doi:
Paper

PAPER

FFIndex: Compressed Indexing for Parallel FAST-Q Parsing

Siddhant Bharti,[1] Prajwal Singhania[1] and Rakrish Dhakal[1]

[1]Department of Computer Science, University of Maryland, College Park, 20740, Maryland, USA

Abstract

Our project's aim is to address the challenge of efficiently parsing compressed FASTQ files. This is critical for genomic data processing pipelines but is hindered by the sequential nature of current decompression and parsing methods. The goal is to develop a tool and accompanying library that facilitates parallel processing by creating and utilizing an index over compressed FASTQ files. We demonstrate how just keeping the 32KB uncompressed data before a series of access-points allows us to deterministically decompress the compressed FASTQ file in parallel. This leads to speedups in decompression. In the results section, we show that with multiple reader-threads, the processing time can be significantly reduced. We compare the speedups in our method to different baselines. The code for this project can be found in our GitHub repository: `https://github.com/siddhant-bharti/CMSC701-Project`

Introduction

Contemporary genomics algorithms and tools are embarrassingly parallel, and the processors these days have many cores that can support hundreds of simultaneous threads of execution. Ingesting and parsing the raw read records is a significant bottleneck for these algorithms and tools. The reasons for this are manyfold: 1) Variable record lengths in FASTQ files can impede scaling because identifying these record boundaries needs to be done sequentially; 2) Synchronization and locking while reading a single FASTQ file by many threads can significantly affect performance; 3) FASTQ files are stored in gzipped compressed formats which makes reading these files using many threads hard since decompression is a sequential process. [6] provides a good analysis over these bottlenecks for the FASTQ files.

With the recent advancements in genomics, the size of the FASTQ files is increasing. For example, the size of the BGISEQ (MGISEQ-2000RS) FASTQ file is around 37GBs [4]. Therefore these files are stored and shared in compressed format. Gzip has shown to achieve best size reduction for FASTQ files [4]. However, Gzip's DEFLATE (mLZ77 and huffman coding) algorithm is sequential. This makes random access in compressed FASTQ files hard. There is a different way to compress the files by first breaking the files into different chunks and then compressing these chunks separately. This would have allowed parallel decompression of the compressed file but the compression quality of this method is not good. Furthermore, most files hosted on Sequence Read Archive (SRA) are not compressed in blocks. This bottleneck is further underscored by the fact that reading compressed data is a slow process. *gunzip*, the decompressor component of *gzip*, can read compressed data at around 30-50 MB/sec which is order of magnitude less than read throughput of current SATA/NVMe solid-state drives [4].

The compression and decompression algorithms used in Gzip are inherently serial. As such, existing FASTQ parsers can only begin reading starting from the beginning of each compressed FASTQ file. However, we found that if we know the 32KB uncompressed data before any random access point of interest, we can then start decompressing only from that point onward. This is the idea behind our method. We create some "access points" and store preceding 32KB uncompressed data to be able to decompress from that point. Using this idea, we can have multiple threads read the compressed FASTQ file, where each thread starts processing from an access point and read some number of records.

We have created FFIndex, the tool that creates this index over compressed FASTQ file, as well as FFParser, a corresponding library that uses this index to be able to read the compressed FASTQ file in parallel. The idea is to incur a one-time cost of building the index, which then significantly accelerates parsing compressed FASTQ files in the future. This index can be shared alongside the FASTQ files in repositories like the NCBI Sequence Read Archive [10], which amortizes the cost of the index creation. The implementation to create the access points and corresponding 32KB uncompressed context is adopted from zran.c, which is an example file in the zlib github repository [2]. zran.c decompresses the input file in its entirety and creates this index. This index is then retained in memory to allow random access from some byte offset. We store this index data structure to disk in compressed format to allow future processing. This index is also augmented with information on record boundaries to allow random access starting from a record boundary. The access points are saved every few records (tuneable parameter). We show the speedups in processing

