e.g., Gzip, LZMA and zlib). In the present study, we adopted a relatively faster algorithm, ZSTD. Besides, LZMA with a higher compression ratio was also used as an alternative compression algorithm.

A series of experiments were carried out to investigate the performance of these advanced algorithms in GBC systematically. We first compared the compression ratio of the GBC with the other three widely used alternative methods, including BGT, PBWT and GTC. According to Danek et al.[6], GTC had the largest compression ratio among the three methods. However, Fig.1a showed that GBC delivered the best compression ratio on the moderate datasets (sample size≤5000), 37% higher than GTC. For example, GBC only needed 1.61GB of space to store the 1000GP3[7], while GTC needed 1.99GB. Fig.1b summarized the compression ratios of the GBC compared with the alternative methods on typical public datasets. With the increase of the samples, the compression ratio of GBC and GTC became gradually similar. In brief, GBC provided a more competitive compression ratio, and it was more advantageous to compress the widely available unphased genotypes.

Then, we investigated the compression and decompression speed of the GBC and the alternative tool GTC (with the highest compression ratio) using simulated and real genotypes. In the term of compression speed, we found that the GBC was much faster than GTC. The relative speed grew exponentially as the sample size increased. Fig.1c showed the compression speed ratio of GBC to GTC on simulated datasets. For instance, GBC was over 1000 times faster than GTC in a simulated sample with 500,000 subjects. GBC took 0.80 minutes and 3.87 minutes to compress the genotype datasets with 30,000 variants of 100,000 and 500,000 subjects, respectively, while GTC spent 180 minutes and 4964.19 minutes doing it. In the real dataset testing, GBC was 454.73 times faster than GTC in compressing the UKBB[2] genotypes of 32,626 variants on chromosome 10 of 487,409 subjects. Besides, for the imputed genotypes of 4,562,904 variants on chromosome 10, GBC took 549.02 minutes for the compression while CTC failed after three weeks' running. This comparison clearly showed that GBC had the superior speed to compress UKBB and even larger-scale genotype datasets. And in the term of decompression speed, GBC was also approximately 1.8 times faster than GTC when decompressing to the BGZ format (Fig.1c). Note that it is more reasonable to describe the speed difference in decompression with BGZ format, as it avoids the inefficiencies of the algorithm caused by the slow disk output speed. Finally, GBC can be several times faster under multi-threads for compression and decompression (Fig.1d).

The time cost of accessing the compressed genotypes is more important for subsequent computation based on genotypes. GTB format and GBC functions help provide the fastest and most flexible access to genotypes. First, we compared the random-access time of several alternative methods in extracting 0.01%, 0.1%, 1% and 10% of the variants from the SG10K[8] dataset. We found that GBC was more than two orders of magnitude times faster than others (Fig.1e). Note that when retrieving multiple uncontiguous variants, GBC can operate on multiple variants simultaneously and return the results at once, while PBWT, BGT and GTC take longer time because they can only retrieve one variant at a time. Second, GBC was 1.27 times faster than the best of the other peer methods (i.e., BGT) on accessing a set of consecutive variants on the same chromosome (Fig.1f). What is more, GBC can quickly find the genotype data for the given samples and the superior speed of GBC over other methods became increasingly striking when the sample size increased. For example, when extracting genotypes in a single subject, GBC was 1.59 times faster than GTC on small-scale datasets, while 5.16 times faster on large-scale datasets (Fig.1g and Fig.1h). We indicated that all forms of data access would eventually approach the speed of decompressing all data as the number of genotypes increases. However, GBC was still 1.8 times faster than the currently fastest method GTC on large sample sizes (Fig.1c). Besides, GBC was also 1.25 times as fast as other methods when filtering out variants within a specified allele frequency range (Fig.1i). Especially, GBC is by far the only method that enables fast retrieval on unordered datasets (instead of sorting before retrieving as BCFtools or others), and the time cost on unordered datasets was similar to that on ordered datasets (Fig.1j). Last but not least, GBC has yet been the only tool that supports querying data in parallel. The query speed of GBC under multiple threads could be several times faster than itself of a single thread, which reveals the great potential of GBC to accelerate complex calculations.

The genotype access and calculation can be carried out in a space-saving and fast manner based on GTB, which helps speed up downstream analyses with genotypes. Here, we showed the advantage of calculating the pair-wise linkage disequilibrium (LD) coefficients with GTB format. At the algorithmic level, GBC used bitwise operations to maximize computational speed, which is similar to PLINK(V1.9)[9] and KGGSeq[10]. However, GBC accessed genotypes for computation based on GTB format rather than the vcf.gz format like other tools (such as PLINK, VCFtools[11], PopLDdecay[12]). Taking the 1000GP3 dataset as an example, the results (Fig.1k) showed that GBC was 3.6 times faster than PLINK(V1.9) under a single thread. Especially, GBC could be even 17.3 times faster than PLINK(V1.9) under 8 threads. Here, we emphasized that the speed advantage of GBC is largely due to the faster genotype parsing process and the cyclic locking mechanism for parallelization. Although the latest research indicated that GPU-based computing (if possible) could significantly improve the speed of LD calculation[13], GBC mainly focused on optimizing the computational speed at the (input/output) IO level (Fig.1l). Therefore, we believe that many tools would have a better performance if integrated with GBC to access and parse genotypes. The GTB format and GBC functions have largely overcome the limitation of IO and made great processes on IO optimization, which are particularly beneficial for the development of genomic tools or platforms analyzing large-scale genotype data.

