Performance Analysis and Optimization of SAMtools Sorting

Nathan T. Weeks[1,2(✉)] and Glenn R. Luecke[2]

[1] Department of Computer Science, Iowa State University, Ames, USA
[2] Department of Mathematics, Iowa State University, Ames, USA
weeks@iastate.edu

Abstract. SAMtools is a suite of tools that is widely-used in genomics workflows for post-processing sequence alignment data from large high-throughput sequencing data sets. A common use of SAMtools is to sort the standard Binary Alignment/Map (BAM) format emitted by many sequence aligners. This can be computationally- and I/O-intensive: BAM files can be many gigabytes in size, and may need to be decompressed before sorting and compressed afterwards. As a result, BAM-file sorting can be a bottleneck in genomics workflows. This paper presents a case study on the performance characterization and optimization of BAM sorting with SAMtools. OpenMP task parallelism to enhance concurrency and memory optimization techniques were employed in both SAMtools and the underlying library HTSlib. Utilizing all 32 processor cores on the benchmark system, the optimizations resulted in a speedup of 3.92X for an in-memory sort of 24.6 GiB of BAM data (102.6 GiB uncompressed), while a 1.55X speedup was achieved for an out-of-core sort.

Keywords: Bioinformatics · High-throughput sequencing · OpenMP

1 Introduction

The rapid decline in DNA sequencing costs has outpaced the growth in transistor density from Moore's Law since 2007 [9]. The resulting increase in genomics data generation is predicted to potentially dwarf Twitter, YouTube, and astrophysics data combined by the year 2025 [7]. Furthermore, application performance has generally not even kept pace with Moore's Law, as many applications are ill-equipped to express the parallelism needed to utilize the extra performance potential. As a result, storing, processing, and analyzing genomics data have already become problematic for many institutions. Improvements in algorithms, computing hardware, and storage technology are needed to prevent this trend from worsening.

Most genomics data being generated is high-throughput sequencing (HTS) comprising numerous, relatively-short sequencing reads. HTS is commonly aligned against a reference sequence, typically resulting in sequence alignment data in the standard text-based Sequence Alignment/Map (SAM) format; its

F. Desprez et al. (Eds.): Euro-Par 2016 Workshops, LNCS 10104, pp. 409–420, 2017.
DOI: 10.1007/978-3-319-58943-5_33

binary analog, the Binary Alignment/Map (BAM) format; or the relatively-recent CRAM format. These formats can be consumed by a number of bioinformatic tools for downstream analysis.

SAMtools is a utility for working with sequence alignment data in the SAM, BAM, and CRAM formats [5]. SAMtools supports operations such as sorting, merging sorted files, indexing, selecting subsets of records, compressing, and reporting various statistics. SAMtools makes use of the code-veloped HTSlib library for reading, parsing, and compressing/decompressing SAM/BAM/CRAM data. As SAMtools and HTSlib are developed in lockstep with the SAM/BAM/CRAM specifications, they are considered to be the reference implementations among software tools/libraries that work with these formats.

While SAMtools is partially parallelized using the pthreads API, this implementation is inefficient in some cases (as illustrated by performance profiling of SAMtools 1.3 in Sect. 3 compared with in performance profiling post-optimization in Sect. 4), and there are other parts of the code that could benefit from multithreading. Performance optimization of this foundational tool could benefit many genomics workflows and many users.

The rest of this paper is organized as follows. Section 2 describes other attempts to improve the performance of SAMtools, as well as other software that exists with the explicit goal of implementing performance-critical SAMtools functionality more efficiently. Section 3 characterizes the performance of a SAMtools sort workflow, and identifies performance bottlenecks. Section 4 describes the various categories of optimizations implemented in this work to address the performance bottlenecks identified in Sect. 3. The impact of the performance optimizations on a benchmark data set is analyzed in Sect. 5. Section 6 lists additional opportunities for performance optimization. Section 7 discusses the significance of this work in the context of other work that has been done to address the performance limitations of SAMtools.

