Journal Title Here, 2022, pp. 1–10

doi: DOI HERE
Advance Access Publication Date: Day Month Year
Paper

PAPER

Compressed Checkpoint Index and Parallel Parser for FASTQ Files

Yishan Zhao,[1,*] Elliot Huang[1] and Erin Lea[1]

[1]Department of Computer Science, University of Maryland, College Park, 8125 Paint Branch Drive, 20742, MD, USA
[*]Corresponding author. yishanzh@umd.edu

FOR PUBLISHER ONLY Received on Date Month Year; revised on Date Month Year; accepted on Date Month Year

Abstract

FASTQ sequencing reads, due to their respectable size, are often stored in a GZIP format. This strategy is compact for storage, but also poses significant challenges to efficient parsing. We present a FASTQ parsing library that significantly improves the decompressing and parsing speed of a GZIP compressed FASTQ file, by generating a corresponding index file that enables parallel parsing.

Key words: parallel, FASTQ, GZIP

Introduction

Ever since the advent of multi-core CPU, parallel computing has become a feasible concept that offers numerous advantages over the traditional synchronous routine, at the cost of requiring specialized variants of the already-complicated algorithm to handle domain-specific challenges. For example, a bottleneck scenario could occur during a critical section(7), where multiple threads demand a shared resource that is slow to retrieve and is designed to serve only one thread at a time.

Another additional complication is the number of running threads. Counter-intuitively, instantiating more threads than necessary could hamper the performance by introducing extra overhead during the initialization phase, as well as additional pressure on the operating system's thread scheduling capability.

Due to these inherent flaws, parallel computing, although beneficial, is not prevalent in small utilities and hobbyist programs. However, in the context of bio-informatics, where performance and efficiency are regarded as top priorities, parallelism can not be ignored even to the slightest degree. Thus we propose our own library for fully utilizing the potential of parallelism. Our benchmarking suggests a significant speedup of up to 7 times faster compared to a sequential decompressor4. The extra index-building phase is also performant, consuming less than a minute when building for a 4.1 GB file. In addition, the library incorporates a lightweight API that encapsulates concurrent resource management so that the compressed FASTQ records behave as if they were in an already-decompressed collection.

Background

FASTQ format is one of the most widely adopted sequencing data formats in the realm, despite its noticeable flaws. In terms of size, the nature of being a plain text file rather than a compressed binary file means it is prone to redundant information. Since sequences are bound to share multiple occurrences of repetitive patterns, file sizes could snowball while the information entropy is low. Thus, it is a common practice to compress a huge FASTQ file into a GZIP file for storage. In this way, file size is significantly reduced at the cost of additional decompression time. This is less than ideal, however, due to the fact that decompression can not happen in parallel, thus rejecting the potential speedup of utilizing multiple CPU threads. A naive attempt could be made to partition the file into chunks and decompress them individually. However, it is not viable because in order to inflate a chunk of data compressed using deflate algorithm, not only a sliding window of symbols is required, but block type information must also be set correctly(14).

This is not a recent issue, and there has been a competitive solution. The `bgzip` utility, following the SAM specification(17), is an improvement over the established GZIP as it uses the same deflate algorithm and produces a corresponding index file. The index comprises checkpoints that record decompressor states at the said point, hence allowing the decompressor to jump to the respective point directly, set its states, and start decompressing. This procedure can happen in parallel, therefore significantly improving the performance.

Being a relatively novel data format comes with one inherent drawback: backward compatibility. The majority of FASTQ files are still in GZIP format, unable to enjoy the benefit brought by bgzip. In fact, by doing a simple search, most, if not all, recent sequencing data in SRA database are still using the old GZIP format. Although technically it is possible to convert every relevant GZIP file into the newer format, bgzip, it would not be effective at all, considering the sheer amount of files waiting to be processed. Thus a practical solution would be to apply the gist of bgzip utility to GZIP format, by implementing a similar index builder for it.

There are also similar tools that adopt the idea of parallelism using indexing, such as `gztool`(5), `indexed_gzip`(?) and `zindex`, all providing indexing with built-in support for parallel decompression for general-purpose random accessing. However, since they are not optimized for FASTQ file format, and the range of requested decompressed text can be arbitrary, there are inevitable but unnecessary overheads.

CIndex(1), on the other hand, is a specialized tool for this exact purpose. It combines various mainstream compression techniques to minimize the compressed file size, and also provides excellent file indexing performance(1). Unfortunately, it is not compatible with existing GZIP-compressed FASTQ files either.

Our project thus aims to reconcile the conflict by providing a compressed index for existing files and supplementing them with parallel parsing capability.

Methods

The basic idea of achieving parallel decompressing is to distribute checkpoints across the compressed file, where each checkpoint contains the required information for decompressors. Thus multiple decompressors can be instantiated concurrently, with each initialized with the given information of the relevant checkpoints. As a general-purpose parallel decompressing tool, `gzindex` distributes the checkpoints in a somewhat uniform manner, so that for any arbitrary decompression task, it is within constant time to jump to the nearest checkpoint and begin the actual procedure.

