Faster Sorting of Aligned DNA-Read Files

Dominik Siebelt

Karlsruhe Institute of Technology

Abstract. The abstract should briefly summarize the contents of the paper in 15–250 words.

1 Introduction

To enable faster downstream analysis, aligned DNA-Read Files are sorted.

1.1 SAMtools

SAMtools [**?**] is a collection of tools to work on alignment data. It relies on the co-developed HTSlib [**?**] for reading and writing information files, e.g. BAM files. SAMtools offers functionality for different operations on alignment data, such as format conversion, statistics, variant calling and many more, including the sorting, which is the focus here.

1.2 SAM and BAM files

A Sequence Alignment/Map (SAM) as specified by Li et al. [**?**] 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. Here, a segment is a continuous sequence or subsequence of a raw DNA read.
The alignment information mainly consists of the ID of the reference sequence the alignment is mapped to, the position where the alignment starts in the reference sequence and a CIGAR string providing the alignment at this position. The CIGAR String as specified in [**?**] lists i.a. sequential matches, mismatches, insertions and deletions and therefore represents the alignment of the corresponding segment and its reference sequence.
A BAM file is the binary representation of a SAM file. The main differences are the usage of a 4-bit encoding for the sequences, 3-bit for CIGAR Symbols and a 0-based instead of 1-based coordinate system for the position.
Furthermore, a BAM file is per default *BGZF* compressed.

1.3 BGZF Compression

BGZF is a lossless compression method proposed at the same time as 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 that matching pieces of the file can be found over a larger span. However, to decrypt such a compressed file, it also needs to be read from the beginning and, depending on the compression method, decompressed at least until the point of interest. Since alignment data can produce very large files but not all of their regions are needed for every use case, it is beneficial to enable some form of random access. To archive this, BGZF utilizes *GZIP* [**?**] and the *DEFLATE* algorithm [**?**] by Phil Katz to compress large files into blocks of less than 64KB size (compressed and uncompressed). These blocks are the concatenated. Thus, fast random access using index files is possible. In an index file, the position of a piece of information is stored in a 64-bit integer. It consists of a 48-bit unsigned integer *coffset* indicating the number of the compressed block and a 16-bit unsigned integer *uoffset* describing the position in the uncompressed block.
Besides this, the BGZF format also provides compatibility with GZIP. As GZIP allows this combination of multiple compressed files to one file, a BGZF-compressed file can be decompressed by any standard GZIP implementation. This can be practical if for example a SAM file that can be viewed with any text editor is compressed using BGZF.
Like GZIP, BGZF supports compression levels ranging from 1 (fastest but worst) to 9 (slowest but best) which are basically the compression levels used for the underlying GZIP compression. The compression levels affect the size of the compressed files as shown in Figure 1. The speed of the compression depends mainly on the compression level, the GZIP-implementation and the number of used threads (see Figure 2). To measure the compression speed while minimizing the influence of the sort and merge processes, the speed of HTSlib's `bgzip` can be measured. This is a tool using BGZF to compress arbitrary files. As its inner mechanisms are exactly the same as the compression part of SAMtools' `sort` tool (use the same underlying HTSlib methods), pipelining `bgzip`'s output to `/dev/null` gives an estimation of the computation time needed only for compression at using SAMtools `sort`. This still holds for compression level 0. At this level, the input is not compressed, but directly written to the output.

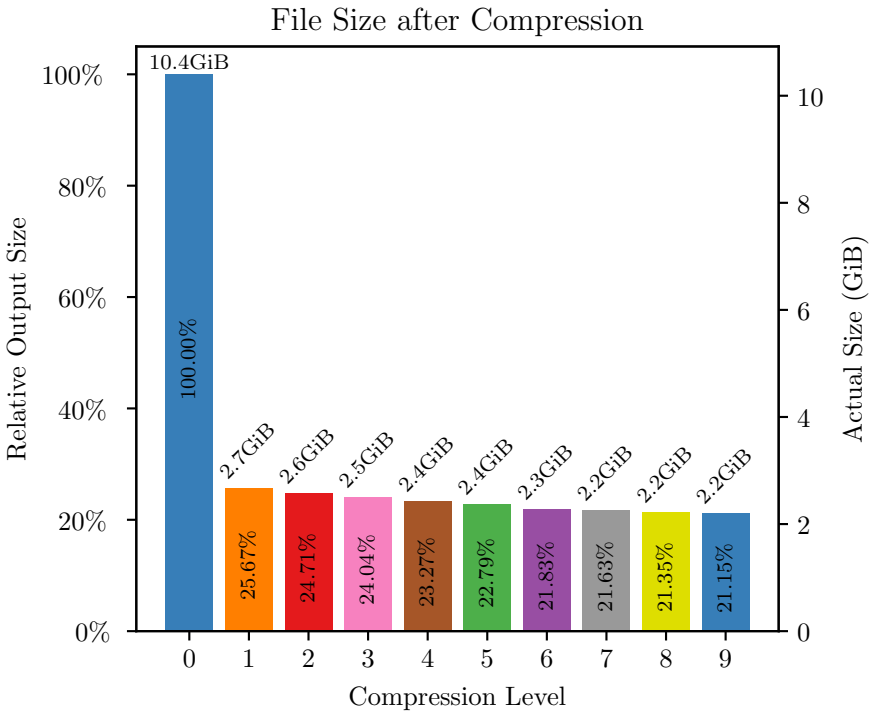

Fig. 1. Comparison of the size of BGZF compressed files on all compression levels, exemplified using a 10.4GiB unsorted BAM file.

