RAIRO-Oper. Res. 50 (2016) 351–361
DOI: 10.1051/ro/2015039

**RAIRO Operations Research**
www.rairo-ro.org

# HIGH-ORDER STATISTICAL COMPRESSOR FOR LONG-TERM STORAGE OF DNA SEQUENCING DATA [*]

Marek Chlopkowski[1], Maciej Antczak[1], Michal Slusarczyk[1], Aleksander Wdowinski[1], Michal Zajaczkowski[1] and Marta Kasprzak[1,2]

**Abstract.** We present a specialized compressor designed for efficient data storage of FASTQ files produced by high-throughput DNA sequencers. Since the method has been optimized for compression quality, it is especially suitable for long-term storage and for genome research centers processing huge amount of data (counted in petabytes). The proposed compressor uses high-order statistical models for range encoding, similar to Markov models, but the whole input is considered in building a symbol context. Compression of DNA reads is performed according to LZ-style with the use of the 5–7th order model, while nucleotides' scores are encoded with the 3rd order model.

**Mathematics Subject Classification.** 68P20, 68P30, 68W32, 92D20.

Received September 8, 2015. Accepted September 21, 2015.

## 1. Introduction

In computer science, compression is a process of transforming an input data stream into an output stream of lower size by removing redundancy [18]. It is used for reducing costs of data storage and shortening data transmission time. Most known and often used measures, which characterize efficiency of compression algorithms, are compression speed and compression ratio. The former is defined as

$$CS = \frac{S_{in}}{t},$$

where $S_{in}$ is the size of the input data stream (in megabytes) and $t$ stands for overall processing time (in seconds). Compression ratio (also called quality) is measured as

$$CR = \frac{S_{out}}{S_{in}} \cdot 100\%,$$

where $S_{out}$ is the compressed stream size [17]. Optimization of one of these criteria usually leads to deterioration of the other. Most compression algorithms offer parametrization, where a user can choose a better ratio at the

*Keywords.* High-throughput DNA sequencing, data compression, FASTQ files.

[*] *The research has been supported by grant No. 2012/05/B/ST6/03026 from the National Science Centre, Poland, and by grant DS 09/91/DSPB/0570 from the Institute of Computing Science, Poznan University of Technology.*

[1] Institute of Computing Science, Poznan University of Technology, Piotrowo 2, 60-965 Poznan, Poland.
marta@cs.put.poznan.pl

[2] Institute of Bioorganic Chemistry, Polish Academy of Sciences, Poznan, Poland

expense of the compression time, or vice versa. Methods designed for long-term data storage can further optimize compression quality at cost of processing time, which is still cost efficient.

As DNA sequencing technology cost is getting lower, the amount of produced raw sequencing data is consequently growing. Nowadays, genome research centers are processing data counted in petabytes. Currently available assembly algorithms, which aim is to combine together high-throughput sequencing data in order to produce longer sequences (parts of a genome), often do not provide results of satisfying quality for data sets of a current size, or they may even be technically incapable to accept such sets at the input. Thus, it is a good idea to store raw sequencing data for further analysis with new, improved assembly software [7]. The available hardware infrastructure that stores such data is shrinking so fast, that there is a great need to develop high-quality compression algorithms that will allow to reduce significantly the dynamics of this process.

FASTQ format for storing sequencing information is commonly accepted and utilized by biological and bioinformatics communities. It is a standard for, among others, Illumina DNA sequencers. Nowadays, most of popular data warehouses that store DNA sequencing data use gzip compression format (*e.g.* ENA at the European Bioinformatics Institute [4], DDBJ [20], or 1000 Genomes [1]). Files compressed in this format are usually much larger than the ones compressed by a specialized software. Our aim is to provide a specialized algorithm that outperforms currently available programs in compression ratio, thus allows for significant reduction of data storage costs.

We present a new lossless compressor designed for long-term data storage of files in FASTQ format. Our method is optimized to provide the best compression quality. Its processing time actually cannot be satisfactory in comparison with others, but it pays off in long term data storage (as shown in Sect. 5). Moreover, the optimization of processing time is the subject of our ongoing research on deep parallelization of the algorithm. We compare our algorithm with the currently available software specialized for lossless compression of FASTQ files: DSRC [6,15], Quip [12] and SCALCE [8], and with general-purpose compressors, namely 7-zip, bzip2 and gzip.

The organization of the paper is as follows. In the next section the FASTQ format is described. The proposed algorithm is presented in Section 3. Computational tests are summarized in Section 4, while Section 5 contains analysis of storage costs estimated on the basis of these tests. Concluding remarks and the outline of future work are presented in Section 6.

