Parallel Computing and Data Science Lab Technical Reports

Computer Science

Fall 2017

Scalable Data Structure to Compress Next-Generation Sequencing Files and its Application to Compressive Genomics

Sandino Vargas-Perez
WMU, sandinonarciso.vargasperez@wmich.edu

Fahad Saeed
Western Michigan University, fahad.saeed@wmich.edu

Scalable Data Structure to Compress Next-Generation Sequencing Files and its Application to Compressive Genomics

Sandino Vargas-Pérez[1] and Fahad Saeed*[,1,2]

[1]*Department of Computer Science, Western Michigan University*
[2]*Department of Electrical and Computer Engineering, Western Michigan University*

Abstract

It is now possible to compress and decompress large-scale Next-Generation Sequencing files taking advantage of high-performance computing techniques. To this end, we have recently introduced a scalable hybrid parallel algorithm, called *phyNGSC*, which allows fast compression as well as decompression of big FASTQ datasets using distributed and shared memory programming models via MPI and OpenMP. In this paper we present the design and implementation of a novel parallel data structure which lessens the dependency on decompression and facilitates the handling of DNA sequences in their compressed state using fine-grained decompression in a technique that is identified as *in compresso* data processing. Using our data structure compression and decompression throughputs of up to 8.71 GB/s and 10.12 GB/s were observed. Our proposed structure and methodology brings us one step closer to compressive genomics and sublinear analysis of big NGS datasets. The code for this implementation is available at https://github.com/pcdslab/PHYNGSD

1 Introduction

With current developments in computational technologies and the increase in communication and information sharing using the Internet, as well as interdisciplinary scientific research and collaboration, massive amounts of data that need to be stored, accessed, distributed, and processed in a timely manner are created. The biological sciences in particular and large-scale studies such as the Human Genome Project [1] [2] and the 1000 Genomes Project[1], alone are responsible for generating petabytes of raw information [3]. Considering the advances in Next-Generation Sequencing (NGS) technologies, the necessity for computing techniques able to handle large-scale genomic information becomes apparent.

Compression and decompression algorithms have become essential tools for storing and distributing large-scale genomic files generated by NGS machines. The use of general purpose compression algorithms, such as gzip, is more widespread than those specialized for NGS data. Some algorithms have been re-designed to efficiently use the computational powers of

*__Corresponding author e-mail__: fahad.saeed@wmich.edu
[1]http://www.internationalgenome.org, The International Genome Sample Resource

High Performance Computing (HPC) and parallel programing models and can achieve modest compression speeds, but their scalability deteriorates as more computational resources are used.

Furthermore, after the process of compression, the data file needs to be decompressed in order to use its content, bringing us back to the issue of handling huge datasets with limited memory resources. A few NGS-specialized compression algorithms offer functionalities like retrieving records [4] [5] from the compressed dataset, but they do not allow the manipulation of data in its compressed form. Hence, better compressive genomic techniques that allow sublinear analysis of the data are required.

In the following sections of this paper we discuss the data structure of the resulting compressed file, analyze the design of *phyNGSC* [6] for decompression, and offer a discussion on compressed data manipulation to lessen the dependency on decompression. We evaluate the performance of *phyNGSC*, presenting comparisons against existing sequential and parallel compression solutions.

2 Background Information

The authors presented a hybrid parallel solution named phyNGSC [6] for the compression of NGS datasets utilizing MPI+OpenMP. In depth analysis of the compression portion of the algorithm was presented. Scalability was maintained by allowing the processes to write the compressed data as soon as it was ready to be written using a shared file pointer and a non-deterministic way of writing. Using a timestamp technique designed for the algorithm and named Timestamp-based Ordering for Concurrent Writing or TOC-W, the order of the written compressed data can be guaranteed. This makes the proposed parallel strategy highly scalable since it removes the bottleneck of ordered-concurrent writing in a shared file space. In this paper we discuss the strategy for decompression and the resulting data structure of the compressed file named NGSC, as well as the ability to manipulate data in its compressed form.

General purpose compression algorithms structure the data in blocks of compressed information [7] [8] [9], ranging from 100 and 900 kB in size for BZIP2 and its parallel derivate. NGS-specialized algorithms, such as DSRC [4] and DSRC2 [5], use a richer structure, which allows structural metadata[2] to be stored together with the actual compressed information. These structures have the ability of achieving better compression ratios and allow for the access of data in its compressed form.

Tools designed for the processing of FASTQ files that are able to work with compressed files are available. The generally used gzip compressor is the preferred format for tools like fqtools [10], which allows viewing FASTQ files, counting sequence reads, converting between FASTQ and FASTA formats, validating files, trimming sequences, and so on.

[2]Structural metadata refers to information about the location and composition of the compressed data within the file.