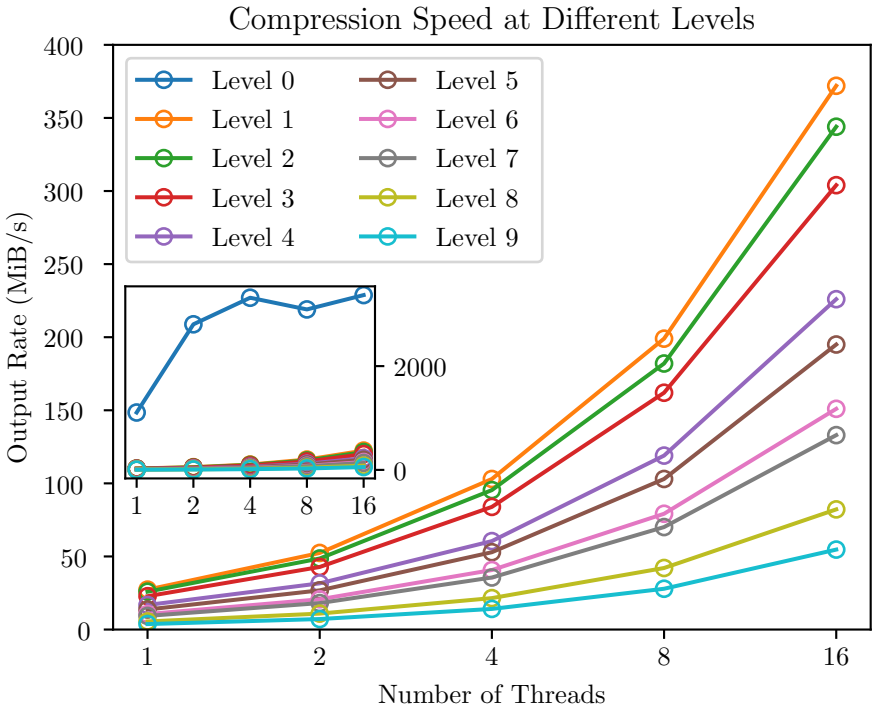

Fig. 2. Comparison of the output rate of HTSlib's `bgzip` which uses BGZF to compress A 10.4GiB unsorted BAM file. For reference, compression level 0 is plotted in the smaller inset plot.

2 Analysis (Version 1.19.2)

2.1 Algorithm

Prerequisites The process of sorting alternates, depending on some internal Constants and command-line-arguments:
We only focus on sorting by the order of the reference, then position and then the REVERSE flag which indicates, if the sequence is aligned forward or backward to the reference. This order is the one used by SAMtools per default, although other sorting criteria e.g. tags or the read name are possible.
The maximum amount of memory used to sorting is calculated by the amount of memory the user specifies via the `-m` option multiplied by the (via `-@` option) assigned number of threads. Here, we refer to the total amount as `max_mem`.
The in- and output formats are per default inferred from the file names.
SAMtools `sort` passes its output to standard output if no output file is specified. In this case, the output format is set to BAM. The maximum number of temporary files is hard-coded as 64 in a constant named `MAX_TMP_FILES`.
The gzip compression level for temporary files is set to 1, while the compression level of the result file can be changed via the `-l` parameter. It defaults to the default compression level used by the library that implements the compression, usually 6, but can be set to a number between 0 (no compression) and 9 (highest and slowest compression).

Sorting SAMtools performs an external sort process using temporary files that are merged in the end. The sorting process flow is represented by the flowchart in Figure 3. The sorting starts by sequentially reading BAM records from the input file using HTSlib for parallel decompression. Once the memory limit given by `max_mem` is exceeded, these records are split into as many blocks as threads are specified and afterward sorted in parallel.
Then, the merge is performed. In the merge, all the sorted in-memory files are written to a single sorted temporary BAM file. In Addition, some of the previously created temporary files are added: The algorithm distinguishes between small files and big files. Small files are files generated by merging one set of in memory blocks. If the number of small files is greater than half of the maximum allowed number of temporary files, all the small files are merged (and afterward deleted). The result of a merge of in-memory and temporary files is a big file. If the total number of files exceeds the limit for temporary files, all temporary files including big files are included in the merge (and afterward deleted). 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

[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.

combinations of very big files and little memory.
In general, the temporary files on the disc can be put into three categories: small files being at most as big as the sorted in-memory blocks 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 bigger size than the other big files.
After the merge, the algorithm repeats the previous steps until the end of the input file is reached. As the last step, the remaining in-memory BAM records are sorted and merged together with all temporary files and written to the output file.