2 Related Work

SAMtools uses the HTSlib library for SAM/BAM/CRAM I/O. HTSlib supports multi-threaded reading/writing of CRAM data using a custom pthreads-based thread pool (originally adapted from the Scramble [2] I/O library). Though HTSlib supports multi-threaded BAM compression/output, it supports only single-threaded BAM decompression/input. While it should be feasible to extend HTSlib to accommodate concurrent compression/output of BAM data using the method implemented by Scramble, this paper presents an alternative approach using OpenMP, a high-level API for shared-memory parallel programming that is widely supported by most C, C++, and Fortran compilers.

Other software projects exist with the explicit goal of providing better performance over SAMtools for performance-critical tasks. Sambamba [8] is intended to be a high-performance replacement for a subset of the SAMtools functionality

(including BAM sorting, the focus of this paper). Written in the D programming language with parallelism as explicit design goal, Sambamba aims to exploit multi-core CPUs better than SAMtools.

elPrep [4] is a multi-threaded Lisp application that focuses on high-performance, in-memory execution of a subset of SAMtools functionality useful for preparing SAM/BAM/CRAM data for variant calling. elPrep has large memory requirements for sorting, however: the elPrep 2.4 documentation states "As a rule of thumb, elPrep requires 6x times more RAM memory than the size of the input file in .sam format when it is used for sorting". In contrast, the BAM data set described in Sect. 3, which is ∼85.5 GiB when converted to SAM, is sorted in-memory by SAMtools on the 128 GiB-memory compute node. elPrep uses SAMtools internally for reading and writing BAM files, and so could benefit from the decompression and compression optimizations to SAMtools described herein.

DNANexus has submitted a patch to SAMtools that improves concurrency in the BGZF compression/writing code[1] (this paper describes a different method in Sect. 4), as well a fork that leverages RocksDB for improved sorting/merging performance[2]. Neither of these contributions have been accepted into the SAMtools code base.

Intel- and CloudFlare-optimized versions of the zlib compression library have been shown to improve compression performance in SAMtools[3].

3 Performance Profiling

A single compute node of the NERSC Cori (phase 1/Data Partition) supercomputer was used for performance profiling and benchmarking. Each compute node contains two 16-core 2.3 GHz Intel "Haswell" Xeon processors (with each processor core supporting two hardware threads), and 128 GiB 2133 MHz DDR4 memory. While there is no local storage on a compute node, a high-speed Aries interconnection network provides a fast path to a Lustre parallel file system capable of >700 GB/s aggregate bandwidth. Using the `dd` command with a 4 MiB block size (optimal per `st_blksize` from the `stat()` system call) to test single-threaded read performance on the data set described below resulted in a throughput of over 500 MiB/s.

SAMtools 1.3 and HTSlib 1.3 were compiled with the gcc 5.2.0 compiler[4] using the default compiler options specified in the SAMtools/HTSlib makefile, with the exception of using the Cray compiler driver to invoke gcc, overridden to use dynamic linking (`cc -dynamic`).

[1] https://github.com/samtools/htslib/pull/51.
[2] http://devblog.dnanexus.com/faster-bam-sorting-with-samtools-and-rocksdb/.
[3] http://www.htslib.org/benchmarks/zlib.html.
[4] The Intel 16.0.2 compiler was initially used, but due to a potential compiler bug affecting subsequently-implemented OpenMP-based optimizations, the gcc compiler was used for the benchmarking described in this paper.

SAMtools was benchmarked with the 19.9 GiB (102.6 GiB uncompressed) BAM for individual HG00109 from the 1000 Genomes Project [3]. As this BAM file was already sorted by position, it was first sorted by query name to "shuffle" it. The resulting BAM had a considerably worse compression ratio: the file size increased to 24.6 GiB.

SAMtools was run with 32 threads under two scenarios:

1. Out-of-core: the default memory per thread (768MiB) results in sublists sorted in-memory, then written to disk for an out-of-core merge at the end.
2. In-memory: using 3664MiB per thread is enough to store and sort the entire uncompressed BAM data set in memory.