Convenient management of large-scale compressed files is also critical for genetic and genomic studies. We compared GBC with the current popular tool BCFtools [14] for managing compressed genotype data. Note that BCFtools can only handle compressed VCF in BGZ formats, while GBC manages compressed genotype blocks. When concatenating genotype files with the same samples, we found GBC was 4197.8 times (Fig.1m) faster than BCFtools. It was even 14639.1 times (Fig.1n) as fast as BCFtools when splitting genotypes by chromosome. Besides, GBC was 10.06 times faster than BCFtools on merging genotypes files with non-overlapping subjects under a single thread (Fig.1o). For retrieving the genotypes of a subset of subjects, the speed of GBC was further increased by 6 times because of the advanced GTB structure (Fig.1p) compared to the BGZ format (Fig.1h). In terms of the sorting speed, GBC was always approximately 3 to 10 times faster than BCFtools for 1,000,000 variants (Fig.1q, the less disordered the data, the better performance the GBC would have). Particularly, GBC can achieve the sorting directly in 4 GB of memory while the BCFtools need external disk space (Fig.S2). Similar to the data access performance, we believe that the speed difference between GBC and BCFtools will be more pronounced as sample sizes increase. The results also illustrated that the independent block structure design efficiently managed large amounts of genomic data and substantially sped up large-scale complex manipulation.

To summarize, we designed a fast compression tool GBC to store large-scale genotypes into a uniform, accessible and space-saving format (i.e., GTB). Remarkably, GBC significantly reduces the running memory required, allowing even large genome projects with tens of thousands of subjects to run in 4 GB of runtime memory. Besides, GBC is a highly parallelizable computational toolset that can respond extremely quickly to various data access methods and management requirements, which would help save the overhead of pay-per-use services on popular cloud platforms. Three commonly used and user-friendly interfaces are also provided, including graphical interface, command-line and API library. Thus, GBC is suitable for a wide range of users, enabling professional researchers and amateurs to use it even on personal laptops easily. Several extensions to GBC are under consideration for further development, including more efficient functions for large-scale genomic computations (e.g., the population PCA) based on its parallelization and low memory consumption advantages.