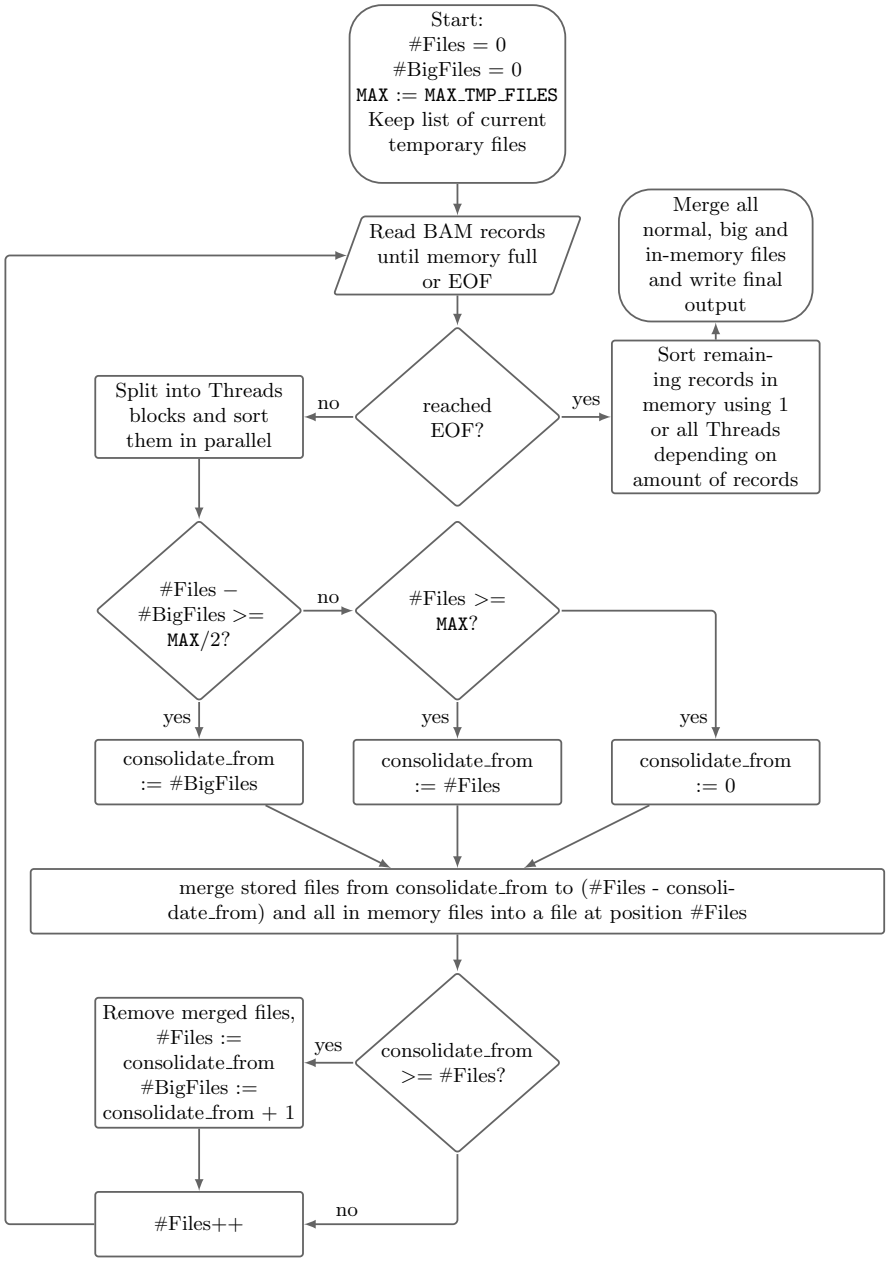

Fig. 3. Flow chart showing the current process of sorting, especially the choosing of files to be merged. The list of files is a 0-based list of their names. In the beginning it is empty, after BAM records are read the second time, there is a single record at position 0 and #Files is 1.

2.2 Time Allocation

Understanding the resource utilization and the time allocation of the different parts of the sorting process is crucial to be able to optimize its computation time. However, the process has different points of constraint on different machines, as we will see in the following.
In general, high time consumption of the SAMtools `sort` method can be traced to three main blocks. In the following, I partition them in temporary files management, compression and Input/Output.

3 Temporary Files

Temporary files are necessary for SAMtools `sort` to work as a stream while processing more data than can be held in memory. Unfortunately, writing temporary files is time-consuming. When only looking at the time between reading and decompressing the input and writing and compressing the output, operations involving temporary files lead to the most time consumption. Thus, the amount of temporary files should be minimized. More specific, a BAM record should be written as infrequently as possible. On the other hand, limitations of the Operating System have to be taken into consideration.

3.1 Analysis

Observing the generation of temporary files in SAMtools `sort`, an irregularity stands out. Sorting a 216GB unsorted BAM file utilizing 16 Cores and a total of 32GiB memory, nearly all temporary files take on average 29.75 seconds from opening the file to closing it. In this time, the 16 (one per core) sorted lists in memory are merged and written to the file. Temporary files are compressed like normal BAM files but with GZIP compression level 1. However, the 33rd file takes 945.38 seconds. That is 31.7 times the amount of time needed for the temporary files before. What caused this significant increase?
As explained in 2.1, SAMtools `sort` performs merges of temporary files if a certain number of temporary files is reached. This is to limit the total number of temporary files needed for the final merge. In previous versions of SAMtools where this behavior did not exist, opening to many files at the same time in the final merge caused the program to crash. To understand how many temporary files are written and when they are merged, one has to look into the algorithm for merging.
SAMtools `sort` has, as mentioned above, a hard coded limit for temporary files. Until reaching half of this limit, it writes all blocks that are results of sorting the amount of BAM records fitting into memory at once to a single temporary file. If the limit is reached, the next temporary file is a merge of all small temporary files written before together with the next block of sorted BAM records in memory. In summary, on writing every 33rd file, SAMtools `sort` performs a merge of 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. As the SAMtools `sort` merges temporary files on half of the limit and generates a single file at every merge, the limit is reached later than the square of half the limit. If the limit is reached and 33 big files exist, SAMtools `sort` merges them again together with all small files and the records currently in memory. For details, refer to section 2.1.
The amount of merges depends on the number of temporary files needed in total. This is mainly determined by the amount of the memory the user gives to SAMtools `sort`. The user can set this limit using the "`-m`" parameter. Defaulting to 768MiB, it gets multiplied by the number of threads. The result is the limit up to which SAMtools `sort` reads BAM records in one block. This is also a good approximation for the size of a small temporary file before compression. At least one MiB per thread is enforced to prevent the creation of a huge amount of temporary files. One might instinctively believe that sorting becomes faster as more memory is utilized. Figure 4 illustrates that this is generally the case, although not in a linear proportion. Moreover, between 400MiB and 12800MiB memory