HPCToolkit 5.4.2 [1] was used to generate a profile trace for the in-memory sort (Fig. 1). This profile revealed that program execution (∼16 min) is divided into four main phases, in which (1) the compressed BAM data is read and decompressed (∼35% run time), (2) the BAM records are sorted by a single thread (∼24% run time), (3) the sorted BAM is compressed and written to disk (∼25% run time), and (4) data structures allocated during program execution are deallocated (∼16% run time). Only the compress/write phase is performed by multiple threads.

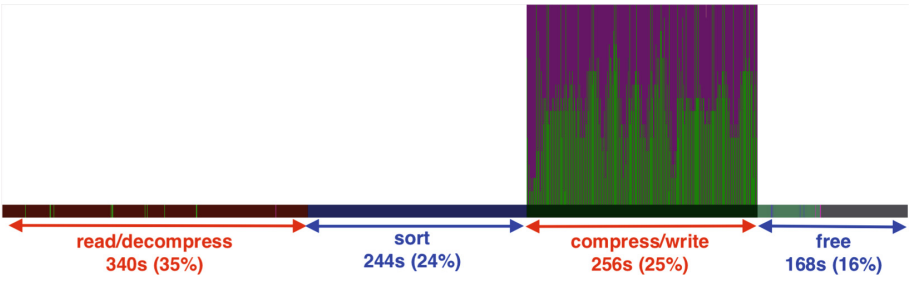

Fig. 1. HPCToolkit performance summary of SAMtools 1.3 for a sort (16 threads) of HG00109 BAM alignment data (102.6 GiB uncompressed). This represents the number of threads that are in a particular procedure at a given time. The white area indicates that only one thread is active in three of the four phases. In the compress/write phase, the magenta area represents the compression routine (`bgzf_compress()`), while the green area phase represents idle threads waiting on a condition variable (`pthread_cond_wait()`). (Color figure online)

The out-of-core sort cycles between phases 1–3 until all input has been processed. Unlike the in-memory sort, the sort phase for the out-of-core sort is multi-threaded, where each thread sorts a separate sublist of BAM records and writes its sublist to disk. Finally, the on-disk sorted BAM files are merged and written to a single sorted BAM file.

The run time of the out-of-core sort (∼15 min) is actually less than the run time of the in-memory sort. This is due to reductions in run time from added

parallelism in the sort phase (∼6% of the run time is spent sorting sublists and writing to temporary files), and in the time spent freeing dynamically-allocated memory associated with BAM data (<3%). The additional I/O overhead from writing the sorted sublists to disk and reading them in again during the final merge did not outweigh the benefit of the parallel sort, at least in part because of the high-bandwidth of the underlying Lustre parallel file system. A conventional (non-parallel) file system might not handle I/O from multiple data streams so effectively.

As a result of performance profiling, a goal was formulated to optimize the performance of each of the four phases of the in-memory sort, with the recognition that many of the optimizations would also benefit the out-of-core sort. These optimizations are described in the next section.

4 Optimizations

Read/Decompress. The HTSlib routines for reading BGZF-compressed BAM data are sequential. While reading an input stream is inherently sequential, in this case the input stream consists of compressed BGZF[5] blocks that can be decompressed independently.

Previously there was an effort to parallelize the decompression of the input BGZF blocks[6]. In this approach, the master thread reads the input data stream, assigning a fixed number of BGZF blocks to each worker thread, while the worker threads wait on a condition variable. When all input buffers have been filled, the master thread executes `pthread_cond_broadcast()` to start the worker threads. Each worker thread inflates each assigned compressed BGZF block into a temporary buffer before copying it into the origin buffer, overwriting the compressed BGZF block. When all worker threads are complete, the master thread adds each buffer pointer to a hash-based cache, allocates new buffers, and repeats the process on new input data. A drawback of this approach is that it limits concurrency: worker threads are idle while the master thread reads input data into the workers' input buffers.