## 2. General concept behind compression of sequencing data

FASTQ is a file format commonly used for storing data produced by high-throughput DNA sequencers. A file holds a set of records, one for each DNA sequence (called a *read*). Every record is composed of the following components [5]:

- sequence identifier – character '@' followed by unique sequence identifier and optional description,
- sequence of nucleotides – for DNA it is a sequence of characters ACGTN or acgtn (N stands for an unknown nucleotide); it can be written in a single or multiple lines,
- quality score tag – character '+', which can be optionally followed by a repeated sequence identifier,
- quality score – ASCII-encoded quality values obtained for every nucleotide of the sequence (encoding standard depends on a sequencing technology used); this can also be written in multiple lines.

According to Illumina's sequencing software documentation [10,11] all the aforementioned components are single lines (thus, a record always takes four lines) and the sequence identifier line consists of:

(1) character '@',
(2) instrument identifier,
(3) run number on the instrument,
(4) flowcell identifier,
(5) lane number,
(6) tile number,

Table 1. Compression of FASTQ files by 7-zip compressor, when a file is compressed as a whole or as a set of files corresponding to the separated components, *i.e.* 3 files containing identifiers, sequences and quality scores, respectively. The compression ratio is given in parentheses

| File name | Size [MB] | Compressed size [MB] for original file | Compressed size [MB] for separated components |
|---|---|---|---|
| SRR027520_1 | 4586 | 1307 (28.5%) | 1195 (26.1%) |
| SRR065390_1 | 8411 | 1792 (21.3%) | 1547 (18.4%) |
| ERR022075_1 | 6789 | 1160 (17.1%) | 946 (13.9%) |

(7) X coordinate of cluster,
(8) Y coordinate of cluster,
(9) read number (1/2 for paired-end experiment),
(10) information on filtering (Y/N),
(11) control number,
(12) barcode sequence (if used).

The read number is preceded by a space and the remaining fields (except the first two) are separated by colons. Fields 3, 5–9, and 11 are numerical. An example record is presented below.

```
@EAS139:136:FC706VJ:2:5:1000:12850 1:Y:18:ATCACG
AAGGCATTCGTAGAGAGATTTCCAACTTGAAAAAAA
+
BBBBCCCC?<A?BC?7@@???????DBBA@@@@A@@
```

Each of the components contains data of different characteristic, including different alphabets. Therefore, a natural concept is to compress each of them separately. Even a simple operation of writing each component to a different file skipping new line characters (but including separators) and then compressing them using a general-purpose data compressor, leads to the reduction of output size as shown in Table 1.

New compressors dedicated for DNA sequencing data (*e.g.* DSRC [6, 15], Quip [12], SCALCE [8]) exploit distinct algorithms for different components. This leads to further improvements in comparison to the simple transformation presented above.

## 3. The algorithm

There are two main approaches to data compression: statistical and dictionary ones. The goal of statistical methods is to assign possibly shortest codewords to most frequent symbols observed at the input data stream. The symbol frequencies are used to build a statistical model, which is next encoded with variable size codes (*i.e.* Huffman codes [9]) or arithmetic/range encoding. Both statistical model and encoding method affect the compression ratio. Dictionary compression allows for a further improvement of quality by replacing repeated substrings with dictionary tokens [17]. Depending on the implementation, a token can be a numerical index of a substring stored in the dictionary (see methods based on LZ78 algorithm [21, 24]), information about an earlier substring occurrence (*i.e.* distance-length pair used in LZ77-based methods [23]) or any other kind of data allowing decoder to identify a repeated substring. LZ78-based methods employ a specialized data structure (list, hash table, suffix tree) for the construction of the dictionary, which is also taken into consideration by a decoder. In LZ77 more sophisticated data structures for finding matching substrings are applied, but the decoder does not need to be aware of its details (since it gets the information on the distance in a stream).

In our work we apply high-order statistical models in conjunction with range encoding, this approach improves compression quality in comparison to the Huffman method [22]. The concept of statistical models is similar to Markov models [17], except that the symbol context is constructed on the basis of the whole input data stream. We also take advantage of the dictionary approach and a sorting-based algorithm for finding repeated information together with LZ77-based token encoding. Data encoding for statistical models is done with the use of range encoder implementation based on a published code [14, 22].

Our algorithm reads the whole input file three times. First, the preprocessing phase is performed for gathering basic information about the content of the file – recognizing alphabets and identifier patterns, storing records for finding LZ matches – that, among others, leads to saving space required by models. The second pass is for building optimal (static) statistical models based on global symbol frequencies. During the third reading the data encoding is performed on the basis of models constucted earlier (static models need to be stored within compressed data).