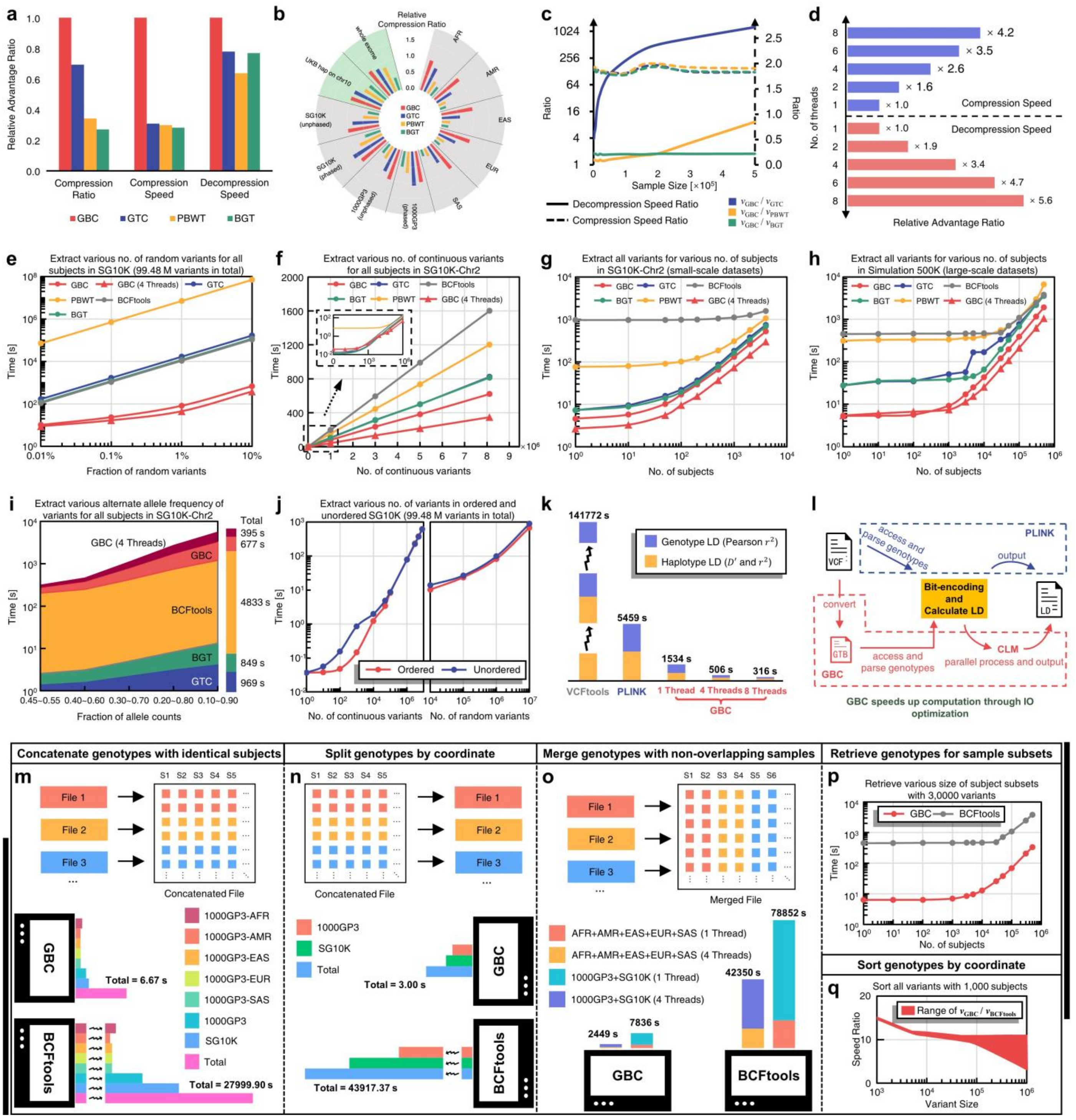

Fig.1 | The comparison of performance among alternative methods (details in Table S5~Table S11). a, The basic performance of different methods on the 1000GP3-Population, 1000GP3 and SG10K datasets. **b**, The compression ratios of GBC and other alternative methods on multiple typical open datasets. **c**, The ratio of compression speed (solid line) and decompression speed (dotted line) of GBC and other alternative methods with the increase of sample size on simulated datasets (Variants number = 30000). **d**, The significant improvement of compression and decompression speed under multi-threads on the 1000GP3 dataset. **e**, Retrieved the genotypes of random variants for all the subjects on the SG10K dataset. This function of random access is only supported by BCFtools and GBC. Thus, the time cost was estimated by the access time of individual sites for methods including GTC, PBWT and BGT. **f**, Retrieved a range of continuous variants for all the subjects on SG10K-chr2 (the genotypes on chromosome 2 of SG10K dataset) dataset. The subgraph showed that when querying the small number of variants, all the methods require very little time. **g**, Retrieved all the variants for a specified subset of subjects on the SG10K-chr2 dataset. **h**, Retrieved all the variants for a specified subset of subjects on Simulation 500K dataset. **i**, Filtered out the variants by alternative allele frequency on the SG10K-chr2 dataset. **j**, Retrieved continuous variants and random variants in ordered and unordered SG10K dataset separately. **k**, The comparison of LD coefficients computational speed between GBC and other popular tools on the 1000GP3 dataset. **l**, GBC speeds up downstream computation (calculating the pair-wise linkage disequilibrium coefficients as an example) through IO optimization. **m**, Concatenate the chromosome-separated files within each dataset. **n**, Split compressed archives by chromosome. **o**, Merge multiple compressed archives with non-overlapping subjects. **p**, Retrieve all the genotypes for a specified subset of subjects and rebuilt the compressed archives. We tested the time cost of fetching different sizes of subject subsets on the Simulation 500K dataset. **q**, Sort the variants by coordinate. We used several disordered simulated data with 1,000 subjects for evaluating the time cost and measured the range of speed ratio of GBC to BCFtools according to the disordered degree of datasets.

Online Method

Overview of the GBC compression procedure

Here, we proposed a general framework for genomic data compression as shown in Fig.S3 and GBC is designed as a typical instance of such framework in a highly scalable way. At the beginning of compression, the input VCF file is divided into several chunks, which can be processed in parallel with multiple threads. Under each thread, every chunk is further divided into numerous smaller blocks containing plenty of variants (Fig.S3a). Here, a block is the smallest unit of compression in our algorithm. Then, the genotypes of the variants are encoded into byte codes (Fig.S3b), and the byte codes of biallelic variants are further combined into one-byte codes by combining three or four consecutive ones (Fig.S3c). Next, the approximate minimum discrepancy ordering (AMDO) algorithm is applied on the variant level (Fig.S3d) to sort the variants with similar genotype distributions together for improving the compression ratio. Finally, the ZSTD algorithm is adopted to compress the sorted data in each block (Fig.S3e). Finally, all the compressed blocks and metadata are written into a single structure GTB (Fig.S3f). GBC compression algorithm has a linear complexity property regarding the number of subjects and variants with little memory usage (less than 4 GB), which provides almost the best compression ratio to date. Built on the unique GTB formats and advanced algorithms, GBC is extremely friendly to projects with very large sample sizes (millions or tens of millions). Besides, its supporting functions, including transferring, extracting, concatenating, merging and splitting genotypes, are much faster and more convenient than alternative tools.

Input VCF files for compression

GenoType Blocking Compressor(GBC) helps compress genotypes in VCF files. One or more input files are allowed. For a single input file, GBC reads the file directly. For multiple files inputs, GBC will first figure out the main file with the largest samples and build the sample primary indexes. Other input files will be handled in turns after matching the sample indexes of the main file. In the matching process, the genotypes of missing subjects will be replaced by '.|.' subsequently, ensuring that all the input files can be compressed together accurately in subsequent steps.

Chunking inputs and blocking each chunk by variants

The processed VCF file is sliced into t chunks (if multi-threading mode is enabled and t threads are specified) with approximate sizes according to the physical size of the input file. Under each thread, the read-in variants of each chunk are divided into several blocks of consecutive entries. The amounts of variants N in each block is set automatically, which is mainly dependent on the subject size M as $N \in \{2^n | 2^n \times M \leq 2^{31} - 2, 7 \leq n \leq 14 \text{ and } n \in \mathbb{N}^*\}$. GBC stores 128 variants of 16,777,215 subjects or 16384 variants of 131,071 subjects at most. Briefly, one block is terminated if (a) the number of variants in the block reaches the maximum; (b) an inconsistent chromosome is scanned; (c)it is at the end of a file. The blocks from different chunks are parallelly processed independently under the multithreaded mode, improving compression speed. This strategy makes it possible for the personal laptop to deal with large datasets conveniently because these blocks occupy small memory during the operation. Besides, we have adopted nine commonly used quality control strategies (described in KGGSeq[10]) in GBC to filter out genotypes and variants with poor quality. The QC is executed by default after read-in, and only genotypes and variants that pass the quality-control are assigned into the blocks. The blocking strategy also allows the decompressing algorithm to only process a small fraction of all the data for queries (fast random access of the compressed data).

Maximized byte-encoded of genotype (MBEG)

Conventional genotype array storage in text format (e.g., 0|1) takes up much redundant information space. Therefore, a more efficient and acceptable strategy is needed to encode the genotypes with less space to enhances the information density (containing as much information as possible in the same byte length). Here, we propose a novel byte-based lossless encoding strategy to store genotypes, which helps reduce memory burden and accelerate the compression process. It has two steps. However, Step 2 is specifically designed for biallelic genotypes.