Using the previous effort as a guide for safely adding concurrency to the relevant HTSlib routines, the code was modified so that the master thread generates OpenMP tasks to decompress BGZF blocks as they are read. The next consideration was how many BGZF blocks should be decompressed by each task. A finer level of granularity (i.e., fewer BGZF blocks per task) would improve load balancing, while a coarser level of granularity (coalescing more adjacent BGZF blocks into a single payload for each task) would reduce synchronization overhead. A microbenchmark was constructed to approximate *task creation overhead* from the time elapsed between the last statement before and the first statement

[5] Each BGZF block is effectively a gzip file of size ≤64K (compressed or uncompressed), with a user-defined field in the gzip header used to represent the length of the BGZF block. The BGZF file format is described in more detail in the SAM/BAM specification.

[6] https://github.com/smowton/htslib/compare/parallel_read.

inside the OpenMP task construct. With 505,455 tasks (the number of BGZF blocks in the HG00109 BAM file) and a 64 KiB OpenMP `firstprivate` payload (the maximum size of a BGZF block) per task, the aggregate task creation overhead for all 32 threads less than 4 s—an average of less than $\frac{1}{8}$ of a second per thread. This indicated that the fine-grained approach of one BGZF block per task would facilitate load balancing without introducing significant synchronization overhead. To eliminate a `memcpy()` for each BGZF block, a new target buffer is dynamically allocated, and a pointer to this buffer is cached.

Memory Allocation. Profiling revealed that a significant fraction (∼4%) of the run time for the in-memory SAMtools 1.3 sort was due to a single line of code that allocated an array to store the variable-length data for each BAM record. In the benchmark data set, this meant almost 207 million calls to `realloc()` (each of which was subsequently paired with a corresponding `free()` upon data structure deallocation).

The original data structure used to represent a single BAM record is listed in Fig. 2.

```
typedef struct {
    bam1_core_t core;
    int l_data, m_data;
    uint8_t *data;
    uint64 t id;
} bam1_t;

bam1_t **buf;
```

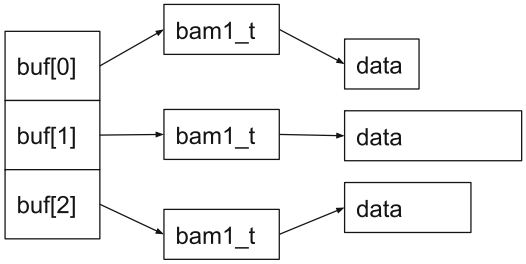

Fig. 2. The original SAMtools represented each BAM record as a dynamically-allocated bam1_t struct, each containing a dynamically-allocated `data` member. The length of each BAM record is stored in the `l_data` member. An array of pointers (`buf[]`) to the BAM records is sorted during the BAM sort.

This memory allocation overhead was addressed by allocating a single, contiguous array of approximately the maximum memory size requested by the user, as well as an ancillary array of pointers of type `bam1_t` into this array to indicate the start of each BAM record. Each subsequent BAM record starts at the address of the previous BAM record + `l_data`, rounded up to the nearest 8-byte boundary to ensure proper memory alignment of all structure members.

To accomplish the partitioning of the contiguous memory region into an array of variable-length structs, a flexible array member (`fam[]`) was added to `bam1_t` (Fig. 3). For backwards compatibility with other HTSlib code, the `data` member was retained, pointing to `fam[]`. Reworking the HTSlib code to remove the dependence on the `data` member could save 8 bytes per `bam_t` BAM record (the flexible array member is not a pointer, and consumes no storage beyond the data contained in the array).