In our implementation, the positions of the checkpoints are determined by a parameter named `chunkSize`, which describes the ideal number of FASTQ records between two adjacent checkpoints. In order to determine the position of checkpoints during the index-building phase, a naive approach would be to carefully and precisely inflate everything up to the point where the immediate output is certain records away from the last checkpoint, and then make this a new checkpoint. However, since the amount of decompressed output produced by `inflate` method can not be manipulated, an `inflate` block boundary is unlikely to coincide with the chunk boundary. In this case, the retrievable decompressor state is not correlated to the desired state. Therefore we took the approach of allowing an `inflate` iteration to stop at a block boundary that is slightly ahead or behind the ideal record checkpoint, and then use the approximate point as a checkpoint, rounding to the nearest byte that is the beginning of the next record.

In this way, each checkpoint in the index consists of:

- an offset in the input stream
- an offset in the output stream
- a sliding window containing 32,768 bytes of uncompressed data preceding the checkpoint
- a small sequence of bytes spanning from the start of the last incomplete record in the block to the block boundary

With the above information of each checkpoint stored in the index, we implemented a method called `ExtractDeflateIndex` to extract the data between two adjacent checkpoints. We can simply pass two adjacent checkpoints, along with the required compressed data determined by the input offset of the two checkpoints, into the method for data extraction. The method will use the output offset of the two checkpoints to determine the length of the data that needs to be decompressed, and then initialize the decompression library from the former checkpoint's sliding window. Subsequently, the method will use `inflate` to decompress the input data until all the input bytes have been read and decompressed, and then return the uncompressed data between the two checkpoints.

Assuming the total size of a FASTQ record follows a uniform distribution from 256 bytes to 1024 bytes, the expected length is, therefore, 640 bytes. The theoretical size of the index file is thus $(32768+640)n = 33408n$ bytes, where n is the number of checkpoints. For a sufficiently large file with relatively sparse checkpoints, the extra space taken by the index file is negligible.

Although having a fully parallelizable decompressor is a great achievement, this does not mark the end of our project since the ultimate goal is to produce a library capable of automatically orchestrating a multi-threaded decompressor. Thus, in addition to the `ExtractDeflateIndex` method mentioned above, we also implemented a supplementary data structure that encapsulates the low-level details of parallelism. The basic idea is to create an iterator that returns one record at a time per request, while continuously performing the IO actions, decompressing, and parsing in the background using multiple threads. To achieve this, the data structure `BatchedFASTQRecords` maintains two caches for the parsed records and file buffers respectively, which are read chunk by chunk in a concurrent manner that exploits the concurrent IO capability of modern SSD. However, it is worth noticing that reading concurrently is extremely inefficient on an HDD due to physical constraints. Thus, we also provide an option to turn off SSD optimization. To further reduce the overhead when claiming resources, threads and buffers are drawn from a pool, and then returned when the designated work is done. It is worth noticing that, `BatchedFASTQ` employs the lazy evaluation strategy, which only parses new records if there are not enough records in the cache. At the same time, it also frees used records from memory to reduce memory pressure further.

The sole purpose of this structure is to simplify the iteration. For example, counting the occurrences of a certain pattern can be written as this:

```
var records = new BatchedFASTQRecords(index,
    pathToGzip);
int count = 0;
foreach (FastqRecord record in BatchedFASTQ)
{
    if (record.Sequence.Contains(pattern)) count++;
}
```

It also supports the functional style:

```
var count = records.Count(r =>
    r.Sequence.Contains(pattern));
```

However, due to the way that it is implemented, there is no guarantee of the order among individual records. In other words, suppose record a precedes b in the original file.

Results

To better understand the scalability of our library, we used various file sizes ranging from 8 MB to 32 GB, with each file being twice the size of its previous one. Initially, we tried to obtain the test data from SRA but soon realized that it is extremely time-consuming to find the file with the exact size. Although SRA Explorer is useful for downloading some large files, others are only available through the SRA Tool command line utility. While this is technically an alternative, it has one major drawback, that is the files are downloaded in FASTQ format and need to be compressed on a local machine. There is another major issue: some records contain `@` characters in the quality strings, which are not compatible with our index generation algorithm. Considering the potential implication, we have to abandon this idea and generate our own mock data.

We chose the ILLUMINA sequencing read as a template. Our mock data follows a uniform distribution between 128 and 512, and the base occurrences are completely random, as we consider this property irrelevant to our purpose, thus introducing little to no biases. The quality strings follow a weighted uniform distribution where `?` has a probability of 95% whereas `*` and `!` are each 2.5%. The number of records in a compressed file is carefully controlled so that the GZIP file sizes strictly grow in exponents of 2.