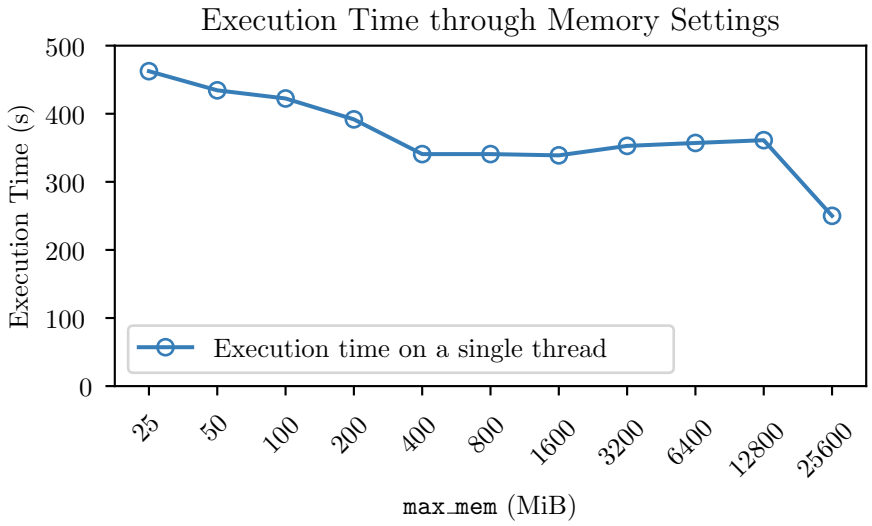

Fig. 4. Execution time of SAMtools `sort` on a 2.4GB BAM file using default parameters except `-m` for memory limitation setting.

allocation the execution time does not decrease - despite SAMtools `sort` using up to 32 times more memory. To investigate further, one can take a look at the amount of temporary files produced. The input file expands to just a little larger than the second-highest memory limitation in Figure 4. Therefore, at the highest setting 25600MiB which equals to 25GiB, SAMtools `sort` produces no

temporary file. At the next highest settings, it produces 1, 2, 4, ... temporary files, as the `max_mem` parameter 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 400MiB, SAMtools `sort` generates 32 temporary files as expected. At 200MiB, it generates 65 temporary files. This indicates, that after having processed 12800MiB of data, 200MiB are not enough to keep the remaining data in memory until the final merge into the output file, but 400MiB are. For this reason, the size of the BAM file must increase to between 13000MiB and 13200MiB in memory.
Now it becomes obvious why there are no speed improvements between 400MiB and 12800MiB. In between those settings, SAMtools `sort` writes exactly the same records to the disk, in exactly the same order. The only difference is the number of files they are split into.
This changes at 200MiB `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 files from disk, decompresses, merges, compresses and writes them to disk a second time.
At 100MiB SAMtools `sort` generates 3 big files, at 50MiB 7 and at 25MiB 15. This is also reflected in the total amount of bytes written. In the parameter settings that produce temporary files but not enough of them to be merged to big files SAMtools `sort` writes a total of 2.4GiB in temporary files. This number goes up to 3.7GiB, 4.3Gi, 4.6GiB and 4.8 GiB for 200MiB, 100MiB, 50MiB and 25MiB. 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 `max_mem` parameter. In Addition, the proportional influence on the total time spend before merging the final result lowers with the number of performed merges: While writing the first big temporary file costs approximately as much as writing all temporary files before, writing the second one costs only a third of all file writing before, the next one 1/5 then 1/7 and so on.
Obviously, the measurements above are unrealistic, as nowadays even Laptops have more memory installed. At the same time, BAM files are usually way bigger than the used sample, which I sampled by randomly taking 1% of BAM records from a real world BAM file. To get an impression of the impacts of increasing the file size, one can 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 method used is a radix sort, which is also in $\mathcal{O}(n)$. For merging, a heap based approach is chosen, which works in $\mathcal{O}(n \log(k))$. Here, k is the number of sorted list to be merged. Thus, in theory, keeping the same ratio of input size and available memory should produce the same amount of temporary files. Together with all other operations being in linear time, results on little files with little memory should transfer proportional to big files and more memory. This is confirmed by the experiment shown in Figure 5.
However, changing only one of these parameters has different effects. Using SAMtools for example locally installed on a laptop to sort a larger BAM file

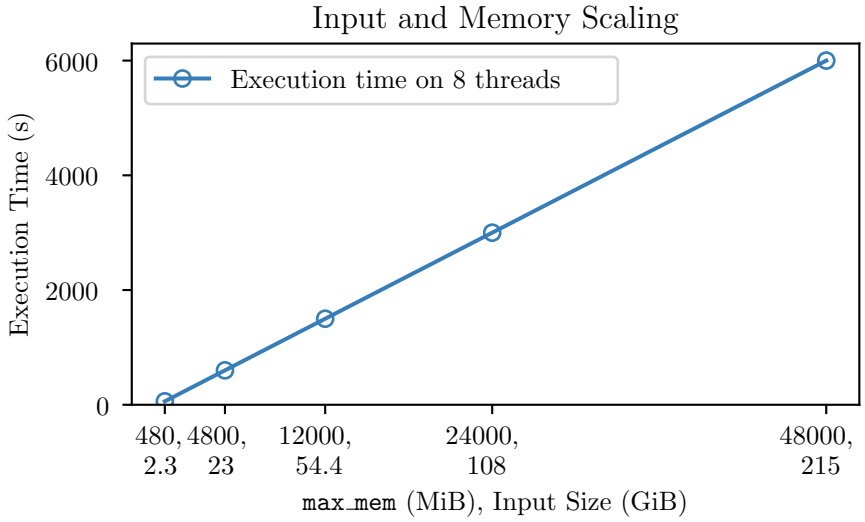

Fig. 5. Execution time of SAMtools `sort` on different input sizes. Keeping the ratio from input size to `max_mem` constant, the execution time grows linear with increasing both parameters.