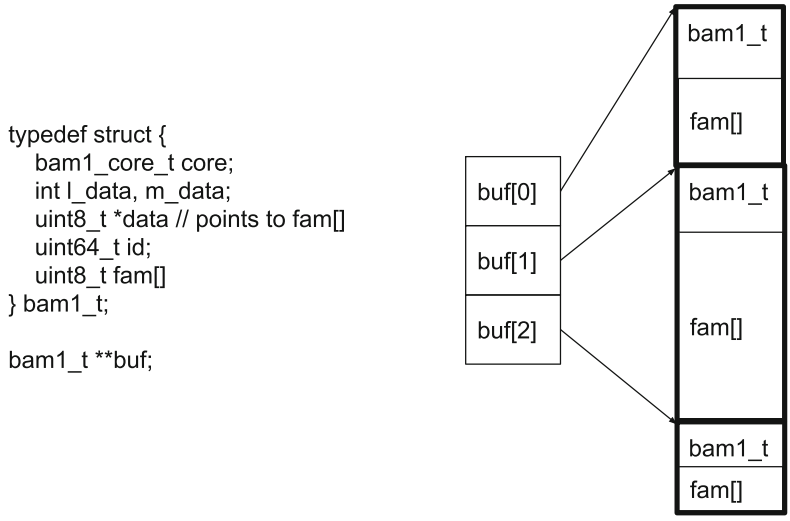

Fig. 3. The use of a flexible array member (`fam[]`) for variable-length data allows BAM records to be stored consecutively in a contiguous memory region. The array of pointers (`buf[]`) points to the beginning of each (8-byte aligned) `bam1_t` record in the array. For backwards compatibility, the `data` member points to `fam`.

Sort. In SAMtools 1.3, if the input BAM records do not fit within the user-specified memory limit, then the BAM records in memory are partitioned into N sublists, where N is the number of threads. Each thread sorts its sublist, then writes it to a separate temporary BAM file. After all input data has been processed in this manner, the master thread merges the sorted temporary BAM files to produce a single sorted output stream.

A shortcoming with the SAMtools 1.3 implementation is that if the BAM input fits within the user-specified memory limit, then the sort will be performed by only a single thread. To address this issue, the merge sort was implemented on the entire in-memory array of BAM records, which dramatically reduced the amount of time spent in the sort phase. However, it was recognized much of the sort phase could be overlapped with the read/decompression phase to make use of the spare computational capacity. This idea was implemented by creating a sort task after reading/decompressing every 2^{20} BAM records (an empirically-chosen value). For the benchmark data set, this approach resulted in more sublists to merge (in-memory), but the result was still faster than the previous approach.

An additional I/O optimization was implemented for the out-of-core sort. Instead of writing each in-memory sorted sublist to a separate file, the sorted sublists are merged while writing to a single file. This reduces both the number of temporary BAM files that must be written and subsequently merged, potentially improving performance on storage systems that benefit from fewer, larger I/O streams.

Compress/Write. SAMtools 1.3 supports multi-threaded compression of output BAM records. However, as with the experimental pthreads read/decompress code, a condition variable is used in a manner that limits concurrency. The master thread fills per-worker-thread input buffers with "work" (by default, 256 up-to-64KiB blocks of uncompressed BAM records) while all worker threads are blocked on a condition variable. Once the buffer for each worker has been filled, the master thread then issues a `pthread_cond_broadcast()` to unblock the worker threads. Each worker thread compresses a block of BAM records into a temporary buffer, then copies it back to the original buffer, overwriting the uncompressed block. After all worker threads are done, the master thread outputs all compressed BGZF blocks (during which time worker threads are idle), and repeats the process with the next subset of the sorted BAM data.

To increase concurrency (and thus CPU utilization) during this process, the limited-concurrency pthreads code was refactored to use OpenMP tasks, with each task both compressing a contiguous list of 256 up-to-64KiB blocks of BAM records, as well as writing the compressed BGZF blocks in input order (see Listing 1.1). To reduce latency, after compressing its blocks, the thread executing the task spins until its turn to write the output (effectively implementing a ticket lock [6]). Because OpenMP `atomic` directives are effectively used for synchronization, an OpenMP `flush` directive must be used before and after the routine that writes the compressed BGZF blocks to ensure memory consistency between threads of any referenced shared data structures. Alternatively, compilers supporting OpenMP 4.0 or newer could specify the `seq_cst` clause to the `atomic` directive, which makes the atomic construct sequentially consistent (implying the `flush`).