In the following subsections particular concepts integrated in the algorithm are described in details.

## 3.1. Sequence ID compression

As mentioned in Section 2, a sequence identifier line contains several fields and most of them are numerical. The order of those parts and their number depend on the machinery and software used to produce a FASTQ file. A natural concept would be to encode numerical fields as binary numbers (*i.e.* 32-bit integers). A further analysis shows that values of certain fields retain a particular pattern of modification. The read numbers (if included) and the pair of coordinates X,Y usually increase throughout all the file, where Y coordinate is reset to a lower value on every change of X.

Our algorithm parses every ID line and extracts a string pattern. Afterwards, the pattern consists of the same formatting characters as used in `sprintf` function of C programming language (with optional width specifiers). In practice, we usually get 1–4 patterns: from one pattern for not filtered single reads up to four different patterns in files containing paired-end reads with filtering information. For example, the identifier

`@ERR022075.1 EAS600_70:5:1:1158:949/1`

is converted into

`@ERR%.6ld.%ld EAS%ld_%ld:%ld:%ld:%ld:%ld/%ld`.

During the parsing procedure numerical field values are converted into binary numbers for further statistical compression with the 1st order statistical model. This is done by encoding a difference to the corresponding value from the preceding record. In most cases it allows us to obtain a significant gain in compression ratio. This method is able to compress the identifier part of a file to 2–4% of its original size. At the output, the identifiers are encoded as the pattern index and the number of numerical fields followed by field values. The decoder simply reads the information and performs `fprintf` function with field values as parameters.

## 3.2. Reads compression

For reads compression we apply a dictionary approach followed by statistical compression of the information about LZ matches and unmatched parts of reads. First, all reads and their reverse complementary equivalents are stored in memory (during the first reading of the input file) with additional information about corresponding record index. The reverse complements are two DNA strings, which can be mutually transformed by reading from the end and translating A to T (and *vice versa*) and C to G (and *vice versa*), for example AGGTAG and CTACCT. In fact, a read in a FASTQ file may represent its reverse complement as well, due to properties of the sequencing procedure used to produce the file. We included reverse complements in order to improve the results of the matching procedure. The routine creates pointer tables, one for every considered shift of input strings, which aim is to enable a fast comparison of their substrings. Entries in the tables point to the considered locations in the reads and their reverse complements, here the starting positions in strings belong to the interval $\langle 0; 14\rangle$ with step 2. The pointers in each table are sorted to get the lexicographical order of the substrings.

Afterwards, a set of longest common substrings is identified, at most one element (*match*) for each input record. When no match is found at a given record index, a whole read is compressed with the use of the 5–7th order statistical model (called *DNA model*). Otherwise, a special character is encoded on the basis of the DNA model, followed by match information encoded with different models (0 and 1st order), which is then followed by remaining characters from the particular read (if any). The match information is represented by the following components: the length of the common substring, the coincidence of directions of its appearances, the offset in the earlier read, and the distance between the reads. For example, the read

```
ACGTCGTTTGGCGCCCTTGGTCGATT
```

with the reverse complementary match of length 20 starting at the 3rd nucleotide will be encoded as

`AC` *0,1,20,4,15342* `GATT`

using a statistical encoder, where:

- '0' is a special character for indicating the match description,
- '1' means the match is found with a reverse complementary read ('0' otherwise),
- '20' stands for the length of the common substring,
- '4' represents the starting position (offset) in the earlier matched read,
- '15342' is the difference between indices of the current and matched reads (distance).

Since there is at most one match per record, the compression quality is improved when the distance is encoded as shown above, instead of encoding huge distances in bytes in a raw sequencing file.

The encoding stage is performed during the last reading iteration of the input file. The algorithm reads DNA sequences nucleotide by nucleotide and compresses every non-matched nucleotide with the DNA model. If a match is encountered, a series of additional models is being engaged. The special symbol '0' is still encoded with the DNA model. Every next field needs a separate model (see Fig. 1). The last field – the distance from the reference read – is encoded even with four independent models (*dist1*–*dist4*), one for each byte representing its value.