We planned to test the performance against a naive C# sequential decompressor as a reference. During benchmarking and performance profiling, we discovered that the stage of parsing is the most computation-intensive one; thus for a controlled comparison, the references must have their parsing methods implemented in some way. `gztool` appears to be a good candidate at first: it is performant and easy to use. However, the best we can do is to read everything in parallel from a GZIP file and immediately write it into an output device, which involves no parsing stage at all. It would be simply unworthy for us who are not familiar with C language. `indexed-gzip` is technically also a valid control group, but there is a similar issue: we need to implement the parser in Python, which due to its interpreted runtime, would not constitute a fair comparison.

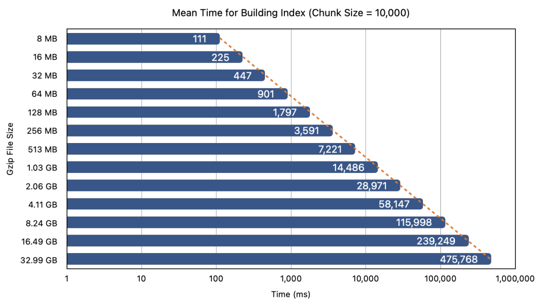

Fig. 1. Mean Time for Building Index with Chunk Size = 10,000

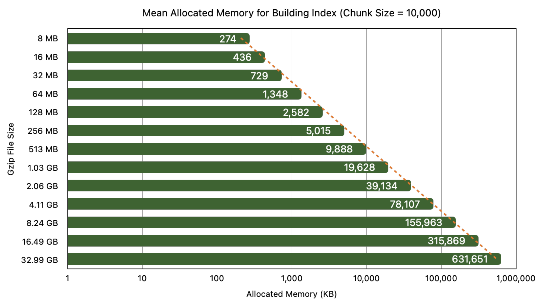

Fig. 3. Mean Allocated Memory for Building Index with Chunk Size = 10,000

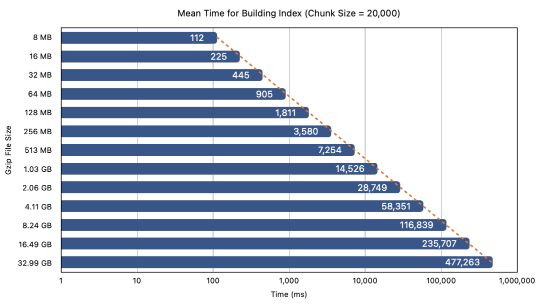

Fig. 2. Mean Time for Building Index with Chunk Size = 20,000

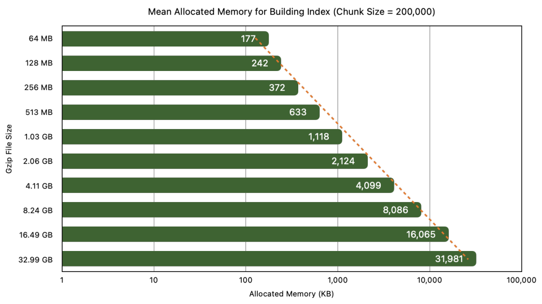

Fig. 4. Mean Allocated Memory for Building Index with Chunk Size = 200,000

To benchmark the performance of index building, we used the aforementioned GZIP files ranging from 8 MB to 32 GB, and built the indexes for each of them with 6 different chunk sizes, 10,000, 20,000, 50,000, 100,000, 200,000, and 1,000,000. We performed each index-building task at least 10 times and measured the mean time to run each task and the mean memory allocated for each task. During data processing, we removed some data for mean memory allocated for smaller files when the chunk size is large. This is because under these conditions, the chunk size is larger than or very close to the total number of records in the file, resulting in the index only storing two checkpoints, one immediately after the header and one at the end of the file. In other words, no meaningful checkpoints are stored in the index. **Fig. 1** to **Fig. 12** shows the results of this benchmark.

The results indicate that for all the chunk sizes tested, the time to build index increases linearly with the increase in GZIP file size. The GZIP file size and the memory allocated for building index have a linear relationship as well. When comparing the mean time and mean allocated memory for building index across the 6 chunk sizes (**Fig. 13** and **Fig. 14**), we can observe that the time to build index is only dependent on the GZIP file size and independent of the chunk size. The memory allocated for building index is positively correlated with GZIP file size and negatively correlated with the chunk size. With the increase in chunk size, the number of records between two adjacent checkpoints increases, and thus the number of total checkpoints decreases; consequently, the memory allocated for building and storing index decreases.

