Karlsruhe Institute of Technology

Faster Sorting of Aligned DNA-Read Files

Bachelor's Thesis of

Dominik Siebelt

At the KIT Department of Informatics
Institute of Theoretical Informatics (ITI)

First examiner:	Prof. Dr. Alexandros Stamatakis
Second examiner:	Prof. Dr. Peter Sanders
First advisor:	M.Sc. Lukas Hübner

01. February 2024 – 03. June 2024

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 03.06.2024

.....
(Dominik Siebelt)

Abstract. In the analysis of DNA sequencing data for finding disease causing mutations, to understand evolutionary relationships between species, and to find variants, DNA-Reads are compared to a reference genome. A reference genome is a representative example for a set of genes of a species. Sorting these aligned DNA-Reads by their position within the reference sequence is a crucial step in many of these downstream analyses. SAMtools `sort`, a widely used tool, performs external memory sorting of aligned DNA-Reads stored in the BAM format (Binary Alignment Map). This format allows for compressed storage of alignment data. SAMtools `sort` provides the most comprehensive set of features while exhibiting demonstrably faster execution times than its open source alternatives. In this work, we analyze SAMtools `sort` for sorting BAM files and propose methods to reduce its runtime. We divide the analysis into three parts: management of temporary files, compression, and input/output (IO). For the management of temporary files, we find that the maximum number of temporary files SAMtools `sort` can open concurrently is lower than the maximum number of open files permitted by the operating system. This results in an unnecessarily high number of merges of temporary files into larger temporary files, introducing overhead as SAMtools `sort` performs extra write and compression operations. To overcome this, we propose a dynamic limit for the number of temporary files, adapting to the operating system's soft limit for open files. For compression, we test seven different libraries for compatible compression and a range of compression levels, identifying options that offer faster compression and result in a speedup of up to five times in single-threaded execution of SAMtools `sort`. For IO, we demonstrate that a minimal level of compression avoids IO overhead, thereby reducing the runtime of SAMtools `sort` compared to uncompressed output. However, we also show that uncompressed output can be used in the pipelining of SAMtools commands to reduce the runtime of subsequent SAMtools commands. Our proposed modifications to SAMtools `sort` and user behavior have the potential to achieve speedups of up to 6. This represents an important contribution to the field of bioinformatics, considering the widespread adoption of SAMtools `sort` evidenced by its over 5,000 citations and over 5.1 million downloads through Bioconda.

Zusammenfassung. Bei der Analyse von DNA-Sequenzierungsdaten, um krankheitsverursachende Mutationen zu finden, evolutionäre Beziehungen zwischen Arten zu verstehen und Varianten zu identifizieren, werden DNA-Reads mit einem Referenzgenom verglichen. Ein Referenzgenom ist ein repräsentatives Beispiel für ein Set von Genen einer Spezies. Das Sortieren dieser aligned DNA-Reads nach ihrer Position innerhalb der Referenzsequenz ist ein entscheidender Schritt in vielen dieser nachgelagerten Analysen. SAMtools `sort`, ein weit verbreitetes Tool, führt das externe Speichersortieren von aligned DNA-Reads durch, die im BAM-Format (Binary Alignment Map) gespeichert sind. Dieses Format ermöglicht die komprimierte Speicherung von Ausrichtungsdaten. SAMtools `sort` bietet das umfassendste Set an Funktionen und weist nachweislich schnellere Ausführungszeiten auf als seine Open-Source-Alternativen. In dieser Arbeit analysieren wir SAMtools `sort` zum Sortieren von BAM-Dateien und schlagen Methoden zur Reduzierung der Laufzeit vor. Wir unterteilen die Analyse in drei Teile: Verwaltung von Zwischendateien, Komprimierung und Eingabe/Ausgabe (IO). Für die Verwaltung von Zwischendateien stellen wir fest, dass die maximale Anzahl Zwischendateien, die SAMtools `sort` gleichzeitig öffnen kann, geringer ist als die maximale Anzahl offener Dateien, die vom Betriebssystem zugelassen wird. Dies führt zu einer unnötig hohen Anzahl von Merges von Zwischendateien zu größeren Zwischendateien, was zusätzlichen Aufwand bedeutet, da SAMtools `sort` zusätzliche Schreib- und Komprimierungsvorgänge durchführt. Um dies zu überwinden, schlagen wir ein dynamisches Limit für die Anzahl von Zwischendateien vor, das sich an das soft-limit des Betriebssystems für offene Dateien anpasst. Für die Komprimierung testen wir sieben verschiedene kompatible Libraries und eine Reihe von Komprimierungsstufen (Level), um Möglichkeiten zu identifizieren, die schnellere Komprimierung bieten und zu einer bis zu fünfmal schnelleren Ausführung von SAMtools `sort` im Single-Thread-Betrieb führen. Für die Ein- und Ausgabe zeigen wir, dass ein minimales Komprimierungsniveau IO-Overhead vermeidet und dadurch die Laufzeit von SAMtools `sort` im Vergleich zu unkomprimiertem Output reduziert. Wir zeigen jedoch auch, dass unkomprimierter Output im Pipelining von SAMtools-Befehlen verwendet werden kann, um die Laufzeit aufeinanderfolgender SAMtools-Befehle zu reduzieren. Unsere vorgeschlagenen Änderungen an SAMtools `sort` und dem Benutzerverhalten haben das Potenzial, Speedups von bis zu 6 zu erreichen. Dies stellt einen wichtigen Beitrag im Bereich der Bioinformatik dar, angesichts der weit verbreiteten Verwendung von SAMtools `sort`, die durch über 5.000 Zitate und über 5,1 Millionen Downloads über Bioconda belegt wird.

Table of Contents

1 Introduction

1.1 Motivation

Analysis of aligned DNA-Read data is a crucial part of modern bioinformatics, with applications for scientific [25,35,39] and medical [36,22,54] purposes. DNA-Reads are short (typically 250–800 bases long) sequences of DNA bases encoding genetic information. They are aligned to a reference sequence, which serves as a standard DNA or RNA sequence for comparison within a species. The alignment can be used to find variants [47,44,15], disease causing mutations [41,30,34] and understand evolutionary relationships between species [40,46,20]. In order to analyze specific parts of a genome, e.g., a single chromosome, DNA-Reads are sorted. This also improves the performance of many other downstream analysis tasks and algorithms [43,37], e.g., finding and removing duplicate reads [23,6].

To address the vast amounts of data (ranging from less than 10 GB up to terabytes per sequencing run) generated by modern DNA sequencing machines, highly efficient tools and data formats are needed. Developed during the 1000 Genome Project [50], the SAM and BAM formats have found widespread use for storing alignment information to DNA sequences. Developed alongside these formats, SAMtools became a standard tool for manipulating SAM and BAM files. Beyond its collection of 38 commands for filtering, merging of aligned DNA-Read files and various other tasks, SAMtools provides the SAMtools `sort` utility to reorder the DNA sequences stored in DNA-Read files such as SAM and BAM files.

We aim to optimize the computational cost and processing time of the SAMtools `sort` command for sorting DNA-Reads in BAM files. BAM is a file format capable of storing alignment information to DNA sequences in a binary and compressed form. Given the substantial storage requirements (uncompressed BAM approximately 400 GB per genome [7]) of DNA data, the BAM file format incorporates compression to minimize storage costs, optimize storage capacity, and facilitate faster network transfer.

SAMtools `sort` sorts DNA-Read files by various sorting criteria, with the default order sorting aligned DNA-Reads by the ID of the reference sequence to which the DNA-Read is mapped, followed by the position of the mapping on this reference sequence. This enables fast random access to specific regions of interest via index files.

BAM files are compressed with BGZF, a compression method allowing to access the content of the file in blocks without the necessity of decompressing all preceding blocks of the file. The compression format internally uses the popular zlib compression library for GZIP compression. Zlib is open source and used extensively in a wide array of applications (e.g., HTML-Compression). Thus, competing libraries have been implemented, offering GZIP compatible compression with higher throughput and smaller resulting files.

In this thesis, we present three approaches to speed up the sorting of DNA-Reads stored in BAM files utilizing SAMtools `sort`:

SAMtools `sort` utilizes an external memory sorting algorithm. In situations with limited memory, it utilizes temporary files and merges them subsequently. As writing temporary files requires additional computation steps, such as compression, writing temporary files is computationally demanding. Especially, merging temporary files into other temporary files invokes computational overhead, as SAMtools `sort` reads and decompresses already written temporary files and compresses and writes their content again. In this thesis, we investigate the runtime implications of SAMtools `sort` writing and merging temporary files. We propose parameter settings and changes in SAMtools `sort`'s merging strategy, reducing the amount of merges and thus lowering the runtime of SAMtools `sort`.

SAMtools `sort` dedicates a substantial amount of its runtime to compressing temporary and output files. To maintain the advantages of compression but reduce its impact on SAMtools `sort`'s total runtime, we examine various GZIP-compatible compression libraries and the effects of different compression levels.

Finally, we assess the impacts of IO operations and limitations of IO devices and propose recommendations to minimize or eliminate IO blocklenecks.

2 Prerequisites

2.1 SAMtools

SAMtools [24] is a collection of tools to work on alignment data, such as aligned DNA-Reads. It relies on the co-developed HTSlib [19] for reading and writing information files, namely SAM, BAM, and CRAM files. With SAMtools `sort`, SAMtools offers functionality for sorting of aligned DNA-Read files, which is the focus here. In addition, SAMtools offers functionality for different operations on alignment data, such as format conversion, statistics, variant calling and many more.

2.2 Aligned DNA-Reads

DNA-Reads are short (typically 250–800 nucleotides long [32]) sequences of nucleotides, the fundamental building blocks of DNA. These nucleotides are denoted by their bases, adenine (`A`), guanine (`G`), cytosine (`C`), and thymine (`T`). A DNA-Read can consist of multiple contiguous sequences.

Aligned DNA-Reads are DNA-Reads aligned to a reference sequence, which serves as a standard DNA or RNA sequence for comparison within a species. The alignment may include insertions, deletions, mismatches, and skipped parts of the reference sequence. Additionally, a step called *clipping* excludes parts of the sequenced fragment with low read qualities from the alignment to improve the alignment of the remaining high-quality sequence with the reference sequence. Also, changes in the direction of the alignment on the reference are possible. In alignment files, changes in direction of the alignment lead to splitting up the DNA-Read into multiple sequences, one for each contiguous part of the DNA-Read aligned in the same direction.

2.3 SAM and BAM files

A SAM (Sequence Alignment Map) file as specified by Li et al. [38] is used to store the alignment of sequences against reference sequences. It consists of a header section and an alignment section. The header section contains meta information such as the format version or the sorting of the content and a dictionary of the reference sequences, whereas the alignment section contains aligned segments with alignment information and meta information such as the read quality. A segment is a continuous sequence or subsequence of a raw DNA-Read. Aligned DNA-Reads are possibly put into multiple records with different segments in the alignment section, as single BAM records can not store changes in directions of the alignment on the reference sequence.

The alignment information primarily includes the ID of the reference sequence to which the alignment is mapped, the position in the reference sequence where the alignment starts, and detail on the alignment at this position (match, mismatch, insertion, or deletion).

A BAM (Binary Alignment Map) file is the binary representation of a SAM file. Compared to the SAM format, this format utilizes a 4-bit encoding for DNA sequences, a 3-bit encoding for CIGAR symbols, and adopts a 0-based coordinate system for positions. Furthermore, a BAM file is per default *BGZF* compressed.

2.4 The DEFLATE format and algorithm

The DEFLATE format [26] is a format specifying data compressed with a combination of an LZ77 algorithm [55] and Huffman Coding [33]. Compressed data is structured as a sequence of consecutive blocks. These blocks contain strings and references to previous strings (matches). The references to previous strings contain the distance to and the length of the matching part. The bytes of the strings which are no matches to previous strings and the pointers, which consist of distances and lengths, are compressed using Huffman coding. The trees representing the Huffman codes are Huffman coded as well. The DEFLATE algorithm is an algorithm capable of producing an output stream in the DEFLATE format given an input data stream.

2.5 GZIP and zlib

GZIP [27] is a container format for data compressed in the DEFLATE format. A GZIP file consists of a series of *members* which consist of a header, a series of compressed blocks, and a footer. The header contains meta information on the compressed files (e.g., the file name and the modification time) and can also contain extra fields which in the BGZF file format are used to identify a member as part of a BGZF file and to store the length of the compressed member. The compressed blocks contain the compressed data and the footer contains a checksum and the size of the uncompressed content of the blocks modulo 2^{32}.

The c library zlib is an implementation of the DEFLATE algorithm. It can produce output in GZIP, ZLIB, or raw DEFLATE format. The ZLIB format

is a wrapper for DEFLATE similar to the GZIP format but with less header information and another checksum algorithm (Adler32 instead of crc32).

2.6 BGZF Compression

The Blocked GNU Zip Format (BGZF) is a lossless compression method proposed together with the BAM format. Widely used compression methods like GZIP compress a file from the beginning to the end in one piece. This has the advantage of allowing matching segments of the file to be located over a greater range. Thus, the compression method is able to reduce the file size more effectively, as repeated sequences can be identified throughout the entire file. However, to decompress such a compressed file, it also must be read from the beginning and, depending on the compression method, decompressed at least until the point of interest.

Given that not all regions of large alignment data files are relevant for every analysis, random access is required for efficient analysis of specific data subsets. To achieve this, BGZF utilizes GZIP [27] to compress large files into blocks of less than 64 KB size (compressed and uncompressed). GZIP uses the DEFLATE algorithm [26] to compress these individual blocks, which it then subsequently concatenates. Thus, fast random access using index files is possible. In an index file, the position of a piece of information is stored as a 64 bit integer. This index is divided into a 48 bit block index and a 16 bit offset in the respective block.

The BGZF format leverages compatibility with GZIP, enabling any standard GZIP decompression tool to handle BGZF-compressed files. This compatibility stems from BGZF exploiting GZIP's ability to combine multiple compressed blocks into a single file. Given that GZIP is highly prevalent as a compression technique, there are numerous compatible compression and decompression libraries for all platforms. Thus, employing the open-source GZIP internally simplifies the development of other legacy tools working with BGZF-compatible compression.

Like GZIP, BGZF supports compression levels ranging from 1 (fastest but largest file size) to 9 (slowest but smallest file size) analogously to the compression levels used for the underlying GZIP compression. We show details on the resulting files sizes and throughput differences between the compression levels in Section 6.3.

3 Related Work

3.1 Sorting Tools for aligned DNA-Read Files

SAMtools is the most commonly used tool for manipulating SAM and BAM files, which store information on aligned DNA-Reads. With over 5.1 Million downloads (05.2024), it is the most downloaded package from the Bioconda [51] channel, which is a common source for bioinformatic tools in the package and dependency manager Conda. However, there are some alternatives for sorting aligned DNA-Read files, such as Sambamba [48], Picard [8] and NovoSort [4]. Picard, written in Java, provides a simpler interface for sorting SAM and BAM files, but its functions are limited compared to SAMtools `sort`, and it does not prioritize efficiency. NovoSort is a commercial program and, therefore, we could not test it. Sambamba, written in the D programming language [14], aims to provide a subset of SAMtools functionality, including sort, but with greater efficiency through higher parallelization. With SAMtools introducing parallelism in version 1.4 in 2017, the advantage Sambamba had over SAMtools diminished. For sorting, we report SAMtools 1.19 to be 3 times faster than Sambamba 1.0.0 for small files (2.3 GiB), both utilizing a total of 16 threads and 48 GiB of memory. For larger files, the speedup of SAMtools `sort` compared to Sambamba `sort` increases to up to 5 at the largest input file we tested (215 GiB). A comparison of the runtime of SAMtools `sort` and Sambamba `sort` on different input file sizes is shown in Figure 1.

3.2 Alternative Compression Methods and File Formats

Compression using BGZF, which is part of the specification of BAM files, utilizes GZIP. As a general purpose compression method, GZIP is not specialized in compressing DNA data. Hence, a variety of compression algorithms have been developed specifically for DNA data [31]. These algorithms aim to minimize the resulting file size, compression speed, memory usage during compression or decompression, or the decompression speed, but typically focus primarily on minimizing the resulting file size. Compression algorithms are divided in *reference-based* and *reference-free* methods.