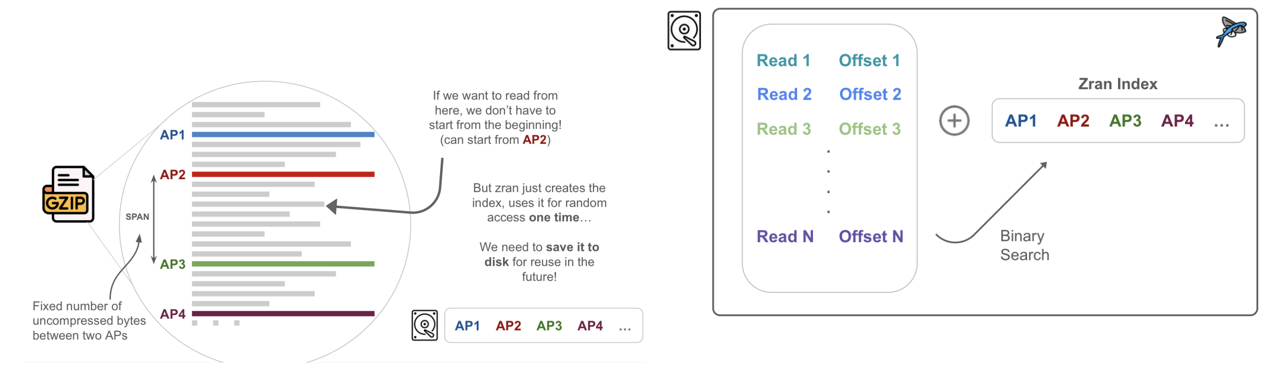

AP1
AP2
AP3
AP4
SPAN
Fixed number of uncompressed bytes between two APs
If we want to read from here, we don't have to start from the beginning! (can start from AP2)
But zran just creates the index, uses it for random access one time…
We need to save it to disk for reuse in the future!
AP1 AP2 AP3 AP4 …
Read 1 Offset 1
Read 2 Offset 2
Read 3 Offset 3
Read N Offset N
Zran Index
AP1 AP2 AP3 AP4 …
Binary Search

Fig. 1. Structure of Zran Index (left) and FFIndex (right). FFIndex is an FAST-Q file based extension of Zran Index that supports record level access. [The logo for FFIndex is taken from [1]]

by using this in a publisher-consumer based framework where publisher reads some records from the FASTQ file and adds it to the queue. Then the consumer picks up these read records and does the processing over them. The downstream task we use in FFParser for experiments is counting the number of bases in the FASTQ file. This application is used because it is not only light weight but also helps us to sanity check the correctness of our implementation.

Background

Sequencing data is typically created and shared in the form of FASTQ files. More specifically, the *gzip* software application is used to compress FASTQ files to reduce their sizes, making them easier to distribute on the Internet on various database. The most prominent example of such a database is the NCBI Sequence Read Archive [10].

The *gzip* application uses the DEFLATE algorithm (Phil Katz, 1993) to compress and decompress binary data. DEFLATE consists of two stages. In the first stage, the data is processed sequentially and LZ77 (Lempel-Ziv 1977) parsing is done. This encodes the data as a sequence of literals or off/length pairs. Off/length is the offset and length of the longest prefix in preceding 32KB context that can be used to replace data at this location. Once the LZ77 is done, Huffman coding is done to further encode the data compactly. The Huffman code trees are reset every block to give best results. DEFLATE decompression is the opposite process - the compressed data is worked upon by Huffman decoding. This writes down the data in a 32KB circular queue, which is then used by LZ77 decoding to decompress the data entirely.

Similar to the standalone *gzip* software application, *zlib* [2] is a software library for compression/decompression of general data streams using the same compression algorithm, DEFLATE. It can be thought of as a generalized version of gzip. We found that zlib contains an example program in its source, called *zran.c*. This example program demonstrates the use of certain zlib API features by accessing a gzip- compressed file from a random offset. It does this by fully decompressing the file, and building an index containing "access points" at uniform locations. Each "access point" includes the starting file offset, bit of each deflate block, and 32KB of uncompressed data immediately preceding the point. This index is then used to fetch some number of bytes from an arbitrary, user-provided location in the original gzip file (starting at the latest access

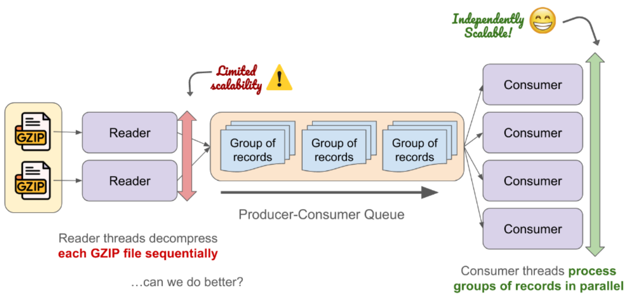


Fig. 2. Structure of FQParser. It can not read a single FASTQ file in parallel.

point ≤ desired location). This program does not save or load the index from disk, but rather stores it in memory for a one-time random fetch (after all, it is an example and not a full program).