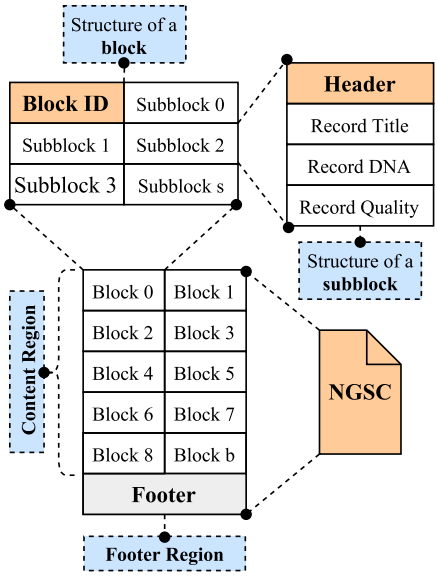


Figure 1: Data structure for the Next-Generation Sequence Compressed (NGSC) file.

3 NGSC File Structure

The use of non-deterministic writing creates the opportunity to design a new data structure to organize the compressed information, referred to as NGS Compressed (NGSC) file, shown in Fig. 1. The compressed data is organized in the content region into subblocks, which are grouped together to form blocks of equal size, 8 MiB[3] each. Structural metadata is organized into both a bit-packed footer (footer region) and bit-packed headers for each compressed subblock. The NGSC file structure facilitates the handling of DNA sequences in their compressed state by using fine-grained decompression in a technique that is identified as *in compresso* data processing.

3.1 Footer Region

The content of the footer region is bit-packed into a compact, easy to retrieve data structure. The last 2 bytes of the NGSC file represent the length in bytes of the footer region and it is used to exactly calculate its starting position within the compressed file. Table 1 shows the fields and number of bits used to encode them.

In order to succinctly bit-pack this information, we use 6 fields (BEPS, BEFS, BEBS, BESS, BELB, and LBES) to indicate the amount of bits used to encode the 6 major fields (PS, FS, BS, SS, CBO, and LBS). These fields are then used to bit-pack the information about the blocks in the content region using the subfields SBC, BCSS, and SBL. These first 6 fields will help to keep the footer size growth proportionally to the size of the FASTQ file, i.e., fewer bits will be required for smaller datasets compared to bigger datasets.

For the first 6 fields we always use 26 bits:

- BEPS, which represents the number of bits that are going to be used to encode the

[3] 8 MiB or mebibyte = 8×1024×1024 bytes

Table 1: Bit-packed footer region structure.

Field	# of bits	Description
BEPS	4	Bits Encoding Processes Size
BEFS	6	Bits Encoding File Size
BEBS	5	Bits Encoding Block Size
BESS	5	Bits Encoding SubBlock Size
LBES	1	Are Last Blocks of Equal Size?
BELB	5	Bits Encoding Last Blocks Size
PS	BEPS	Process Size
FS	BEFS	FAST File Size
BS	BEBS	Block Size
SS	BESS	Subblock Size
CBO	BS×BEPS	Concurrent Block Order
LBS	PS×BELB	Last Block Sizes
SBC	4	Subblock Count
BCSS	2	Block Contains Split Subblocks?
SBL	SBC×22	Subblocks Length

field PS. 4 bits are used to encode this field. The greatest value this field can hold is $2^4 - 1$.

- BEFS, which represents the number of bits required to encode the size of the FASTQ file. 6 bits are used to encode this field, holding a value of at most $2^6 - 1$.
- BEBS and BESS, which represent the number of bits required to encode the total number of blocks and subblocks in the NGSC file, respectively. 5 bits are used for each of the fields, allowing a maximum value of $2^5 - 1$.
- LBES, uses 1 bit to indicate if the last blocks written to the NGSC file are of the standard size of 8 MiB. A value of 0 in this field indicates that the last blocks written are of different sizes.
- BELB, if LBES has a value of 0, then this field will encode the number of bits use to encode the maximum size of the last blocks written to the NGSC file. If the value of LBES is 1, then this field won't be present in the footer region.

The 6 major fields are encoded using the amount of bits indicated by the first fields as follows:

- PS, which encodes the number of processes used for the compression of the FASTQ file. Since we are using BEPS bits to encode this field, the maximum value it can hold is $2^{(2^4-1)} - 1$, meaning that the algorithm can use a maximum of 32,767 processes to compress a FASTQ file.
- FS, which encodes the original size of the FASTQ file. Using BEFS bits for this field allows it to hold a maximum value of $2^{(2^6-1)}-1$, hence the maximum file size phyNGSC

can work with is 9 EiB[4] or approximately 9 exabytes.

- BS and SS, which encode the total number of blocks and subblocks in the content region of the NGSC file, respectively. These two fields use BEBS bits for BS and BESS bits for SS, allowing a maximum of $2^{(2^5-1)} - 1$ blocks and subblocks.
- CBO, which encodes the order in which the blocks were written to the NGSC file. It uses BEPS-1 bits to encode the id of BS blocks, for a total of BS×(BEPS-1) bits to encode the whole list.
- LBS, if LBES has a value of 0, then this field will contain the sizes of the last blocks written to the NGSC file, each encoded using BELB bits, for a total of PS×BELB bits. If the value of LBES is 1, then this list won't be present in the footer region.