Step 1: A byte-based encoding strategy of genotype (byte-encoded genotype, BEG, Fig.S1b) is proposed when dealing with the genotype. We define n_v as the number of alleles at a given variant v, and encoders with a different base will be used to encode the genotype of a variant according to n_v. In detail, when $n_v = 2$, a biallelic encoder will work; For multiallelic variants ($n_v \in [3, 15]$), a multiallelic encoder is used. Therefore, for a given variant v, the non-missing phased genotype '$a \mid b$' will be encoded as:

$$a \mid b \to \begin{cases} (a+1)^2 - b & ,a \geq b \\ b^2 + a + 1 & ,a < b \end{cases} \tag{1}$$

Besides, for a missing genotype '.|.', it is encoded to 0. For an unphased genotype 'a/b', it is encoded as above after being transformed to '$\min\{a,b\} \mid \max\{a,b\}$'. For the sake of unification, a genotype 'a' of a variant in a male's chromosome X and Y is converted into '$a \mid a$' and is encoded by the above formula. Note that the strategy ensures the coherent genotype encoding values with different allele numbers.

Step 2: However, the BEG array (BEGs) will still be redundant and may reduce the compression efficiency in a project with large samples. In consideration that a biallelic variant has 4(unphased)~5(phased) possible genotypes and the most majority of variants in human genomes are biallelic, we further combine 3(phased)~4(unphased) multiple consecutive BEG codes into a single byte for biallelic variants as follow:

$$[\mathrm{BEG}_0, \mathrm{BEG}_1, \mathrm{BEG}_2] \to 5^2 \cdot \mathrm{BEG}_0 + 5 \cdot \mathrm{BEG}_1 + \mathrm{BEG}_2 \tag{2}$$

$$[\mathrm{BEG}_0, \mathrm{BEG}_1, \mathrm{BEG}_2, \mathrm{BEG}_3] \to 4^3 \cdot \mathrm{BEG}_0 + 4^2 \cdot \mathrm{BEG}_1 + 4 \cdot \mathrm{BEG}_2 + 4 \cdot \mathrm{BEG}_3 \tag{3}$$

Conversion (2) and (3) are designed for phased and unphased BEGs, respectively. If the number of BEG for a variant is not enough to make a group of three or four, the inadequate part is called incomplete MBEG (e.g., one variant of 1000 subjects with phased genotypes will form 334 groups, and two nulls to make up the last group) the null BEG_i will be set to the same as the previous one according to $\mathrm{BEG}_i = \mathrm{BEG}_{i-1}$ (if $i \geq 1$). In comparison, the storage space with the maximized byte-encoded genotype array (MBEGs) will be reduced by nearly 11/12 or 15/16, compared to the common text format in VCF. Finally, BEG and MBEG can be pre-calculated and stored in an encoding table with $o(1)$ access time. The encoding table can be loaded in memory for a fast encoding or decoding by direct mapping without computing. For instance, the encoding value of genotype $a \mid b$ is the element in row a and column b of the encoding table.

Approximate minimum discrepancy ordering of variants (AMDO)

For a given variant, the diverse genotype (especially for the non-conservative region) will result in alternating light and dark bands of genotype distribution (Fig.S3d), and similar contiguous sequences may be distributed at rare variants with close physical coordinates.

Thus, we propose an algorithm named Approximate Minimum Discrepancy Ordering (AMDO), which considers both the allele frequency and genotype distribution in the sequence. AMDO provides an $o(nm)$ time complexity in the fast sorting process (compared to GTC with at least $o(nm^2)$ time complexity), and significantly improves the compression ratio.

AMDO starts with capturing the genotype accumulated down-sampling features. Supposing that each block contains M variants and N subjects, a zero-count matrix is denoted as $C = [c_{mn}]_{M\times N}$, where c_{mn} is the count of REF alleles (namely 0 alleles) of the m^{th} variant for the n^{th} subject. Then, the sequence of a variant m $C_m = [c_{m0}, c_{m1}, \cdots, c_{m(N-1)}]$ is merged into a shorter s-element vector,

$$C_m^{(l)} = \left[C_{m,0}^{(l)}, C_{m,1}^{(l)}, \cdots, C_{m,s-1}^{(l)}\right] \tag{4}$$

where $C_{m,i}^{(l)}$ covers a maximum of $l = \lceil N/s \rceil$ consecutive genotypes and s is 24 by default. Each element $C_{m,i}^{(l)}$ in the vector is defined as an accumulated count, i.e. :

$$C_{m,i}^{(l)} = \sum_{j=i\cdot l}^{\min\{N-1,(i+1)l-1\}} \sum_{k=i\cdot l}^{j} c_{mk} = \begin{cases} \sum_{j=i\cdot l}^{(i+1)l-1} \big((i+1)l - j\big)c_{mj} & ,i < s-1 \\ \sum_{j=i\cdot l}^{N-1} (N-j)c_{mj} & ,i = s-1 \end{cases} \tag{5}$$

The accumulation helps discriminate genotype distribution effectively. For example, according to (5), two zero-count vectors [0,1,1,2] and [2,1,1,0] have different accumulated counts 7 and 9, respectively, although they have the same alternative allele counts.

All the variants in a block are divided into two groups, i.e., the biallelic and the multiallelic groups. In the biallelic variants group, the order of the variant v_i and the biallelic variant v_j is defined as the dictionary order of $C_i^{(l)}$ and $C_j^{(l)}$, which are described as below:

- If $\exists k_0 \in [0, s-1], \forall k \in [0, k_0 - 1]$, such that $C_{i,k_0}^{(l)} < C_{j,k_0}^{(l)}$, $C_{i,k}^{(l)} = C_{j,k}^{(l)}$, then $v_i > v_j$;
- If $\exists k_0 \in [0, s-1], \forall k \in [0, k_0 - 1]$, such that $C_{i,k_0}^{(l)} > C_{j,k_0}^{(l)}$, $C_{i,k}^{(l)} = C_{j,k}^{(l)}$, then $v_i < v_j$;
- If $\forall k \in [0, s-1]$, such that $C_{i,k}^{(l)} = C_{j,k}^{(l)}$, then $v_i = v_j$.