These compressed FASTQ files are the first step of any pipeline for processing sequencing data. A recent such pipeline is Dr. Rob Patro's FQFeeder [11]. FQFeeder is a producer-consumer-based implementation that uses a producer thread pool for single and paired-end FASTQ parsing. A corresponding consumer thread pool does the actual computation over the parsed read records. As seen in Figure 2, the producer thread pool cannot have more threads than the number of files being parsed (since each file is parsed sequentially). We see that this is the major scalability bottleneck for the pipeline, which our tool solves. KSeq++[3] is used for parsing the files, and moodycamel's ConcurrentQueue [8] is used for producer and consumer queues. We want to have a similar producer-consumer structure to our library but with the parsing replaced with a parallel parser that can make use of an index for Gzipped files.

Related Work

pugz [5]

Pugz is a parallel algorithm that enables fast decompression of gzipped files. It can achieve this by being able to decompress from a random offset by creating the fully decompressed 32KB context using a two-pass heuristic approach. The 32KB context can be constructed almost always at low compression levels, while some approximations are needed at higher levels.

	Compressed FASTQ file size	FFIndex file size	FFIndex creation time	FFIndex reading time	% of bytes to store uncompressed states	% of bytes to store offset locations of read records
Salmonella	1.4 MB	39 KB	56 ms	<1ms	99.9%	0.1%
Ecoli	39 MB	5 MB	2819 ms	58 ms	62.9%	37.1%
Dataset C - Illumina	520 MB	38 MB	18846 ms	315 ms	47.5%	52.5%
Nematode	655 MB	48 MB	25816 ms	400 ms	61.2%	38.8%
ENCFF000FEU	8.1GB	543MB	267 s	4.5 s	45%	54%

Table 1. File size and runtime statistics on FFIndex run for various inputs.

mgzip [7] and pgzip [12]

Python libraries built on Python's `zlib` and `gzip` wrappers. pgzip, a maintained fork of mgzip, compresses uncompressed buffers by breaking them into blocks, and compressing each block in parallel. The metadata of all blocks is inserted into the `FEXTRA` field within the gzip file. This allows for parallel decompression by reading this metadata. While they maintain backward compatibility with `gzip` (as in, the `gzip` utility can decompress the generated `gzip` file), they generate a different `gzip` file due to independent block compression, lacking the desired "original `gzip` + supplementary index file" functionality we want.

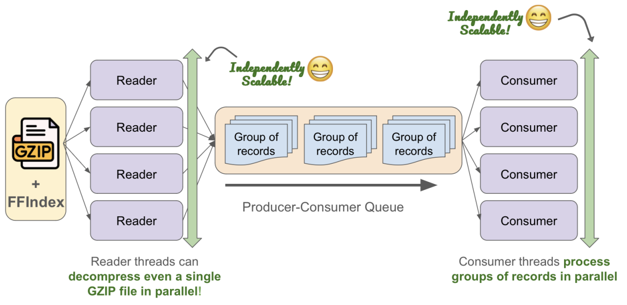


Fig. 3. Structure of FFParser and how it can be used to independently scale on a single FAST-Q file.

Our Approach

FFIndex

In this section we describe our design for FFIndex, the index structure we use to enable parallel parsing of gziped FAST-Q files.

Base framework

We built our solution on top of the `zran.c` codebase [2], which demonstrates building an index that help access a random offset within a gzip file. The main idea behind the index is that DEFLATE algorithm used by gzip requires a 32-KB maximum context window to decompress data at any offset. Zran uses this property to build an index that stores access points at fixed `SPAN` of uncompressed byte offsets. At each access point, zran stores 32KB of uncompressed data prior to the access point. For reading from a random offset, Figure 1 (left) shows the structure of the zran index.