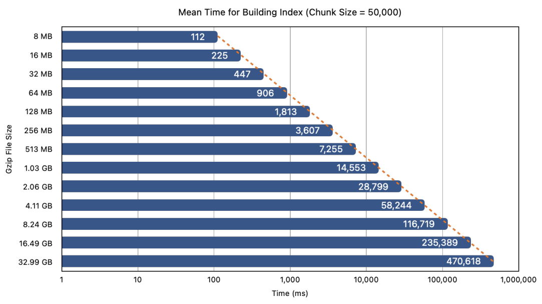

Fig. 5. Mean Time for Building Index with Chunk Size = 50,000

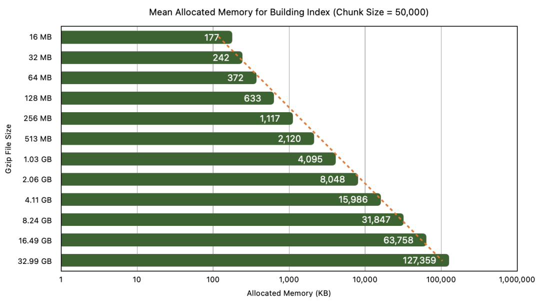

Fig. 6. Mean Allocated Memory for Building Index with Chunk Size = 50,000

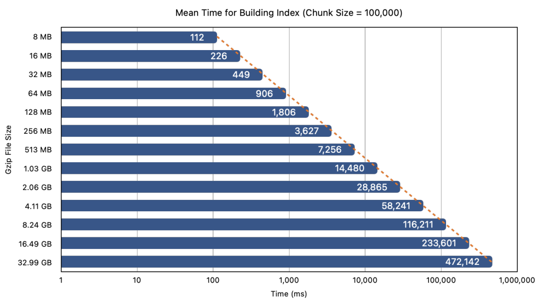

Fig. 7. Mean Time for Building Index with Chunk Size = 100,000

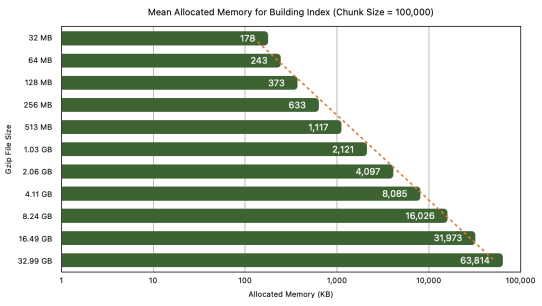

Fig. 8. Mean Allocated Memory for Building Index with Chunk Size = 100,000

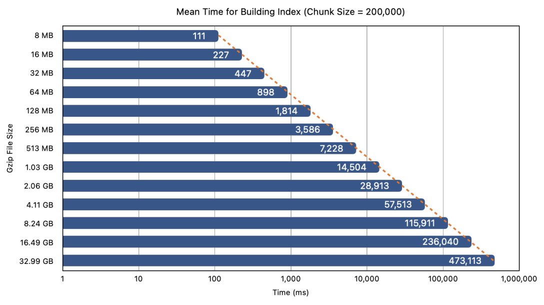

Fig. 9. Mean Time for Building Index with Chunk Size = 200,000

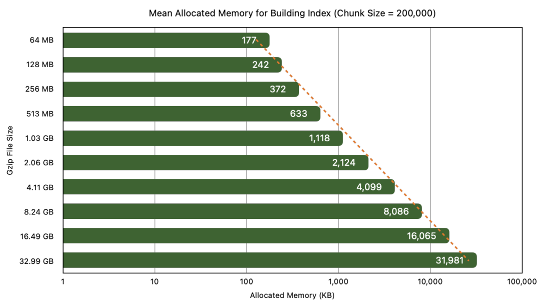

Fig. 10. Mean Allocated Memory for Building Index with Chunk Size = 200,000

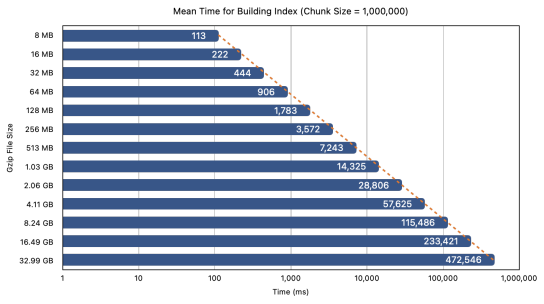

Fig. 11. Mean Time for Building Index with Chunk Size = 1,000,000

Next, we tested the performance of the `ExtractDeflateIndex` method. We measured the mean time for extracting data using pre-built indexes on GZIP files with various sizes. The results (**Fig. 15**) demonstrate that the time to extract data using indexes and the GZIP file size has a linear relationship as well.

Before benchmarking the performance of our parallel decompressor and parser, we need to set a baseline so that we know how much speedup we will gain compared to sequential reading, decompressing, and parsing. Therefore, we built a simple decompressor that can achieve these and measured the meantime for data extraction and parsing using the simple decompressor. The results are shown in **Fig. 16**.