can produce many temporary files if memory is limited. If e.g. 8GB are available for SAMtools `sort`, it cannot process files bigger than 50GB without merging temporary files. This behavior gets worse if the ratio of the input file to the `max_mem` setting grows further. An important point that should not be exceeded is reaching 1120 temporary files. At this point, SAMtools `sort` merges all "big 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 1700GiB file using 8GiB of memory, which is an unlikely use case.
In conclusion, writing larger amounts of temporary files leads to merging of temporary files, which is time-consuming. This is mainly affected by the ratio of the size of the input file to the amount of available memory.

3.2 Recommendation

Since the most time-consuming part of sorting is compressing and writing, the frequency of writing a single BAM record should be minimized. Therefore, SAMtools `sort` should perform as few merges as possible.
To archive this without changing any source code, the user can only change the "`-m`" parameter for memory limitation setting. The more memory the user gives to the process, the less likely SAMtools `sort` needs to merge temporary files. Therefore, the user should set this limitation as high as possible. 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.
Especially on laptops or for working on large files, the computing device provides not enough physical memory to avoid merging. Because of this, I recommend enlarging the limit for open temporary files. At the moment, it is set to 64 while 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 reaches 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 the computer I used for most of the experiments I present in this work). The hard limit can not be increased.
On a Unix operating system, a program can obtain its soft limit using the `getrlimit` [**?**] system call. Knowing that SAMtools `sort` only opens all the files to merge, an output file (or standard output), possibly an index file and has standard input and standard error open, `sort` should recognize how many files can be opened and set the limit accordingly. Then, the user can also increase the soft limit, making merging of temporary files obsolete for realistic use cases. For compatibility reasons, if the system call fails, the limit can be kept. This is e.g. on Windows machines necessary due to Windows not having a limit for open file handles and thus not supporting `getrlimit`.
Notice, that the necessary file size until the limit is reached grows quadratic to the maximum amount of temporary files SAMtools `sort` allows. On the other hand, the file size up to which SAMtools `sort` does not perform a merge of temporary files grows only half as fast as the maximum number of temporary files.

3.3 Evaluation

Increasing the number of allowed temporary files to 1019 ($= 1024 - 5$) while keeping everything else the same leads to a 15.5-fold increase in the potential file size before triggering a merge. SAMtools `sort` then performs the first merge of temporary files at the 513th file instead of at the 33rd. Having a limited amount of 8GiB of memory, the change to 1019 allowed temporary files raises the tipping point, after which the first merge of temporary files occurs, from around 50GiB input size to around 775GiB input size. Figure 6 shows, that in the example I presented before, a noticeable speedup occurs only at the lower memory settings but not at the lowest. One can understand this by referring to Figure 7. Figure 7 shows the number of times, a block of BAM records in size of the available `max_mem` is compressed and written into a temporary file. Having a limit of 64 temporary files, files are merged relatively often compared to having a limit of 1019 temporary files. Therefore, the graph for the smaller limit

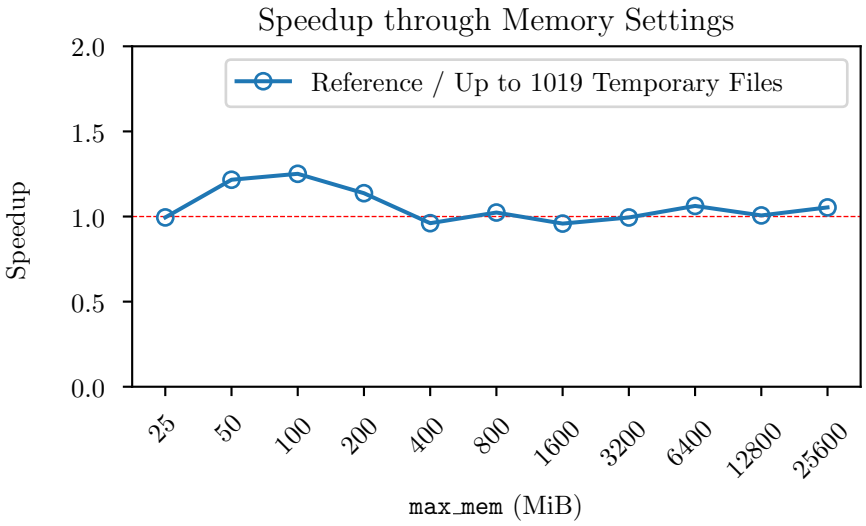

Fig. 6. Speedup after setting the limit for temporary files to 1019. Calculated by dividing the values from Figure 4 by the values of the increased temporary file limit. All other parameters are the same as in Figure 4.