Augmenting zran.c for sequencing data

FAST-Q files contains read records that are not of fixed size and so each record can span a variable number of bytes. Any application that reads FAST-Q files needs to read records as atomic blocks and needs to be able to index records to be able to match them in paired end reads. Therefore, we needed to augment zran's index to support record level access. We do so by maintaining an auxillary mapping from record index to the corresponding uncompressed byte offset. We augment the KSeq++ [3] to also compute the byte offset from the starting of the file while parsing the FASTQ file. This mapping is built when the original zran index is built stored, alongside the list of access points. Figure 1 (right) shows augmentation of zran's index with our read to offset mapping.

Storing FFIndex

An important point to note here is that we observed that storing the index in uncompressed form led to large file size. This depends on the number of access points stored but it was a significant overhead, so we also store the index in a gzip compressed form. This leads to the index size being around 8-10% of the original FAST-Q file size. This increases the index creation time and index read time. Index creation time is a one time cost, therefore we believe that this is a reasonable trade-off. The read time is considerably less then overall runtime of the programs. This can be further optimized by also reading the index in parallel by creating an index for the index using the same approach. Statistics around index size and creation/read time can be found in table 1

FFParser

Next, we describe our design for the producer-consumer based parallel reader that reads a FAST-Q file in parallel using the FFIndex. For the producer-consumer queue, we use moodycamel's ConcurrentQueue [8]: a lock-free queue implementation that supports multiple producers and consumers. This is similar to FQFeeder's design but we have not optimized the queue as well as FQFeeder has. This is because the focus of our project is how to build and read the index which is the main source of parallelism in the system. The index is read once and shared among all the producers that read blocks of records from the FAST-Q file. Once a parser is done reading its block of records, it moves to the next block not being read by any other parser. The records read are parsed using KSeq++ [3] and converted to a well formed C++ structure that can be used by downstream tasks. Figure 3 shows a basic structure of FFParser and how it can be used to independently scale on a single FAST-Q file, as opposed to FQFeeder which needs multiple FAST-Q files to scale the number of producers.

Evaluation

In this section, we discuss our experimental setup and results for benchmarking our tool.

Experimental setup

We take a multi-faceted approach to evaluate our tool. We need to evaluate two critical aspects of our tool: (a) How lightweight is the FFIndex? (b) Is FPParser correct and how much parallelism can we achieve with it?.

For the the first aspect, we evaluate the index creation times, reading times and most importantly the index sizes for a range of FAST-Q files of different sizes. For the evaluation we kept the span of access points to be constants, which generated different index sizes for different FAST-Q files. Ideally, we would like to also evaluate how many access points are needed for a given file size but we leave this for future work. We focus on showing that the index size is small enough with enough access points to allow for parallel reading of the FAST-Q file by anywhere between 16-128 threads. This number represents the number of cores on a typically high-end server machine and since our solution does not span multiple machines, we believe this is a reasonable assumption. The index creation times and read times are measured on a single core of an M2 Pro Macbook Pro.

For the second aspect, we evaluate the speedups we can achieve with FFParser. We run FFParser on two different FAST-Q files: Nematode [9] and ENCFF000FEU [13], which are 655MB and 8.1GB respectively in compressed state. We run the parser in each case with a fixed number of consumer threads and vary the number of producer threads. The parser is run on a single node of Zaratan cluster [14] where each compute node has an AMD EPYC 7763, 128 core CPU. FFParser is run for different indices with different number of access points. The downstream task used for benchmarking is a simple task of counting the number of read records in the FAST-Q files by the consumer threads. We compare our tool's performance with a sequential reader and FQFeeder [11], using the same number of consumer threads in the latter case. Please note that we also ran the experiment using pgzip (python) over nematode file. It took almost 10x more time than the C++ based vanilla sequential baseline, therefore we are not using it as a baseline for further evaluation.