When designing the benchmark to perform on our parallel parser, we came up with two tasks–counting the

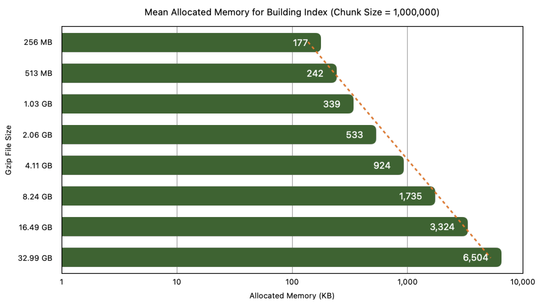

Fig. 12. Mean Allocated Memory for Building Index with Chunk Size = 1,000,000

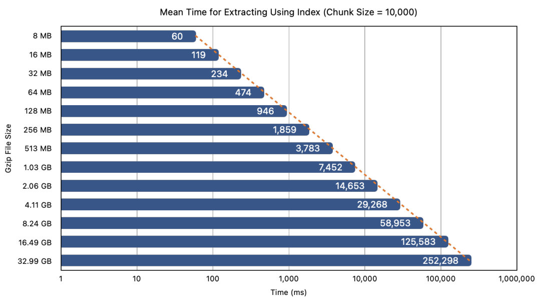

Fig. 15. Mean Time for Extracting Using Index (Chunk Size = 10,000)

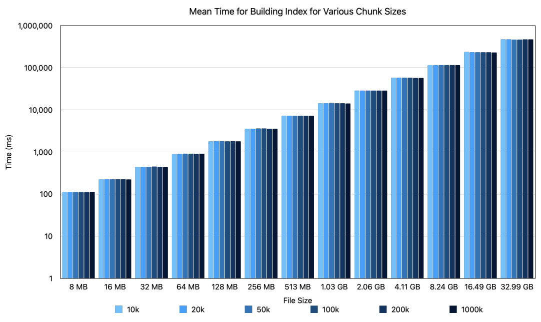

Fig. 13. Mean Time for Building Index for Various Chunk Sizes

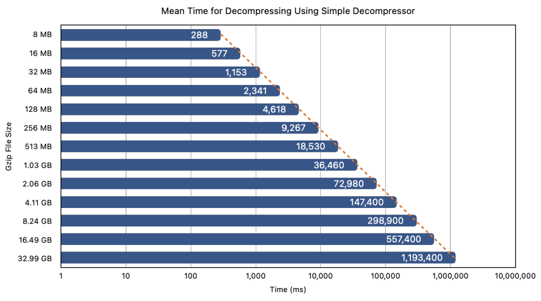

Fig. 16. Mean Time for Decompressing Using Simple Decompressor

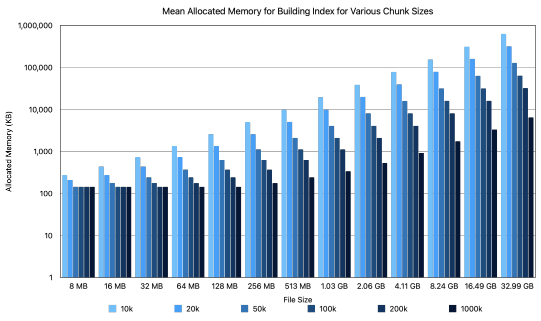

Fig. 14. Mean Allocated Memory for Building Index for Various Chunk Sizes

total records in the uncompressed data and counting the occurrences of a pattern in the uncompressed data. Since counting the total records is achieved by simply calling the `Count()` method on the return value of the parallel parser, we can assume the performance of this task is equivalent to the performance of parallel parsing using our parallel parser.

We considered a simple task to try to capture the fundamentals of processing a sequence of records, which is to count the occurrence of a given pattern as mentioned above. We specifically chose the pattern to be a sequence of 12 random bases, as the number of matches is within an observable range.

Assume the pattern is of length a. Since a random string of length a has a probability of 4^a to be an exact match of the said given pattern, as well as the fact that a string of length n contains exactly $n-a+1$ substrings, we can deduce that for a string with length n, the probability would be $4^{-a}(n-a+1)$. Assuming the sequences in mock data follow $X \sim U(128, 512)$, we have the expectation for the occurrences in a single sequence:

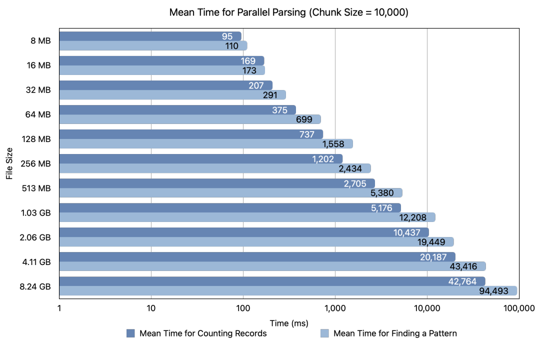

Fig. 17. Mean Time for Parallel Parsing (Chunk Size = 10,000)

$$E(X) = \frac{1}{384}\sum_{i=128}^{512} 4^{-a}(i-a+1) \tag{1}$$

$$= \frac{1}{4^a}\cdot\frac{1}{384}\sum_{i=128+a-1}^{512+a-1} i \tag{2}$$

$$= \frac{1}{4^a}\cdot\frac{1}{384}(640+2a-2)\cdot 192 \tag{3}$$

$$\simeq 320\frac{1}{4^a} \tag{4}$$

Because X follows uniform distribution, their addition is transitive. Thus the number of occurrences in an entire file with s sequences is simply $320s\frac{1}{4^a}$. If we choose a to be 12, then in the smallest sample where there are only 48000 records, the expectation becomes 0.9 which is acceptable.

