4-26-2017

Optimization of SAMtools sorting using OpenMP tasks

Nathan T. Weeks
Iowa State University, weeks@iastate.edu

Glenn R. Luecke
Iowa State University, grl@iastate.edu

Optimization of SAMtools Sorting Using OpenMP Tasks

Nathan T. Weeks · Glenn R. Luecke

Published 28 May 2017.
The final publication is available at Springer via `https://dx.doi.org/10.1007/s10586-017-0874-8`

Abstract SAMtools is a widely-used genomics application for post-processing high-throughput sequence alignment data. Such sequence alignment data are commonly sorted to make downstream analysis more efficient. However, this sorting process itself can be computationally- and I/O-intensive: high-throughput sequence alignment files in the de facto standard Binary Alignment/Map (BAM) format 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 describes a case study on the performance analysis and optimization of SAMtools for sorting large BAM files. OpenMP task parallelism and memory optimization techniques resulted in a speedup of 5.9X versus the upstream SAMtools 1.3.1 for an internal (in-memory) sort of 24.6 GiB of compressed BAM data (102.6 GiB uncompressed) with 32 processor cores, while a 1.98X speedup was achieved for an external (out-of-core) sort of a 271.4 GiB BAM file.

Keywords Bioinformatics, High-Throughput Sequencing, OpenMP, Sorting, Burst Buffer

N. Weeks
Department of Computer Science, Iowa State University, Ames, IA, USA
E-mail: weeks@iastate.edu

G. Luecke
Department of Mathematics, Iowa State University, Ames, IA, USA
E-mail: grl@iastate.edu

1 Introduction

The advent of high-throughput sequencing (HTS) has resulted in a rapid decline in DNA sequencing costs, outpacing the growth in transistor density from Moore's Law [23]. As a result, genome sequencing is increasing at a rapid pace. Genomics data generation is predicted to potentially dwarf Twitter, YouTube, and astrophysics data combined by the year 2025 [19]. However, performance-sensitive genomics applications have generally not even scaled with Moore's Law, mainly because many such applications are unprepared to fully utilize increasingly-parallel processors. Consequently, algorithmic improvements, specialized computing hardware, and new storage technologies are needed to cope with storing, processing, and analyzing increasing amounts of genomics data.

HTS workflows often include sequence alignment to a reference genome. SAMtools [12] is a utility for performing operations on the resulting sequence alignment data such as sorting, indexing, selecting subsets, compressing, and reporting various statistics. These sequence alignments can be represented in one of three industry-standard formats: the Sequence Alignment/Map (SAM) text format; its binary equivalent, the Binary Alignment/Map (BAM) format; or the more recent CRAM format, which utilizes reference-sequence-based compression and lossy quality scores.

SAMtools utilizes the codeveloped 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, many consider these to be the reference implementations among similar software that work with these formats.

Sequence alignments are typically sorted to make downstream analysis more efficient. However, the sorting operation can itself be a bottleneck in HTS workflows [8, 18]. Sorting large data sets is well-known to be a resource-intensive problem that can benefit from parallelism [7]. The SAMtools 1.3.1 sorting pipeline is already parallelized using the POSIX threads (pthreads) API. But, as the performance analysis in Section 3 illustrates, this parallelization is incomplete, and exhibits inefficiencies. Because of the widespread adoption of SAMtools—including by HTS workflows that run on large-scale public cloud resources, such as Churchill [11], and massively parallel supercomputers, such as MegaSeq [16]—performance enhancements to it would have broad impact.

This work[1] extends our previous effort to improve the performance of SAMtools for the purpose of sorting BAM files [22], primarily by 1) utilizing an alternative algorithm employing fine-grained tasking for compression during BAM encoding to allow adequate load balancing with a larger number of threads, 2) utilizing circular buffers to reduce calls to `memcpy()` and enhance concurrency, 3) improving the performance of external sorting by keeping sorted BAM records in memory where possible instead of writing them to secondary storage before merging, and 4) benchmarking the external sort with a much larger data set, leveraging Cori's newly-available Burst Buffer to store input, output, and intermediate BAM files.