The remaining fields of the footer have information about blocks and their subblocks. Since a block contains many subblocks and they can be split in order to truncate the size of a block, the remaining fields are encoded as shown below:

- SBC, for each block that a process wrote to the NGSC file, this field will contain the number of subblocks in them, encoding the value with 4 bits.
- BCSS, a 2 bits fields with four possible values:
  - 0: First and last subblocks in the block are not split.
  - 1: Last subblock in the block is split.
  - 2: First subblock in the block is split.
  - 3: First and last subblocks in the block are split.
- SBL, which encodes the size in bytes of each subblock in the block using 22 bits. This field requires SBC×22 bits to encode the size of all the block's subblocks.

Fig. 2a shows an example of an NGSC file footer for a FASTQ dataset of 100 MB. The file was compressed using 2 processes and created 4 blocks and 14 subblocks total. Each field contains the value (represented in base 10) that will allow the reconstruction of the original FASTQ file. In the example, the CBO field indicates that process p_1 wrote its first block, then p_0 wrote its first and second block, and then p_1 wrote its last block. Fig. 2b shows the actual representation of the footer in the NGSC file. Bold and highlighted numbers are used to indicate the different fields. Bytes are represented by groups of 8 bits (since this is the smaller unit for value representation). The 'X' character at the end of Fig. 2b indicates the number of bits wasted bit-packing the footer (1 bit in the example), in general it will be 7 bits or less.

3.2 Content Region

The content region of the NGSC file is formed by multiple blocks of compressed data. Each block contains a 2 bytes 'block id' to identify which process wrote it to the NGSC file and multiple subblocks. In order to truncate the size of a block to 8 MiB, the last subblock of a

[4] 9 EiB $= 8 \times 1024^6$ bytes

BEPS	*BEFS*	*BEBS*	*BESS*	*LBES*	*BELB*	*PS*	*FS*
2	27	3	4	0	23	2	113042902
BS	*SS*	*CBO*				*LBS*	
4	14	1	0	0	1	6329446	6409697
SBC	*BCSS*	*SBL*					
4	1	2213141		2178420		2166240	1830805
SBC	*BCSS*	*SBL*					
4	2	327215		2170630		2204791	1626808
SBC	*BCSS*	*SBL*					
4	1	2209411		2205828		2199836	1773531
SBC	*BCSS*	*SBL*					
4	2	428696		2182692		2190193	1608114

(a) Field values of the footer.

```
00100110 11000110 01000101 11110110 10111100
11100101 11010110 10011101 00111000 00100101
00011001 10110000 11100110 11110000 10100011
00001110 00101000 10101100 00100111 10101110
10010000 10000110 11110000 00110111 11011111
00101010 10010000 10011111 11000101 11110001
10101000 10000100 10000110 10010001 11011101
10001101 00101011 10000100 01100001 10110110
10000011 10000100 11111000 00110100 00110010
00100011 10001101 10000111 11101101 10100100
00110100 01010100 11000100 00101001 11000100
10010000 10110101 10111000 10110110 00011111
1011011X
```

(b) Bit values of the footer (actual representation in NGSC).

Figure 2: Example of a footer for an NGSC file of a 100 MB.

block needs to be split if it doesn't fit. The remaining bytes of the split subblock are written to the next new block. The 'block ids' of the blocks containing a subblock that was split between the two need to be the same.

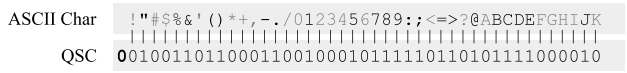
Subblocks are composed by a header and all the compressed titles, DNA sequences, and quality score lines of every record (Fig. 1). The header of a subblock contains multiple fields to help the decompression process, as shown in Table 2. These fields are:

- NR, which encodes the number of records compressed in the subblock. It uses 19 bits, which allows a maximum value of $2^{19} - 1$ records.
- MTL and MSL, which represent the maximum title and DNA sequence length, respectively, for records in the subblock. Their value is encoded using 8 bits, which allows for a maximum length of $2^8 - 1$.
- QSC, which uses 44 bits to encode the alphabet used for the quality score line of the records. The first bit will represent the base used for the quality score; 0 if base 33 and 1 if base 64 [11]. The remaining 43 bits will indicate if the character is present in the quality score by having its bit with a value of 1, or 0 if the character is not present, as shown in Fig. 3a and 3b.
- NF, which represent the total frequency of each nucleotide (A,C, G, and T) in the DNA sequence line of all records in the subblock. It uses 20 bits per nucleotide.

Table 2: Bit-packed subblock header structure.

Field	# of bits	Description
NR	19	Number of Records in subblock
MTL	8	Maximum Title Length
MSL	8	Maximum DNA Sequence Length
QSC	44	Quality Score Symbol Count
NF	20×4	Nucleotide Frequency
SOF	BEFS	Subblock Offset in FASTQ File
CTL	21	Compressed Title Length
CQL	21	Compressed Quality Score Length

- SOF, which encodes the offset in the FASTQ file from where the raw data was extracted. It uses BEFS bits to encode this value. Once the offset is decoded and the subblock data is decompressed, a process will know exactly where to write its content in the FASTQ file using explicit file pointers.
- CTL and CQL, which encode the length of compressed titles and quality scores. These fields are used to divide the decompression of the title, DNA sequences, and quality scores among multiple threads within the different processes, as will be explained in Section 4.