Listing 1.1. Conceptual routine invoked by the master thread to concurrently BGZF compress ≤64 KiB blocks of BAM records and serialize output in input order

```
void compress_and_output(BAM *master_thread) {
  char blocks[SIZEOF_BLOCK*NUM_BLOCKS_PER_TASK];
  static uint64_t now_serving_shared = 0;
  memcpy(blocks, master_thread->blocks, sizeof(blocks));
  uint64_t my_ticket = master_thread->ticket++;
  #pragma omp task firstprivate(blocks, my_ticket)
  { uint64_t now_serving_private;
    compress(blocks); // concurrent with other tasks
    do { // wait until this task's turn to output
      #pragma omp atomic read
      now_serving_private = now_serving_shared;
    } while (now_serving_private != my_ticket);
    #pragma omp flush
    output(blocks); // serialized
    #pragma omp flush
    #pragma omp atomic update
    now_serving_shared++; // let the next task output
  } }
```

The task generation is continuous until input (uncompressed blocks) has been exhausted.

Memory Deallocation. Approximately 16% of the 32-thread in-memory SAMtools 1.3 sort runtime was spent deallocating over 100 GiB of dynamically-allocated memory comprising approximately 207 million BAM records. This required two calls to `free()` for each BAM record: one for the (fixed-size) `bam1_t` data structure, and one for the variable-length `data` member. As this was done at the end of execution, an initial workaround to avoid this excessive memory deallocation overhead was to not explicitly free the memory, instead allowing the operating system to reclaim allocated memory upon process termination. However, storing the BAM records in a single contiguous memory region (allocated with a single `malloc()`, and deallocated with a single `free()`) as described in the *Memory Allocation* subsection obviated the need for this workaround.

5 Benchmark Results

The benchmark hardware/software environment and data set are described in Sect. 3. Minor optimizations to avoid the overhead of dynamic memory allocation with multiple threads were implemented in several places using automatic variables on the stack. To accommodate the extra per-thread stack usage, the `OMP_STACKSIZE` environment variable was set to 64M. Approximately 112 GiB total, divided by the number of threads, was specified for the per-thread memory argument to the `samtools sort` command to allow the in-memory sort.

Both the original SAMtools 1.3 and SAMtools with the optimizations described in Sect. 4 were used to perform `samtools sort` on the benchmark data set with 1, 2, 4, 8, 16, 32, and 64 threads. The 64-thread run utilized both hardware threads in each core (HyperThreading).

Single-threaded performance was similar for both SAMtools 1.3 and the optimized SAMtools. Moving to two threads activated different code paths in each code base, and resulted in a performance regression in the optimized SAMtools. The reason for this regression may be due to a combination of overhead associated with creating OpenMP tasks, and lack of work stealing in the GNU OpenMP runtime, leading to load imbalance with one dedicated task "producer" and one dedicated "consumer". With $\geq$4 threads, the optimized SAMtools performed between 29% and 73% faster than the SAMtools 1.3. The performance of the optimized SAMtools was slightly slower with HyperThreading (64 threads, or 2 threads per core) than without (32 threads), whereas the performance of the original SAMtools was slightly better with HyperThreading than without.

For the in-memory sort, the optimized SAMtools saw a modest single-threaded performance boost (7%) over the SAMtools 1.3, likely due to the memory optimizations described in Sect. 4. As with the out-of-core sort, the performance of the optimized SAMtools with 2 threads was worse than SAMtools 1.3. With more than 2 threads, the optimized SAMtools demonstrated significant speedups: 1.58X at 4 threads, 2.41X at 8 threads, 3.6X at 16 threads,

3.92X at 32 threads, and 3.49X at 64 threads. Performance at 32 threads was not substantially better than performance at 16 threads, as the extra computational capacity was mostly idle (Fig. 5). Profiling with 32 threads (Fig. 5) revealed that code for performing the N-way merge of sorted sublists using a heap data structure became a bottleneck (∼15% of total thread 0 run time). Thus, thread 0 could not generate sorted blocks for the merge/compress/write tasks fast enough to keep the remaining threads busy.