Results

Evaluating FFIndex

Table 1 shows the index size for FASTQ files of different sizes. The index size is around 5-10% when compressed, and this size can be adjusted by controlling the number of SPAN bytes after which access points are created. Furthermore, the index size can be reduced by minimizing the data stored for byte offsets of read records. This data constitutes around 40-50% for large FASTQ files. In the current implementation, the byte offset data is stored for each read record. This data is not needed for each record because consecutive records are read by a single reader thread. By storing the offset locations only for the read records of interest, the index size can be significantly reduced. We need to investigate the ideal number of read records after which we can store this data while ensuring strong scaling of performance.

We also report index creation times, which can grow to a few minutes for large files. For example, for 8.1GB ENCFF000FEU FASTQ file, it took 4 minutes to generate the index. However, this is acceptable as it is a one-time cost and can be amortized over multiple runs. The read times are also reported which are significantly less than the overall runtime. This performance can be further improved by reading the index in parallel using the same approach.

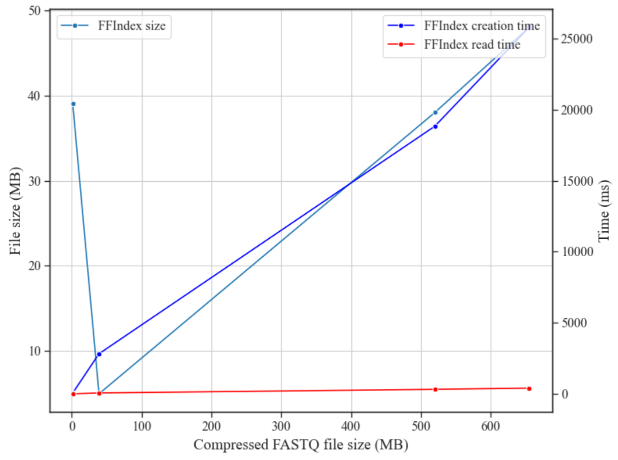


Fig. 4. Scaling of FFIndex creation time, read time, and size as input FASTQ file increases.

Evaluating FFParser

First, we evaluate the correctness of FFParser by comparing the output of our downstream nucleotide counting task with the sequential reader. Figure 6 shows the output of the nucleotide counting task for the ENCFF000FEU file for the sequential reader and FFParser. We can see that the outputs match exactly. We also ran the same test for other FAST-Q files and found the same results. This demonstrates that FFParser is correct and can be used for downstream tasks.

Figure 5 shows the strong scaling results for FFParser on the two different FAST-Q files - Nematode (left) and ENCFF000FEU (right). We can see that FFParser scales near linearly with the number of producer threads for both the files. We also see that FFParser outperforms the sequential reader and FQFeeder with the same number of consumer threads by scaling the number of producer threads.

We notice that for lower producer threads, the performance of FFParser is worse than the sequential reader and FQFeeder. Ideally, our tool should match FQFeeder's performance with the same number of consumer threads and 1 producer thread. We believe that the difference in performance is due to the fact that we have not optimized the producer-consumer queue as well as FQFeeder has. Furthermore, we fix the number of records to be read per producer thread to be 1000. This means that even if we have a single producer thread, we will read 1000 records at a time. Currently, in our implementation, such a case is handled by the thread calling a binary search on the index to find the access point to start from. Clearly, for a single producer thread, this is not needed as we can just read the records sequentially. This case needs to be handled in our implementation by having the ability to start reading directly from the last byte read by that thread if no other thread is reading that block of records. We leave this for future work.

Secondly, we notice that the performance of FFParser varies greatly with the number of access points in the index. This is expected again because every producer thread reads a fixed