(a) QSC field for base 33 quality score.

(b) QSC field for base 64 quality score.

Figure 3: Example of QSC field for a subblock header in NGSC file.

3.3 *in compresso* Data Processing

The *in compresso* data processing technique is designed to handle the data in the NGSC format. *in compresso* uses a fine-grained decompression approach which consists of partially decompressing records to solve a particular task, without the need for full decompression. With this technique the processing of a compressed FASTQ file can be accelerated since the system will be handling fewer bytes than the original representation.

The data structure of NGSC allows for quick retrieval of certain statistical details of the FASTQ file by unpacking the footer and subblock headers, and gathering the metadata where this information is stored. Subblock header fields, such as NC, QSC, and NF help in the gathering of total number of records in the file, nucleotide frequency distribution,

and guessing the quality score encoding (for Solexa, Sanger, or Illumina encoding variants), respectively.

NGSC also helps with DNA sequence and pattern finding by identifying the sequences as they are decoded and discarding the ones that don't match the pattern searched. Since all the DNA sequences in a subblock are compressed as one single stream, once the pattern is found for the first time by any of the processes involved, the algorithm will stop and return the record information from where the pattern was found. In the worst case scenario, this process will take $\mathcal{O}(\frac{R}{p})$ if the DNA sequence is located in the last record of the last subblock being decompressed by p_i, or $\mathcal{O}(1)$ if the DNA sequence appears in the first record of the first subblock (best case).

The following list of tasks are possible with our *in compresso* data processing technique:

- FASTQ to FASTA format conversion.
- Basic FASTQ file statistics: nucleotide frequency distribution, number of records, quality score encoding.
- DNA sequence pattern finding.
- DNA sequence extraction and trimming by filtering sequences based on quality scores.

4 Methodology

In order to understand how the use of fine-grained decompression will help with *in compresso* data processing and manipulation, we will describe the methodology used by *phyNGSC* to achieve the decoding of the NGSC data structure to its original FASTQ file.

For our hybrid parallel implementation, an NGSC file of SB subblocks is partitioned equally among p homogeneous processes to be decompressed into the original FASTQ file. Decompression begins with each p reading the footer region (fr) of the NGSC file and bit-unpacking its content.

With the information of fr processed, p_i, $\forall i \in \mathbb{N} : 0 \leq i < p$, will calculate the number of subblocks that it will decompress and get their precise offset from where to start fetching in the content region (cr) of the file. p_i will have $D_i = \frac{SB}{p}$ subblocks to decompress, such that $0 \leq |D_i - D_j| \leq 1$, where $i \neq j$ and $\forall i\ j \in p$. The pseudo-code for the work distribution step is shown in Fig. 4. This load balancing procedure is used when the number of processes used for compression (C_P) is different than the number of processes used in decompression (D_P). When $C_P = D_P$, then each block with its subblocks will be assigned to a p for decompression, such that the 'block id' of a block match the rank (id) of the particular process p.

Decompression continues with p_i processing subblock's offset from D_i, one by one. A subblock s is read from the NGSC file and its header is bit-unpacked. With the information obtained from the header, p_i starts the decompression step by creating a threaded region with t threads. This is a "sections worksharing" region in which a thread t_h, $\forall h \in \mathbb{N} : 0 \leq h < t$ will be assigned the completion of one of the k tasks in a one-to-one relationship. This threaded region will decompress the titles, DNA, and quality scores of all the records in the block.

```
1: procedure WORK_DISTRIBUTION(p_i, p, SB, cr)
2:     |D_i| = ⌊SB/p⌋
3:     SB_counter = 0
4:     r = SB mod p
5:     x = p − r
6:     add_to_SB_s = 0
7:     add_to_SB_e = 0

8:     if r ≥ 1 and p_i ≥ x then
9:         add_to_SB_s = p_i − x
10:        add_to_SB_e = 1
11:    end if

12:    SB_s = p_i × |D_i| + add_to_start
13:    SB_e = SB_s + |D_i| − 1 + add_to_end

14:    for all s in SB do
15:        if SB_counter ≥ SB_s and SB_counter ≤ SB_e then
16:            add offset of s in cr to D_i
17:        end if
18:        SB_counter increments by 1
19:    end for
20: end procedure
```

Figure 4: Pseudo-code for *phyNGSC* compression using p processes and t threads per process and a FASTQ file.