Reference-based algorithms utilize the reference sequence a DNA-Read is aligned to for compression. By storing only differences between each DNA-Read and the reference sequence, they archive better compression than reference-free methods. NGC [45] exemplifies this approach. It leverages reference sequences for compression and further enhances efficiency by splitting data types within a BAM file, like sequences, sequence names, and read qualities, into separate blocks for independent compression. Other algorithms and data formats like Goby [21], DeeZ [29], and CRAM [28] are also reference-based, but keep the advantage of enabling random access through index files. CRAM (Compressed Reference-oriented Alignment Map) is also built into HTSlib, the library SAMtools utilizes for file operations and the user can choose to use it as output for SAMtools `sort`. NGC, Goby and CRAM also offer lossy compression. Lossy compression

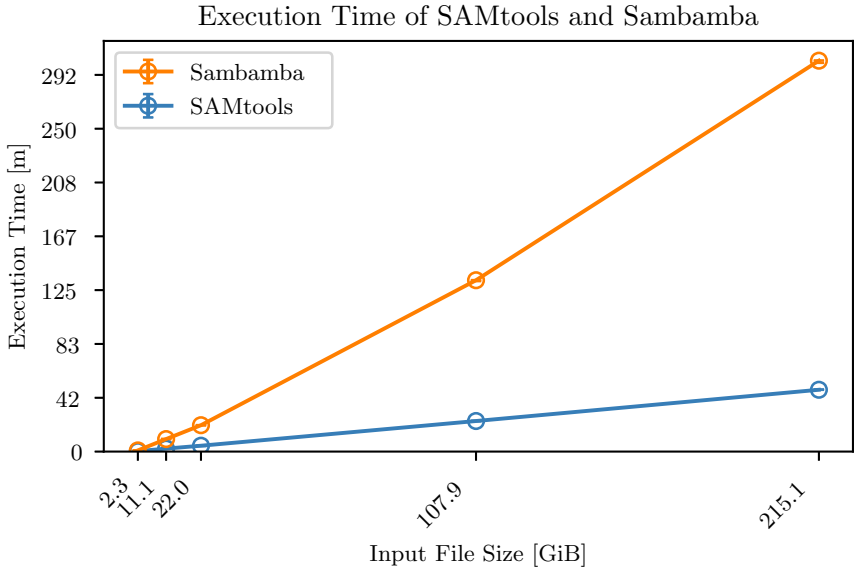

Fig. 1. Runtime of Sambamba and SAMtools for sorting BAM files of different sizes. Both tools are default installations via Bioconda and utilize a total of 16 threads and 48 GiB of memory. Data points represent median values across three replicate runs. Error bars depict the minimum and maximum values observed in these runs.

primarily targets meta information, including read qualities. Read qualities may not be crucial for every analysis and are independent of the reference sequence, making them less compressible. Thus, lossy compression of read qualities offers substantial storage savings.

Although reference-based compression methods for SAM files are still actively developed [16], they lack interoperability, as these compression methods are not widely used. An exception to this is CRAM, which is supported by HTSlib. Although CRAM files offer several advantages over BAM files, including better compression, BAM files remain more popular due to their wider support among software tools.

Reference-free compression methods like BGZF, which per specification BAM files are compressed with, do not require the reference sequence for compression and decompression, making them more flexible, as e.g., no reference is required for sorting. Furthermore, reference-based compression methods often necessitate sorted DNA-Reads, making them inapplicable to inputs for SAMtools `sort` and outputs sorted by read names or other criteria. However, the only commonly used reference-free compression method is the BAM format, which uses binary encoding for the bases a DNA sequence consists of and BGZF compression.

In summary, the BAM file format employing BGZF compression, which internally utilizes GZIP, is the most commonly used format for storing aligned DNA-

Reads. Given its widespread use, optimizing the speed of generating BAM files using SAMtools `sort` is essential. However, increasing adoption of the CRAM format might lessen the future significance of BAM file generation.

4 Algorithm (Version 1.19.2)

4.1 Prerequisites

The process of sorting alternates depending on some internal Constants and command-line-arguments: We focus on sorting by the ID of the reference a DNA-Read is aligned to, then the position where the alignment on the reference starts, and then the REVERSE flag which indicates, if the sequence is aligned forward or backward to the reference. This order is used by SAMtools per default, although other sorting criteria e.g., tags or the read name are possible.

The maximum amount of memory SAMtools `sort` utilizes for storing BAM records during the sorting process is calculated by the amount of memory the user specifies via the `-m` option multiplied by the (via `-@` option) assigned number of threads. We refer to the total amount as `max_mem`.

Users can specify the number of threads to use for SAMtools `sort` by the `-@` parameter. If it is set to 1, the operation is single threaded. If the user sets it to a number greater than one, SAMtools `sort` uses the specified number of threads in addition to the main thread.

SAMtools `sort` infers the in- and output formats from the input and output file names the user specifies. SAMtools `sort` passes the sorted aligned DNA-Read files it outputs as BAM file to standard output if no output file is specified.

The maximum number of temporary files is hard-coded as 64 in a constant named `MAX_TMP_FILES`.

Temporary files utilized in the sorting process are compressed with compression level 1. The compression level of the result file defaults to the default compression level used by the library that implements the compression. Usually this is compression level 6. The user can change the compression level of the output file via the `-l` parameter and set it to a number between 0 (no compression) and 9 (highest and slowest compression).

4.2 Sorting

SAMtools performs an external memory sort utilizing temporary files that are merged in the end. The sorting starts by sequentially reading BAM records from the input BAM file, or stream using HTSlib for parallel decompression. Once BAM records, which contain alignment information for DNA-Reads, exceed the memory limit given by `max_mem`, SAMtools `sort` splits these BAM records into as many in-memory vectors as threads are specified and afterward sorts them in parallel. For sorting, each thread used for sorting (on multithreading, each thread except the main thread) performs a radix sort.

Then, SAMtools `sort` performs a heap based merge. In the merge, SAMtools `sort` keeps a binary heap containing the smallest entry from each file, and in-memory vector that should be merged. Each of these entries is a BAM record with the lowest order from the respective file or in-memory vector, together with a reference to the file or in-memory vector. The merge works by outputting the entry of the heap with the lowest order, inserting the next entry from the file or

in-memory vector this entry came from into the heap, and adjusting the heap using the *sift-down* algorithm [18].

In the merge, SAMtools `sort` writes all the sorted in-memory vectors of BAM records to a single sorted temporary BAM file. In addition, SAMtools `sort` includes some of the previously created temporary files into the merge: The algorithm distinguishes between *small files* and *big files*. Small files are files generated by merging the sorted in-memory vectors of BAM records resulting from sorting them in parallel. If the number of small files is greater than half of the maximum allowed number of temporary files, SAMtools `sort` includes all small files it generated before into the merge (and deletes them afterward). The result of a merge of in-memory vectors of BAM records and temporary files is a big file. If the total number of files (small files and big files) exceeds the limit for temporary files, all previously generated temporary files including big files are included in the merge and deleted afterward. The resulting file is also counted as a big file, despite possibly being much larger than other big files generated by merging only small files. However, as the first merging of big files occurs at the 1120th temporary file[1], this is only relevant for combinations of enormous files and little memory.

In general, temporary files can be put into three categories: small files being at most as big as the sorted in-memory vectors of BAM records together, big files being at most as big as half of the maximum number of allowed temporary files times the maximum size for small files and one big file growing depending on the ratio of allocated memory to the size of the input file possibly to much larger size than the other big files.

After the merge, the algorithm repeats the previous steps from reading the input to merging, until SAMtools `sort` reaches the end of the input file or stream. As a last step, SAMtools `sort` sorts the remaining in-memory BAM records and merges them together with all temporary files into the output file. The sorting process flow is represented by the flowchart in Figure 2.

[1] This number is the result of adding $33 \cdot 33$ temporary files already merged into big files to 31 small files. Here we have to square 33, as 32 small files can exist, and the 33rd file is the big file which the result of a merge, but not counted among the small files. If 32 big files exist, there is still space for 32 small files, and they are merged to a 33rd big file, leaving only space for 31 small files in the next merging process.

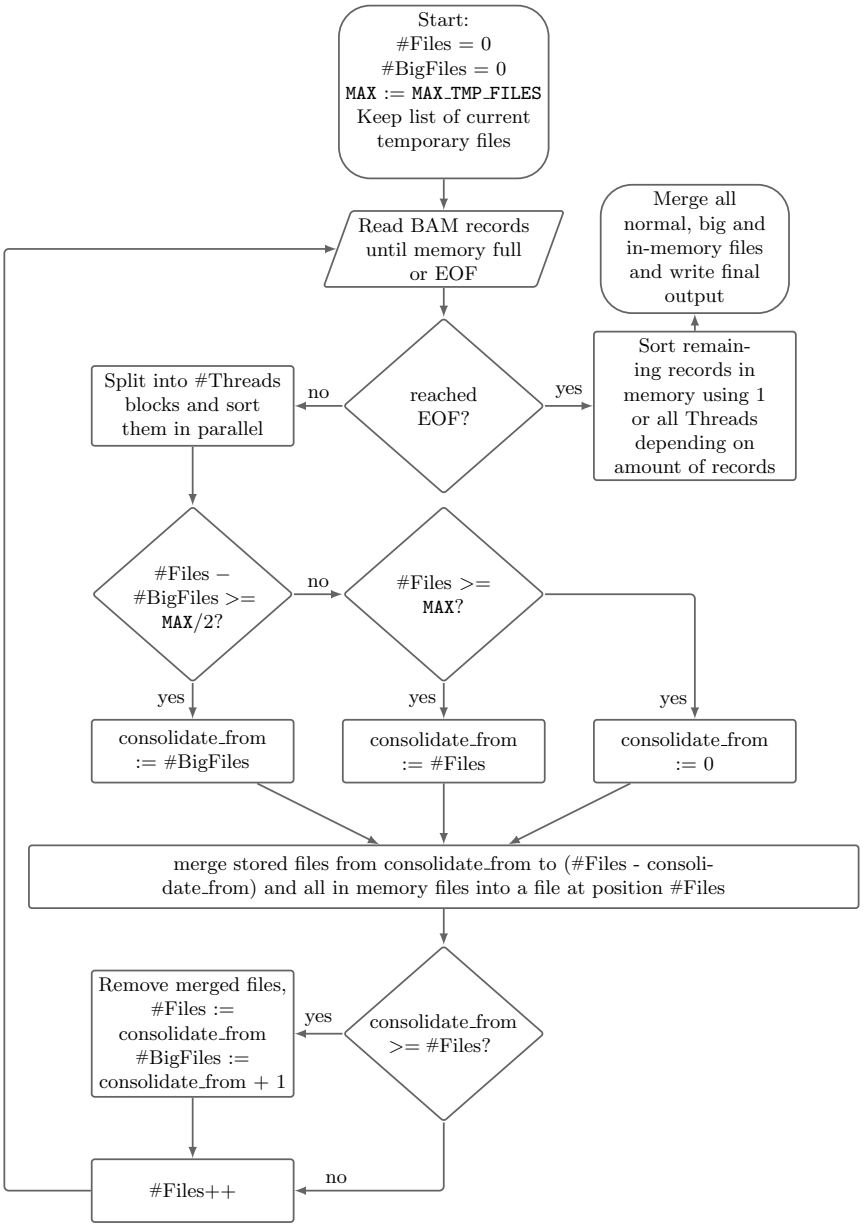

Fig. 2. Flow chart showing the current process of sorting, especially the choosing of files to be merged. The list of files is a 0-indexed list of their names. In the beginning it is empty, after BAM records are read the second time, there is a single record at position 0 and #Files is 1.

5 Temporary Files

5.1 Overview

Temporary files act as buffers for SAMtools `sort`, allowing it to process large datasets that exceed the available memory, while supporting streams as input and output. However, writing temporary files is time-consuming, as SAMtools `sort` compresses each temporary file, writes it to disk, and afterward decompresses and reads it again in order to merge the temporary files.

When looking at the time between reading and decompressing the input, and writing and compressing the output, decompressing and compressing temporary files during the sorting process is more time-consuming than the actual sorting of BAM records, which store the alignment information of DNA-Reads in a BAM file.

Most operating systems limit the number of files that a process is allowed to keep open simultaneously. To address this constraint, SAMtools `sort` employs a merging strategy, reducing the number of open files in the final merge that creates the output files. However, in every merge of temporary files, BAM records, that have been compressed and written to a temporary file before, are compressed and written again. This introduces overhead, as the computation time for decompressing and compressing the content of the merged temporary files again is added to the overall computation time. Thus, we aim at reducing the amount of temporary files SAMtools `sort` utilizes in order to minimize the frequency of SAMtools `sort` compressing and writing each BAM record. In addition, a dynamic merging strategy that adapts to the limitations of the operating system can reduce the number of merges needed for a given number of temporary files, leading to performance improvements.

5.2 Analysis

SAMtools `sort` creates a total of 40 temporary files when sorting a 215 GiB unsorted BAM file, utilizing 16 threads and a total of 32 GiB memory. Temporary files are BAM files like the output file, but since the main objective for temporary files is processing speed, not disk space, SAMtools `sort` compresses them with compression level 1. This is to maximize the throughput compared to higher compression levels but reduce IO overhead on writing uncompressed BAM files (Section 7.2). During writing, nearly all temporary files consume an average of 29.75 seconds from opening the file to closing it. In this time, SAMtools `sort` merges the 16 (one per thread) sorted vectors of BAM records in memory, compresses them, and writes them to the file. However, the 33rd file takes 945.38 seconds. That is 31.7 times the amount of time needed for each temporary file before.

SAMtools `sort` merges temporary files if it has written a certain number of temporary files. This is to limit the total number of temporary files SAMtools `sort` opens in the final merge. If a program opens too many files concurrently, the

operating system kills it (Section 5.3). Thus, merging of temporary files is necessary for SAMtools `sort`. To understand how many temporary files are written and when they are merged, we examine SAMtools `sort`'s merging strategy:

SAMtools `sort` enforces a predefined maximum number of 64 temporary files concurrently stored on disk. Until reaching half of this limit for temporary files, it writes all blocks that are results of sorting the amount of BAM records fitting into memory at once into a single temporary file (a "small file"). Whenever SAMtools `sort` has 32 (half of the limit for temporary files) small temporary files stored on disk concurrently, the next temporary file (a "big file") is a merge of all small temporary files written before together with the next in-memory vectors of sorted BAM records. In summary, on writing every 33rd file (half of the limit for temporary files plus one), SAMtools `sort` performs a merge of 32 small temporary files. This explains the increase in time at writing the 33rd temporary file from the example above: SAMtools `sort` reads every temporary file written before again, merges them and writes their content a second time.

The number of temporary files on the disk concurrently reaches the limit for temporary files at writing a total of more than the square of half the limit for temporary files (Footnote 1). If 33 big temporary files and 31 small temporary files are stored on disk concurrently, SAMtools `sort` merges all small and big temporary files into the next temporary file. For details, refer to Section 4.2.

The amount of merges depends on the number of temporary files SAMtools `sort` utilizes in total. This is mainly determined by the amount of the memory the user allocates to SAMtools `sort`. The user can configure the memory usage of SAMtools `sort` via the "`-m`" parameter. Defaulting to 768 MiB, it gets multiplied by the number of threads. We refer to the result as `max_mem`. It is the limit up to which SAMtools `sort` reads BAM records, which contain alignment information to a DNA-Read, into memory, before sorting them in parallel. SAMtools `sort` enforces at least 1 MiB of memory per thread to prevent the creation of a huge amount of temporary files. In general, sorting is faster the more memory the user allocates to SAMtools `sort`, although not in a linear proportion, as shown in Figure 3.

The execution time of SAMtools `sort` does not decrease on sorting a 2.3 GiB BAM file using between 400 MiB and 12800 MiB of memory.[2] To investigate further, we examine the amount of temporary files produced. The input file expands to larger than 12800 MiB, the second-highest memory limitation in Figure 3, as the internal representation of BAM records is not compressed in contrast to BAM records in a BAM file. At the highest setting (25600 MiB = 25 GiB), SAMtools `sort` utilizes no temporary file. At the next highest settings, it produces 1, 2, 4, ... temporary files, as the memory limitation `max_mem` halves to every next highest value.

Looking at the amount of temporary files generated, it is also possible to approximate the size of the BAM file in memory. At the memory limitation of

[2] Sorting a 2.3 GiB input file utilizing a memory limitation of 400 MiB, SAMtools `sort` creates approximately the same amount of temporary files as sorting a 200 GiB input file utilizing a memory limitation of 32 GiB.

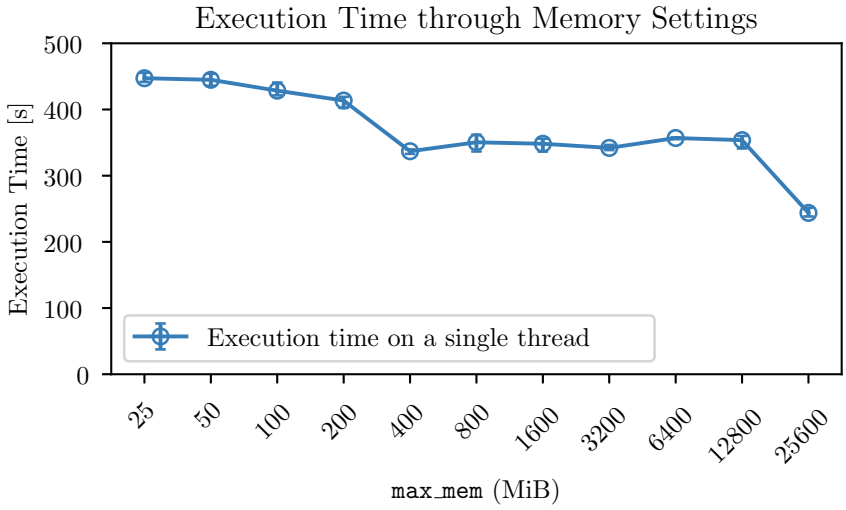

Fig. 3. Execution time of SAMtools `sort` on a 2.3 GiB BAM file on a single thread at different memory limitation settings. Data points: Median, Error bars: fastest and slowest of 3 runs.

400 MiB, SAMtools `sort` generates 32 temporary files. At 200 MiB `max_mem`, it generates 65 temporary files. This indicates, that after having processed 12800 MiB (= 400 MiB · 32) of data, 200 MiB are not enough to keep the remaining data in memory until the final merge into the output file, but 400 MiB are. For this reason, the size of the BAM file must increase to between 13000 MiB and 13200 MiB in memory, which represents an increase in size by factor 5.52.

These observations explain the missing speedup between the memory limitations of 400 MiB and 12800 MiB. In between these memory limitations, SAMtools `sort` writes exactly the same records to the disk, in exactly the same order. The only difference is the number of temporary files it splits them into.