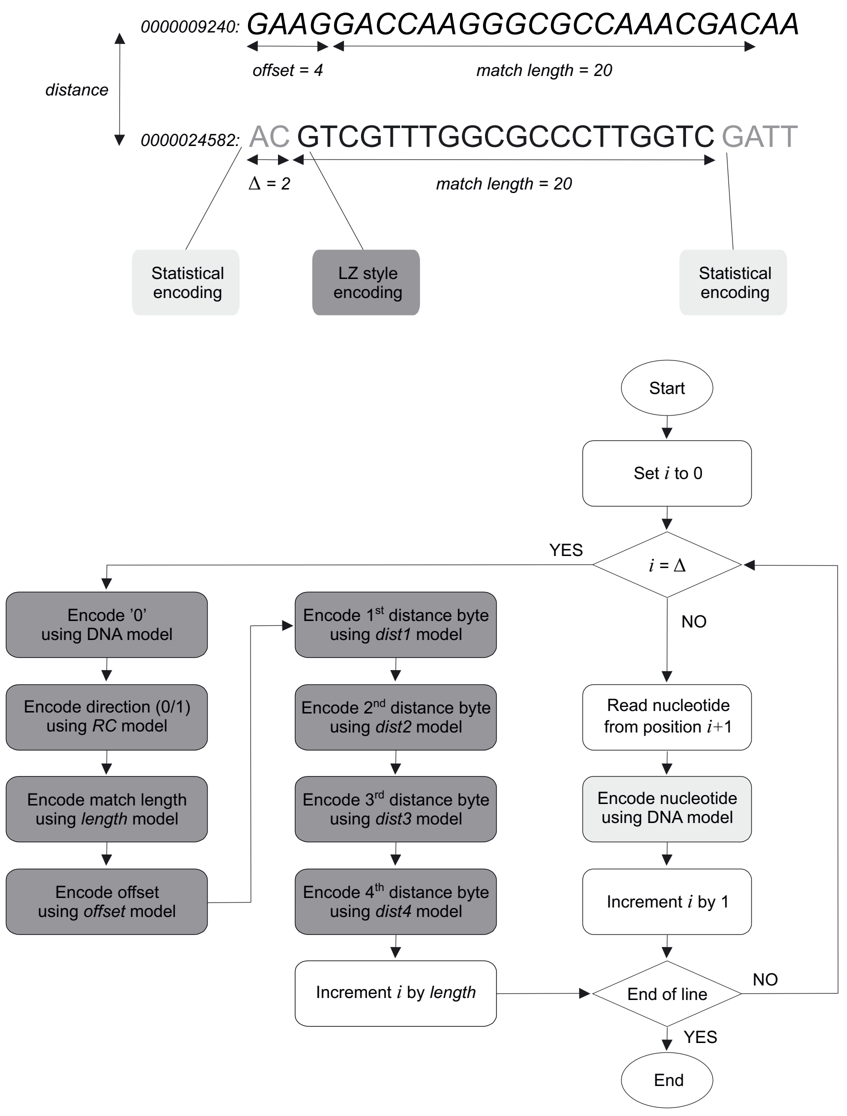
## 3.3. Score compression

The values representing quality scores also possess some properties, which, when rightly exploited, lead to improving compression quality. Actually, the quality score at $i$th position is usually close to value at position $i-1$ [13]. This observation is, among others, applied in Quip [12], where the differences between neighboring scores are encoded. We exploit this property by the use of the 3rd order statistical model for direct values, where probability of $i$th score depends on three preceding values (at positions $i-1$, $i-2$ and $i-3$) (see Fig. 2). This approach, realized by our implementation of the statistical models and the range encoder [22], allowed us to improve the results in comparison to the approach of encoding differences. As nucleotide quality scores usually degrade with growing indexes, the context of the statistical model is reset at the beginning of each line. This concept ensures that the probability of a value at the first position in a line is independent of values at the end of the preceding line.

## 4. Results

The proposed algorithm, FastQComp, was encoded in the standard C++ programming language (C++11) and compiled by GCC–4.9.2. Moreover, the source code management system Git was used to support non-linear team workflow.

In order to validate quality and time efficiency of our solution, we performed a computational experiment using publicly available files coming from six sequencing experiments as shown in Tables 2 and 3. Table 2 contains our basic dataset, while Table 3 presents an extension of the dataset where larger FASTQ files originated from other species are considered. The latter files were used in additional tests with specialized compressors only.



0000009240: GAAGGACCAAGGGCGCCAAACGACAA
offset = 4
match length = 20
distance
0000024582: AC GTCGTTTGGCGCCCTTGGTC GATT
Δ = 2
match length = 20
Statistical encoding
LZ style encoding
Statistical encoding
Start
Set i to 0
i = Δ
YES
NO
Encode '0' using DNA model
Encode direction (0/1) using RC model
Encode match length using length model
Encode offset using offset model
Encode 1st distance byte using dist1 model
Encode 2nd distance byte using dist2 model
Encode 3rd distance byte using dist3 model
Encode 4th distance byte using dist4 model
Increment i by length
Read nucleotide from position i+1
Encode nucleotide using DNA model
Increment i by 1
End of line
NO
YES
End

FIGURE 1. Encoding a read on the basis of the match information.