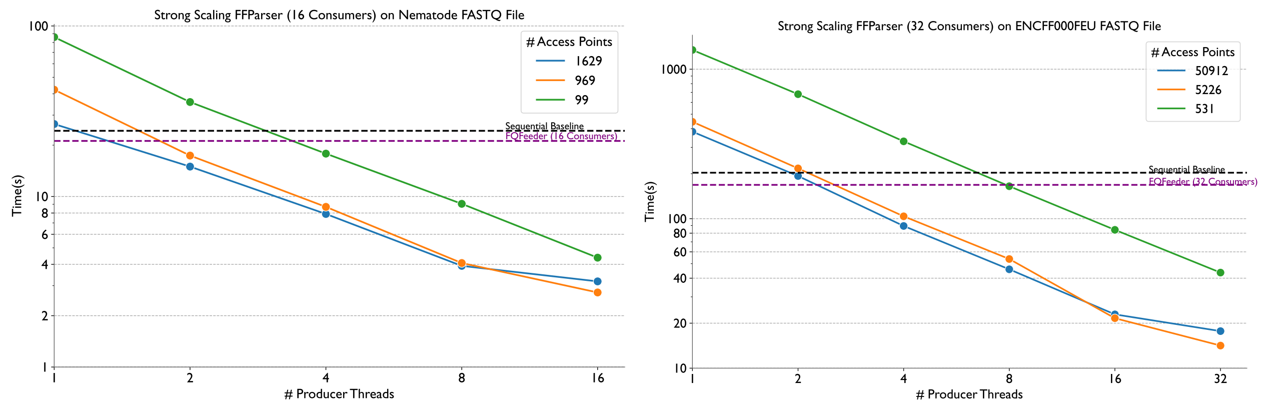
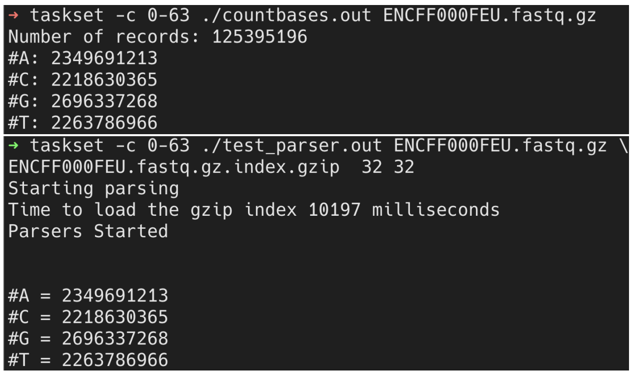


Fig. 5. Strong scaling results for FFParser on Nematode (left) and ENCFF000FEU (right) FAST-Q files. We can see that we acheive near linear scaling for both the files and outperform the sequential reader and FQFeeder with higher number of producer threads. The plots also show a trend that the performance of FFParser varies greatly with the number of access points in the index with fewer access points leading to worse performance.



```
➜ taskset -c 0-63 ./countbases.out ENCFF000FEU.fastq.gz
Number of records: 125395196
#A: 2349691213
#C: 2218630365
#G: 2696337268
#T: 2263786966
```

```
➜ taskset -c 0-63 ./test_parser.out ENCFF000FEU.fastq.gz \
ENCFF000FEU.fastq.gz.index.gzip  32 32
Starting parsing
Time to load the gzip index 10197 milliseconds
Parsers Started


#A = 2349691213
#C = 2218630365
#G = 2696337268
#T = 2263786966
```

Fig. 6. Correctness of FFParser. We compare the output of the nucleotide counting task for the sequential reader (top) and FFParser (bottom) on the ENCFF000FEU FAST-Q file. We can see that the outputs match exactly.

number of records. If the number of access points → 1, then it is equivalent to having no index at all and each thread will search for the the nearest access point to start reading from, which in this case is the beginning of the file. Thus each thread will read the file sequentially to get to the block of records it needs to read. This is extremely inefficient. We beleive that this can be alleviated by configuring the number of records read per producer thread as follows:

$$\text{records per thread} \propto \frac{\text{total records}}{\text{number of access points}}$$

This way, each thread will read a larger number of records if the number of access points is less and vice versa. This will also ensure that ideally only one thread reads a block of records between two access points. We were not able to implement this in our current version of FFParser and leave this for future work.

Despite, these identified bottlenecks in the implementation (which are easy to fix), we can see that FFParser scales extremely well and still outperforms the sequential reader and FQFeeder by almost 10x with the largest count of producer threads in each experiment. This demonstrates that FFParser greatly benefits from the parallelism introduced by the FFIndex.

Limitations and Future Work

To further demonstrate the effectiveness of FFIndex and FFParser, as well as to study scalability bottlenecks, we can test our implementations using FAST-Q files of other sizes. Because building the FFIndex is a one time work, efforts should be spent optimizing the runtime of FFParser instead. One way of accomplishing this is by collecting detailed traces of FFParser's executions, and understanding where the bottlenecks are located.