This changes at the memory limitation of 200 MiB `max_mem`. The total of 65 produced temporary files means that SAMtools `sort` has to perform a single merge and generate a single big file before the final merge. This comes with additional time consumption because SAMtools `sort` reads the content of the first 32 temporary files from disk, decompresses, merges, compresses, and writes them to disk into a temporary file a second time.

At the memory limitation of 100 MiB SAMtools `sort` generates 3 big files, at 50 MiB 7, and at 25 MiB 15. This is also reflected in the total amount of bytes written. With the memory limitation settings larger than 200 MiB SAMtools `sort` utilizes temporary files, but does not merge them into big files. With these memory limitation settings, SAMtools `sort` writes a total of 2.4 GiB in temporary files. This number goes up to 3.7 GiB, 4.3 Gi, 4.6 GiB and 4.8 GiB for the memory limitations of 200 MiB, 100 MiB, 50 MiB, and 25 MiB `max_mem`.

Here, the increase in total written bytes for temporary files is not proportional to the amount of merges, as the size of the merged files shrinks with lowering the memory limitation `max_mem`. In addition, the proportional influence, merging of temporary files has on the runtime of SAMtools `sort` lowers with the number of performed merges: While writing the first big temporary file costs approximately the same as writing all temporary files before, writing the second one costs only a third of writing all files before, the next one 1/5 then 1/7 and so on.

The measurements above are unrealistic, as nowadays even Laptops have more memory installed. However, aligned DNA-Read files are usually several times larger than the used sample BAM file, which we sampled by randomly selecting 1% of BAM records from a real world BAM file. To get an impression of the impacts of increasing the file size, we look at the changes that come with the size increase.

Both compression and decompression work in $\mathcal{O}(n)$, ensured by the blockwise compression. The sorting algorithm SAMtools `sort` uses is a radix sort, which also is in $\mathcal{O}(n)$. For merging, SAMtools `sort` employs a heap based approach, which operates in $\mathcal{O}(n \log(k))$. Here, k is the number of sorted lists to be merged. Thus, in theory, keeping the same ratio of input size and available memory produces the same amount of temporary files. As all operations (except merging, which also depends on the number of temporary files) are in linear time, results on small files with less memory transfer proportional to large files and more memory. This is confirmed by the experiment shown in Figure 4.

However, changing only one of these parameters has different effects. Using SAMtools for example installed locally on a laptop for sorting a larger BAM file can produce many temporary files if the laptop's memory is limited. If, e.g., 8 GB are available for SAMtools `sort`, it cannot process files bigger than 50 GB without merging temporary files. The file size up to which SAMtools `sort` does not merge temporary files decreases further, if the ratio of the input file to the memory limitation `max_mem` grows further.

An important point that should not be exceeded, is reaching 1120 (Footnote 1) temporary files. At this point, SAMtools `sort` merges all "big files" (temporary files resulting of a merge of other temporary files) into one single file. This means SAMtools `sort` writes every single BAM record it processed before to disk once more. This occurs approximately at sorting a 1700 GiB file using 8 GiB of memory.

In conclusion, writing larger numbers of temporary files, specifically more than 32, leads to merging of temporary files, which is time-consuming. The amount of temporary files SAMtools `sort` utilizes is mainly affected by the ratio of the size of the input file to the amount of available memory.

5.3 Recommendation

Since the most time-consuming part of sorting aligned DNA-Read files utilizing SAMtools `sort` is compressing and writing, we aim to minimize the frequency of SAMtools `sort` writing a single BAM record. Consequently, our primary strategy

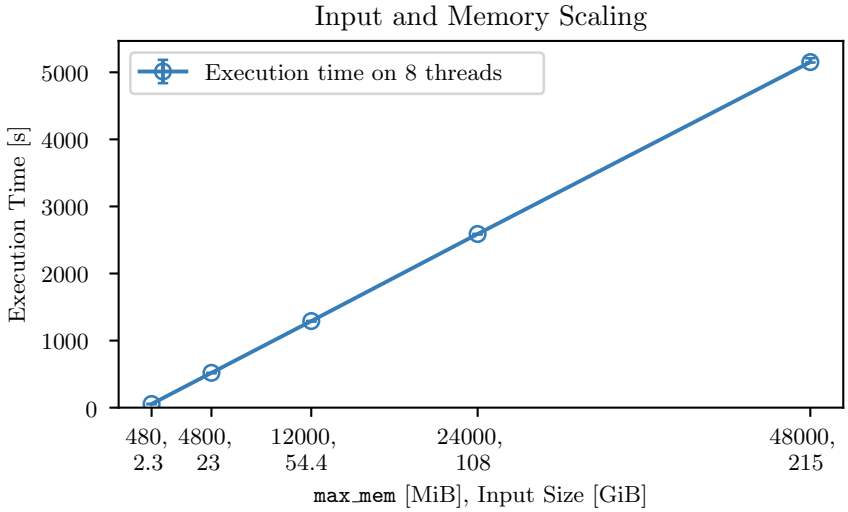

Fig. 4. Execution time of SAMtools `sort` on different input sizes. Keeping the ratio from input size to the memory limitation `max_mem` constant, the execution time grows linear to both parameters. Data points: Median, Error bars: fastest and slowest of 3 runs.

focuses on reducing the number of merge steps performed by SAMtools `sort` during the overall sorting process.

To accomplish this without changing any source code, the user can change the "`-m`" parameter for per thread memory limitation setting. The more memory the user gives to the process, the less likely SAMtools `sort` needs to merge temporary files. Therefore, we recommend the user to allocate the highest possible memory allowance to SAMtools `sort` within system constraints. To avoid merging of temporary files, the user must set the memory limitation per thread ("`-m`") to a value larger than $\frac{\text{InputSize} \cdot 6}{31 \cdot \#\text{Threads}}$ for any typical input file compressed with zlib on compression level 6 (the default compression level).[3] However, as the memory limitation the user sets via "`-m`" is an upper bound only for storing BAM records in memory, SAMtools will most likely exceed it. Thus, the user should not set "`-m`" to the whole available amount of memory divided by the number of used threads, but keep some memory for SAMtools internal resource allocation.

However, on devices with limited physical memory, such as laptops, or when working with exceptionally large files (ranging into terabytes of size), allocating enough memory to entirely avoid merging in SAMtools `sort` might not be feasible. Because of this, we recommend enlarging the limit for the number of open

[3] 6 for the blowup of the input file in memory (Section 5.2), 31 for the temporary files written before merging minus 1 to avoid edge cases, #Threads, as the `-m` parameter specifies the memory limitation per thread.

temporary files in SAMtools `sort`. At the moment, SAMtools `sort` has a predefined limit of 64 for temporary files stored on disk concurrently. Yet, modern computers are able to keep much more files open without noticeable performance losses.

On Unix systems, there exist two kinds of limits for the number of open files. The operating system differentiates between *soft limits* and *hard limits*. A soft limit is a limit set by the user. If a process attempts to open more files than the soft limit, the operating system kills it. On most modern systems, the soft limit is set to 1024 by default.

The hard limit is the limit up to which the user can increase the soft limit. Its size differs from system to system, but is typically much larger than the hard limit (e.g., 262144 on our testing machine). The hard limit can not be increased by users of the operating system.

To maximize the limitation for the number of open temporary files in SAMtools `sort`, SAMtools `sort` must obtain the soft limit for open files and calculate the number of files it opens in addition to temporary files. On a Unix-like operating system, a program can obtain its soft limit using the `getrlimit` [1] system call. SAMtools `sort` only opens temporary files to merge, an output file (or standard output), possibly an index file, and has standard input and standard error open. We use this knowledge to propose a dynamic merging strategy. This strategy recognizes how many files SAMtools `sort` can open concurrently without exceeding the soft limit for open files. Subtracting the maximum number of files (including standard output, standard error, and standard input) SAMtools `sort` opens concurrently to temporary files, the strategy sets an adapted limit for temporary files in SAMtools `sort`. This also enables users to reduce the amount of merges further by increasing the soft limit for open files for SAMtools `sort`. For compatibility reasons, if the system call fails, SAMtools `sort` can still enforce a limit of 64 for temporary files.

Notice, that the file size of the input, at which SAMtools `sort` stores as many temporary files as its limit for temporary files allows on disk concurrently, grows quadratic to this limit. At the same time, the file size of the input up to which SAMtools `sort` does not perform a single merge of temporary files grows only half as fast as the maximum number of temporary files.

5.4 Evaluation

Increasing the number of allowed temporary files to 1019 (= 1024 − 5 = the default soft limit minus potential other files opened concurrently with temporary files), leads to a 15.5-fold increase in the potential file size of the input file before triggering a merge. With the limit of 1019 for temporary files, SAMtools `sort` performs the first merge of temporary files at the 510th file instead of at the 33rd. Having a limited amount of 8 GiB of memory, the change to 1019 allowed temporary files raises the file size of the input file, after which the first merge of temporary files occurs, from around 50 GiB to around 775 GiB.

Comparing the increased limit for temporary files to the current limit of 64 temporary files, we report a speedup of up to 1.25 at sorting a 2.3 GiB BAM

file with 40 MiB of memory (Figure 5). At the memory limitations between 400 MiB and 25600 MiB, we observed identical execution times for both limits for temporary files on sorting a 2.3 GiB input file. This is due to SAMtools `sort` utilizing between 0 and 32 temporary files. Therefore, it does not merge temporary files with both limitations for temporary files. Sorting the 2.3 GiB BAM file with memory limitations between 50 MiB and 200 MiB (equivalent to sorting between 1.6 TiB and 400 GiB with a memory limitation of 32 GiB), SAMtools `sort` performs between 1 and 7 merges of temporary files with the limit of 64 for temporary files. In contrast, with the limit for temporary files increased to 1019, sort does not merge temporary files in between these memory limitations.

However, on the lowest tested memory limitation setting of 25 MiB for sorting the 2.3 GiB BAM file (equivalent to sorting a 2.9 TiB file with a memory limitation of 32 GiB), the increased limitation for temporary files of 1019 has no speedup compared to the limitation for temporary files of 64. On this memory limitation setting, SAMtools `sort` utilizes a total of 527 temporary files. With the limitation for temporary files set to 1019, SAMtools `sort` merges temporary files the first time at writing the 510th temporary file. Therefore, SAMtools `sort` merges 509 temporary files into the 510th file on the limitation of 1019 for temporary files.

On the limitation of 64 for temporary files, SAMtools `sort` performs 15 merges while writing 527 temporary files. However, as each of these merges only merges 32 temporary files into a single file, only the content of 480 temporary files is written and compressed two times, compared to the content of 509 temporary files at the limitation of 1019 for temporary files. The reason for the difference between the two limits for temporary files is, that SAMtools `sort` includes the in-memory vectors of sorted BAM records, which are the results of parallel sorting, into each merge. Therefore, writing the 510th file, SAMtools `sort` performs less compression and writing operations with the limit of 64 for temporary files, than with the limit of 1019, as it compresses and writes content in the size of 15 temporary files only once. In addition, SAMtools `sort` merges temporary files at the 495th and again at the 528th temporary file if its limit for temporary files is 64. Thus, with this limit for temporary files, at the 510th temporary file, SAMtools `sort` has written 15 temporary files, which are "small files" and not merged into a "big file" until SAMtools `sort` writes the 528th temporary file. The total amount of times, SAMtools `sort` compresses and writes a block of BAM records in size of the available `max_mem` into a temporary file is illustrated in Figure 6 for both limitations for temporary files.

In summary, increasing the limit for temporary files results in a noticeable speedup if it prevents merging of temporary files. However, with an increased limit for temporary files, each merge of temporary files leads to a higher increase in runtime than a merge with the limitation of 64 for temporary files. If the limit is calculated from the soft limit defined by the operating system, it is maximized and the user can increase it if necessary, preventing merging of temporary files even for sorting terabyte-sized aligned DNA-Read files.

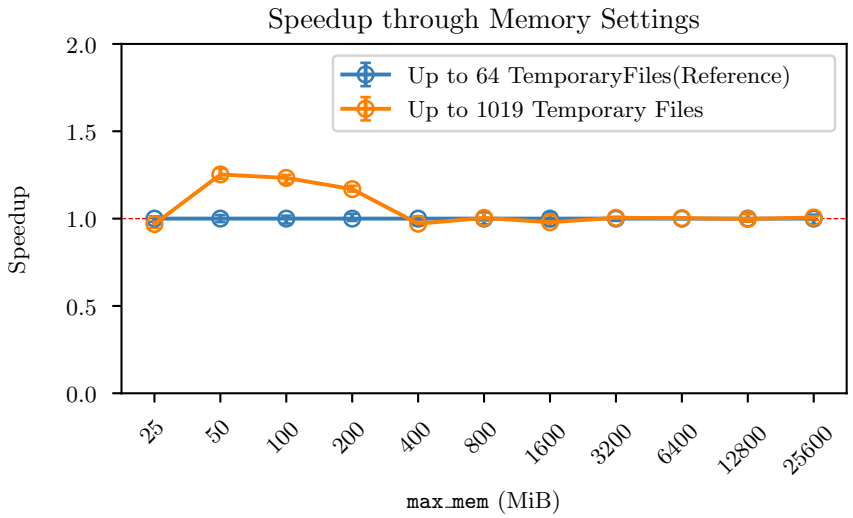

Fig. 5. Speedup after setting the limit for temporary files to 1019. On the memory limitations of more than 200 MiB `max_mem`, for both limitations for the number of temporary files, no merges are performed. At 200 MiB, 100 MiB, and 50 MiB SAMtools `sort` performs 1, 3, and 7 merges with the limit of 64 for temporary files. With the limit of 1019 for temporary files, SAMtools `sort` merges temporary files only at 25 MiB `max_mem`. Data points: Median, Error bars: fastest and slowest of 3 runs.

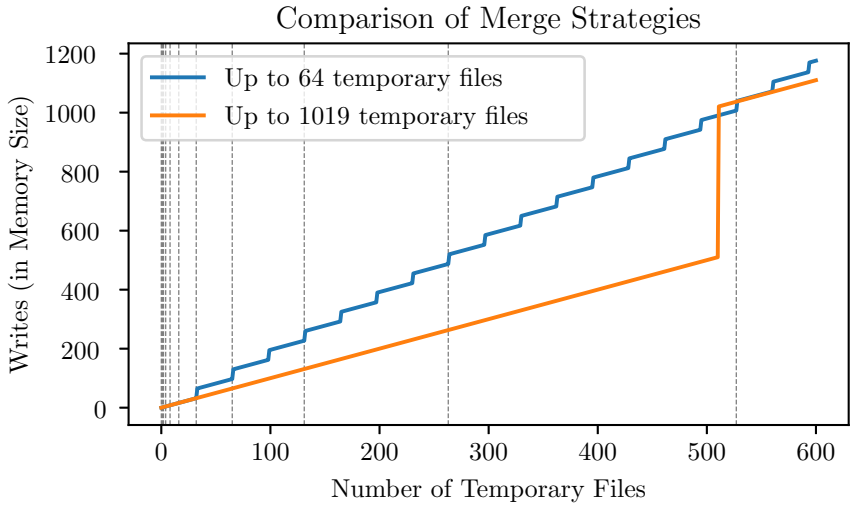

Fig. 6. Write and compression operations for blocks of BAM records in size of SAMtools `sort`'s memory limitation `max_mem` on using a limit of 64 and 1019 for temporary files. If 32 files are merged together with one in-memory block, it counts as 33 Operations. Vertical gray lines mark numbers of temporary files produced in the examples in Figures 3 and 5. (E.g., 527 at 25 MiB `max_mem`.)

5.5 Future Work

Despite setting the limit for the number of temporary files to 1019, the program initiates merging as early as at writing the 510th temporary file, effectively utilizing only half of the available capacity. Due to merging small temporary files at half of the limit for temporary files, SAMtools `sort` only ever utilizes the full capacity of 1019 temporary files, when the total number of written temporary files exceeds 260,000. While users can potentially eliminate merging by increasing the soft limit, this approach necessitates an unlikely amount of user awareness of this optimization option. To minimize the number of merge steps and reduce the runtime, SAMtools `sort` should ideally utilize the temporary file limit to a greater extent before initiating the merging process. This can be achieved by writing small temporary files not only up to half of the limit for temporary files, but to the limit minus the current amount of big files before merging them. For the first merges, this change would lead to reducing the number of merges by half.

6 Compression

6.1 Overview

Compression of aligned DNA-Read files reduces storage space and thus storing costs as well as transfer-times of the files over network. Per specification, compression is a part of writing BAM files, which are files containing information on the alignment of DNA sequences to a set of reference sequences. The compression method used in BAM files is BGZF (Section 2.6). BGZF is an extension of the popular GZIP compression format. BGZF compresses files in blocks of less than 64 KB (compressed and uncompressed), enabling fast random access to each block. Internally, BGZF compresses data in the DEFLATE format, which GZIP is a wrapper for. The library BGZF utilizes for compression is zlib. While compression is beneficial for saving storage space and thus reducing storage costs, it introduces additional processing steps compared to uncompressed output. Therefore, applying compression to the sorted BAM files SAMtools `sort` outputs leads to increased execution times for SAMtools `sort`.

In this chapter, we analyze the influences the default compression using zlib has on the sorting process. Then, we discuss possible changes in the compression levels. Afterward, we investigate seven different compression libraries offering alternative implementations for zlib. Based on the findings, we recommend settings to fasten up SAMtools `sort` and evaluate them.