On the contrary, the order will be inverted for multiallelic variants, which helps maximize the length of similar sequences. After re-ordering, the corresponding information of positions, alleles and MBEGs for variants are sorted according to the ordered variants ($I = [m_0, m_1, \cdots, m_{M-1}]$).

Merge data stream after compressing with advanced compressors

All the sorted MBEG codes in each block are further compressed by advanced compressors. Popular compression algorithms (e.g., Gzip, LZMA and zlib) can achieve a compression ratio of 100x or more on genotypes. We chose the ZSTD (short for Zstandard[15]) by default because it provides the fastest speed with a similar compression ratio among the widely used compression algorithms. In detail, the MBEG codes of each variant are concatenated into a byte array B_1 directly. Next, the position of each variant is converted into 4 bytes, and then all variants' positions are concatenated into a byte array B_2. Finally, the alleles of all variants are concatenated into another byte array B_3 with a '/' delimiter. Then, these concatenated data (B_1, B_2 and B_3) are compressed by the latest ZSTD respectively. Each packed block has two types of information, including abstract information and data entity, subsequently written to the GTB file. Abstract information summaries a block, containing the chromosome number (1 byte), minimum and maximum positions

(4 bytes respectively), number of biallelic variants (2 bytes), number of multiallelic variants (2 bytes), length of compressed positions, alleles and genotypes data. Besides, the data entity is a long vector composed of three sections of compressed data, including encoded genotypes, positions, alleles.

GBC can be integrated with different compression algorithms. At present, ZSTD and LZMA algorithms have been embedded to compress each independent GTBs. We also reserve two types of compressors for other developers to extend.

A novel uniform structure for storing the compressed data

The Genotype Block format (Fig.S1a and Supplementary Note 1) is elaborately designed to store the compressed genotype data and auxiliary meta information. Here is a simple process for generating a GTB file. GBC first writes 5 bytes of placeholder information for Magic Code and Block Numbers at the beginning of compression. The Reference Information and Subjects Information are added subsequently. Next, the data entity and abstract information for each block are written to the disk and memory synchronized to ensure that the order of records in memory is the same as when data is written to the file in the hard disk. After all the blocks are compressed, all the abstract information in the memory is written to the end of the GTB file. Finally, the file pointer returns to the head of the file, and GTB modifies the placeholder information according to the compression results, including block size, compressor parameters, the state of genotype and whether the file is in order.

GTB is a flexible structure with multiple independent compression submodules (or blocks), which facilitates the management of large genotype data. The file merging and splitting can be finished in seconds based on these independent modules because decompression is unnecessary. This is particularly important for large-scale genotype datasets because the submodules facilitate fast data modification and access locally. These independent and small blocks also make GBC a highly parallelizable tool. Its multi-threading model further improves the data process speed substantially when compared with the alternative tools. Notably, especially for decompression, GBC is the first algorithm that supports the parallel extraction of genotype data, mainly due to the independent compression submodules.

A two-level index table for fast access and management

For accessing or managing a GTB file, the block summaries and variant coordinates in the GTB file are loaded into the memory to construct the first-level index table (a GTBTree) instantly. In detail, a GTBNode is built from the abstract information and the range of entity positions of a block (calculated by accumulating previously lengths of compressed positions, alleles and genotypes data). Then, all GTBNodes of the same chromosome are grouped. Finally, the GTBNodes of all chromosomes constitute the GTBTree. Due to the blocks are out-of-order after compression, the GTBTree is sorted according to the chromosome, minimum position and maximum position of each block. The address of a variant is determined jointly by three indexes: the GTB file's id (if merged from multiple GTB datasets), GTBNode and the index within the block. The coordinates of all variants form the second-level index table. Therefore, searching for a variant with a given position can be done by looking up the GTBNodes based on the boundary coordinates of the blocks at first. Then, the candidate GTBNode can decompress the position data to verify its location within a block. Noticeably, the two-level index table enables fast data access even if the original file is out of order. Also, file sorting and multi-files merging based on the two-level index table can be done in parallel with an extremely low memory burden (Supplementary Note 2).

The addressing algorithm to access MBEGs

The variant-centric data-organization strategy helps query all information within a specified range of variant positions, but it works relatively poorly on querying variants information of specified subjects. Thus, we develop an addressing algorithm to helps quickly access a genotype from the MBEGs directly, which has two steps: (1) finding the start pointer of the m^{th} variant; (2) decoding the genotype of the n^{th} subject.

In the MBEGs of a decompressed GTB block i, which contains $b_i^{\text{biallelic}}$ biallelic variants and $b_i^{\text{multiallelic}}$ multiallelic variants, the start pointer of the m^{th} variant in the MBEGs is calculated as:

$$P_m = \begin{cases} m \cdot \left\lceil \frac{N}{l} \right\rceil & , m < b_i^{\text{biallelic}} \\ b_i^{\text{biallelic}} \cdot \left\lceil \frac{N}{l} \right\rceil + \left(m - b_i^{\text{biallelic}}\right) \cdot N & , \text{else} \end{cases} \tag{6}$$

where l is dependent on the state of genotypes (phased: $l = 3$; unphased: $l = 4$), and N is the number of subjects.