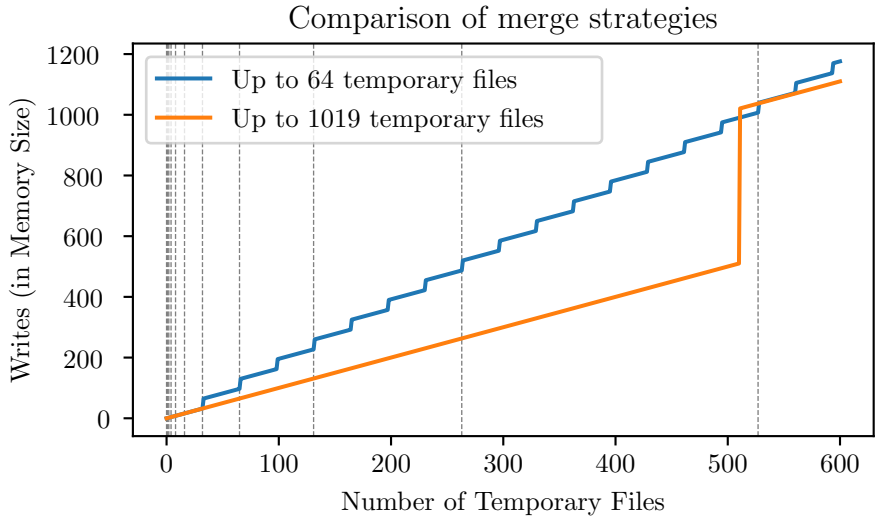

Fig. 7. The y-axis shows the amount of compress and write operations for blocks in size of the available memory. This means, if one small temporary file is produced, it counts as one write operation. If 32 files are merged together with one in-memory block, it counts as 33 Operations. Vertical gray lines mark numbers of temporary files produced in the example in Figure 4 and 6. (E.g. 527 at 25MiB `max_mem`.)

has much more steps. On the other hand, changing the limit for temporary files also means, that if temporary files are merged, much more of them are merged at once. The gray, vertical lines mark the number of temporary files SAMtools `sort` creates at the different settings in the example in Figure 4 and 6. If the amount of temporary files is below 33 files, which is true for the seven highest memory settings, the total amount of decompression and write operations (in memory-sized blocks) is equal for both limits. After this, the larger limit gains advantage until the 509th temporary file. Writing the 510th temporary file with a limit of 1019 temporary files means, that SAMtools `sort` rewrites the content of all 509 files it generated before again. For some temporary files, the smaller limit even needs less writes in total. This happens, because after the first merge at the greater limit, the content of every file SAMtools `sort` generated before is written twice, except the content of the in-memory block of BAM records that is taken into the merge that results in file number 510. However, considering the smaller limit, at the same number of files nearly all content of temporary files is written twice as well. Here, exceptions are again the BAM records being an exclusive part of a big file, as they are taken into a merge together with small files but never written into a small file on their own. As this happens much more frequent at the smaller limit, the smaller limit needs less compress and write operations every time the greater limit performs a merge. This changes again at the next merge at the smaller limit.
In summary, increasing the limit for temporary files results in a noticeable speedup if it prevents merges. However, if a merge with a greater limit is performed, the impact on the execution time is stronger than with a smaller limit. 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.

3.4 Future Work

Even after setting the limit for temporary files to 1019, SAMtools `sort` merges temporary files the first time at writing the 510th temporary file. Due to merging, SAMtools `sort` reaches the real limit of 1019 files after writing more than 260000 files. While with the proposed changes the user can prevent merges by changing the soft limit, knowing about this option is unlikely for an average user. Thus, SAMtools `sort` should use as much of the limit as early as possible to keep the number of merges as low as possible. This can be archived 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.

4 Compression

Compression is a part of writing BAM files, as per default compression is applied to all BAM files and even part of the specification. Although compression of BAM files is beneficial in the long term in order to reduce storing costs and transfer speed, it comes with a significant resource overhead.

4.1 Analysis

Running on 16 cores, with a total of 32GiB of memory, SAMtools `sort` takes 71 minutes and 57 seconds to sort a 215GB BAM file. However, SAMtools `sort` only uses 2 minutes and 35 seconds, which are 3.6% of the total time, for sorting (merging not included). What is the rest of the time spend on?
Performing SAMtools `sort` on a laptop, trough various settings compression and decompression together account for around 95% of the CPU time. Approximately 80% are solely required by the *deflate* method that is used for the compression. SAMtools has outsourced all file operations to HTSlib. HTSlib depends on zlib for compression and decompression. Compression is done in blocks using the DEFLATE algorithm. Thus, compression can and is parallelized: Every time a block is to be compressed, HTSlib gives it to a thread pool of workers that compress blocks in parallel.

4.2 Alternative zlib Implementations

Being build into the Linux kernel, zlib is seen as the de facto standard of file compressing using the DEFLATE algorithm. The first version of zlib was published in 1995 to be used in the PNG handling library *libpng*. Although still maintained, other libraries have been created that surpass zlib in both compression speed and ratio.
For Example, *libdeflate*[**?**] offers fester compression while at the same time archiving a better compression ratio. Libdeflate archives this through various improvements such as using word access instead of byte access in input reading and match copying, which is a part of the DEFLATE algorithm. Furthermore, it uses a speed-up Huffman decoding process, loads the whole block into a buffer before compressing and utilizes BMI2 instructions on x86_64 machines if supported.
As the developers of SAMtools are aware of the advantages of libdeflate against zlib, support for libdeflate is already built into SAMtools. Moreover, the usage is recommended and if libdeflate libraries are found, they are automatically used instead of zlib. To decide manually between using zlib and libdeflate, the HTSlib `configure` script can be run with the `--with-libdeflate` resp. `--without-libdeflate` option.
In addition, the user can choose to use other zlib implementations by using `LD_PRELOAD` [**?**]. `LD_PRELOAD` is an environment variable telling the loader to load shared libraries. A shared library is a code object which is not part of another program but can be used by multiple programs. Functions and symbols from the shared library are connected to another program by the linker. If two different definitions for symbols exist, the linker prefers the one from a shared library in `LD_PRELOAD`. For example, per default, HTSlib uses the `deflate` method of `libz.so`. However, the user can compile e.g. *zlib-ng* [**?**], 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, the deflate implementations of zlib-ng are used instead of the implementations of zlib. However, this approach is only

possible, if the replacement implementation supports the zlib API. Therefore, other libraries, which also produce gzip compatible output but benefit from an adjusted API, can not be used for this approach.