A "loop worksharing" threaded region is created to copy all the decompressed records to the writing buffer (wb). In this threaded region all the R records are divided equally among the t threads using a for loop, such that t_h will copy $\frac{R}{t}$ records to wb. Once wb is ready to be written to the FASTQ file, p_i will utilize a non-blocking file writing function with explicit offsets, which will allow to write the current decompressed data in its original position on the FASTQ file, without waiting for the writing process to be done in order to continue. A sketch of the decompression work flow is presented in Fig. 5

4.1 Parallel I/O and Concurrency

To overcome the overhead of concurrently writing to a single disk, a strategy of stripping the datasets among different object storage targets (OST) was devised. Taking advantage of a Lustre parallel file system, both FASTQ and NGSC datasets were divided in chunks of 8 MiB, allowing for more concurrent access to the files, since a process p_i accessing a section of a dataset won't interfere with another process p_j accessing another section. This is because, in most cases, bytes being accessed by different processes are located in different OSTs.

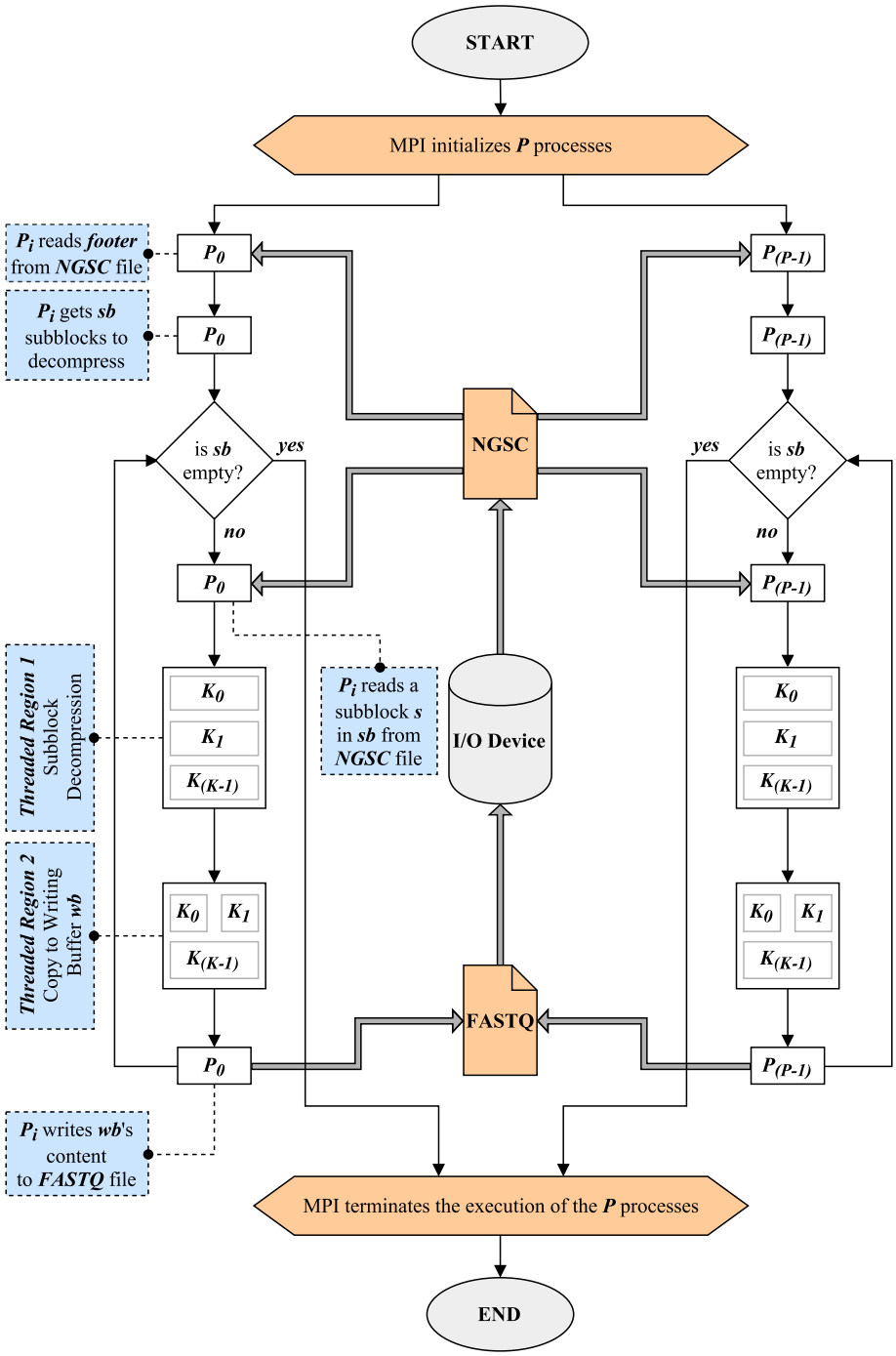


Figure 5: Sketch of the work flow of *phyNGSC* for decompression.

Table 3: Datasets used for the test. The fastq file SRR622457 was append nine times to itself to get 1TB.