The rest of this paper is organized as follows. Section 2 lists other software tools that aim to efficiently implement performance-critical SAMtools functionality. Section 3 characterizes the performance of SAMtools for internal sorting of a large BAM file. Section 4 describes optimizations implemented in this work to address the identified performance bottlenecks. The impact of this work on performance is measured in Section 5. Section 6 explores other possible optimizations. Section 7 summarizes this effort to address the performance limitations of SAMtools.

2 Related Work

Picard [1] from the Broad Institute is a Java analog to SAMtools, providing similar HTS-processing functionality. Picard currently supports only serial SAM/BAM sorting.

Similar software exists with the primary goal of outperforming SAMtools. One such software, Sambamba [20], aims to be a high-performance replacement for a subset of performance-critical SAMtools functionality. Written from the ground up in the D programming language to implement parallelism, Sambamba strives to fully exploit multi-core CPUs.

elPrep [9] is a Common Lisp/Python application for multi-threaded, in-memory execution of the subset of SAMtools functionality that focuses on preparing sequence alignment data for variant calling. While elPrep's in-memory processing benefits performance, it requires a lot of memory for sorting: the elPrep 2.5 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". For comparison, SAMtools can accomplish an internal sort of the 24.6 GiB BAM ($\sim$85.5 GiB SAM) data set described in Sect. 3 on a 128 GiB-memory compute node. elPrep can sort such BAM files within an accessible memory limit at the cost of extra disk space and I/O overhead: the protocol involves splitting the input BAM files (each containing alignments with respect to a different reference sequence/chromosome), sorting each BAM file independently, and merging the resulting sorted BAM files to produce a single sorted BAM file. Since elPrep uses SAMtools as a library for decoding/encoding BAM files, the optimizations described in this paper could benefit elPrep as well.

Similar to elPrep, biobambam2 [21] focuses on alignment preprocessing in preparation for variant calling. Coded in C++, biobambam2 can perform a multi-threaded SAM/BAM/CRAM sort by coordinate under the condition that the input is already sorted by query (read) name, or sort by query name when the input is in any order.

Some sequence aligners, such as Isaac [17], sort alignments in-memory before output. This strategy avoids the I/O overhead associated with emitting alignments from a sequence alignment process and reading them into a separate sort process.

DNANexus has proposed a fork of SAMtools that leverages RocksDB for improved sorting/merging performance [13], as well as a patch to HTSlib that improves concurrency in BGZF encoding[2].

Both Intel and CloudFlare have created optimized versions of the zlib compression library. These have been shown to benefit SAMtools compression performance[3].

The HTSLib library utilized by SAMtools for SAM/BAM/CRAM I/O supports multi-threaded encoding/decoding of CRAM data using a custom pthreads-based work-stealing thread pool (adapted from the implementation in the Scramble [4] library).

[1] Source code for SAMtools optimizations available at https://doi.org/10.5281/zenodo.262169, and HTSlib optimizations at https://doi.org/10.5281/zenodo.262161

[2] `https://github.com/samtools/htslib/pull/51`

[3] `http://www.htslib.org/benchmarks/zlib.html`

As of this writing, support for multi-threaded BAM encoding/decoding using this thread pool implementation has been committed to the development branch of HTSlib[4]. The alternative approach presented in this paper uses OpenMP 4.0 constructs to implement concurrency. The OpenMP API is supported by almost all major C compilers today [15], and is therefore portable and understood by a comparatively-wide swath of developers. OpenMP runtimes are widely deployed, and therefore relatively well tuned and tested. The authors thus believe that OpenMP can facilitate a more robust, performant, and maintainable solution.