6.2 Analysis

To measure the impacts of compression and decompression on the computation time that SAMtools `sort` requires, we examine the proportional time consumption of SAMtools `sort`'s processing steps. Running on 16 threads, with a total of 32 GiB of memory, we report SAMtools `sort` to require 71 minutes and 57 seconds to sort a 215 GiB BAM file. However, SAMtools `sort` uses only 2 minutes and 35 seconds, (3.6 % of the total time), for sorting[4] (merging not included), while it uses the rest of the time for reading, writing, compression, decompression, and merging.

Profiling SAMtools `sort` locally on a laptop reveals that a substantial amount of the remaining time is dedicated to compression: Compression and decompression of the temporary and output BAM files, which contain the alignment information to aligned DNA-Reads, together account for 97 % of the CPU time when performing SAMtools `sort` on a laptop with the default compression level. On this compression level, the *deflate* method that is used for the compression and a part of zlib requires approximately 81 % of the CPU time, the *inflate* method that is used for decompression approximately 11 %, and the calculation

[4] Sorting time calculated by adding up the time spans from dividing the blocks of BAM records in memory into one block per thread and the time when all the sorting threads are finished. The times are retrieved by changing SAMtools `sort`'s source code to output the current Unix-Timestamp to standard error before and after sorting.

of the crc32 checksum, which is part of the compression as well as of the decompression, approximately 5 %. Setting the compression level to 0 reduces the relative amount of CPU time required for compression and decompression by 1 % to 96 %. On this compression level, the deflate method requires approximately 68 % of SAMtools `sort`'s computation time, inflate 19 %, and the checksum calculation 9 %.

SAMtools utilizes HTSlib for file operations and compression. HTSlib serves as an API for various high-throughput sequencing data formats, such as BAM files, and provides functionalities for reading and writing them. HTSlib utilizes the zlib library for compression and decompression of compressed file formats (e.g., BAM). A file compressed in the BGZF format, the compression method used in the BAM format, consists of a series of blocks of independently compressed data (each smaller than 64 KB both compressed and uncompressed). To make optimal use of the available processors, HTSlib compresses these blocks in parallel: SAMtools sequentially passes single BAM records containing alignment information on DNA-Reads to HTSlib. HTSlib buffers these BAM records until the next BAM record does not fit into a 64 KB block together with the other buffered BAM records. Then HTSlib creates a compression and writing job in a thread pool. The thread pool contains queues for each job kind, in case of SAMtools `sort` compression and decompression jobs, and a counter for the amount of queued jobs. The thread pool never wakes up more threads than pending jobs. Threads work through jobs of the same queue until the queue is empty. This leads to threads likely staying at the same job and to some threads running in idle, if less jobs then threads are pending. Since all uncompressed blocks are approximately equal in size, active threads receive a balanced workload distribution during the compression or decompression process.

6.3 Compression Levels

SAMtools utilizes HTSlib for compression tasks. HTSlib utilizes zlib for compression to the DEFLATE format, which the BGZF compression format applied to BAM files is a wrapper for. In zlib, compression levels offer a configurable balance between computational intensity and resulting file size. The DEFLATE format, internally used in all of zlib's output formats, is a combination of LZ77 and Huffman codes. LZ77 is a dictionary-based compression method, which finds matching sequences in a sliding window and replaces following sequences with a reference to the first appearance of a match. The reduction of the file size depends mainly on the amount of matches found and the length of the matching sequences. As shorter matches occur more frequently, finding them is less computational intensive.

The match finding algorithm in zlib hashes 3 bytes for every position in the string it compresses and inserts a pointer to the position into a list in a hash map. If the current position is not a part of a previous match, zlib looks within the hash table for substrings starting with the same bytes. Then it iteratively compares this substrings to the string at the current position to find the longest match. The parameters after which match-length zlib stops searching for longer matches

depend on the compression level. E.g, for compression level 1, zlib compares at most 4 substrings per position, reduces this number to 2 if it finds a match of length 4-7, and stops searching for longer matches if it finds a match of length 8 or longer. For compression level 9, zlib compares at most 4096 substrings per position, reduces this number to 2048 if it finds a match of length 32-255, and stops searching for longer matches if it finds a match of length 256 or longer. For compression levels 4-9, zlib also computes the matches for the next position before eventually accepting a match. This leads to approximately 6 times higher throughput and an approximately 21 % larger resulting file size on using compression level 1 instead of compression level 9 for compressing BGZF files in HTSlib utilizing zlib. We show differences between the compression levels' output sizes in Figure 7 and differences between the compression levels' throughput in Figure 8.

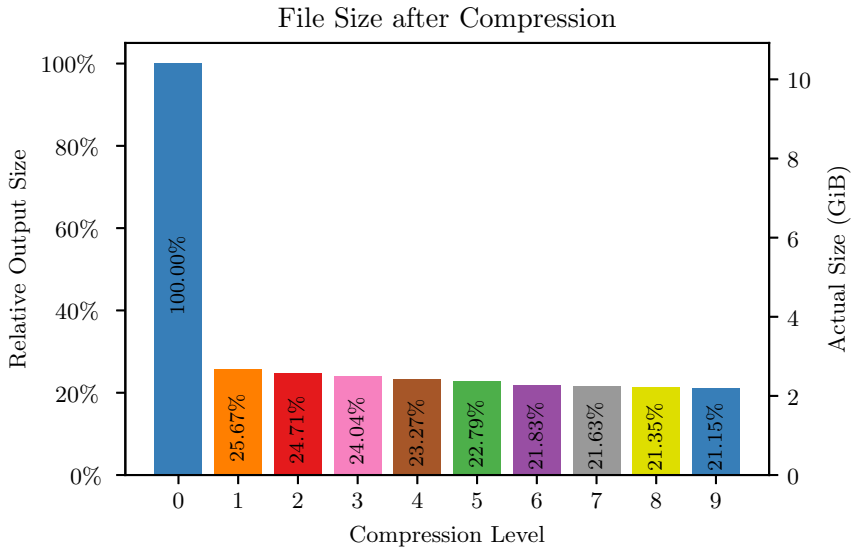

Fig. 7. Size of BGZF compressed files on compression levels 0-9, exemplified using a 10.4 GiB unsorted BAM file. Although no compression yields a file four times as large, the distinctions between compression levels are less substantial.

To measure the compression speed, we measure the speed of HTSlib's `bgzip`, a tool to compress arbitrary files using BGZF. SAMtools `sort` uses the same methods as `bgzip` of HTSlib, the library utilized by SAMtools for compression and file operations. This still holds for compression level 0. For this compression level, `bgzip` as well as SAMtools `sort` do not compress, but directly write the output. Therefore, the compression speed of `bgzip` is relevant for sorting because

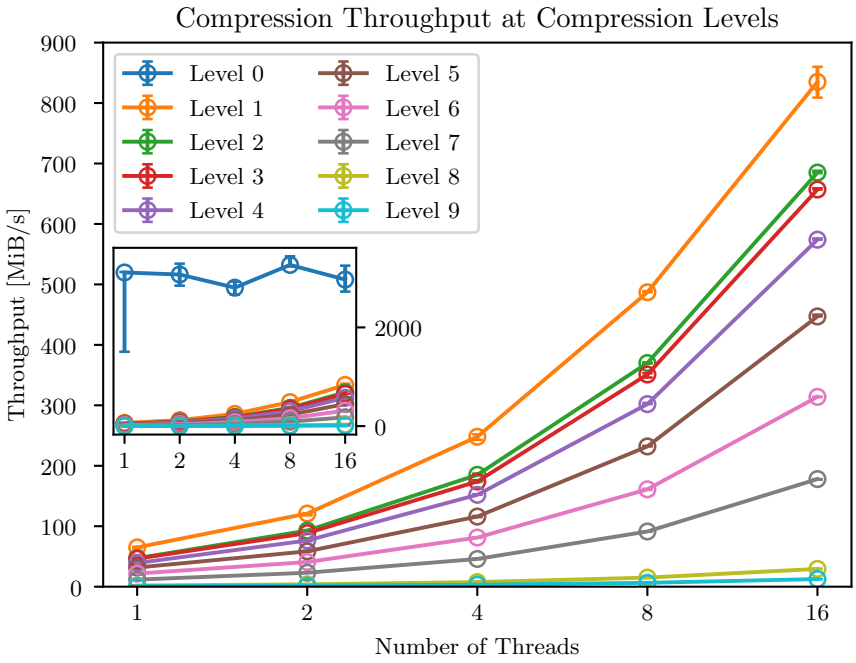

Fig. 8. Output rate of HTSlib's `bgzip` which uses BGZF to compress A 10.4 GiB uncompressed BAM file. For reference, we show the throughput of `bgzip` for compression level 0 in the smaller inset plot. Data points: Median, Error bars: fastest and slowest of 3 runs.

it sets a lower boundary on writing the output of SAMtools `sort`, considering that compression is a part of creating BAM files.

Per default, SAMtools `sort` uses two different zlib compression levels for the BGZF compression of temporary and output BAM files. For writing output files, it uses the default compression level of the zlib implementation (compression level 6 for zlib) as long as the user has not set a specific compression level.[5] For temporary files, SAMtools uses compression level 1. In current SAMtools versions, this can not be changed without changing the source code.

6.4 Alternative zlib Implementations

Being build into the Linux kernel, zlib is seen as the de facto standard of file compressing using the DEFLATE algorithm. However, other libraries have been created that surpass zlib in both compression throughput and smaller file size of the resulting files.

[5] For instructions to set compression levels in SAMtools, refer to Appendix A.2.

For Example, *libdeflate* [17] offers faster compression than zlib (e.g., 2.3 times faster on single threaded compression with HTSlib's `bgzip` compression tool on compression level 6) while achieving a better compression ratio at the same time (0.5 % smaller file on compression level 6 (Figure 9)).

Remember, SAMtools utilizes zlib and libdeflate to compress data into the DEFLATE format wrapped by BGZF for BAM files, which contain alignment information on DNA-Reads. The DEFLATE format is a combination of Huffman Codes and LZ77, which finds identical substrings in a string and replaces matches of previous substrings with references to the position of the previous substring.

To speed up the compression process, libdeflate employs various improvements: It uses an optimized algorithm for generating Huffman Codes and chooses dynamically between the two types of Huffman Codes that are supported by the DEFLATE format, *fixed* and *dynamic* Huffman Codes. Moreover, depending on the compression level, libdeflate uses different kinds of hash maps for the match finding: Hash tables with the matches stored directly in the table for low compression levels, hash maps with lists of matches in external arrays for intermediate compression levels, and hash maps with binary heaps containing lexicographically sorted matches for the higher compression levels. Libdeflate also incorporates five different algorithms for DEFLATE compression and selects them depending on the compression level. These algorithms include greedy algorithms, algorithms which calculate matches for one or two positions ahead and accept matches only, if there is not a longer match at the following positions, and an algorithm utilizing a graph based approach to find near optimal matches. Similar to zlib, the length at which libdeflate accepts a match and stops comparing a substring to possible matches in the hash map, depends on the compression level, as well as the maximal amount of string comparisons for each position. Furthermore, libdeflate makes use of various precomputed values, e.g., for hashing, and uses optimized instructions on compatible machines.

Libdeflate also offers a decompressor for DEFLATE compressed formats, incorporating various improvements such as using word access instead of byte access in input reading and match copying, which are parts of decoding the DEFLATE format. Furthermore, it uses a speed-up Huffman decoding process and utilizes BMI2 instructions on x86_64 machines if they support them.

Moreover, libdeflate contains a crc32 checksum implementation, which it uses for both decompression and compression of files in the GZIP format. This implementation leverages properties of the XOR operation to reduce computation steps and utilizes multi-bit units. The implementation relies on an iterative approach to calculate the checksum. It processes the data four bytes at a time, leveraging pre-computed values for each byte.

Support for libdeflate is already built into SAMtools. Moreover, the developers recommend using libdeflate instead of zlib. If HTSlib's configure script finds libdeflate libraries, HTSlib uses them automatically instead of zlib.[6]. Since libdeflate offers 12 compression levels compared to zlib's 9 compression levels, SAM-

[6] To decide manually between zlib and libdeflate see Appendix A.3

tools implements a mapping scheme (Appendix A.4) to translate user-specified compression levels when utilizing libdeflate for compression tasks.

In addition, the user can choose to use other zlib implementations by using `LD_PRELOAD` [42]. The `LD_PRELOAD` environment variable instructs the dynamic linker to prioritize specific shared libraries during program execution. Shared libraries are reusable code modules that can be loaded by multiple programs. If two different definitions for methods or variables exist, e.g., one definition in the shared library a program uses by default and one in a library in `LD_PRELOAD`, the linker prefers the one from a shared library in `LD_PRELOAD` over the one from a shared library that is not in `LD_PRELOAD`.

For example, HTSlib uses the `deflate` method of `libz.so`. However, the user can compile e.g., *zlib-ng* [13], which is API compatible to zlib, to a shared object. Then he can specify the path to the compiled shared object in `LD_PRELOAD`. As a result, every time HTSlib calls zlib methods, it uses the implementations in zlib-ng.

However, this approach is only possible, if the replacement implementation supports the zlib API. `LD_PRELOAD` also allows for partial compatibility. If the dynamic linker does not find a method or variable within the preloaded libraries, it uses the implementation from the default shared libraries.

6.5 7BGZF: Testing Non-API-Compatible Compression Libraries

7BGZF [53] is a tool for testing different compression libraries for BGZF compression in SAMtools. It works by overwriting the method HTSlib uses anytime it outputs BGZF compressed files, `bgzf_compress`, via `LD_PRELOAD`. Users can choose the compression library and the compression level of 7BGZF via an environment variable.[7]

This approach simplifies the evaluation of various compression libraries. It eliminates the need for individual library installations and accommodates libraries with non-zlib-compatible API implementations.

However, the evaluation of compression libraries utilizing 7BGZF does not transform directly to implementing their usage into SAMtools. SAMtools has to link to HTSlib as a shared library rather to linking to the static library, as the method 7BGZF overwrites is a method of HTSlib.[8] Moreover, 7BGZF disregards the compression level passed to the `bgzf_compress` method. Instead, 7BGZF receives the compression level to be used via an environment variable. Therefore, it applies the same compression level on every written BGZF compressed file, in context of SAMtools `sort` the sorted output files and temporary files. This means temporary files, which contain memory-sized chunks of sorted BAM records and are merged into the final output file (Section 5), have the same compression level as output files. Without using 7BGZF, SAMtools `sort` compresses temporary files with compression level 1, regardless of the specified output compression level. With using 7BGZF, employing a compression level

[7] For more information on using 7BGZF, refer to Appendix A.5.
[8] To change SAMtools to link to HTSlib as a shared library, refer to Appendix A.6.

greater than one for the output file will result in increased compression time, compared to an implementation of the compression library into SAMtools, which would make use of different compression levels for output and temporary files. Therefore, using files that require SAMtools `sort` to produce temporary files distort comparisons with the default zlib or libdeflate compression.

To get insights of the runtime of SAMtools `sort` utilizing the different compression libraries of 7BGZF on different levels, we use 7BGZF on sorting a BAM file small enough not to produce any temporary files. For libdeflate and zlib, SAMtools `sort` achieved similar runtimes on average when using 7BGZF compared to plain HTSlib, with variances ranging from 3 to 5 percent.

6.6 Compression Libraries in 7BGZF

We tested the following seven compression libraries in 7BGZF: *zlib-ng* [13] is a merge of optimizations of a zlib version by Intel [12] and a zlib fork by Cloudflare [9]. Both of these zlib implementations can be found in old comparisons of zlib implementations for using them in SAMtools. Zlib-ng aims to provide a version of Zlib that is more receptive to code changes. Therefore, it makes use of newer compiler features, adapts faster to code improvements than zlib, and removes many of zlib's workarounds for older systems and compilers.

The *slz* [49] compression library supports only a single compression level. It reduces CPU and RAM usage for web servers by limiting the size of the hash map containing previous matches, using precomputed fixed Huffman Codes only, and precomputed coding for the distances which are parts of the references to previous matches. These changes also come with lower runtimes but larger resulting files.

igzip [52], which is a part of the Intelligent Storage Acceleration Library (ISA-L) [11] by Intel, prioritizes runtime over result size as well, but without slz's goal of reducing the CPU and RAM usage for the compressor.

In contrast, Google's *zopfli* [10] is an algorithm designed to enable the best possible deflate-compatible compression by iteratively modeling the entropy of the data and finding a minimal path in a graph of all possible DEFLATE compressions. However, previous experiments showed its implementation on default settings being slower than zlib by a factor of 80.

miniz [5] is another zlib implementation written from scratch. It presents its main advantage as being contained in a single source file.

7BGZF also supports zlib and libdeflate (Section 6.4). A comparison of the supported compression levels and features of the tested compression libraries in 7BGZF is shown in Table 1.

Implementation	zlib	libdeflate	miniz	igzip	slz	zlib-ng	zopfli
Levels	1-9	1-12	1-9	1-3	1	1-9	-[9]
Decompression	yes	yes	yes	yes	no	yes	no
Drop-In[10]	-	no	yes	no	no	yes[11]	no

Table 1. Comparison of features of the compression libraries tested in 7BGZF.

6.7 7BGZF Results

In the following, we present the results of using the compression libraries supported by 7BGZF for compression of aligned DNA-Read files in the BAM format using SAMtools `sort`. We tested each library on their fastest and on their default compression level. To work around 7BGZF's limitation regarding separate compression levels for output and temporary files, we sort a file small enough to avoid producing temporary files in SAMtools `sort`. The results can be used to see which libraries have potential for replacing zlib in HTSlib. We primarily compare the compression libraries to zlib and libdeflate, since HTSlib already supports these.