IDs	Platform	Records	Size(GB)	Renamed
ERR005195	ILLUMINA	76,167	0.0093	10MB
ERR229788	ILLUMINA	889,789	0.113	100MB
ERR009075	ILLUMINA	8,261,260	1.01	1GB
ERR792488	—	90,441,151	15	10GB
SRR622457	ILLUMINA	478,941,257	107	100GB
SRR622457*	ILLUMINA	4,310,471,313	1,106	1TB

5 Implementation Results

The implementation of *phyNGSC* for decompression (and compression) was tested in SDSC Comet[5], a high performance compute and I/O cluster of the Extreme Science and Engineering Discovery Environment (XSEDE). Comet has 1944 compute nodes, each with two 12-core 2.5 GHz Intel Xeon E5-2680v3 (Haswell) processors and 128 GB of DDR4 DRAM. The network topology is a Hybrid Fat-Tree with 56 Gb/s (bidirectional) link bandwidth and 1.03-1.97 μs MPI latency. The cluster implements SLURM resource manager and has a Lustre-based parallel file system. C++ was used to code our implementation using the OpenMP library to provide multithread support and the MVAPICH2 implementation of MPI (version 2.1). The GNU Compiler Collection (version 4.9.2) was used for compilation.

NGSC datasets used for decompression where obtained by compressing the FASTQ files in Table 3. For decompression the label 10MB, 100MB, 1GB, 10GB, 100GB, and 1TB indicate an NGSC file resulting from compressing the FASTQ file of the corresponding size.

Fig. 6 shows the time to compress and decompress different dataset sizes using 2, 4, 8, 16, 32, 64, and 128 processes, and 12 threads per process. The average decompression time, measured as the wallclock time elapsed executing code between MPI initialization and finalization, decreases sharply as more processes are added with different NGSC datasets.

Compression times were improved by using the stripping methods established in Subsection 4.1. By implementing Lustre stripping, the time to compressed 1TB dataset was reduced from 8 minutes (no stripping) to 5 minutes, using 64 processes and 3.5 minutes using 128 processes. For the most part decompression is faster than compression, except for some datasets (100GB and 1TB) where decompression took more time. This is due to the content of the data. Datasets with larger DNA sequence reads require more time to decode.

Tables 4 and 5 show the average compression and decompression times of *phyNGSC* against DSRC sequential and DSRC 2, multithreaded. Each experiment was repeated several times and the average compression and decompression time was taken. The compression of a FASTQ file of size 1 terabyte took approximately 3.55 minutes, reporting a 90% time decrease against DSRC 2 with multithreaded parallelism and more than 98% time decrease for DSRC running sequential code—i.e., 10 to 50 times faster, respectively—improving the execution time proportionally to the added system resources.

[5]https://portal.xsede.org/sdsc-comet

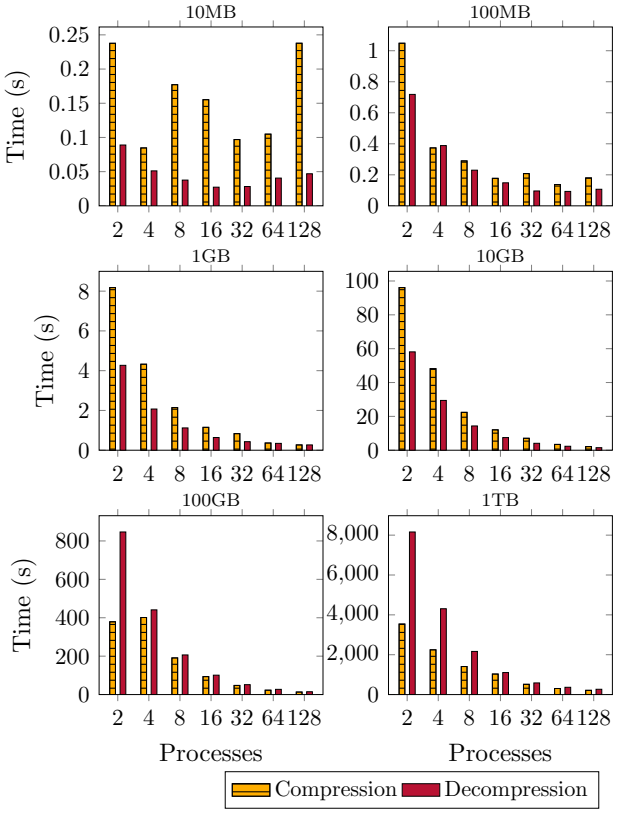


Figure 6: Compression and decompression times for *phyNGSC*, using 2, 4, 8, 16, 32, 64, and 128 processes and 12 threads per process.

Table 4: Compression Time for *phyNGSC* vs DSRC and DSRC2.