The address of the n^{th} subject in the multiallelic encoding sequence is n, whereas in the biallelic encoding sequence, which is retrieved by triples (index, groupIndex, codeIndex), where:

- index: The index of the n^{th} subject in the GTB Subjects Information, that is n;
- groupIndex: The index of the MBEG code containing the genotype of n^{th} subject in the whole encoding sequence, calculated as: $\lfloor n/l \rfloor$;
- codeIndex: The index of the n^{th} subject in the MBEG code containing the genotype of n^{th} subject is calculated as: $n \,\%\, l$.

The biallelic genotype for the m^{th} variant of the n^{th} subject is stored in the $\text{codeIndex}^{\text{th}}$ BEG of the $(P_m + \text{groupIndex})^{\text{th}}$ MBEG. And the multiallelic genotype for the m^{th} variant of the n^{th} subject is stored in the $(P_m + n)^{\text{th}}$ BEG.

Memory control further improve the efficiency of GBC for large-scale data

Controlling memory usage is critical for large-scale projects, determining whether an ordinary user can easily access the genotype data. GBC overcomes the high memory load in processing large-scale datasets through three strategies. First, it has sufficiently reusable buffers (also called context structures). For example, once created, all computing activities can re-use the buffers within a task throughout the whole process. All the reusable buffers are destructed after finishing the process. Second, it can adaptively adjust the variant counts per block. The amounts of variants N contained in each block is set automatically according to the subject size M, which ensures that at most $2^{31} - 2$ genotypes (approximately 2 GB size) are included in each block. Third, it can estimate the compression size. The memory required by the compressor is estimated initially using the compression boundary estimation model. The details are as follows:

Suppose a byte array of length s, whose real and estimated compressed sizes are R_s and E_s respectively. For small-scale data (length < 4 KB.), the file head information (e.g., Hash Code, Magic Code, etc.) is larger than genotype data in the compressed data. Thus, we set a fixed upper boundary β, which indicates the minimum memory assigned for compression. For large-scale data, the main part of compressed data is the genotypes. A good compression algorithm should ensure that the compressed data will not exceed $\alpha(<\inf)$ times of the original size. Here, we aim to estimate the α, which affects the compression efficiency of large-scale data. As shown in the following:

$$E_s = \max\{\hat{\alpha} \cdot s, \hat{\beta}\} \tag{7}$$

In reality, the true value of α, β is unknown, so we estimated them by large-scale random data simulation experiments. The estimated value of the parameter is derived as follows:

$$\underset{\hat{\alpha},\hat{\beta}}{\arg\min}\, \mathbb{I}(E_s < R_s) + \hat{\alpha} + \frac{\hat{\beta}}{10 \times 1024^2} \tag{8}$$

For the compression of large-scale data, β can often be ignored. Thus, we defined $\hat{\beta} = 512, 1024, 1536, \cdots$, and the minimum $\hat{\alpha}$ is obtained with grid search. By estimation and calculation, the compression boundary estimation model of ZSTD is $E_s^{\mathrm{ZSTD}} = \max\{1.0014 \cdot s, 7168\}$, while LZMA is $E_\mathrm{s}^{\mathrm{LZMA}} = \max\{1.0167 \cdot s, 7680\}$ and Gzip is $E_\mathrm{s}^{\mathrm{Gzip}} = \max\{1.0031 \cdot s, 7680\}$.

Cyclic locking mechanism based parallel algorithms for compression and decompression

In GBC, we have designed powerful parallel compression and decompression algorithms, which outperform the existing methods for large-scale projects in terms of the time burden. During compression, GBC physically chunks the input files firstly, and each thread processes an independent chunk without interfering with each other. Each thread is responsible for processing one block, and the order problem is solved by a cyclic locking mechanism (CLM) for decompression in which the current thread holds the lock of the next thread. When a reusable thread finishes decompression, it stays in memory until the previous thread has released its lock. And the thread releases the lock of the next thread only when it has acquired the previous lock and finishes writing the current data to disk. In detail, a parallel decompression process with t threads, thread#0 holds the lock of thread#1, thread#1 holds the lock of thread#2, …, thread#$(t-1)$ holds the lock of thread#0. At the beginning of decompression, the lock of thread#0 is released. Then, any thread other than thread#0 cannot write to the disk because their previous threads do not release the lock. After thread#0 writes to the disk, it releases the lock of thread#1. The program continues this loop to decompress all the data.

CLM ensures that the decompressed data are output in the order of the GTB nodes. Inevitably, the slow disk I/O speed will limit the ability of the parallel decompression algorithm. Thus, GBC also provides a way to directly decompress the data to BGZ format, which improves parallel decompression speed. Based on CLM, we have also developed a java version of the parallel-bgzip compression algorithm, which has now been integrated into GBC as an auxiliary tool.

Evaluation and comparison

Three typical publicly available datasets were used for testing in this study, 1000GP3[7], SG10K[8], and UKBB[2]. Besides, simulated datasets (Supplementary Note 3) were also used to systematically investigate the speed among the tools with various sample sizes. All of these datasets were first compressed by GBC and then decompressed to vcf.gz format to exclude the interference of non-genotypic data.

For the comparison of basic compression performance, we mainly compared with PBWT, BGT, and GTC because they were designed with a similar purpose as GBC – compressing genotypes into an accessible format. The evaluation metrics include compression ratio, compression speed, decompression speed, and speed sensitivity with different sample sizes. We also added a comparison of BCFtools for the data query performance. The evaluation metrics include the speed of sample selection and variant selection (accessing random variants and contiguous variants, filtering by allele frequency). Note that these two subset selection methods can be used in combination. Finally, for the performance of managing files, both GBC and BCFtools are based on compressed datasets, which avoids the process of decompression and re-compression.

Data availability

In this study, partial genotype data from UK Biobank was accessed through a collaboration with application no.15422. Data are available for bona fide researchers upon application to the UK Biobank. The SG10K dataset was performed by Jun Li, a researcher with data resources and access permission to the dataset. The 1000GP3 dataset in this paper is shown in the https://pmglab.top/genotypes, and they do not require access rights.