The best performance could be reached by igzip. On both tested compression levels, it compressed faster than all other compression libraries on their fastest compression level. Using one or two threads, igzip achieved a speedup of up to 5 compared to the default zlib compression. However, igzip's resulting file size on both tested compression levels turned out larger than libdeflate on compression level 1 (30 % respective 23 % of the original size with igzip on compression level 1 respective compression level 3 against 22 % of the original size with libdeflate on compression level one, see Figure 9).

While slz achieves a speedup of 4.5 on one and two threads and is therefore nearly as fast as igzip on compression level 3, it compresses the tested files to 30 % of their uncompressed size, which is more comparable to igzip on compression level 1.

Zlib-ng's runtime and resulting file size are comparable with libdeflate. Zlib-ng on compression level 1 achieves a speedup of 4 on one or two threads, surpassing libdeflate on the same level, but also producing with 28.9 % of the original file size 33 % larger files. However, zlib-ng on compression level 6 produces a 3 % smaller file than libdeflate on the same level, but achieves a speedup of less than 2, making it noticeably slower than libdeflate on compression level 6.

[9] zopfli does not use compression levels but can specify iterations of the algorithm. Here, the level in experiments is always used as iterations.

[10] Drop-In does not infer, that the API provides every symbol of zlib, but that the most important symbols are implemented.

[11] The user has to enable the API compatibility in a configuration step before compiling.

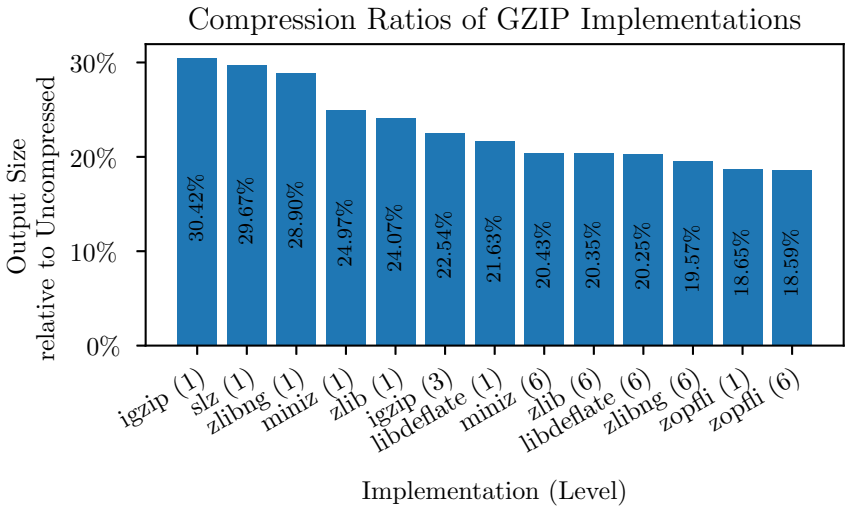

Fig. 9. Compression ratio of different zlib implementations. We report the file sizes relative to the uncompressed file.
Igzip on compression level 1 as well as slz and zlib-ng on compression level 1 produce files 50 % larger than zlib on its default compression level. Miniz on compression level 1 and zlib on compression level 1 produce files which are 20 % larger than the reference (zlib on compression level 6). All other compression libraries and compression levels resulted in differences of up to 10 % compared to zlib on compression level 6, which are 2 % of the uncompressed file.

Libdeflate with compression level 6 has a speedup of 2.3 compared to zlib on compression level 6 (default) for up to 4 used threads, while producing a 0.5 % smaller file than the default zlib compression. This speedup is even larger than the speedup of 2.1 resulting from using zlib on compression level 1. On compression level 1, libdeflate achieved a speedup between 3.5 and 4 for up to 4 threads, while producing a 6.5 % larger file than the zlib compression on compression level 6.

Reducing the compression level to 1, zlib reaches a speedup of approximately 2 for up to 8 threads, while producing a 20 % larger file than on the default compression level of 6.

Previous experiments showed that zopfli produced around 10 % smaller files than the default zlib compression, but took 70 to 90 times longer. Miniz provides for each compression level a marginally worse compression ratio than zlib while having longer processing times, as previous experiments showed. In Figure 10, zopfli and miniz are excluded for clarity.

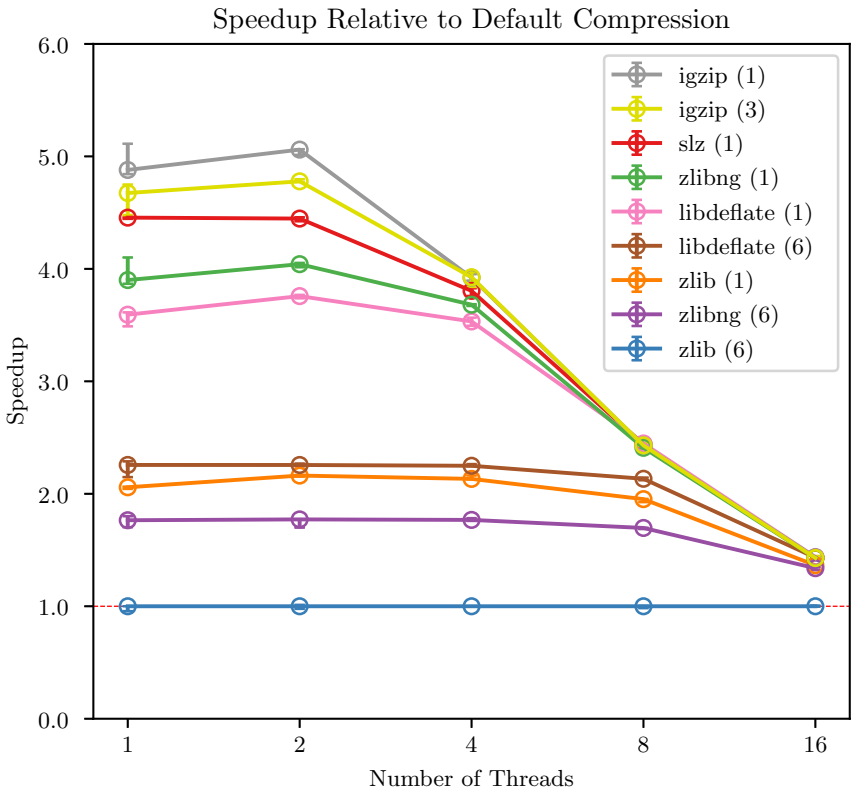

Fig. 10. Speedup of SAMtools `sort` using compression libraries in 7BGZF relative to SAMtools `sort` using zlib compression level 6 (the default) on the same number of threads. Sorting the 2.3 GiB BAM file with 48 GiB of memory did not utilize a temporary file. Numbers in parentheses indicate the compression level. One thread is single threaded computation; for a higher thread count, SAMtools `sort` uses the threads in addition to the main thread. Data points: Median, Error bars: fastest and slowest of 3 runs.
The plot shows that using faster compression libraries and lower compression levels improves SAMtools `sort`'s runtime for a smaller number of threads. With more threads, the improvement decreases.

Increasing the number of threads gradually reduces the speedup achieved by using faster compression libraries. The speedup of 5 achieved by igzip on one or two threads diminishes to 4 on 4 threads, to 2.5 on 8 threads, and finally to 1.5 on 16 threads, the highest tested number of threads. For all other compression libraries and compression levels, igzip with compression level 1 serves as an upper limit. The speedup of all other libraries stays close to their single-thread speedup until the number of threads, where the speedup of igzip on compression level 1

drops below their single-thread speedup. Then, their speedup approaches the speedup of igzip on compression level 1. Therefore, on 16 threads, all alternative zlib implementations on their tested compression levels achieve a speedup of approximately 1.5.

The speedup of the different compression libraries and compression levels compared to their single-threaded execution time increases slower for the settings achieving a higher single-thread speedup, e.g., igzip on compression level 1. The speedup of zlib on compression level 6 increases with every higher number of threads, reaching a speedup of 12 on 16 threads. In contrast, faster compression libraries such as igzip show a less substantial increase in speedup: All tested compression libraries and corresponding compression levels that achieved a single-thread speedup (compared to zlib on compression level 6) which is as high or higher than libdeflate on compression level 1, reached their highest speedup against their single-thread performance on 8 cores with a speedup of less than 5. When raising the number of threads from 8 to 16, their speedup decreases slightly, as shown in Figure 11.

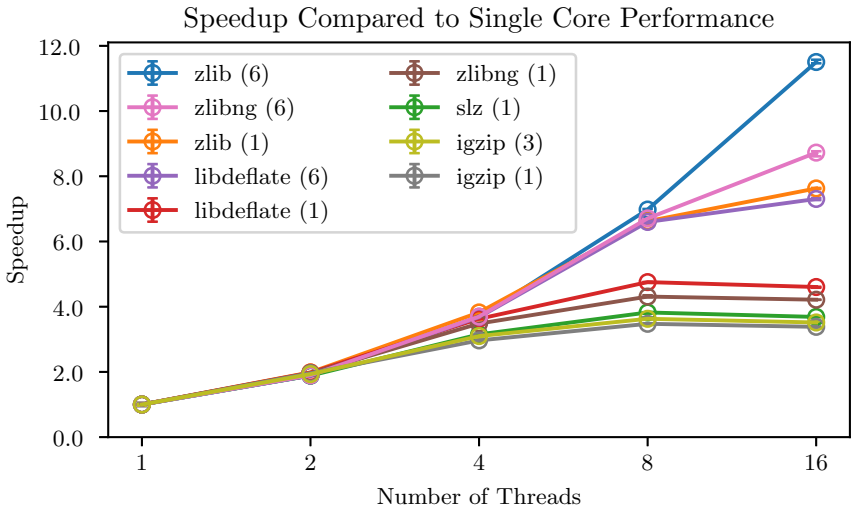

Fig. 11. Speedup against their respective single-threaded performance of 7BGZF compression libraries and compression levels. Sorting the 2.3 GiB BAM file with 48 GiB of memory did not utilize a temporary file. Numbers in parentheses indicate the compression level. One thread is single threaded computation; for a higher thread count, SAMtools `sort` uses the threads in addition to the main thread. Data points: Median, Error bars: fastest and slowest of 3 runs.
The plot illustrates the extent to which the compression libraries and compression levels benefit from an increasing number of threads.

This suggests that beyond a certain number of threads, compression, which is fully parallelized, is no longer the limiting factor. This is supported by the observation that with increasing the amount of used threads, the time SAMtools `sort` spends on decompression, sorting and merging together with compression relative to the total time converges to a similar ratio for all compression libraries and tested compression levels.

In the following, decompression stands for the time from starting SAMtools `sort` until starting the parallel sorting, sorting for the timespan from starting to ending the sorting, and compression the timespan from end of sorting to the end of the SAMtools `sort` process. Recall, having multiple threads available, SAMtools `sort` at first reads the BAM records containing alignment information on DNA-Reads while decompressing the input BAM file in parallel. Then, SAMtools `sort` splits the BAM records in memory and sorts them in parallel. After the sorting, SAMtools `sort` merges the lists of sorted BAM records from the threads, writing each record immediately after merging it (buffered and with compression). Therefore, in the following, compression time includes merging.

On 16 threads, all compression libraries on their tested levels use 21 % to 31 % of their computation time for decompression, 6 % to 9 % for sorting and 60 % to 71 % for compression. Here, the faster compression methods use more of their computation time for decompression than slower methods like zlib on compression level 6. However, on a single thread, the faster compression methods use around 50 % of their time for decompression, while zlib on level 6 only uses 9 % of its computation time for decompression. On two threads, the percentage of the computation time the faster compression methods spend on compression decreases further. However, with further increasing the number of threads, faster compression methods reveal a trend of higher relative allocation of processing time towards compression (around 60 % on 16 threads against 35 % on 2 threads) compared to decompression (around 30 % on 16 threads against 56 % on 2 threads). In contrast, for the compression methods with less single-thread speedup (compared to zlib on compression level 6), the relative computation time spend on compression decreases gradually down to around 70 % on 16 threads (against 88 % on 1 thread). This is illustrated in Figure 13. This analysis reveals that the compression stage within SAMtools `sort`, including the merging process, exhibits limitations in scalability compared to decompression as the number of threads increases.

The underlying reasons for this behavior warrant further investigation. It's likely that the merging step, currently included within the compression time measurement, contributes to the compression time being a bottleneck for a larger amount of threads. Also, influences of SAMtools `sort` producing temporary files remain uninvestigated. In a sorting process utilizing temporary files, SAMtools `sort` would decompress temporary files at the same time as it compressed the output file. In the scenario we investigate here, decompression and compression are strictly separated by the sorting process between both processes.

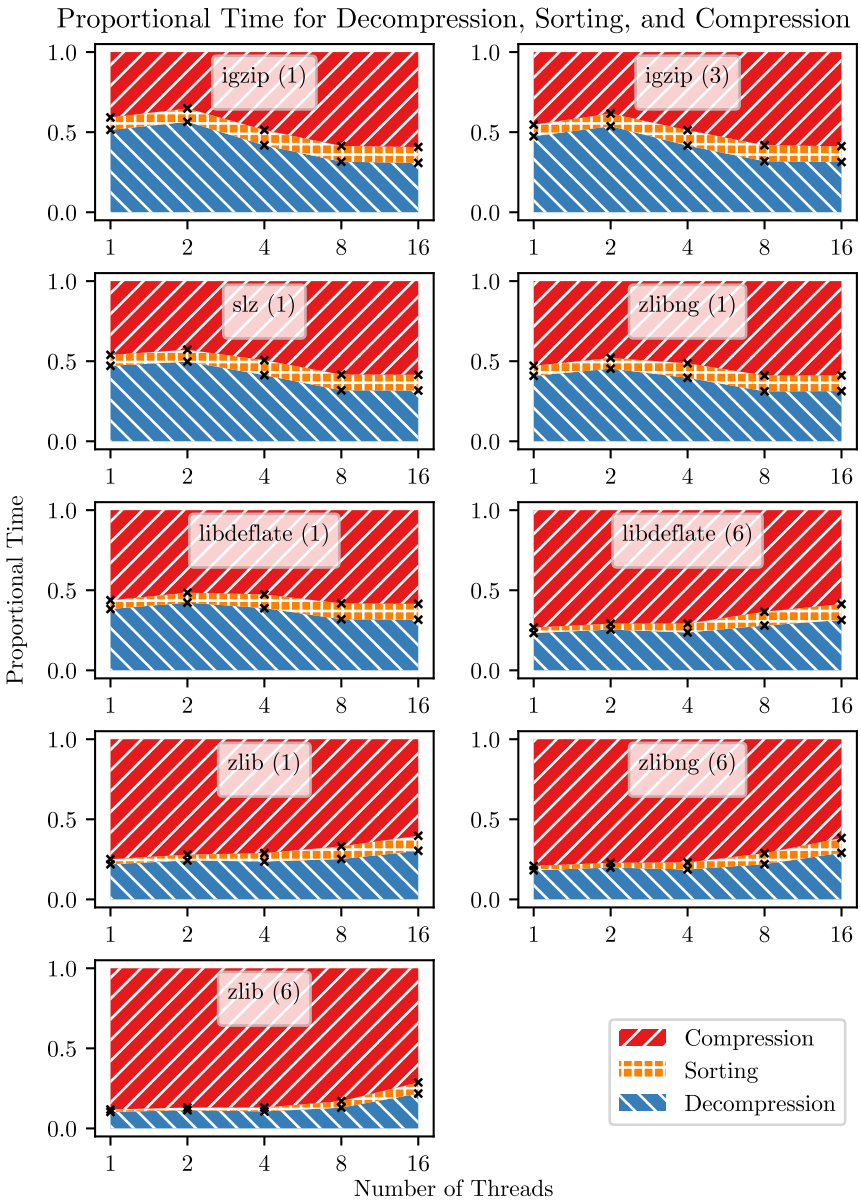

Fig. 13. Procentual amounts of the runtime SAMtools `sort` uses for decompression, sorting, and compression. Decompression: walltime from program startup of SAMtools `sort` until start of the parallel sorting step. Sorting: time spent sorting the BAM records containing aligned DNA-Reads in memory. Compression: walltime from end of sorting to the end of the SAMtools `sort` process. We count the merging of BAM records towards compression, as SAMtools `sort` performs the merge concurrently to compressing the DNA-Reads. Sorting the 2.3 GiB BAM file with 48 GiB of memory did not utilize a temporary file. Numbers in parentheses indicate the compression level. One thread is single threaded computation; for a higher thread count, SAMtools `sort` uses the threads in addition to the main thread. Data points: Median, Error bars: fastest and slowest of 3 runs.
While the compression methods with a higher single-thread speedup compared to the default zlib compression on level 6 use a lower percentage of their computation time on compression on a single thread, the usages converge to a similar ratio on 16 threads.

Besides compression, decompression speed should be taken into account for the full picture. Here, differences between decompressing files compressed by compression libraries in 7BGZF are within a range of 15 % around the average decompression time for compressed files. This holds for both, the zlib and the libdeflate decompressor. Which of the files compressed utilizing compression libraries of 7BGZF decompresses the fastest, depends on the used decompressor. Nevertheless, there is a trend indicating that files with smaller sizes, thus higher compression rates, are decompressed faster, as shown in Figure 12.