FastQComp was compared to other specialized FASTQ compressors: Quip [12], SCALCE [8], and the latest version of DSRC [15]. All programs were executed in the following modes for best possible quality of lossless compression:

- dsrc2 c -d3 -q2 -b1024
- quip -a
- scalce -bin -T8

It should be mentioned, that SCALCE does not offer quite lossless compression, because it discards additional information from identifier lines (a part after the gap) and does not preserve the original order of input records – the records are reordered to improve the compression ratio. Nevertheless, it is present here, because the remaining information is strictly encoded and the program is very efficient.

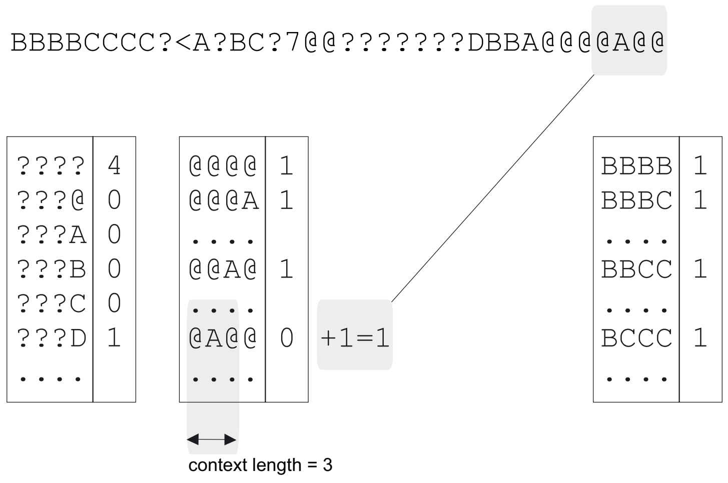


Figure 2. Counting scores for the 3rd order model construction.

Table 2. Dataset A: raw FASTQ files.

| File No. | File name | Source | Size [MB] |
|---|---|---|---|
| 1 | ERR022075_1 | Escherichia coli [DDBJ] | 6789 |
| 2 | ERR022075_2 | | 6876 |
| 3 | SRR065390_1 | Caenorhabditis elegans [ENA] | 8411 |
| 4 | SRR065390_2 | | 8411 |
| 5 | SRR359032_1 | Metagenomic DNA [DDBJ] | 5362 |
| 6 | SRR359032_2 | | 5362 |
| 7 | SRR027520_1 | Homo sapiens [1000 Genomes] | 4586 |
| 8 | SRR027520_2 | | 4586 |

Table 3. Dataset B: large raw FASTQ files.

| File No. | File name | Source | Size [MB] |
|---|---|---|---|
| 9 | ERR268195_1 | Picea abies [DDBJ] | 23 188 |
| 10 | ERR268198_1 | | 28 007 |
| 11 | ERR268210_1 | | 26 867 |
| 12 | SRR327341_1 | Mus musculus [DDBJ] | 27 438 |
| 13 | SRR328541_1 | | 16 685 |
| 14 | SRR329036_1 | | 17 562 |

In the comparison, general-purpose compressing tools have been also involved, since they are commonly used in practice by biologists and bioinformaticians. We considered the following tools: gzip v. 1.3.12, bzip2 v. 1.0.5, and 7-zip (64) v. 9.20, that were also executed with parameters allowing for best possible compression ratio. The tests were performed on Linux server with Intel Xeon X5670 CPU @2.93 GHz and 42 GB RAM.

The results of the comparison are presented in the following tables. Table 4 contains the values of compression ratio obtained for dataset A, where the overall ratio is calculated as the sum of the sizes of all compressed files

TABLE 4. Comparison of compression ratio (dataset A).