Code availability

The underlying algorithm has been implemented in Java. The GBC software package and API functions are publicly available at http://pmglab.top/gbc, which can be easily integrated into other tools and sequencing projects.

References

[1]. Danecek, P., et al., The variant call format and VCFtools. Bioinformatics, 2011. 27(15): p. 2156-8.

[2]. Bycroft, C., et al., The UK Biobank resource with deep phenotyping and genomic data. Nature, 2018. 562(7726): p. 203-209.

[3]. Durbin, R., Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT). Bioinformatics, 2014. 30(9): p. 1266-72.

[4]. Li, H., BGT: efficient and flexible genotype query across many samples. Bioinformatics, 2016. 32(4): p. 590-2.

[5]. Layer, R.M., et al., Efficient genotype compression and analysis of large genetic-variation data sets. Nat Methods, 2016. 13(1): p. 63-5.

[6]. Danek, A. and S. Deorowicz, GTC: how to maintain huge genotype collections in a compressed form. Bioinformatics, 2018. 34(11): p. 1834-1840.

[7]. Auton, A., et al., A global reference for human genetic variation. Nature, 2015. 526(7571): p. 68-74.

[8]. Wu, D., et al., Large-Scale Whole-Genome Sequencing of Three Diverse Asian Populations in Singapore. Cell, 2019. 179(3): p. 736-749.e15.

[9]. Purcell, S., et al., PLINK: a tool set for whole-genome association and population-based linkage analyses. Am J Hum Genet, 2007. 81(3): p. 559-75.

[10]. Li, M., et al., Robust and rapid algorithms facilitate large-scale whole genome sequencing downstream analysis in an integrative framework. Nucleic Acids Res, 2017. 45(9): p. e75.

[11]. Danecek, P., et al., The variant call format and VCFtools. Bioinformatics, 2011. 27(15): p. 2156-2158.

[12]. Zhang, C., et al., PopLDdecay: a fast and effective tool for linkage disequilibrium decay analysis based on variant call format files. Bioinformatics, 2019. 35(10): p. 1786-1788.

[13]. Theodoris, C., et al., quickLD: An efficient software for linkage disequilibrium analyses. Mol Ecol Resour, 2021. 21(7): p. 2580-2587.

[14]. Danecek, P., et al., Twelve years of SAMtools and BCFtools. Gigascience, 2021. 10(2).

[15]. Collet Y, T.C., Smaller and faster data compression with Zstandard. 2016.

Acknowledgements

This work was funded by National Natural Science Foundation of China (31771401 and 31970650), National Key R&D Program of China (2018YFC0910500), Science and Technology Program of Guangzhou (201803010116), and Guangdong project (2017GC010644).

Author information

Affiliations

Zhongshan School of Medicine and The Fifth Affiliated Hospital, Sun Yat-sen University, Guangzhou, China

Liubin Zhang, Yangyang Yuan, Wenjie Peng, Bin Tang & Miaoxin Li

Center for Precision Medicine, Sun Yat-sen University, Guangzhou, China

Liubin Zhang, Yangyang Yuan, Wenjie Peng, Bin Tang & Miaoxin Li

Center for Disease Genome Research, Sun Yat-sen University, Guangzhou, China

Liubin Zhang, Yangyang Yuan, Wenjie Peng, Bin Tang & Miaoxin Li

The Province and Ministry Co-sponsored Collaborative Innovation Center for Medical Epigenetics, Tianjin Medical University, Tianjin, China.

Mulin Jun Li

Behavioral Health Services, Henry Ford Health System, Detroit, MI, USA

Hongsheng Gui

Center for Health Policy & Health Services Research, Henry Ford Health System, Detroit, MI, USA

Hongsheng Gui

Key Laboratory of Tropical Disease Control (SYSU), Ministry of Education, Guangzhou, China

Miaoxin Li

Contributions

M. L. and L. Z. conceived this project. M. L. supervised this research and provided hardware support. M. L., L. Z. and Y. Y. designed detailed implementations. L. Z., W. P. and B. T. developed and optimized the java code of GBC. L. Z., J. L. and H. G. conducted the experiments. M. L. and L. Z. directed the experiments and data analysis. L. Z. and Y. Y. wrote the user manual of GBC. M. L., L. Z. and Y. Y. wrote the manuscript. All authors read and provided feedback on the final manuscript.

Corresponding author

Correspondence to Miaoxin Li, Email: limiaoxin@mail.sysu.edu.cn.

Competing interests

The authors declare no competing interests.

Additional information

System requirements

GBC is developed based on Oracle JDK 8. It is available on any computer device that supports or is compatible with Oracle JDK 8. Users are required to download and install the Oracle JDK or Open JDK firstly.

Computing environment

Experiments for UKBB datasets were run on the following configuration: 106 GB memory, Intel(R) Xeon(R) CPU X5560 @ 2.80GHz 16 cores and a SSD with sequential write speeds of up to 111 MB/s and sequential read speeds of up to 115 MB/s. All Other experiments were run on following configuration: 32 GB 2933 MHz DDR4, Intel Core i7-10700 2.9 GHz 8 cores, and a NvMe SSD with sequential write speeds of up to 1950MB/s and sequential read speeds of up to 2400MB/s.