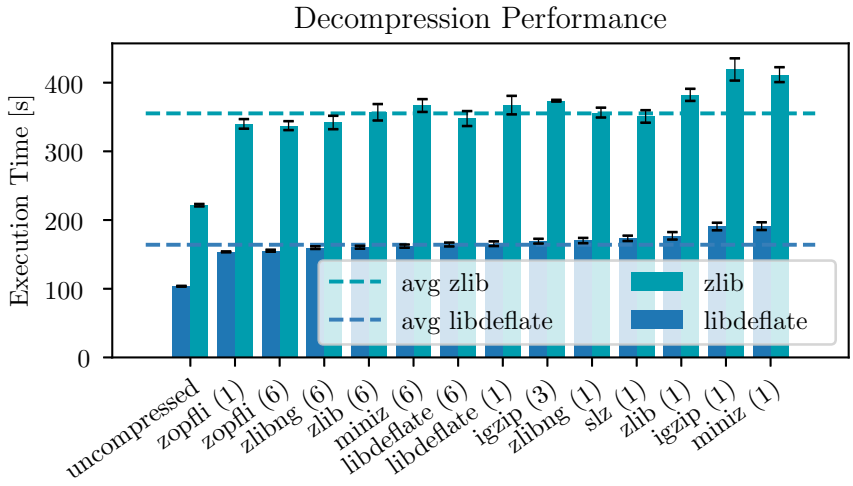

Fig. 12. Decompression speed by SAMtools using zlib compared to using libdeflate for decompression. Measured as execution time of SAMtools `view` with `-u` flag on a single thread, piped to `/dev/null`. The test file is a 104 GB uncompressed BAM file, compressed by SAMtools `sort` using the 7BGZF settings shown on the x-axis. Numbers in parentheses indicate the compression level. One thread is single threaded computation; for a higher thread count, SAMtools `sort` uses the threads in addition to the main thread. Data points: Median, Error bars: fastest and slowest of 3 runs. HTSlib was built with `--with-libdeflate` for decompression utilizing libdeflate respective `--without-libdeflate` for the decompression utilizing zlib.
Decompressing with Libdeflate takes about half the time as with zlib. The trend is for smaller files to decompress faster than larger ones.

Additionally, it is evident that libdeflate is substantially faster than zlib for decompression (on average a speedup of 2). This still holds for the uncompressed file. This is due to a checksum calculation. For each read block, HTSlib calculates a crc32 checksum. Profiling SAMtools `view` (which we used for decompression) revealed that the implementation of the crc32 checksum within libdeflate is ap-

proximately ten times faster than the zlib implementation.

In conclusion, igzip, slz as well as zlib-ng and libdeflate on compression level 1 are very fast and suitable for temporary files and files immediately used for further analysis. On compression level 6, zlib-ng and libdeflate are good default values and provide a trade-off between file size and computation time.

6.8 Recommendation

To increase compression and decompression speed, the amount of threads SAMtools `sort` sort uses should be increased. This can be done using the "`-@`" parameter. Threads are also used for parallel sorting of in memory blocks of read BAM records. This further speeds up the process. The optimal number of threads to use depends on the compression settings. If SAMtools `sort` uses the default zlib implementation with the default compression level 6, using up to 16 threads is reasonable for our setup. For faster compression settings, like zlib-ng or libdeflate on compression level 1, using up to 8 threads is optimal for our setup.

The optimal selection of the compression level depends on the intended usage of the sorted BAM file. For archiving, users can set the compression level parameter of SAMtools `sort` to 9 for maximal compression. If the HTSlib version SAMtools uses was built with libdeflate support, this maps to libdeflate's compression level 12.

However, most of the time, sorting of aligned DNA-Read files is a step in a larger pipeline, and the data is read and processed further soon. To speed this up, we recommend to use compression level 0 ("`-l 0`" which is equal to "`-u`") if the data is not written directly to the disc or transferred over network with limited throughput.

However, if the data is written directly to disk or transferred over a network, the selection hinges on the specific I/O conditions (discussed in the next section). In most cases, compression level 1 ("`-l 1`") serves as a good starting point for these use cases.

For the zlib implementation to use in HTSlib for compression and decompression tasks, we recommend libdeflate, as it is already supported by SAMtools. While other zlib implementations like igzip offer faster compression, libdeflate is already supported by HTSlib. This means the stability is much higher as its usage is tested within SAMtools. Also, libdeflate provides not only a compressor but a decompressor as well. Moreover, even for uncompressed in- and output, the faster crc32 implementation leads to performance improvements. If SAMtools is installed via Bioconda, it is built with libdeflate per default. Users, concerned about output sizes but prioritizing a low execution time, can consider using igzip on compression level 1 via 7BGZF. Still, they should compile HTSlib with libdeflate support.

6.9 Evaluation

Using igzip on compression level 1 via BGZF together with libdeflate for decompression leads to a speedup of up to 5 for single core sorting. This remains valid even if SAMtools `sort` generates a temporary file due to insufficient memory relative to the input file size.

Using the libdeflate integration of HTSlib we achieve a speedup of 2 against zlib on its default level 6 for libdeflate's default compression level (6). Lowering the compression level of libdeflate to 1, we achieve a speedup of 3.2 on 1 to 4 threads (Figure 14).

Zlib on default compression level, as well as libdeflate on default compression level, profit from utilizing up to 16 cores (Figure 15). Libdeflate on compression level 1 and igzip via 7BGZF gain their highest speedup increase from using up to 8 threads. Still, changing from 8 to 16 used threads, they become slightly faster (around 4 %), contrary to the previous results, where for igzip the execution time increased at changing from 8 to 16 used threads (Figure 11).

In conclusion, to reduce computation time of the compression done by SAMtools `sort`, the user can lower the compression level and choose a different zlib implementation. Libdeflate emerged to provide higher compression with lower computation time in both compression and decompression. Other libraries like igzip offer faster compression than libdeflate, but are currently not supported by HTSlib. Nevertheless, with certain restrictions, they can still be used via 7BGZF.

6.10 Future Work

To enhance user awareness, a warning message for the compression library could be implemented within SAMtools. This message would inform users that compiling HTSlib with libdeflate is recommended if HTSlib is currently linked against zlib but not explicitly configured to do so.

Intel's igzip performs even better than libdeflate. Although this comes with the downside of larger files, implementing igzip support in HTSlib would enable even faster compression for temporary files. E.g., a mapping could be used mapping compression level 1 and 2 to compression level 1 and 3 of igzip and the higher levels to libdeflate levels.

For improving 7BGZF, a differentiation between output files and temporary files could be implemented.

Also, the reduced speedup at using faster compression libraries on a larger number of threads could be investigated further. E.g., the merging process could be removed to see if it limits the execution time of SAMtools `sort`.

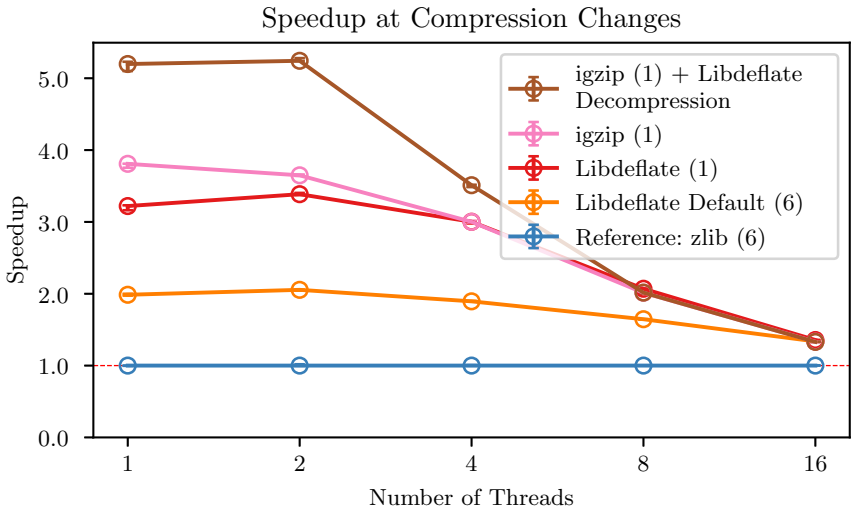

Fig. 14. Speedup of SAMtools `sort` after changing the compression library and the compression level. Reference is SAMtools `sort` using HTSlib with default zlib compression on compression level 6. Libdeflate is used via the HTSlib integration, igzip via 7BGZF. The input file is a 23.6 GB unsorted BAM file. SAMtools `sort` uses up to 48 GiB memory. The output is piped to `/dev/null` to minimize IO impacts. One temporary files is written to disk. SAMtools `sort` compresses the 23.6 GB compressed input file to 21,1 GB using zlib (the reference), 20.5 GB using libdeflate on level 6, 22.6 GB using libdeflate on level 1 and 31.5 GB using igzip on level 1 (in both settings). Numbers in parentheses indicate the compression level. One thread is single threaded computation; for a higher thread count, SAMtools `sort` uses the threads in addition to the main thread. Data points: Median, Error bars: fastest and slowest of 3 runs.
The plot illustrates that while alternative compression methods possess the potential to improve the runtime of SAMtools `sort`, this advantage diminishes as the number of threads employed for sorting increases.

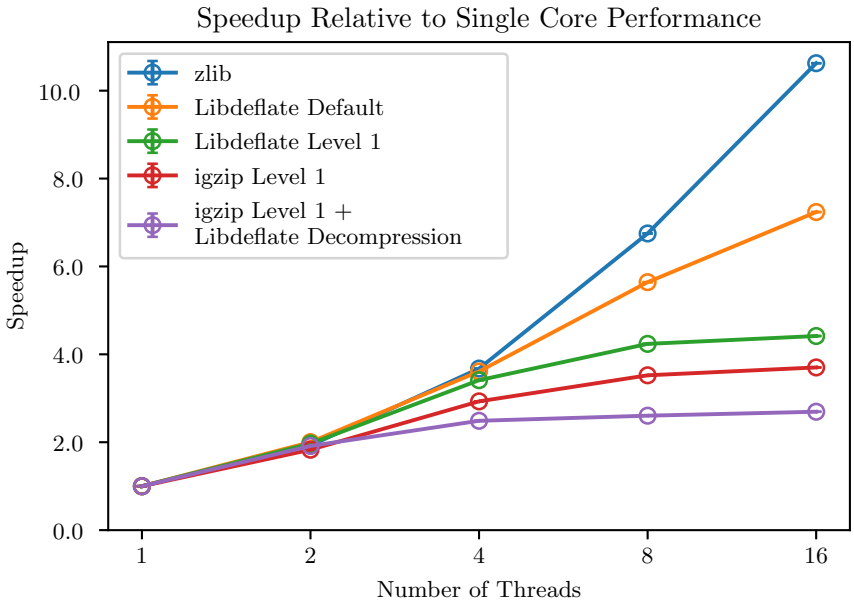

Fig. 15. Speedup of SAMtools `sort` after changing the compression library and the compression level. The reference is the respective single-core performance (strong scaling). The input file is a 23.6 GB unsorted BAM file. SAMtools `sort` uses up to 48 GiB memory. The output is piped to `/dev/null` to minimize IO impacts. Numbers in parentheses indicate the compression level. One thread is single threaded computation; for a higher thread count, SAMtools `sort` uses the threads in addition to the main thread. Data points: Median, Error bars: fastest and slowest of 3 runs.
The default zlib implementation on compression level 6 benefits most from a higher number of threads. The faster the compression speed, the less benefits come from using a higher number of threads.

7 Input/Output

7.1 Overview

While the internal algorithms of SAMtools `sort` are highly optimized for parallel processing, the runtime of SAMtools `sort` can also be constrained by limitations of the input and output devices. In contrast to a program's behavior or used libraries, often the IO devices can hardly be changed. However, in certain scenarios, optimization strategies can be employed to improve processing speed.

7.2 Compression

SAMtools `sort` outputs sorted BAM files containing aligned DNA-Reads in a BGZF compressed format. The BGZF compression format employed by SAMtools `sort` internally relies on GZIP compression provided by the compression library zlib. Zlib offers various compression levels (0-9), with compression level 0 resulting in uncompressed data. The user can achieve lower runtimes of SAMtools `sort` by using a faster implementation for GZIP compatible compression, or a lower compression level (Section 6). If the user sets the compression level to 0, SAMtools `sort` bypasses the compression step and outputs uncompressed data, leading to a higher throughput of SAMtools `sort`.

However, taking IO requirements into account, outputting compressed files turns out to be faster than writing them uncompressed if the disk does not match the compression throughput (Figure 16). As HTSlib compresses blockwise with every block compressed individually, compression scales well with increasing the amount of threads. Therefore, IO bottlenecks occur with higher probability when using a higher amount of threads.

On our test system, removing the output compression leads to a speedup of approximately 3.5 on a single thread. However, on 16 threads, running SAMtools `sort` with the default compression level 6 takes only 62 % (speedup of 0.62) of the time it takes with uncompressed output (see Figure 16).

To investigate further, we measure the IOWait time incurred during SAMtools `sort`'s execution. The IOWait time is the time a CPU waits in IDLE for the completion of IO operations, such as writing to disk.[12] For both compressed outputs from the example above, the IOWait time is below 2 seconds for every tested amount of threads. Moreover, for both compression levels, 1 and 6, the IOWait time is nearly identical, despite a difference in throughput. This observation casts doubt on the assumption that the IOWait primarily arises during the output stage for compressed output.

For the uncompressed output, the IOWait time is at 2 seconds for single threaded sorting and increases with the number of threads used up to 6 seconds on using 16 additional threads, as shown in Figure 17. These observations suggest an IO bottleneck on SAMtools `sort` with uncompressed output. However, the testing system's disk-write speed exceeds 2000 MB/s. In comparison,

[12] Waiting for memory access is not counted as IOWait [2]

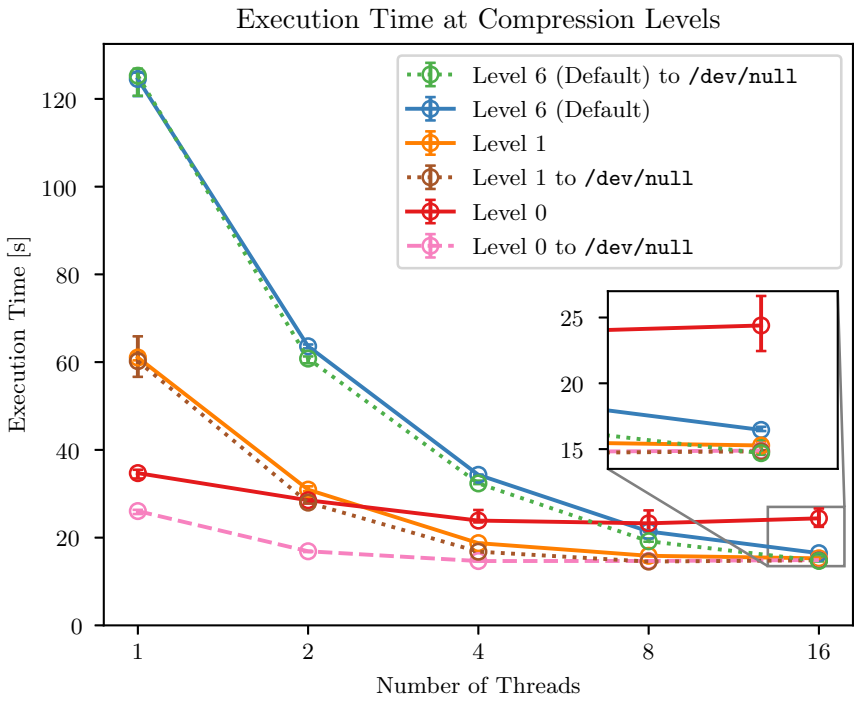

Fig. 16. Execution time of SAMtools `sort` with output written to disk and to `/dev/null`. One thread is single threaded computation; for a higher thread count, SAMtools `sort` uses the threads in addition to the main thread. Data points: Median, Error bars: fastest and slowest of 3 runs.
Writing uncompressed data is faster than writing compressed data for up to two threads. While writing to `/dev/null` is faster on all settings, the difference between the writing targets is higher for uncompressed output.

SAMtools `sort` writes 2.22 GB in 10.5 s (only the time from starting the merging of temporary files to the termination of SAMtools `sort`), which equals a throughput of 211.7 MB/s, if writing uncompressed output to `/dev/null`. Thus, the disk's write speed should be adequate even for writing uncompressed output. Therefore, future work could focus on a more comprehensive investigation into the reasons behind the observed increase in execution time when writing to disk compared to writing to `/dev/null`.

7.3 Unix Pipelines

Pipelines are a way of forwarding the output of one program to the input of another program. In Unix-like operating systems, they connect the standard output of one program with the standard input of another program. In the

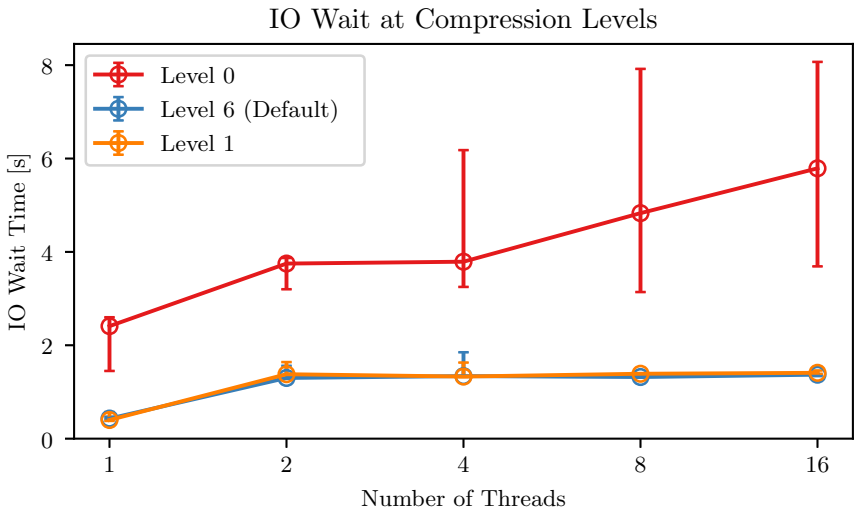

Fig. 17. IOWait time of the experiment shown in Figure 16. IOWait time is the time all processors spend in IDLE because they are waiting for IO requests. One thread is single threaded computation; for a higher thread count, SAMtools `sort` uses the threads in addition to the main thread. Data points: Median, Error bars: fastest and slowest of 3 runs.
With both compressed output, the IOWait time is nearly identical, while substantially higher with uncompressed output. This suggests an IO bottleneck at uncompressed output.