The most significant potential for future optimizations is reducing the size of FFIndex. Since FFIndex builds on zran's index construction algorithm, we currently create access points that are around SPAN bytes apart. We also store a mapping of record boundaries to their respective byte offsets in the gzipped file. One potential optimization that may be possible is to construct access points exactly at record boundaries, so that these two data structures can be represented as one, helping us reduce the size of the index.

Each of the reader threads of FFParser performs a binary search on the access points to determine where to start decompressing the gzipped file. By collapsing the two data structures, we could theoretically avoid this binary search. However, these binary searches are performed independently in parallel, and take a very insignificant amount of time compared to downstream tasks that dominate in runtime. Because this increase in complexity may not be worth the runtime benefit, we are not sure this is worth considering in further work.

Lastly, the producer-consumer model in FFParser is a proof-of-concept of how FFIndex can be utilized in an end-to-end Fast-Q processing pipeline. However, our evaluation from section 5 shows that the current implementation of FFParser is not optimised for cases where the number of producer threads is very small and the number of access points are small. We have identified the cause for this and present easy potential solutions to these problems in section 5. We were not able to incorporate these solutions in our current implementation due to time constraints. The implementation of our architecture is therefore, not production ready and is to show the potential of FFIndex in a parallel processing pipeline. We also beleive a more productive use of time would be to adding Index as a plug-in to FQFeeder, which does have an optimized producer-consumer pipeline. Using FFParser, FQFeeder can fully exploit parallelism and avoid the scalability bottleneck in reading files, if provided the FFIndex.

Conclusion

Our work addresses a significant bottleneck faced by members of the genomics community: efficiently parsing large, compressed FASTQ files in parallel. In this project, we present FFIndex, a lightweight supplementary index to gzipped FASTQ files that enables the parsing of a single highly compressed FASTQ file in parallel.

To demonstrate its utility, we also implemented FFParser, a multi-threaded FASTQ parser that achieves a near-linear speedup compared to serial parsing. By using FFIndex, each of the readers in FFParser can parse groups of records within a compressed FASTQ file in parallel, and enqueue them for downstream processing. Our results show that FFParser achieves strong scaling as we add the number of producers.

By overcoming the bottleneck of needing to parse each FASTQ file sequentially, our work enables much better utilization of available modern hardware. We hope that researchers find FFIndex and FFParser useful in speeding up their workflows, cutting costs, and potentially making prohibitive tasks accessible.

References

1. Flaticon. Flaticon. `https://www.flaticon.com/free-icon/flying-fish_6255787?term=flying+fish&page=1&position=3&origin=search&related_id=6255787`, 2024.
2. Jean-loup Gailly and Mark Adler. zlib. `https://www.zlib.net/`, 2024.
3. Ali Ghaffaari. Kseq++. `https://github.com/cartoonist/kseqpp`.
4. Maël Kerbiriou and Rayan Chikhi. Parallel decompression of gzip-compressed files and random access to dna sequences. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 209–217. IEEE, 2019.
5. Maël Kerbiriou. pugz. `https://github.com/Piezoid/pugz`.
6. Ben Langmead, Christopher Wilks, Valentin Antonescu, and Rone Charles. Scaling read aligners to hundreds of threads on general-purpose processors. *Bioinformatics*, 35(3):421–432, 2019.
7. Vincent Li. mgzip. `https://github.com/vinlyx/mgzip`.
8. moodycamel. moodycamel's concurrentqueue. `https://github.com/cameron314/concurrentqueue`.
9. National Library of Medicine. nlm. `https://www.ncbi.nlm.nih.gov/sra?term=nematode+SRR28592514`, 2024.
10. National Library of Medicine. Sequence read archive. `https://www.ncbi.nlm.nih.gov/sra`, 2024.
11. Rob Patro. Fqfeeder. `https://github.com/rob-p/FQFeeder`.
12. pgzip. pgzip. `https://github.com/pgzip/pgzip`.
13. ENCODE Project. Encff000feu. `https://www.encodeproject.org/experiments/ENCSR000COU/`, 2024.
14. UMD. The zaratan hpc cluster. `https://hpcc.umd.edu/hpcc/zaratan.html/`.