The profiled times for the in-memory sort (Figs. 4 and 5) exclude 30–35 s after the call to `exit()`. The timings in Fig. 6 time the SLURM `srun` command, and thus include this time not counted by HPCToolkit. Subsequent testing with a microbenchmark that allocated a large amount of memory indicated that the extra time was likely due to operating system overhead (e.g., freeing large page tables). Specifying huge pages (via the `cray-hugepages2M` environment module) at compile time reduced the time to process termination on the microbenchmark; however, a `libhugetlbfs` error occurred at run time when this was attempted with `samtools` on the HG00109 data set.

Interestingly, there wasn't a substantial performance difference between the in-memory and out-of-core sorts for SAMtools 1.3 with ≥8 threads, indicating that the added parallelism in the sort phase in the out-of-core version compensated for the extra I/O. As mentioned previously, this is due to the reduction in

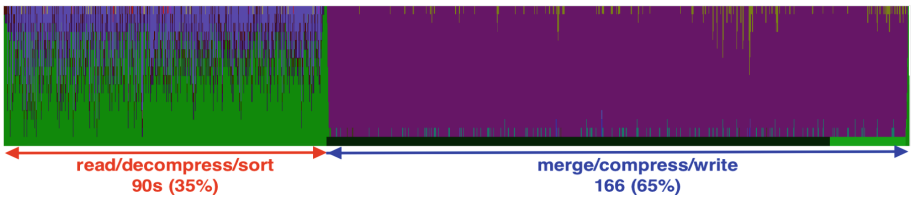

Fig. 4. HPCToolkit performance summary of the optimized SAMtools for the in-memory sort (16 threads). The green area represents overall time among all threads spent "idle" (waiting in `gomp_barrier_wait_end()`). In the merge/compress/write phase, the magenta area indicates that most of the time is spent in the compression routine. (Color figure online)

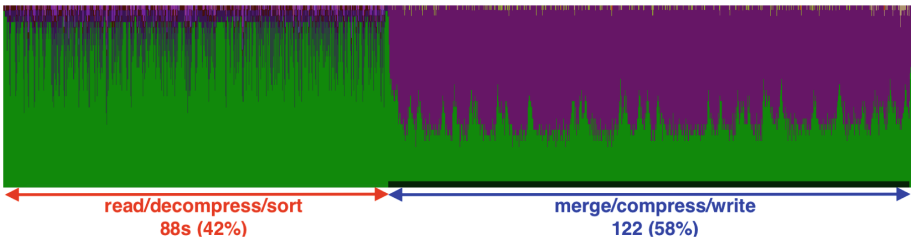

Fig. 5. HPCToolkit performance summary of the optimized SAMtools for the in-memory sort with 32 threads. Unlike with 16 threads, there is noticeable thread idle time (green area) in the merge/compress/write phase. (Color figure online)

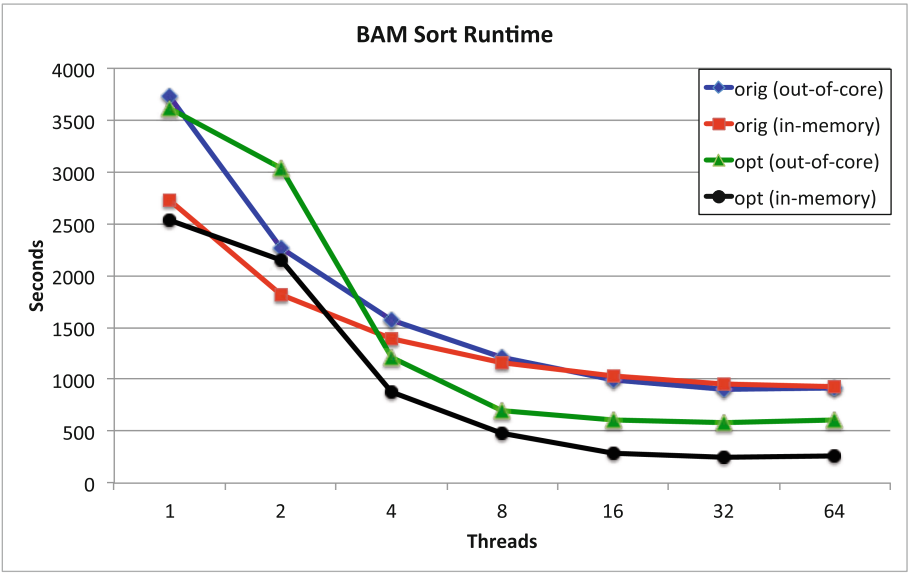

Fig. 6. Runtimes in seconds of SAMtools sort on HG00109 BAM for SAMtools 1.3 (labeled "orig") and the optimized SAMtools (labeled "opt"). The average of 3 timings for each thread count is reported.