Linux Kernel, this is implemented by a ring buffer in memory [3]. When the output of a program is piped to the input of another, the first one writes to the buffer until the buffer is full, or the write operation is finished. If the buffer is full, the operating system halts the first program and allows the second program to read the data from the buffer. In shell scripting, users typically indicate a pipe by placing a vertical bar symbol ("|") between commands.

7.4 Pipelining in SAMtools

SAMtools' commands are designed to work on a stream. They process data sequentially, reading input and writing output in a single pass, without revisiting previous data. Therefore, users can combine several SAMtools commands with Unix Pipelines. For SAMtools `sort`, this has the following consequences: SAMtools `sort` can read BAM records as soon as a potential preceding command has written them. If the potential preceding command supports this, it can forward uncompressed BAM data to SAMtools `sort`. This eliminates the necessity for decompression in SAMtools `sort` and for compression for the potential preceding command. The potential IO bottleneck in writing uncompressed data is

prevented by pipelining, as the outputs of the piped commands are not written to the disk but stored in memory.

Using the default compression (zlib on compression level 6), the speedup on pipelining SAMtools `sort` into a SAMtools `view` command compared to SAMtools `sort` writing a file which is afterward read by SAMtools `view`, is for all tested amounts of threads below 1.2.(Figure 18). However, with SAMtools `sort` outputting uncompressed data, the speedup increases to approximately 1.8 for up to 8 used threads and 1.6 for 16 used threads.

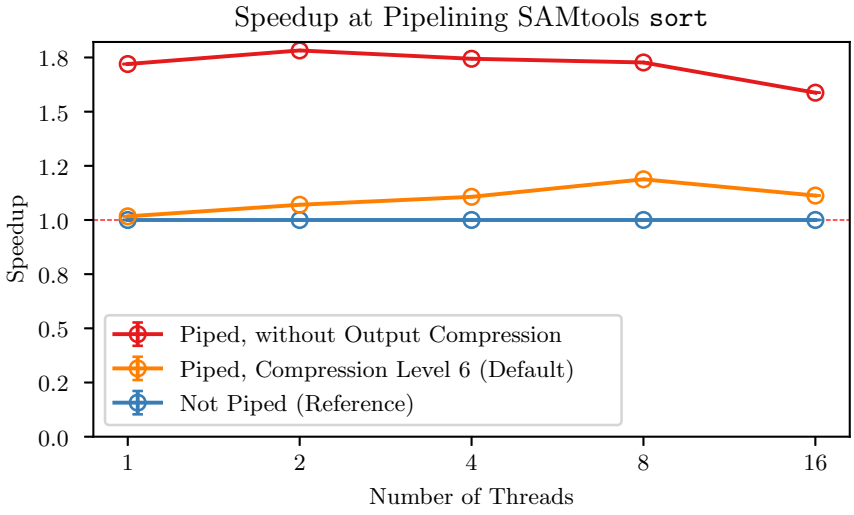

Fig. 18. Execution time comparison between different methods of chaining SAMtools `sort` with SAMtools `view`: *Not Piped* uses "`&&`" for chaining, both of the others use "`|`". SAMtools `sort` utilizes a total of 8 GB of RAM. With this memory setting, SAMtools `sort` utilizes one temporary file. One thread is single threaded computation; for a higher thread count, SAMtools `sort` uses the threads in addition to the main thread. Data points: Median, Error bars: fastest and slowest of 3 runs. The input file is a 2.3 GB unsorted BAM file.

7.5 Prefixes for Temporary Files

SAMtools `sort` utilizes temporary files as buffers in its external memory sort. If no prefix is specified, it writes them into the same directory as the output. If the output is to standard output, SAMtools `sort` places them in the current working directory. Users can set prefixes for temporary files via the "`-T`" parameter. To reduce IOWait times, we recommend choosing a directory located on a local, high-speed disk. As temporary files are deleted automatically after successful

sorting, the disks capacities are used only at the time of sorting. However, it is important to remember, that temporary files are less compressed (compression level 1) than input and output files compressed with default zlib compression (compression level 6). In combination, they typically require about 20% more disk space than the input file, possibly exceeding the capacity of small disks.

In scenarios with limited computational power but very fast and readily available storage, e.g., sorting on a laptop, the compression of the intermediate files can be omitted. At the moment, this is only possible by changing the source code of SAMtools `sort` (Appendix A.7). While this reduces the amount of computations SAMtools `sort` performs, it can lead to IO bottlenecks (Section 7.2).

7.6 Recommendation

The effectiveness of the IO-related recommendations presented here hinges on the specific characteristics of the employed storage devices. While these guidelines hold for our testing system, they may require adjustments based on the relative balance between computational speed and the read/write performance of the used disk.

IO operations can limit the runtime of SAMtools `sort`, if it outputs uncompressed files, such as uncompressed BAM files and writes them to disk. To reduce the amount of IO operations while also eliminating compression associated overhead in the output stage of SAMtools `sort`, we recommend using pipelines together with uncompressed output wherever possible.

To automate the process of SAMtools `sort` switching to uncompressed output, detecting if its output is piped to another SAMtools command would be necessary. As this is hardly possible and files piped into non-SAMtools commands might benefit from compression, we do not recommend implementing such a behavior. However, a warning should be displayed, if the output file remains unspecified. This situation commonly arises when the program is used within a pipeline. In this case, the filename of the output file is set to "`-`" and the output is forwarded to standard output using HTSlib. The change in the file name can be detected, and a warning can be printed to standard error, allowing users unfamiliar with compression options to adjust their parameters and save on computation time.

If pipelining is not possible, we recommend a compression level of 1, if SAMtools `sort` has more than a single thread available. If SAMtools `sort` is run single-threaded, writing uncompressed output leads to runtime improvements.

In addition, we recommend users to change the prefix for temporary files to a local, high-speed disk. This is especially important to remember if pipelining is used. In this case, SAMtools `sort` writes temporary files to the user's current working directory, which can lead to unexpected IO bottlenecks if the current working directory is, e.g., a network mount.

Modifying SAMtools `sort` to use uncompressed temporary files might yield performance benefits in specific scenarios. It should only be considered if computational resources are limited, while high-speed disk space is readily available.

8 Failed Approaches

8.1 Storing Pointers

The initial idea to speed up the sorting process consisted of the following steps:

1. Read once through the whole input file. For every BAM record, store a pointer to the location of the BAM record on the disk together with the attributes needed for sorting: the ID of the reference sequence, the DNA-Read is aligned to, the starting position of the alignment on this reference sequence, and the REVERSE flag.
2. Sort the resulting list based on the extracted attributes. Due to their smaller memory footprint compared to complete BAM records (10 bytes for the attributes needed for sorting compared to on average 250 bytes for BAM records in our test BAM file), sorting these attributes can be efficiently performed in-memory.
3. Iterate over the resulting sorted list. For every entry, read the referenced BAM record from disk using random reads and write it sequentially into the output file.

Although this method eliminates the need to write intermediate files, which currently consumes a substantial portion of the time needed for sorting (Section 5), it has some drawbacks:

BAM files are binary compressed representations of SAM files containing alignment information of DNA-Reads. While compression is beneficial to store and transfer the huge amounts (up to multiple terabytes per file) of data an aligned DNA-Read file can consist of, it makes random access a lot harder. Usually, a compressed file has to be decompressed from start to at least the position the user is interested in. To address this, BAM files are compressed in the BGZF file format. As a file in the BGZF file format consists of small blocks (less than 64 KB uncompressed) compressed individually in the DEFLATE format, for a random read only the number of the block the BAM record for the aligned DNA-Read is in, together with an offset into the compressed block is necessary.

However, this method is not suitable for accessing every single record in a file in random order: As mentioned before, the DEFLATE compressed blocks in a BGZF file typically have sizes of 64 KB of uncompressed data. Within our main test file, BAM records have on average a size of about 250 bytes. Therefore, a DEFLATE compressed block on average contains 256 BAM records. To extract every record in random order, the block has to be decompressed 256 times on average to halfway. Moreover, if the input file is very large in comparison to the available memory, caching the uncompressed blocks is not feasible.

We can now approximate the compression and decompression operations per BAM record at the current state of SAMtools `sort`, compared to this approach:[13] Currently, SAMtools `sort` decompresses the input file once, accounting

[13] For simplicity, we ignore possible caching of uncompressed blocks in the approach using random reads.

for one decompression operation for every 265 BAM records. Then, the record is written to the temporary file, resulting in one compression operation for every 256 records. In the final merge, SAMtools `sort` reads the temporary file again (one decompression operation per 256 records) and writes the output file (again, one compression operation for every 256 records). Input and output decompression and compression are necessary for both approaches, therefore they account for the same amount of compression and decompression operations in both approaches.

The approach using random reads, however, does not use compression operations in between reading the input file and writing the output file, but for every BAM record on average one compression operation on half a block, accumulating to around 128 decompression operations on whole blocks per 256 BAM records. Therefore, for the approach using random reads to be faster than the current behavior of SAMtools `sort`, compression of a single block (together with writing) would have to be 127 times slower than the combination of reading and decompressing a compressed block. However, with the default zlib compression library, compression on compression level 6 is approximately 9 times slower than decompression of a file.

These considerations apply only to sorting without merging of temporary files. If SAMtools `sort`'s memory limitations are too limited to allow for sorting without merging of temporary files, each merge of temporary files, would add one compression and decompression step for each block of BAM records which is part of the merge. These considerations are not supported by experiments. However, due to the multiple decompression operations for each block, a speedup from this approach seems unlikely.

In addition, having to read the file two times breaks the ability of SAMtools sort to work on a stream. As this is a core feature of SAMtools, breaking it should be avoided.

8.2 Removing Compression of Temporary Files

Our first measurements showed, that SAMtools `sort` spends most of its computation time for compression, even if it outputs uncompressed aligned DNA-Read files. Based on this observation, our initial assumption was that removing the compression of the temporary files would reduce the runtime of SAMtools `sort`. However, experimenting with faster compression libraries and utilizing a larger amount of processor cores, removing the compression of files turned out to be slower than keeping it on a low level (Section 7.2). Therefore, we decided not to change the compression SAMtools `sort` applies to temporary files.

9 Conclusion and Outlook

In this work, we analyzed SAMtools `sort` for sorting aligned DNA-Read files, specifically BAM files. We found that the most time-consuming part of sorting is compression and writing of output and temporary files. To reduce the runtime of SAMtools `sort`, we proposed setting a higher limit for the number of temporary files concurrently stored on disk, analyzed alternative implementations of the compression library used by SAMtools, and examined the impact of IO requirements on the runtime of SAMtools `sort`.

Setting a higher limit for temporary files concurrently stored on disk reduces the number of merges SAMtools `sort` performs, leading to lower runtimes when sorting large files with limited memory. By using libdeflate as the compression library, which is automatically the case if SAMtools is installed via Bioconda, we achieved a single-thread speedup of 2.3 compared to using zlib. On 16 threads, this results in a speedup of 1.6. By utilizing Unix pipelines, we can remove the output compression of SAMtools `sort`, achieving a speedup of 1.8 when SAMtools `sort` is piped to SAMtools `view` (SAMtools `view` with zlib compression at level 6). To increase user awareness of better compression options, we recommended implementing warnings if zlib is used instead of libdeflate, and if the output of SAMtools `sort` is piped but still compressed.

Combining our optimizations and using libdeflate for decompression, igzip with compression level 1 for compression of temporary and output files, and an increased limit for temporary files, we could archive a speedup of 6 compared to SAMtools `sort` with the default zlib compression on compression level 6 for single-threaded sorting of a 215 GiB BAM file utilizing 32 GiB of memory. Compared to SAMtools `sort` with libdeflate compression on compression level 6, this equals a speedup of 3.[14] Utilizing 16 additional threads for sorting a 215 GiB BAM file making use of 32 GiB of memory, our optimizations lead to a speedup of 2 compared to SAMtools `sort` with the default zlib compression on compression level 6, and a speedup of 1.5 compared to SAMtools `sort` with libdeflate compression on compression level 6.

These improvements for the runtime of SAMtools `sort` represent an important contribution to the field of bioinformatics, considering the widespread adoption of SAMtools `sort` evidenced by its over 5,000 citations and over 5.1 million downloads through Bioconda.

Future projects can investigate the merging process further, as this appears to be a bottleneck for very fast compression libraries. Additionally, they can implement igzip support into SAMtools and its file operation library HTSlib, as igzip has lower runtimes than libdeflate. Furthermore, the merging strategy of SAMtools `sort` can be enhanced by writing temporary files not only to half the limit for temporary files but to the limit minus existing "big files", which are results of previous merges, thereby halving the merges for the first few merges.

[14] Runtimes single-threaded: with zlib (6): 41395 s, with libdeflate (6): 20815 s, with libdeflate decompression and igzip (1) compression: 6887 s. Runtimes utilizing 16 additional threads: with zlib (6): 4287 s, with libdeflate (6): 3488 s, with libdeflate decompression and igzip (1) compression: 2217 s. (Compression levels in parentheses)

References

[1] getrlimit(2) - Linux manual page, https://man7.org/linux/man-pages/man2/getrlimit.2.html

[2] iostat(1) - Linux manual page, https://man7.org/linux/man-pages/man1/iostat.1.html

[3] linux/fs/pipe.c at master · torvalds/linux, https://github.com/torvalds/linux/blob/master/fs/pipe.c

[4] NovoSort | Novocraft, https://www.novocraft.com/products/novosort/

[5] richgel999/miniz: miniz: Single C source file zlib-replacement library, originally from code.google.com/p/miniz, https://github.com/richgel999/miniz/tree/master

[6] samtools-markdup(1) manual page, http://www.htslib.org/doc/samtools-markdup.html

[7] Storage and Computation Requirements | Strand NGS (May 2011), https://www.strand-ngs.com/support/ngs-data-storage-requirements

[8] Picard toolkit. https://broadinstitute.github.io/picard/ (2019)

[9] cloudflare/zlib (May 2024), https://github.com/cloudflare/zlib, original-date: 2014-03-05T22:14:29Z

[10] google/zopfli (May 2024), https://github.com/google/zopfli, original-date: 2015-03-09T10:32:36Z

[11] intel/isa-l (Apr 2024), https://github.com/intel/isa-l, original-date: 2016-01-25T23:10:55Z

[12] intel/zlib (Mar 2024), https://github.com/intel/zlib, original-date: 2013-12-13T19:36:32Z

[13] zlib-ng/zlib-ng (May 2024), https://github.com/zlib-ng/zlib-ng, original-date: 2014-10-13T15:47:27Z

[14] Alexandrescu, A.: The D Programming Language. Addison-Wesley Professional (Jun 2010), google-Books-ID: bn7GNq6fiIUC

[15] Audano, P.A., Sulovari, A., Graves-Lindsay, T.A., Cantsilieris, S., Sorensen, M., Welch, A.E., Dougherty, M.L., Nelson, B.J., Shah, A., Dutcher, S.K., Warren, W.C., Magrini, V., McGrath, S.D., Li, Y.I., Wilson, R.K., Eichler, E.E.: Characterizing the Major Structural Variant Alleles of the Human Genome (Jan 2019). https://doi.org/10.1016/j.cell.2018.12.019, https://linkinghub.elsevier.com/retrieve/pii/S0092867418316337

[16] Banerjee, S., Andorf, C.: ABRIDGE: An ultra-compression software for SAM alignment files (Jan 2022). https://doi.org/10.1101/2022.01.04.474935, https://www.biorxiv.org/content/10.1101/2022.01.04.474935v1

[17] Biggers, E.: ebiggers/libdeflate (Apr 2024), https://github.com/ebiggers/libdeflate, original-date: 2014-12-28T05:10:42Z

[18] Bojesen, J., Katajainen, J., Spork, M.: Performance Engineering Case Study: Heap Construction. In: Vitter, J.S., Zaroliagis, C.D. (eds.) Algorithm Engineering. pp. 301–315. Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48318-7_24

[19] Bonfield, J.K., Marshall, J., Danecek, P., Li, H., Ohan, V., Whitwham, A., Keane, T., Davies, R.M.: HTSlib: C library for reading/writing high-throughput sequencing data. GigaScience **10**(2) (Feb 2021). https://doi.org/10.1093/gigascience/giab007, https://doi.org/10.1093/gigascience/giab007

[20] Branstetter, M.G., Longino, J.T., Ward, P.S., Faircloth, B.C.: Enriching the ant tree of life: enhanced UCE bait set for genome-scale phylogenetics of ants and other Hymenoptera. Methods in Ecology and Evolution **8**(6), 768–776 (2017). https://doi.org/10.1111/2041-210X.12742, https://onlinelibrary.wiley.com/doi/abs/10.1111/2041-210X.12742, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/2041-210X.12742

[21] Campagne, F., Dorff, K.C., Chambwe, N., Robinson, J.T., Mesirov, J.P.: Compression of Structured High-Throughput Sequencing Data. PLOS ONE **8**(11) (Nov 2013). https://doi.org/10.1371/journal.pone.0079871, https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0079871, publisher: Public Library of Science