4.3 7BGZF

7BGZF [**?**] is a tool developed by Taiju Yamada for testing different GZIP compatible compression libraries. It works by overwriting `bgzf_compress`, the method HTSlib uses for compressing. Then it chooses the library to use for compression based on the `BGZF_METHOD` environment variable which the user can set before. This approch has the advantage, that it simplyfies testing different compression libraries. To test 9 libraries, the user only has to do one installation as 7BGZF's only dependency is `libc`. On the other hand, using 7BGZF has some disadvantages. As the method 7BGZF overwrites is a method of HTSlib, SAMtools has to link to a shared library rather to linking to the static library. However, to archive this the user only has to change a single line in SAMtools' makefile and change `@Hsource@HTSLIB = $(HTSDIR)/libhts.a` to refer to `libhts.so`. Another disadvantage is, that 7BGZF does not distinguish between different compression settings which are a parameter of the overwritten `bgzf_compress` method. Instead, 7BGZF receives the compression level to be used as part of the `BGZF_METHOD` environment variable. Therefore, it applies the same compression level on every written BGZF compressed file. In the context of SAMtools `sort`, this means temporary files have the same compression level as output files. This leads to more time consumption if the output file should use a compression level which is not level 1, possibly changing the results of benchmarks.
Testing 7BGZF on sorting a BAM file small enough not to produce any temporary files, still gives hints on which libraries to use for faster sorting. Results of such an experiment are shown in Figure 8 and 9.

Compression Levels are the gzip way of trading computation time against space requirements. They can be set using the "`-l`" parameter in SAMtools sort. Incidentally, this is also possible for other SAMtools commands 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`).
Here it comes down to what the purpose of the sorted data is when deciding the level to be used. If the file should be archived, setting the level to 9 for maximal compression is an option. However, most of the time, sorting is a step in a larger pipeline, and the data is read and processed further soon.
To speed this up, it is recommended to use compression level 0 ("`-l 0`" which is equal to "`-u`") if the data is not written directly to disc or transferred over network with limited throughput.
In the other case, it is recommended to use compression level 1 ("`-l 1`").

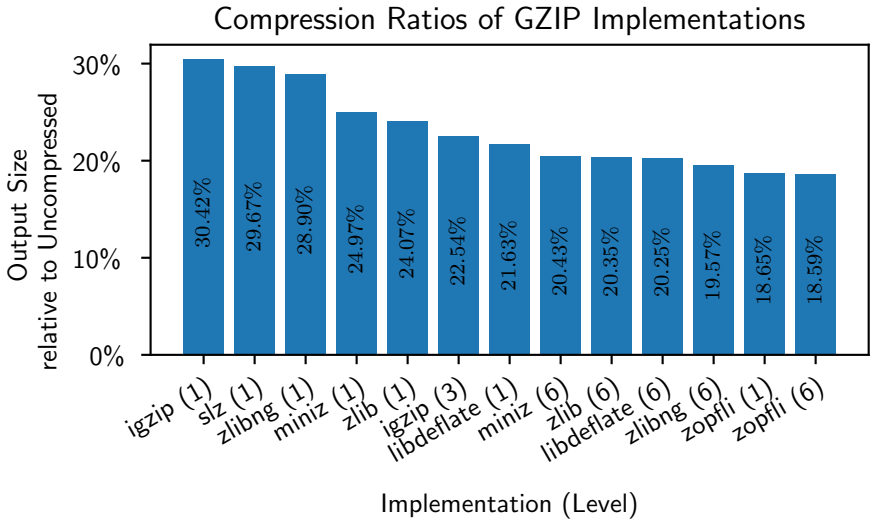

Fig. 8. Comparison of the compression ratio of different zlib implementations. Sizes are relative to the uncompressed file.

4.4 Recomendation

he most obvious way to increase compression speed is to increase the number of available threads. 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.