Figures

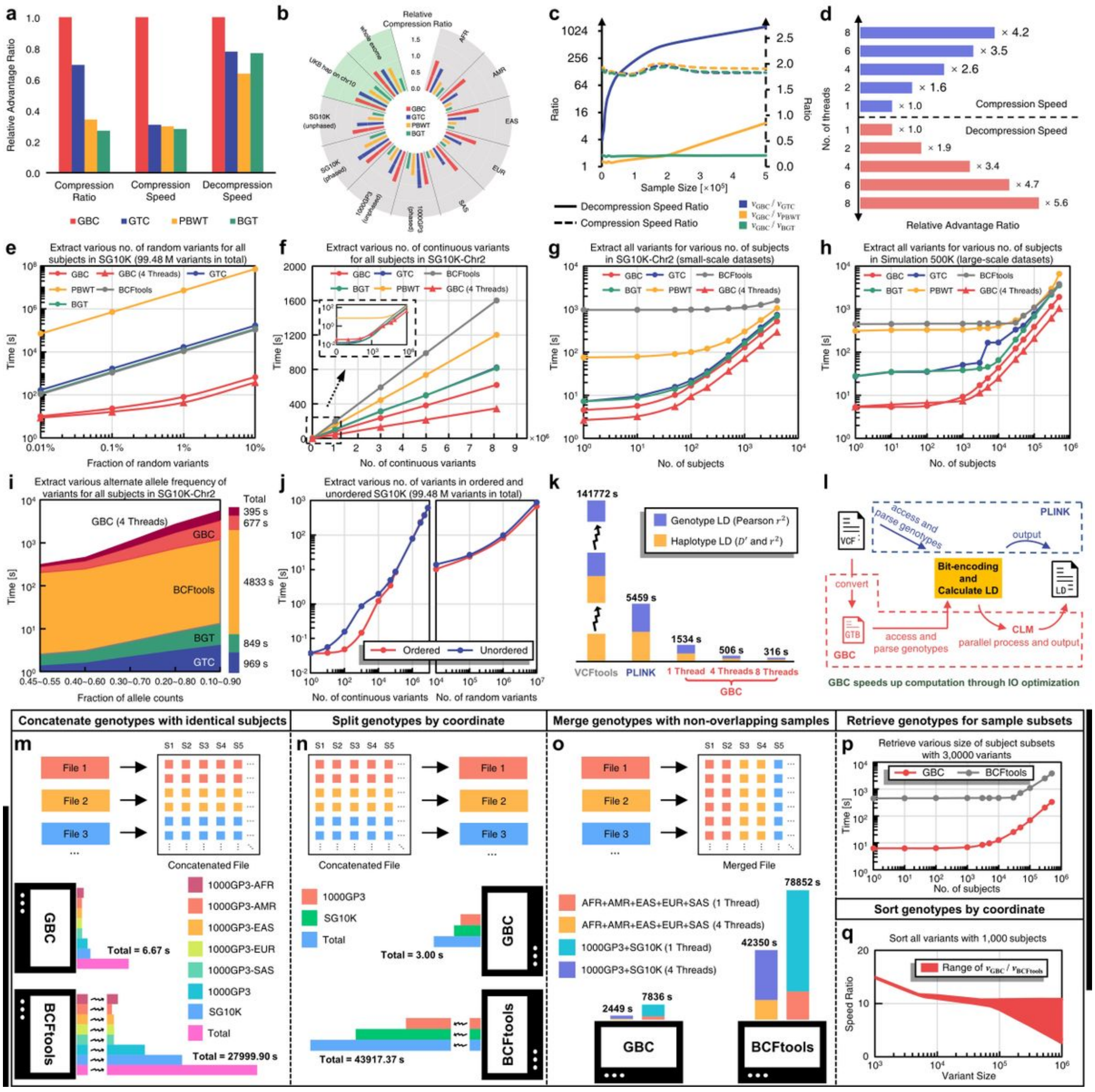

Figure 1

The comparison of performance among alternative methods (details in Table S5~Table S11). a, The basic performance of different methods on the 1000GP3-Population, 1000GP3 and SG10K datasets. b, The compression ratios of GBC and other alternative methods on multiple typical open datasets. c, The ratio of compression speed (solid line) and decompression speed (dotted line) of GBC and other alternative methods with the increase of sample size on simulated datasets (Variants number = 30000). d, The significant improvement of compression and decompression speed under multi-threads on the 1000GP3 dataset. e, Retrieved the genotypes of random variants for all the subjects on the SG10K dataset. This function of random access is only supported by BCFtools and GBC. Thus, the time cost was estimated by the access time of individual sites for methods including GTC, PBWT and BGT. f, Retrieved a range of continuous variants for all the subjects on SG10K-chr2 (the genotypes on chromosome 2 of SG10K dataset) dataset. The subgraph showed that when querying the small number of variants, all the methods require very little time. g, Retrieved all the variants for a specified subset of subjects on the SG10K-chr2 dataset. h, Retrieved all the variants for a specified subset of subjects on Simulation 500K dataset. i, Filtered out the variants by alternative allele frequency on the SG10K-chr2 dataset. j, Retrieved continuous variants and random variants in ordered and unordered SG10K dataset separately. k, The comparison of LD coefficients computational speed between GBC and other popular tools on the 1000GP3 dataset. l, GBC speeds up downstream computation (calculating the pair-wise linkage disequilibrium coefficients as an example) through IO optimization. m, Concatenate the chromosome-separated files within each dataset. n, Split compressed archives by chromosome. o, Merge multiple compressed archives with non-overlapping subjects. p, Retrieve all the genotypes for a specified subset of subjects and rebuilt the compressed archives. We tested the time cost of fetching different sizes of subject subsets on the Simulation 500K dataset. q, Sort the variants by coordinate. We used several disordered simulated data with 1,000 subjects for evaluating the time cost and measured the range of speed ratio of GBC to BCFtools according to the disordered degree of datasets.

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- SupplementaryMateria.submit.pdf
- MBEGIntroduction.pdf
- flatNATCOMPUTSCI210849Tspc.pdf
- flatNATCOMPUTSCI210849Tepc.pdf