Similar to benchmarking index building performance, to test the performance of our parallel parser, we performed the two tasks on various GZIP files ranging from 8 MB to 8 GB with 6 different chunk sizes. The results of the mean time needed to perform these tasks are shown in **Fig. 17** to **Fig. 22**. From the results we can observe that with the increase of the GZIP file size, the time needed to perform both tasks will increase as well, and overall, pattern lookup requires more time than keeping track of the count of the records. For larger files, the percentage of the time pattern lookup requires more than the time for counting records is also higher.

When we compare the performance of parallel parsing across different chunk sizes (**Fig. 23** and **Fig. 24**), we can see that the performance of parallel parsing is also dependent on the chunk size; larger chunk size would result in longer time, and this is true for both pattern lookup and record counting.

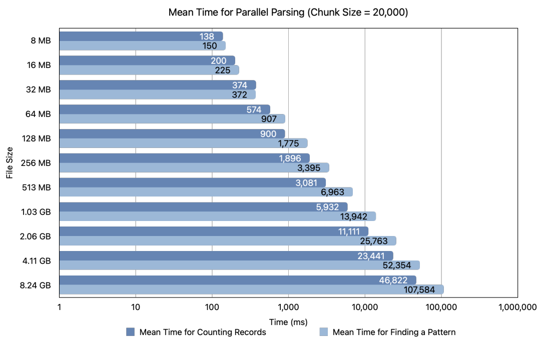

Fig. 18. Mean Time for Parallel Parsing (Chunk Size = 20,000)

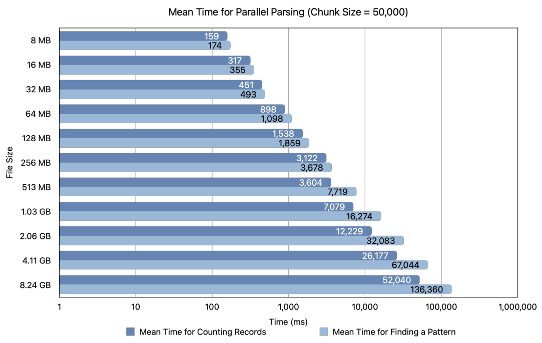

Fig. 19. Mean Time for Parallel Parsing (Chunk Size = 50,000)

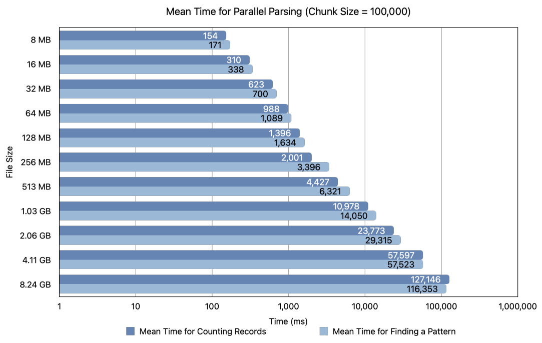

Fig. 20. Mean Time for Parallel Parsing (Chunk Size = 100,000)

We then compared the speed of sequential extracting and parsing achieved by our simple decompressor with the speed of parallel parsing (**Fig. 25**). We were delighted to see that our parallel parser is about 6 to 7 times faster

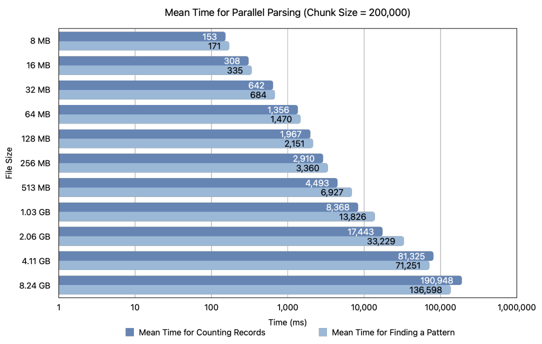

Fig. 21. Mean Time for Parallel Parsing (Chunk Size = 200,000)