[22] Chen, R., Mias, G.I., Li-Pook-Than, J., Jiang, L., Lam, H.Y.K., Chen, R., Miriami, E., Karczewski, K.J., Hariharan, M., Dewey, F.E., Cheng, Y., Clark, M.J., Im, H., Habegger, L., Balasubramanian, S., O'Huallachain, M., Dudley, J.T., Hillenmeyer, S., Haraksingh, R., Sharon, D., Euskirchen, G., Lacroute, P., Bettinger, K., Boyle, A.P., Kasowski, M., Grubert, F., Seki, S., Garcia, M., Whirl-Carrillo, M., Gallardo, M., Blasco, M.A., Greenberg, P.L., Snyder, P., Klein, T.E., Altman, R.B., Butte, A.J., Ashley, E.A., Gerstein, M., Nadeau, K.C., Tang, H., Snyder, M.: Personal Omics Profiling Reveals Dynamic Molecular and Medical Phenotypes. Cell **148**(6), 1293–1307 (Mar 2012). https://doi.org/10.1016/j.cell.2012.02.009, https://www.cell.com/cell/abstract/S0092-8674(12)00166-3, publisher: Elsevier

[23] Chen, S., Zhou, Y., Chen, Y., Huang, T., Liao, W., Xu, Y., Li, Z., Gu, J.: Gencore: an efficient tool to generate consensus reads for error suppressing and duplicate removing of NGS data. BMC Bioinformatics **20**(23), 606 (Dec 2019). https://doi.org/10.1186/s12859-019-3280-9, https://doi.org/10.1186/s12859-019-3280-9

[24] Danecek, P., Bonfield, J.K., Liddle, J., Marshall, J., Ohan, V., Pollard, M.O., Whitwham, A., Keane, T., McCarthy, S.A., Davies, R.M., Li, H.: Twelve years of SAMtools and BCFtools. GigaScience **10**(2) (02 2021). https://doi.org/10.1093/gigascience/giab008, https://doi.org/10.1093/gigascience/giab008, giab008

[25] Derrien, T., Estellé, J., Sola, S.M., Knowles, D.G., Raineri, E., Guigó, R., Ribeca, P.: Fast Computation and Applications of Genome Mappability. PLOS ONE **7**(1), e30377 (Jan 2012). https://doi.org/10.1371/journal.pone.0030377, https://journals.plos.

org/plosone/article?id=10.1371/journal.pone.0030377, publisher: Public Library of Science

[26] Deutsch, L.P.: DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (May 1996). https://doi.org/10.17487/RFC1951, https://www.rfc-editor.org/info/rfc1951

[27] Deutsch, P.: Rfc1952: Gzip file format specification version 4.3 (1996)

[28] Fritz, M.H.Y., Leinonen, R., Cochrane, G., Birney, E.: Efficient storage of high throughput DNA sequencing data using reference-based compression. Genome Research **21**(5), 734–740 (May 2011). https://doi.org/10.1101/gr.114819.110, https://genome.cshlp.org/content/21/5/734, company: Cold Spring Harbor Laboratory Press Distributor: Cold Spring Harbor Laboratory Press Institution: Cold Spring Harbor Laboratory Press Label: Cold Spring Harbor Laboratory Press Publisher: Cold Spring Harbor Lab

[29] Hach, F., Numanagic, I., Sahinalp, S.C.: DeeZ: reference-based compression by local assembly. Nature Methods **11**(11), 1082–1084 (Nov 2014). https://doi.org/10.1038/nmeth.3133, https://www.nature.com/articles/nmeth.3133, publisher: Nature Publishing Group

[30] He, X., Chen, S., Li, R., Han, X., He, Z., Yuan, D., Zhang, S., Duan, X., Niu, B.: Comprehensive fundamental somatic variant calling and quality management strategies for human cancer genomes. Briefings in Bioinformatics **22**(3), bbaa083 (May 2021). https://doi.org/10.1093/bib/bbaa083, https://doi.org/10.1093/bib/bbaa083

[31] Hosseini, M., Pratas, D., Pinho, A.J.: A Survey on Data Compression Methods for Biological Sequences. Information **7**(4), 56 (Dec 2016). https://doi.org/10.3390/info7040056, https://www.mdpi.com/2078-2489/7/4/56, number: 4 Publisher: Multidisciplinary Digital Publishing Institute

[32] Hu, T., Chitnis, N., Monos, D., Dinh, A.: Next-generation sequencing technologies: An overview. Human Immunology **82**(11), 801–811 (Nov 2021). https://doi.org/10.1016/j.humimm.2021.02.012, https://www.sciencedirect.com/science/article/pii/S0198885921000628

[33] Huffman, D.A.: A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the IRE **40**(9), 1098–1101 (Sep 1952). https://doi.org/10.1109/JRPROC.1952.273898, https://ieeexplore.ieee.org/abstract/document/4051119?casa_token=PQmK8Fl0oV0AAAAA:d9eNzHPI2IkXxAOYnLSQ37HH1mmUFEP3_dMm99oKrUhUOfTkWFFp5YuMBCOdVQvDPR1UE7uvow

[34] Kim, T.M., Yang, I.S., Seung, B.J., Lee, S., Kim, D., Ha, Y.J., Seo, M.k., Kim, K.K., Kim, H.S., Cheong, J.H., Sur, J.H., Nam, H., Kim, S.: Cross-species oncogenic signatures of breast cancer in canine mammary tumors. Nature Communications **11**(1), 3616 (Jul 2020). https://doi.org/10.1038/s41467-020-17458-0, https://www.nature.com/articles/s41467-020-17458-0, publisher: Nature Publishing Group

[35] Kumar, S., Banks, T.W., Cloutier, S.: SNP Discovery Through Next-Generation Sequencing and Its Applications. In: Crop Breeding. Apple Academic Press (2016), num Pages: 36

[36] Lang, F., Schrörs, B., Löwer, M., Türeci, O., Sahin, U.: Identification of neoantigens for individualized therapeutic cancer vaccines. Nature Reviews Drug Discovery **21**(4), 261–282 (Apr 2022). https://doi.org/10.1038/s41573-021-00387-y, https://www.nature.com/articles/s41573-021-00387-y

[37] Li, H.: Tabix: fast retrieval of sequence features from generic TAB-delimited files. Bioinformatics **27**(5), 718–719 (Mar 2011). https://doi.org/10.1093/bioinformatics/btq671, https://doi.org/10.1093/bioinformatics/btq671

[38] Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., Subgroup, .G.P.D.P.: The Sequence Alignment/Map format and SAMtools. Bioinformatics **25**(16), 2078–2079 (06 2009). https://doi.org/10.1093/bioinformatics/btp352, https://doi.org/10.1093/bioinformatics/btp352

[39] Logsdon, G.A., Vollger, M.R., Eichler, E.E.: Long-read human genome sequencing and its applications. Nature Reviews Genetics **21**(10), 597–614 (Oct 2020). https://doi.org/10.1038/s41576-020-0236-x, https://www.nature.com/articles/s41576-020-0236-x, publisher: Nature Publishing Group

[40] McCormack, J.E., Hird, S.M., Zellmer, A.J., Carstens, B.C., Brumfield, R.T.: Applications of next-generation sequencing to phylogeography and phylogenetics. Molecular Phylogenetics and Evolution **66**(2), 526–538 (Feb 2013). https://doi.org/10.1016/j.ympev.2011.12.007, https://www.sciencedirect.com/science/article/pii/S1055790311005203

[41] Meyerson, M., Gabriel, S., Getz, G.: Advances in understanding cancer genomes through second-generation sequencing. Nature Reviews Genetics **11**(10), 685–696 (Oct 2010). https://doi.org/10.1038/nrg2841, https://www.nature.com/articles/nrg2841, publisher: Nature Publishing Group

[42] Myers, D.S., Bazinet, A.L.: Intercepting Arbitrary Functions on Windows, UNIX, and Macintosh OS X Platforms

[43] Neph, S., Kuehn, M.S., Reynolds, A.P., Haugen, E., Thurman, R.E., Johnson, A.K., Rynes, E., Maurano, M.T., Vierstra, J., Thomas, S., Sandstrom, R., Humbert, R., Stamatoyannopoulos, J.A.: BEDOPS: high-performance genomic feature operations. Bioinformatics **28**(14), 1919–1920 (Jul 2012). https://doi.org/10.1093/bioinformatics/bts277, https://doi.org/10.1093/bioinformatics/bts277

[44] Pabinger, S., Dander, A., Fischer, M., Snajder, R., Sperk, M., Efremova, M., Krabichler, B., Speicher, M.R., Zschocke, J., Trajanoski, Z.: A survey of tools for variant analysis of next-generation genome sequencing data. Briefings in Bioinformatics **15**(2), 256–278 (Mar 2014). https://doi.org/10.1093/bib/bbs086, https://doi.org/10.1093/bib/bbs086

[45] Popitsch, N., von Haeseler, A.: NGC: lossless and lossy compression of aligned high-throughput sequencing data. Nucleic Acids Research **41**(1), e27 (Jan 2013). https://doi.org/10.1093/nar/gks939, https://doi.org/10.1093/nar/gks939

[46] Sexton, N.R., Smith, E.C., Blanc, H., Vignuzzi, M., Peersen, O.B., Denison, M.R.: Homology-Based Identification of a Mutation in the Coronavirus RNA-Dependent RNA Polymerase That Confers Resistance to Multiple Mutagens. Journal of Virology **90**(16), 7415–7428 (Jul 2016). https://doi.org/10.1128/jvi.00080-16, https://journals.asm.org/doi/full/10.1128/jvi.00080-16, publisher: American Society for Microbiology

[47] Tamura, T., Ito, J., Uriu, K., Zahradnik, J., Kida, I., Anraku, Y., Nasser, H., Shofa, M., Oda, Y., Lytras, S., Nao, N., Itakura, Y., Deguchi, S., Suzuki, R., Wang, L., Begum, M.M., Kita, S., Yajima, H., Sasaki, J., Sasaki-Tabata, K., Shimizu, R., Tsuda, M., Kosugi, Y., Fujita, S., Pan, L., Sauter, D., Yoshimatsu, K., Suzuki, S., Asakura, H., Nagashima, M., Sadamasu, K., Yoshimura, K., Yamamoto, Y., Nagamoto, T., Schreiber, G., Maenaka, K., Hashiguchi, T., Ikeda, T., Fukuhara, T., Saito, A., Tanaka, S., Matsuno, K., Takayama, K., Sato, K.: Virological characteristics of the SARS-CoV-2 XBB variant derived from recombination of two Omicron subvariants. Nature Communications **14**(1), 2800 (May 2023). https://doi.org/10.1038/s41467-023-38435-3, https://www.nature.com/articles/s41467-023-38435-3, publisher: Nature Publishing Group

[48] Tarasov, A., Vilella, A.J., Cuppen, E., Nijman, I.J., Prins, P.: Sambamba: fast processing of NGS alignment formats. Bioinformatics **31**(12), 2032–2034 (Jun 2015). https://doi.org/10.1093/bioinformatics/btv098, https://academic.oup.com/bioinformatics/article/31/12/2032/214758

[49] Tarreau, W.: wtarreau/libslz (May 2024), https://github.com/wtarreau/libslz, original-date: 2020-02-16T17:36:32Z

[50] The 1000 Genomes Project Consortium, Clarke, L., Zheng-Bradley, X., Smith, R., Kulesha, E., Xiao, C., Toneva, I., Vaughan, B., Preuss, D., Leinonen, R., Shumway, M., Sherry, S., Flicek, P.: The 1000 Genomes Project: data management and community access. Nature Methods **9**(5), 459–462 (May 2012). https://doi.org/10.1038/nmeth.1974, https://www.nature.com/articles/nmeth.1974

[51] The Bioconda Team, Grüning, B., Dale, R., Sjödin, A., Chapman, B.A., Rowe, J., Tomkins-Tinch, C.H., Valieris, R., Köster, J.: Bioconda: sustainable and comprehensive software distribution for the life sciences. Nature Methods **15**(7), 475–476 (Jul 2018). https://doi.org/10.1038/s41592-018-0046-7, https://www.nature.com/articles/s41592-018-0046-7

[52] Tucker, G., Oursler, R., Stern, J.: ISA-L Igzip: Improvements to a Fast Deflate. In: 2017 Data Compression Conference (DCC). pp. 465–465. IEEE, Snowbird, UT, USA (Apr 2017). https://doi.org/10.1109/DCC.2017.88, http://ieeexplore.ieee.org/document/7923748/

[53] Yamada, T.: 7bgzf: Replacing samtools bgzip deflation for archiving and real-time compression. Computational Biology and Chemistry **85**, 107207

(Apr 2020). https://doi.org/10.1016/j.compbiolchem.2020.107207, https://www.sciencedirect.com/science/article/pii/S1476927119311375

[54] Yang, C., Luo, T., Shen, X., Wu, J., Gan, M., Xu, P., Wu, Z., Lin, S., Tian, J., Liu, Q., Yuan, Z., Mei, J., DeRiemer, K., Gao, Q.: Transmission of multidrug-resistant Mycobacterium tuberculosis in Shanghai, China: a retrospective observational study using whole-genome sequencing and epidemiological investigation. The Lancet Infectious Diseases **17**(3), 275–284 (Mar 2017). https://doi.org/10.1016/S1473-3099(16)30418-2, https://www.thelancet.com/journals/laninf/article/PIIS1473-3099(16)30418-2/fulltext, publisher: Elsevier

[55] Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory **23**(3), 337–343 (May 1977). https://doi.org/10.1109/TIT.1977.1055714, https://ieeexplore.ieee.org/abstract/document/1055714?casa_token=o6sVfpkKOzsAAAAA:N2nObMKmOGCfeeTa-y-ER_qXHROSJwrTQn1FsCKWHryfwcLW3kwwJKg1CnUCgO8aHohDf-5SqQ

Appendices

A Methods

A.1 Computational Environment

We conducted the experiments on a machine with the following specifications:

- Operating System: AlmaLinux release 8.9 ([core-4.1 kernel])
- CPU:
  - Architecture: x86_64 (64-bit capable)
  - Model: AMD Ryzen 9 3950X 16-Core Processor
  - Cores: 16 physical cores, 32 logical cores
- Memory: 64 GB total RAM
- Storage: NVMe device (KXG60PNV2T04 NVMe KIOXIA 2048 GB)

7BGZF was compiled with clang (16.0.6). All other programs were compiled with gcc (8.5.0).

A.2 Setting Compression Levels In SAMtools

In SAMtools `sort`, the user can set the compression level of the output file using the "`-l`" parameter. Possible options are integers from 0 to 9. Compression level 0 is equal to no compression (equal to the `-u` parameter).

For other SAMtools commands where the "`-l`" parameter does not exist, the user can still change the compression level of the output via adding `--output-fmt-option level=1` to the arguments of the command (Put the desired compression level between 0 and 9 instead of `1`).

A.3 Configuring Libdeflate Support in HTSlib

To decide manually between using zlib and libdeflate, the user can run the HTSlib `configure` script with the `--with-libdeflate` resp. `--without-libdeflate` option. To use `LD_PRELOAD` for changing the zlib implementation, the user must build HTSlib without libdeflate.

A.4 Libdeflate Compression Level Mapping

zlib	1 2 3 4 5 **6** 7 8 9
libdeflate	1 2 3 5 6 **7** 8 10 12

Table 2. Mapping between zlib compression levels and libdeflate compression levels in HTSlib. The default level is marked **bold**.

A.5 Configuring 7BGZF

To configure the compression library and the compression level, 7BGZF uses to compress BAM files in the BGZF format, the user can set the `BGZF_METHOD` environment variable to a compression library's name concatenated with a compression level before running a SAMtools command.

For the compression libraries, users can choose one of `zlib`, `miniz`, `slz`, `libdeflate`, `zlibng`, `igzip`, and `zopfli`. Their possible compression levels vary:

- `zlib`, `miniz`, and `zlibng` offer levels from `1` to `9`.
- `libdeflate` offers levels from `1` to `12`.
- `igzip` offers levels from `1` to `3`.
- `slz` only supports level `1`.
- While `zopfli` does not use compression levels in the traditional sense, it allows specifying an amount of iterations (greater than or equal to `1`) within the compression level parameter of 7BGZF.

Example for calling SAMtools `sort` with igzip and compression level 1:
`BGZF_METHOD=igzip1 LD_PRELOAD=/path/to/7bgzf.so samtools sort ...`

A.6 Using HTSlib as a Shared Library

SAMtools uses HTSlib as static library by default. To override methods from HTSlib, SAMtools must use HTSlib as a shared library. To use HTSlib as a shared library, the user has to change a single line in SAMtools' `config.mk.in` and change `@Hsource@HTSLIB = $(HTSDIR)/libhts.a` to refer to `libhts.so` instead. Running SAMtools `./configure` script and `make` leads to SAMtools using HTSlib as a shared library. If SAMtools does not find the shared object, export the location of HTSlib in the `LD_LIBRARY_PATH` environment variable.

A.7 Writing Uncompressed Temporary Files

To remove the compression of temporary files, the source code of SAMtools `sort` must be changed. This can be done by replacing the parameter `mode` of the first call of `bam_merge_simple` in the `bam_sort_core_ext` method, which is located in `bam_sort.c`. Current values are, depending on the existence of a position too large to be stored in a BAM file, "`wzx1`" for BGZF compressed SAM files on compression level 1 and "`wbx1`" for BAM files with compression level 1. Those can be changed to "`w`" for SAM files and "`wbx0`" for uncompressed BAM files.