| File No. | FastQComp | SCALCE | Quip | DSRC 2.0 | 7-zip | bzip2 | gzip |
|---|---|---|---|---|---|---|---|
| 1 | **10.51%** | 11.09% | 11.52% | 16.94% | 17.09% | 22.62% | 27.38% |
| 2 | **11.60%** | 12.24% | 11.99% | 17.74% | 18.25% | 23.73% | 28.60% |
| 3 | **11.64%** | 13.27% | 16.00% | 17.03% | 21.31% | 22.50% | 27.18% |
| 4 | **12.20%** | 13.79% | 16.37% | 17.35% | 21.64% | 22.85% | 27.54% |
| 5 | **12.64%** | 13.02% | 12.99% | 17.89% | 18.57% | 24.09% | 28.86% |
| 6 | **13.20%** | 13.50% | 13.54% | 18.47% | 19.16% | 24.72% | 29.66% |
| 7 | 22.37% | 23.68% | **22.24%** | 22.82% | 28.51% | 29.25% | 34.75% |
| 8 | 22.83% | 24.19% | **22.73%** | 23.36% | 29.25% | 29.99% | 35.62% |
| Overall | **13.84%** | 14.86% | 15.51% | 18.52% | 21.24% | 24.45% | 29.36% |

TABLE 5. Comparison of compression time [s] (dataset A).

| File No. | FastQComp | SCALCE | Quip | DSRC 2.0 | 7-zip | bzip2 | gzip |
|---|---|---|---|---|---|---|---|
| 1 | 1135 | 169 | 86 | **62** | 2702 | 757 | 1531 |
| 2 | 1246 | 178 | 85 | **62** | 2817 | 780 | 1439 |
| 3 | 2449 | 279 | 95 | **87** | 3395 | 832 | 2398 |
| 4 | 2318 | 278 | **93** | **93** | 3667 | 804 | 2299 |
| 5 | 1089 | 157 | 459 | **73** | 2197 | 582 | 888 |
| 6 | 1127 | 161 | 460 | **69** | 2208 | 600 | 978 |
| 7 | 1996 | 193 | 337 | **80** | 2038 | 487 | 722 |
| 8 | 1960 | 196 | 348 | **78** | 2073 | 546 | 658 |
| Average | 1665 | 201 | 246 | **76** | 2637 | 674 | 1364 |

TABLE 6. Comparison of decompression time [s] (dataset A).

| File No. | FastQComp | SCALCE | Quip | DSRC 2.0 | 7-zip | bzip2 | gzip |
|---|---|---|---|---|---|---|---|
| 1 | 719 | 117 | 555 | 68 | 150 | 297 | **59** |
| 2 | 746 | 120 | 556 | 71 | 155 | 339 | **69** |
| 3 | 1024 | 163 | 702 | 84 | 194 | 373 | **78** |
| 4 | 1138 | 168 | 698 | 85 | 207 | 372 | **81** |
| 5 | 580 | 92 | 456 | 65 | 126 | 245 | **53** |
| 6 | 631 | 93 | 455 | 64 | 129 | 246 | **54** |
| 7 | 858 | 101 | 398 | 74 | 147 | 248 | **47** |
| 8 | 857 | 106 | 403 | 73 | 150 | 253 | **50** |
| Average | 819 | 120 | 528 | 73 | 157 | 297 | **61** |

divided by the sum of the sizes of all raw files. Processing time, counted as the wall time elapsed during computations, is shown in Tables 5 and 6.

FastQComp provides the best compression ratio for six greatest files, for two smallest ones Quip achieved slightly better outcomes. The latter files appeared to be less compressible regardless of a program used. The overall compression ratio falls below 14% for FastQComp only and it is over 1% higher compression quality than the one of SCALCE. Let us recall, that SCALCE does not offer a "truly" lossless compression and that favorizes it in terms of the output size. In practice, the results mean that using FastQComp to store this small

Table 7. Comparison of compression ratio (dataset B).

| File No. | FastQComp | SCALCE | Quip | DSRC 2.0 |
|---|---|---|---|---|
| 9 | **14.58%** | 14.89% | 15.19% | 16.62% |
| 10 | **14.94%** | 15.31% | 15.60% | 17.05% |
| 11 | **15.55%** | 15.93% | 16.14% | 17.60% |
| 12 | **7.81%** | 8.48% | 9.07% | 9.71% |
| 13 | **11.42%** | 11.56% | 15.59% | 16.92% |
| 14 | **12.14%** | 12.72% | 16.20% | 17.25% |
| Overall | **12.83%** | 13.24% | 14.43% | 15.65% |

set of FASTQ files leads to saving 0.5 GB of disk space in comparison to SCALCE and 0.8 GB comparing to Quip. In comparison to bzip2 and gzip, it is 5.2 and 7.6 GB of saved space for the considered files, respectively.