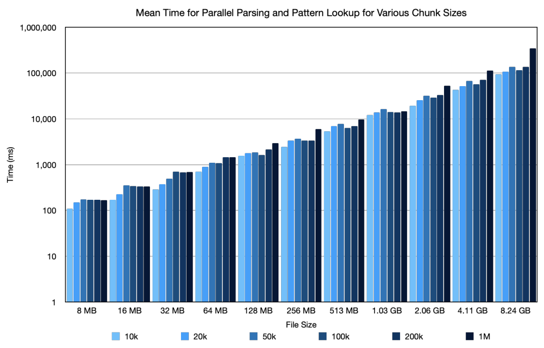

Fig. 24. Mean Time for Parallel Parsing and Pattern Lookup for Various Chunk Sizes

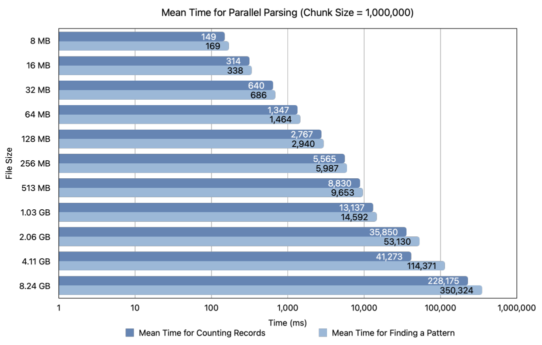

Fig. 22. Mean Time for Parallel Parsing (Chunk Size = 1,000,000)

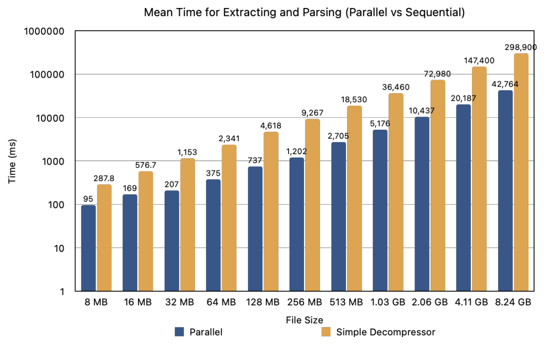

Fig. 25. Mean Time for Extracting and Parsing (Parallel vs Sequential)

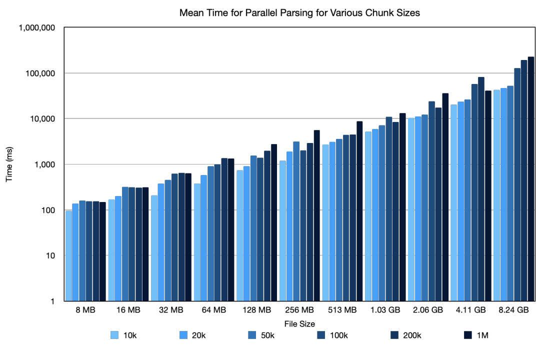

Fig. 23. Mean Time for Parallel Parsing for Various Chunk Sizes

than the simple decompressor when we set the chunk size to be 10,000.

To further improve the performance, we implemented a way to optimize the read speed for SSDs. In short, once all the compressed data has been processed, a new request for reading more will be instantiated, in which 8 new tasks will be spawned and scheduled to unoccupied threads. Each task will then read a specified and separate partition. The number of new tasks, in this case, 8, is an empirical estimate that comes with the optimal file reading efficiency under ideal circumstances. However, this deliberate decision is not as effective as we imagined, as there is no consistent speedup over the unoptimized version (**Fig. 26** and **Fig. 27**) . An educated guess would be that the file IO, in our case, actually is not a bottleneck, considering the parsing phase consumes much of the running time and computational resources.

Discussion and Future Work

At first, we decided to implement the deflate algorithm and the decompressor in general in C#, but soon deemed it too realistic to begin with: our implementation will be

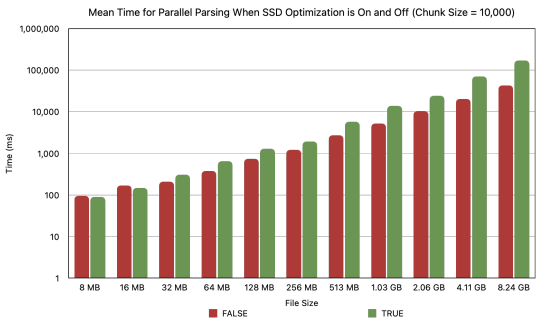

Fig. 26. Mean Time for Parallel Parsing When SSD Optimization is On and Off (Chunk Size = 10,000)

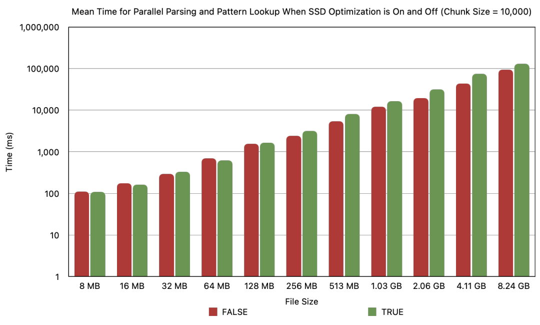

Fig. 27. Mean Time for Parallel Parsing and Pattern Lookup When SSD Optimization is On and Off (Chunk Size = 10,000)

prone to errors and bugs, and the following debugging and optimization will inevitably occupy a significant portion of the work, leaving insufficient time for the real problem that is to devise a scheme for the index file and a thread-safe decompressor.