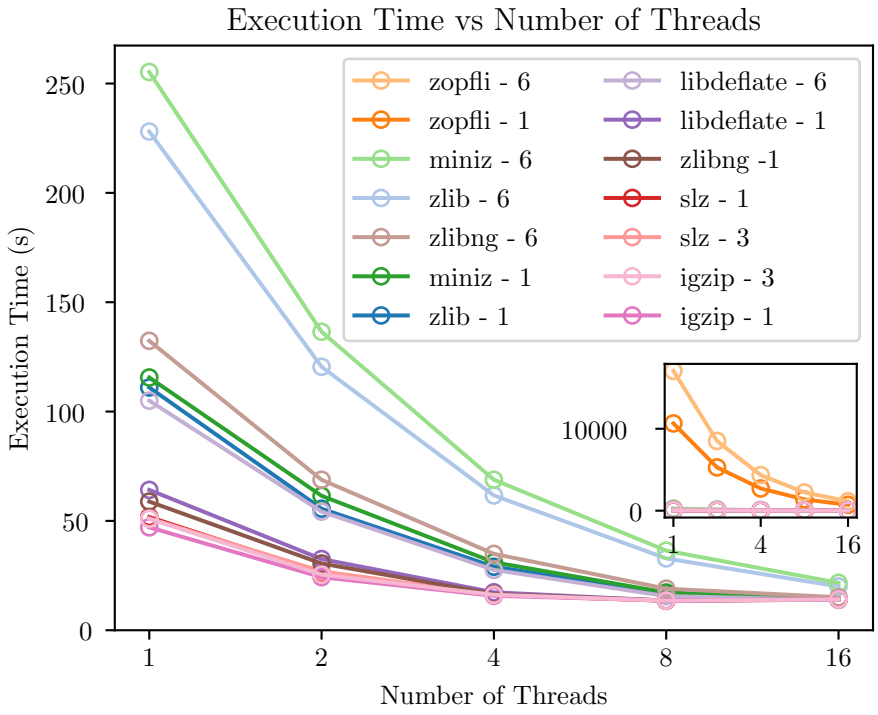

Fig. 9. Execution time of SAMtools `sort` using 7BGZF to test different compression libraries. Libraries are sorted after their execution time on a single thread. The compression level is the number abter the library. The inset figure also shows the performance of zopfli on level 1 and 6.

5 Input/Output

Input and Output can also be constraints of the sorting process. As the internal mechanisms of SAMtools usually work very fast and are highly parallel, but have to process huge amounts of data, input and output devices can also limit the computation speed.

5.1 IO

In contrast to the used compression library, often the IO devices can hardly be changed. However, in some cases there are possibilities to speed up the process.

Pipelining enables all advantages of working with streams. For SAMtools `sort`, this has the following consequences: Records can be read as they are produced by the previous command. This can be done without compression and without

having to write files to the disk using UNIX pipelines. Note that the process generating the input for SAMtools `sort` is most likely halted once the memory limit of `sort` is reached. This is due to the buffer offered by the pipeline being full and only emptied again, after a temporary files is written. Looking at the output of sort, there is also no need for compression and writing to disk if piped.

Prefixes for temporary files can be set via the "`-T`" parameter. If no prefix is specified, temporary files are written into the same directory as the output. If the output is to standard output, they are placed in the current working directory. Here, a directory on a fast disk should be chosen. As temporary files are deleted automatically after successful sorting, only at the time of sorting the disks capacities are used. However, it is important to remember, that temporary files are less compressed than regular input and output files. In combination, they require about 20% more disk space, than the input file. Moreover, if the hard drive is very fast and additionally offers enough storage space, the compression of the intermediate files can be omitted. Unfortunately, this is not possible without changing the source code at the moment. This can be done by simply 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.

6 Approaches

6.1 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 record on the disk together with the attributes needed for sorting, i.e. the reference ID, the position and the REVERSE flag.
2. Sort the resulting list. As the attributes extracted are much smaller than whole records, this should be possible in memory.
3. Iterate over the sorted list. For every entry, read the BAM record placed at the location the pointer points to 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 significant portion of the time needed for sorting, it has some drawbacks:
BAM files are binary compressed representations of SAM files. While the compression usually is beneficial to store and transfer the huge amounts of data a

SAM 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. Especially with the used compression and decompression algorithms, DEFLATE and INFLATE both being streams, every random read would require decompressing the compressed file from the beginning. To solve this, BAM files are compressed using BGZF. As only small blocks are compressed by DEFLATE, for a random read only the number of the block the read is in, together with an offset into the compressed block is needed.
However, this method is not suitable for accessing every single record in a file in random order:
Blocks typically have sizes of 64KB of uncompressed data. Within our main test file, BAM records had on average a size of about 250 bytes. Therefore, a 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. To make things worse, if the input file is very large in comparison to the available memory, caching the uncompressed blocks gets less effective.
If no merging of temporary files has to be performed, we can now calculate the deflate and inflate operations per BAM record at the current state of SAMtools sort compared to this approach: Currently, the Input file is decompressed once accounting for one INFLATE call for every 265 BAM records. Then, the record is written to the temporary file, resulting in one DEFLATE call for every 256 records. After some time, the temporary file is read again (one INFLATE call per 256 records) and the output file is written (again, one DEFLATE call per 256 records). As the input and the output decompression and compression are necessary for both approaches, they can be ignored.
The approach using random reads, however, does not use any DEFLATE call in between input and output, but for every BAM record on average one execution INFLATE on half of a block, accumulating to around 128 INFLATE calls on whole blocks per 256 BAM records. Therefore, to speed up the operation, a single DEFLATE execution (together with writing) would have to be 127 times slower than the combination of reading and INFLATE. As this seems unlikely on most systems, no improvement is expected from this approach.
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.

6.2 Adjust Compression of Temporary Files

The first measurements showed, that the biggest part of computation time is used for compression even if the compression of the output format is removed. Based on this observation it seemed obvious, that removing the compression of the temporary files would speed up the process. More experiments on another computer seemed to confirm this hypothesis. However, in final experiments above a threshold of used threads, removing the compression of files turned out to be slower than keeping it on a low level. Where does this change of directions come from?

If more or less compression is faster depends on the proportion between compression speed and write speed. The compression speed is mainly influenced by the gzip compression level, the number of available cores, their speed and the implementation of the DEFLATE algorithm. As the output consists of many independly compressed blocks, this can be done in parallel. Thus, increasing the number of threads significantly increases the compression speed if enough physical cores are available. Lowering the compression level also results in a speedup, but as the compression level of temporary files is already 1 the only way to reduce it further would be to disable compression (level 0). Of course, changing to a faster compression library also increases the compression speed.
If the write speed the disk offers is less than the compression output, there is an IO bottleneck. To detect this, we can have a look on *IOWait* time. IOWait time is the amount of time, the CPU spends in IDLE because it has to wait for IO Operations. This is measured for all cores together.