time spent deallocating data structures and added parallelism in the sort phase, with the cost of extra I/O to temporary files offset by the large amount of I/O bandwidth afforded by the Lustre parallel file system.

6 Future Work

A dedicated read-ahead thread could improve concurrency and allow greater processor utilization while reading/decompressing sequentially-read BAM files. Similarly, a dedicated writer thread with a work queue could reduce busy-waiting and thus improve concurrency while compressing/writing BAM data.

The out-of-core sort implementation requires all sorted BAM sublists to be written to disk before the final merge. A more efficient approach would be to allow as much data as possible to remain in memory.

7 Conclusions

As an important component in many HTS pipelines, SAMtools processes large amounts of HTS data every day worldwide. Therefore, improvements to this fundamental tool have the potential to positively affect a vast audience. In particular, performance improvements collectively reduce time to solution for many

scientific workflows in diverse life sciences fields such as agriculture, oncology, pathology, and pharmacology.

This work significantly enhanced the performance of SAMtools for sorting BAM data, both in-memory (3.92X speedup 32-threads) and out-of-core (1.55X speedup with 32 threads). This may obviate the need to use alternative tools that are more performant than the original SAMtools for this task. Although not analyzed in this paper, many of the implemented performance improvements should benefit other SAMtools functionality (including in-memory CRAM sorting) and applications that utilize HTSlib (e.g., bgzip).

Leveraging OpenMP's high-level task parallelism proved to have a number of advantages over the existing pthreads-based implementation. Besides offering additional opportunities for concurrency, the use of OpenMP enhanced code conciseness and clarity, which should facilitate maintainability.

Acknowledgment. The authors thank Marina Kraeva for her careful proofreading of the manuscript.

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCToolkit: tools for performance analysis of optimized parallel programs. Concurr. Comput.: Pract. Exp. **22**(6), 685–701 (2010). http://dx.doi.org/10.1002/cpe.1553
2. Bonfield, J.K.: The scramble conversion tool. Bioinformatics **30**(19), 2818–2819 (2014). http://bioinformatics.oxfordjournals.org/content/30/19/2818.abstract
3. 1000 Genomes Project Consortium, et al.: A global reference for human genetic variation. Nature **526**(7571), 68–74 (2015)
4. Herzeel, C., Costanza, P., Decap, D., Fostier, J., Reumers, J.: elPrep: high-performance preparation of sequence alignment/map files for variant calling. PLoS ONE **10**(7), 1–16 (2015). http://dx.doi.org/10.1371/journal.pone.0132868
5. Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., Subgroup, G.: The sequence alignment/map format and SAMtools. Bioinformatics **25**(16), 2078–2079 (2009). http://bioinformatics.oxfordjournals.org/content/25/16/2078.abstract
6. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. **9**(1), 21–65 (1991). http://doi.acm.org/10.1145/103727.103729
7. Stephens, Z.D., Lee, S.Y., Faghri, F., Campbell, R.H., Zhai, C., Efron, M.J., Iyer, R., Schatz, M.C., Sinha, S., Robinson, G.E.: Big data: astronomical or genomical? PLoS Biol. **13**(7), 1–11 (2015). http://dx.doi.org/10.1371/journal.pbio.1002195
8. Tarasov, A., Vilella, A.J., Cuppen, E., Nijman, I.J., Prins, P.: Sambamba: fast processing of NGS alignment formats. Bioinformatics **31**(12), 2032–2034 (2015). http://bioinformatics.oxfordjournals.org/content/31/12/2032.abstract
9. Wetterstrand, K.: DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP). http://www.genome.gov/sequencingcostsdata