Datasets	**DSRC**	**DSRC 2**	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
10MB	0.15s.	0.29s.	0.24s.	0.08s.	0.18s.	0.16s.	0.10s.	0.10s.	0.24s.
100MB	1.69s.	1.06s.	1.05s.	0.37s.	0.29s.	0.18s.	0.21s.	0.14s.	0.18s.
1GB	18.01s.	2.07s.	8.18s.	4.32s.	2.14s.	1.15s.	0.83s.	0.36s.	0.27s.
10GB	3.73m.	36.514s.	1.60m.	48.19s.	22.38s.	12.05s.	7.05s.	3.43s.	2.15s.
100GB	21.00m.	6.59m.	6.32m.	6.69m.	3.18m.	1.56m.	47.54s.	22.76s.	13.12s.
1TB	3.06h.	36.76m.	58.97m.	37.43m.	23.46m.	17.16m.	8.59m.	5.09m.	3.55m.

Table 5: Decompression Time for *phyNGSC* vs DSRC and DSRC2.

Datasets	**DSRC**	**DSRC 2**	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
10M	0.13s.	0.109s.	0.09s.	0.05s.	0.04s.	0.03s.	0.03s.	0.04s.	0.05s.
100M	1.72s.	0.402s.	0.72s.	0.39s.	0.23s.	0.15s.	0.10s.	0.09s.	0.11s.
1G	10.9s.	3.674s.	4.27s.	2.07s.	1.12s.	0.64s.	0.43s.	0.34s.	0.27s.
10G	3.24m.	46.473s.	58.12s.	29.43s.	14.35s.	7.47s.	4.10s.	2.34s.	1.50s.
100G	24.75m.	6.92m.	14.11m.	7.35m.	3.44m.	1.68m.	51.56s.	27.19s.	14.73s.
1T	3.85h.	43.50m.	2.27h	1.20h.	36.09m.	18.42m.	9.75m.	6.10m.	4.45m.

Fig. 7 shows a comparison of the compression and decompression times obtained by running *phyNGSC* against MPIBZIP2, DSRC, and DSRC2. It can be observed that *phyNGSC* is very unbalanced when compressing the smaller dataset of 10MB, since the FASTQ file to be compressed is smaller than the chunks of raw data read from it (approximately 8 MiB). For larger datasets *phyNGSC* demonstrated very good scalability, both in compression and decompression. The other algorithms suffered from poor scalability when decompressing. MPIBZIP2 didn't complete the decompression of the 1TB dataset using more than 8 processes.

Experiments for the *in compresso* tasks of record counting, nucleotide frequencies, and DNA sequence finding were performed on the 1GB and 10GB datasets. Using 16 processes, information on the number of records was obtained 9.14x faster than having to decompress the whole 1GB file, and 5.75x faster for the 10GB. It is useful to compared parsing a FASTQ file to gather nucleotide frequencies to our approach to look for the subblock header field NF and reduce the information found by the processes involved. Our approach had the effect of 98.3% time reduction for the 1GB file and 95.1% for the 10GB file. To perform DNA sequence finding, we located a DNA sequence near the beginning and near the end of both files. The time taken to find the sequence near the beginning was very similar for both the 1GB and 10GB dataset. To find the sequences near the end, the time taken was always smaller than the time taken to decompress the entire dataset, indicating that the time to find a DNA sequence within the NGSC data structure will always have an upper bound less or equal to the time taken by decompressing the whole file.