The compression time of our algorithm is significantly higher than the time needed by other FASTQ compressors, but comparable to processing times of the general-purpose ones. As one can see, the programs coped differently with particular files. Files 1–4 appeared to be much easier (in terms of time) for Quip than files 5–8, and the other way round for gzip. Although the average time obtained for gzip is lower than for FastQComp, the latter program processed files 1–2 faster. The results of decompression shown in Table 6 complete the comparison.

Considering decompression time, gzip worked most efficiently, but DSRC definitely wins the comparison of the total processing time. The difference between FastQComp and other specialized compressors, especially Quip, is now much more acceptable than the corresponding difference of compression times. Summing up the average compression and decompression times, FastQComp worked about three times longer than Quip. However, FastQComp is still cost-effective in terms of a long-term data storage, to which purpose it was designed. This aspect is discussed in Section 5.

Additional tests were performed for specialized compressors only, because the general-purpose ones proved to be far behind them (concerning compression ratio). Larger FASTQ files originated from other species (Tab. 3) were used in these tests, and the resulting compression ratio is presented in Table 7.

The ranking of the programs driven by the overall compression ratio is preserved with reference to the results shown in Table 4. Particular files differ in the content and, as a result, in the ratio, but for every file the ranking of the programs stays the same.

## 5. Simulation of costs of long-term data storage

Let us assume power cost at \$0.2 per KWh, power consumption equal to 1000 W (which is far from the real consumption, especially for server systems) and storage cost at \$0.125 for 1 GB per month (according to [16]). The resulting cost simulation is presented in Tables 8 and 9. Power consumption during compression is taken into account and added to the cost of the first year of storage.

With only a few files gathered in the considered dataset, the savings are not so impressive, but when we consider a 5000 times larger database of an estimated overall size 240 TB, they become the real issue.

## 6. Conclusions

The usage of data storage space is a very important measure when it comes to long-term storage of data from high-throughput DNA sequencing experiments. We have shown that it is profitable to minimize the compression ratio even at the cost of much longer processing, which is performed once for a FASTQ file. Our algorithm, FastQComp, achieved better compression ratio than other compressors specialized for FASTQ files: DSRC, Quip and SCALCE. Our further studies are focused on a deep parallelization of the algorithm, including GPGPU

Table 8. Total cost of first year of storage calculated for dataset A.

| | Data size [GB] | Compression time [h] | Compression cost [$] | Storage cost per year [$] | Total cost for first year [$] |
|---|---|---|---|---|---|
| Raw data | 49.20 | 0.00 | 0.00 | 73.80 | 73.80 |
| FastQComp | 6.81 | 3.70 | 0.74 | 10.22 | 10.96 |
| SCALCE | 7.31 | 0.45 | 0.09 | 10.97 | 11.06 |
| Quip | 7.63 | 0.55 | 0.11 | 11.45 | 11.55 |
| DSRC 2.0 | 9.11 | 0.17 | 0.03 | 13.67 | 13.70 |
| 7-zip | 10.45 | 5.86 | 1.17 | 15.67 | 16.84 |
| bzip2 | 12.03 | 1.50 | 0.30 | 18.04 | 18.34 |
| gzip | 14.45 | 3.03 | 0.61 | 21.67 | 22.28 |

Table 9. Simulation of costs of 10 years storage of FASTQ files. The basic dataset is composed of files specified in Table 2 (dataset A). The overpayment in the last column is related to FastQComp.

| | 1 dataset | | 5000 datasets | |
|---|---|---|---|---|
| | Cost [$] | % of FastQComp | Cost [$] | Overpayment [$] |
| Raw data | 738.03 | 717% | 3 690 148 | 3 175 673 |
| FastQComp | 102.90 | 100% | 514 475 | — |
| SCALCE | 109.78 | 107% | 548 877 | 34 402 |
| Quip | 114.56 | 111% | 572 817 | 58 342 |
| DSRC 2.0 | 136.69 | 133% | 683 468 | 168 993 |
| 7-zip | 157.90 | 153% | 789 476 | 275 001 |
| bzip2 | 180.71 | 176% | 903 555 | 389 080 |
| gzip | 217.29 | 211% | 1 086 473 | 571 998 |

processing, in order to reduce the processing time in a significant way. We will refer to some of our ideas of dictionary construction on CUDA from [3].

The time optimization of the algorithm can be driven by nature of the data being processed. For every data type, namely the sequence identifier, the sequence itself and the quality score line, a separated thread can be used. This approach is very intuitive, but does not ensure a uniform load balancing. Actually, the sizes of a sequence and its score line are equal, but much greater than the identifier's length. This brings the expected speedup close to 2, even for three threads, regardless of a computing environment. Thus, we are currently working on the optimization driven by available resources of a computing environment: the number of cores, amount of RAM, *etc.* To fully exploit a multicore infrastructure, we will create a pool of processing threads, where every thread will process a particular set of reads (a block). The number of threads will depend on the number of cores and the block size will depend on the currently available RAM. All lines describing a particular read, in the appropriate order, will be assigned to the same thread. We plan two additional specialized threads for reading and splitting the input file into blocks, and for storing compressed blocks in the output file, respectively.