Therefore, our actual study started from researching an existing tool, `gztool`. It is a command line utility; therefore, in order to adapt it for our use, we need to first convert it to a library by removing the `main()` function and various command line options, condensing it to the extreme where only the core feature is present. However, the code itself is not well documented, and the underlying program logic is hidden beneath cryptic pointer arithmetic that we failed to fully comprehend.

On the other hand, the Python utility `gzindex` offered an inspiration for us. It is based on the C library, `zran.c`, which entails virtually everything we need: a simple and concise index builder and a decompressor that is able to perform random read on a GZIP file using the generated index file. It provides an appropriate template where we can refer to the comments and notices to modify it to suit our need.

In general, the performance is within our expectations. Due to the short span of this final project, we deliberately made some design decisions and assumptions to simplify the code logic.

For instance, a FASTQ file could contain `'@'` character in its quality string while remaining completely valid. This is, however, not supported by our parser due to the complication it implies. Thus when encountering such a case, the parser would intentionally throw an exception, signaling the end of the parsing procedure to prevent bogus results from corrupting the FASTQ records.

Another limitation is the maximum size of a chunk, i.e., the distance between two checkpoints. Due to the way that C# array is implemented, it is unable to contain more than 2^{31} entries which is 2 GB of data in our case. Assuming each record is at most 1024 bytes long, the max chunk size is thus set to 2 million. Admittedly it is not difficult to carefully design the code to overcome this limitation, but we reckon that for our purpose, the said max limit is more than enough.

Finally, the fact that C# is a GC language constrains us from many delicate optimizations. Although there are ways to manually manage memory to minimize the impact of garbage collection, the core components of the underlying runtime are beyond our control. For any potential future plans, it is wise to rewrite the logic using non-GC languages with minimal runtimes such as C and C++.

We assume this library would be relatively useful due to its high performance, user-friendliness, and extensibility. However, due to its reliance on the native library, namely zlib, it might not perform as intended on Windows as neither of us has a Windows machine.

References

1. H. Huo, P. Liu, C. Wang, H. Jiang, and J. S. Vitter CIndex: Compressed indexes for fast retrieval of FASTQ files *Bioinformatics, vol. 38, no. 2, pp. 335–343* Sep. 2021.
2. Paul McCarthy indexed_GZIP *https://github.com/pauldmccarthy/indexed_GZIP*
3. Mark Adler zran *https://github.com/madler/zlib/blob/master/examples/zran.c*
4. Mark Adler zlib *https://github.com/madler/zlib*
5. circulosmeos gztool *https://github.com/circulosmeos/gztool*
6. Matt Godbolt zindex - index and search large GZIPped files quickly

https://xania.org/201505/zindex-index-your-GZIP-files May 2015

7. Langmead, Ben and Wilks, Christopher and Antonescu, Valentin and Charles, Rone 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) Bioinformatics, vol 35, no. 3, pp. 421-432 July 2018
8. Josh Dullen zlib.net *https://github.com/JoshDullen/zlib.net*
9. celophi zlib.extended *https://github.com/celophi/zlib.extended*
10. Andrey Akinshin, Adam Sitnik, Matt Warren, Wojciech Nagórski, Alina Smirnova, Yegor Stepanov, Igor Fesenko, Nathan Ricci, Mark Tkachenko, Matthias Wolf, et. al BenchmarkDotNet *https://github.com/dotnet/BenchmarkDotNet*
11. Mike Barber csharp-priority-scheduling-tools *https://github.com/mike-barber/csharp-priority-scheduling-tools*
12. Rasko Leinonen, Hideaki Sugawara, Martin Shumway on behalf of International Nucleotide Sequence Database Collaboration The Sequence Read Archive *Nucleic Acids Research, vol. 39, suppl_1, pp. D1-D21*January 2011
13. Phil Ewels, Andrew Duncan and James A. sra-explorer *https://github.com/ewels/sra-explorer*
14. L. Peter Deutsch and Jean-loup Gailly ZLIB Compressed Data Format Specification version 3.3 *RFC Editor*1950
15. L. Peter Deutsch DEFLATE Compressed Data Format Specification version 1.3 *RFC Editor*1951
16. L. Peter Deutsch GZIP file format specification version 4.3 *RFC Editor*1952
17. Bob Handsaker, Heng Li Sequence Alignment/Map Format Specification The SAM/BAM Format Specification Working GroupAugust 2022