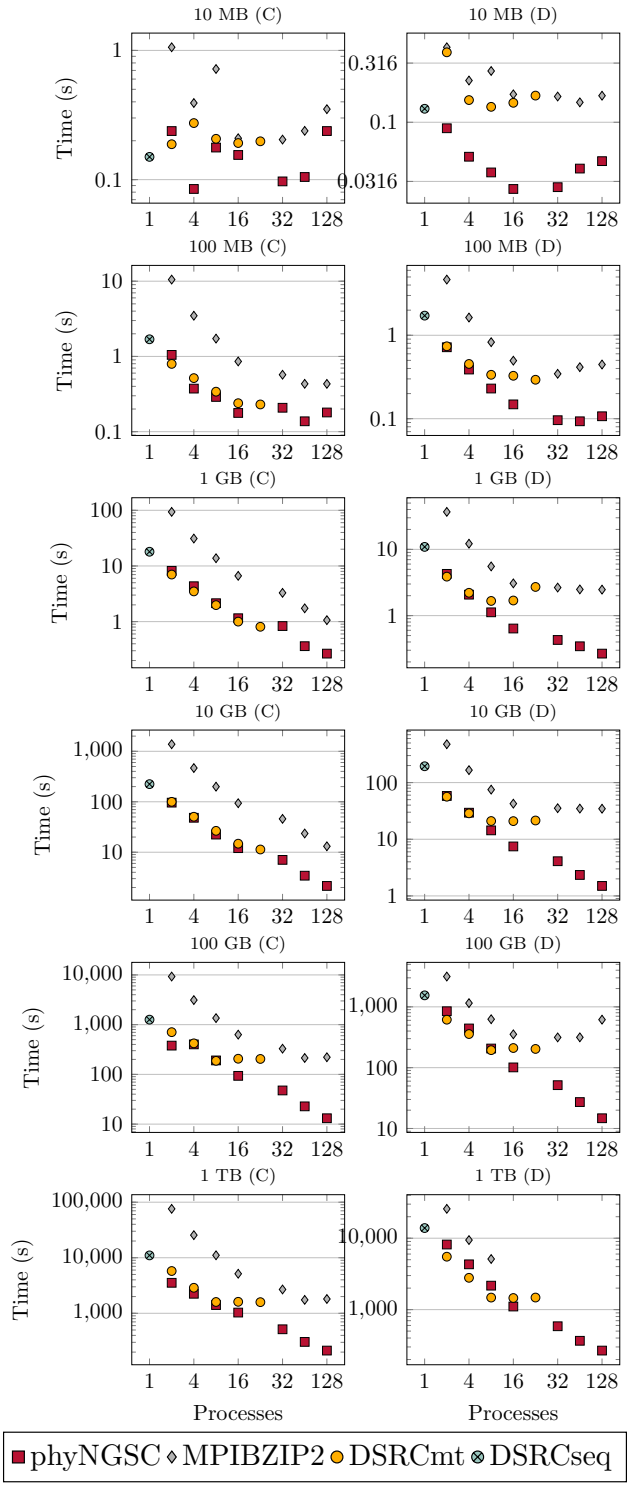


Figure 7: Compression and decompression times for different NGS datasets for DSRC sequential, DSRC multi-threaded, *phyNGSC* and MPIBZIP2 with 2, 4, 8, 16, and 32 processes.

6 Conclusion

In this paper we presented a decompression strategy for FASTQ files compressed using *phyNGSC* [6]. Compression and decompression were accelerated using techniques like Lustre's file stripping, timestamping with TOC-W, shared file pointer and non-deterministic writing, and a data structure that allows easy and efficient retrieval of the compressed information. The proposed NGSC file structure facilitates the handling of DNA sequences in their compressed state by using fine-grained decompression in a technique that is identified as *in compresso* data manipulation. The technique can be further extended to enrich compressive genomics and sub-linear analysis of big genomic data.

Experimental results from this research suggest strong scalability, with many datasets yielding super-linear speedups and constant efficiency. Performance measurements were executed to test the limitations imposed by Amdahl's Law when doubling the number of processing units. The compression of a FASTQ file of size 1 terabyte took less than 3.5 minutes, reporting a 90% time decrease against compression algorithms with multithreaded parallelism and a more than 98% time decrease for those running sequential code—i.e., 10 to 50 times faster, respectively—improving the execution time proportionally to the added system resources. Decompression for the same dataset using 128 processes and 12 threads per process took 4.45 minutes. Compression and decompression throughputs of up to 8.71 GB/s and 10.12 GB/s, respectively, were reported.

Findings of this research provide evidence that hybrid parallelism can also be implemented using CPU+GPU models for compressive genomic, potentially increasing the computational power and portability of the approach. This topic will be explored as future work. Additionally, our *in compresso* technique will be further expanded to analyze its functionality and efficiency when compared to tools in the market such as fqtools and others.

Acknowledgment

This material is based in part upon work supported by the National Science Foundation under Grant Numbers NSF CRII CCF-1464268 and NSF CAREER ACI-1651724. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] M. V. Olson, "The human genome project." *Proceedings of the National Academy of Sciences*, vol. 90, no. 10, pp. 4338–4344, 1993.

[2] F. S. Collins, M. Morgan, and A. Patrinos, "The human genome project: lessons from large-scale biology," *Science*, vol. 300, no. 5617, pp. 286–290, 2003.

[3] E. E. Schadt, M. D. Linderman, J. Sorenson, L. Lee, and G. P. Nolan, "Computational solutions to large-scale data management and analysis," *Nature reviews. Genetics*, vol. 11, no. 9, p. 647, 2010.

[4] S. Deorowicz and S. Grabowski, "Compression of DNA sequence reads in FASTQ format," *Bioinformatics*, vol. 27, no. 6, pp. 860–862, 2011.

[5] Ł. Roguski and S. Deorowicz, "DSRC 2industry-oriented compression of FASTQ files," *Bioinformatics*, vol. 30, no. 15, pp. 2213–2215, 2014.

[6] S. Vargas-Perez and F. Saeed, "A hybrid MPI-OpenMP strategy to speedup the compression of big next-generation sequencing datasets," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2017.

[7] J. Seward, "bzip2 and libbzip2, version 1.0. 5: A program and library for data compression," 2007. [Online]. Available: http://www.bzip.org

[8] J. Gilchrist. Parallel BZIP2 (PBZIP2) data compression software. HTML. [Online]. Available: http://compression.ca/pbzip2/

[9] ——. Parallel MPI BZIP2 (MPIBZIP2) data compression software. HTML. [Online]. Available: http://compression.ca/mpibzip2/

[10] A. P. Droop, "Fqtools: an efficient software suite for modern FASTQ file manipulation," *Bioinformatics*, vol. 32, no. 12, pp. 1883–1884, 2016.

[11] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice, "The sanger FASTQ file format for sequences with quality scores, and the solexa/illumina FASTQ variants," *Nucleic acids research*, vol. 38, no. 6, pp. 1767–1771, 2010.