Solutions implemented in FastQComp can be applied not only on the ground of pure compression, but also in the process of DNA sequencing or resequencing, which could gain advantage from the direct usage of compressed data [2,19]. Nowadays, one of the main challenges in bioinformatics is an efficient processing of huge data volumes produced by biotechnological instruments. The information volume often exceeds the abilities of bioinformatics software. The possibility of direct access to compressed files or the usage of packed data structures is a way to tackle the problem and another subject of our studies.

## References

[1] G.R. Abecasis, D. Altshuler, A. Auton, L.D. Brooks, R.M. Durbin, R.A. Gibbs, M.E. Hurles and G.A. McVean, A map of human genome variation from population-scale sequencing. *Nature* **467** (2010) 1061–1073.

[2] J. Blazewicz, M. Bryja, M. Figlerowicz, P. Gawron, M. Kasprzak, E. Kirton, D. Platt, J. Przybytek, A. Swiercz and L. Szajkowski, Whole genome assembly from 454 sequencing output via modified DNA graph concept. *Comput. Biol. Chem.* **33** (2009) 224–230.

[3] M. Chlopkowski and R. Walkowiak, A general purpose lossless data compression method for GPU. *J. Parallel Distrib. Comput.* **75** (2015) 40–52.

[4] G. Cochrane *et al.* Facing growth in the European Nucleotide Archive. *Nucleic Acids Res.* **41** (2013) D30–D35.

[5] P.J.A. 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 Res.* **38** (2010) 1767–1771.

[6] S. Deorowicz and S. Grabowski, Compression of DNA sequence reads in FASTQ format. *Bioinform.* **27** (2011) 860–862.

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

[8] F. Hach, I. Numanagic, C. Alkan and S.C. Sahinalp, SCALCE: boosting sequence compression algorithms using locally consistent encoding. *Bioinform.* **28** (2012) 3051–3057.

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

[10] Inc. Illumina, CASAVA v1.8 changes. [on-line] `http://support.illumina.com/documentation.html`, January (2011).

[11] Inc. Illumina, BaseSpace user guide. [on-line] `http://support.illumina.com/documentation.html`, May (2013).

[12] D.C. Jones, W.L. Ruzzo, X. Peng and M.G. Katze. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic Acids Res.* **40** (2012) e171.

[13] C. Kozanitis, C.T. Saunders, S. Kruglyak, V. Bafna and G. Varghese. Compressing genomic sequence fragments using SlimGene. *J. Comput. Biol.* **18** (2011) 401–413.

[14] M. Nelson. [on-line] `http://marknelson.us/1991/02/01/arithmetic-coding-statistical-modeling-data-compression/`.

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

[16] D.S.H. Rosenthal, D. Rosenthal, E.L. Miller, I. Adams, M.W. Storer and E. Zadok, The economics of long-term digital storage. In *The Memory of the World in the Digital Age: Digitization and Preservation*, September (2012).

[17] D. Salomon, Data Compression: The Complete Reference. With contributions by Giovanni Motta and David Bryant. Springer, London (2007).

[18] C.E. Shannon, A mathematical theory of communication. *The Bell Syst. Tech. J.* **27** (1948) 379–423, 623–656.

[19] A. Swiercz, B. Bosak, M. Chlopkowski, A. Hoffa, M. Kasprzak, K. Kurowski, T. Piontek and J. Blazewicz, Preprocessing and storing high-throughput sequencing data. *Comput. Methods Sci. Technol.* **20** (2014) 9–20.

[20] Y. Tateno, T. Imanishi, S. Miyazaki, K. Fukami-Kobayashi, N. Saitou, H. Sugawara and T. Gojobori, DNA Data Bank of Japan (DDBJ) for genome scale research in life science. *Nucleic Acids Res.* **30** (2002) 27–30.

[21] T.A. Welch. A technique for high-performance data compression. *Computer* **17** (1984) 8–19.

[22] I.H. Witten, R.M. Neal and J.G. Cleary, Arithmetic coding for data compression. *Commun. ACM* **30** (1987) 520–540.

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

[24] J. Ziv and A. Lempel, Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory* **24** (1978) 530–536.