3 Performance Analysis

Performance analysis was conducted on one Haswell compute node of the NERSC Cori supercomputer [6], a Cray XC40. Each such compute node contains two 16-core / 32-thread 2.3 GHz Intel "Haswell" Xeon processors and 128 GiB 2133 MHz DDR4 memory. There is no direct-attached storage on a compute node; rather, a Lustre parallel file system provides a maximum aggregate bandwidth of >700 GB/second to the entire Cori system. To analyze the performance of SAMtools sort, unsorted BAM input was read from and sorted BAM output was written to this Lustre file system, mirroring typical real-world usage.

SAMtools 1.3.1 and HTSlib 1.3.2 were compiled with the gcc 6.2.0 compiler. The default compiler options specified in the HTSlib and SAMtools makefiles were used, with the exception that gcc was invoked with the Cray compiler driver, overridden to use dynamic linking (`cc -dynamic`). Huge pages (2 MiB, the smallest supported in the Cray Linux Environment with an Aries interconnect) were utilized by loading the `craype-hugepages2M` environment module before compiling/linking and running; this resulted in a significant performance improvement (frequently 10-15% decrease in run time) vs. the default 4 KiB page size for the internal sort. The resulting executable was run using the NERSC-recommended SLURM options for a single-node multi-threaded executable (`srun -n 1 -c 64 --cpu_bind=cores`).

The BAM file used to analyze the performance of the internal sort, HG00109.mapped.ILLUMINA.bwa.GBR.low_coverage.20130415.bam from the 1000 Genomes Project [5], contained approximately 207 million alignments of 100 bp paired-end Illumina reads. The BAM file was already sorted by position; to "shuffle" it, it was re-sorted by query name. Interestingly, the compression ratio of the resulting BAM file worsened considerably: originally 19.9 GiB (102.6 GiB uncompressed), the file size increased to 24.6 GiB. This could be attributed at least partially to the reduced likelihood of overlapping sequences appearing within the same 64 KiB BGZF block, providing less repetition for the compression algorithm to leverage.

The SAMtools 1.3.1 sort was run with 32 threads, specifying 3648 MiB per thread (114 GiB total) via the `-m` option-argument to allow the entire uncompressed BAM data set to be sorted in memory. The internal SAMtools sort was run using HPCToolkit [2] (commit 80bc010 from Nov. 8, 2016) to collect call path trace data sampled at 5 millisecond intervals (Fig. 1). The results, which were visualized using the associated `hpctraceviewer` GUI, revealed that program execution time ($\sim$817 seconds) is divided into four phases, in which 1) compressed BAM records are read and decompressed (or "decoded") ($\sim$41% of run time), 2) the BAM records are sorted by a single thread ($\sim$24% of run time), 3) the sorted BAM records are compressed (or "encoded") and output ($\sim$21.5% of run time), and (perhaps most surprisingly) 4) data structures that stored the BAM records are deallocated ($\sim$13.5% of run time). For the internal sort, only the encoding phase is performed by multiple threads.

Armed with insight gleaned from the preceding performance analysis, a plan was formulated to address performance bottlenecks posed by each distinct phase. The resulting optimizations, described in Section 4, were applied to a fork of SAMtools 1.3.1 henceforth called SAMtools OpenMP. SAMtools OpenMP yielded greatly different performance and CPU utilization characteristics, as visualized in Fig. 2.

4 Optimizations

4.1 Decoding

As noted in Section 3, a significant portion (41%) of the $\sim$13.6-minute internal sort run time was spent reading and decoding the 24.6 GiB compressed BAM file. A lower-bound on the run time of this phase is related to the rate at which the BAM file can be sequentially read. To test single-threaded read performance from the Lustre file system, the `dd` command was used to read the benchmark BAM file. Using a 32 KiB block size (corresponding to the read transfer size that HTSlib 1.3.2 is capped to), a throughput of approximately 1 GiB/sec was achieved, indicating that the file system itself was not a significant bottleneck.

The HTSlib 1.3.2 routines for reading compressed BAM are sequential. While reading an input stream is an inherently-sequential operation, compressed BAM

[4] https://github.com/samtools/htslib/pull/397

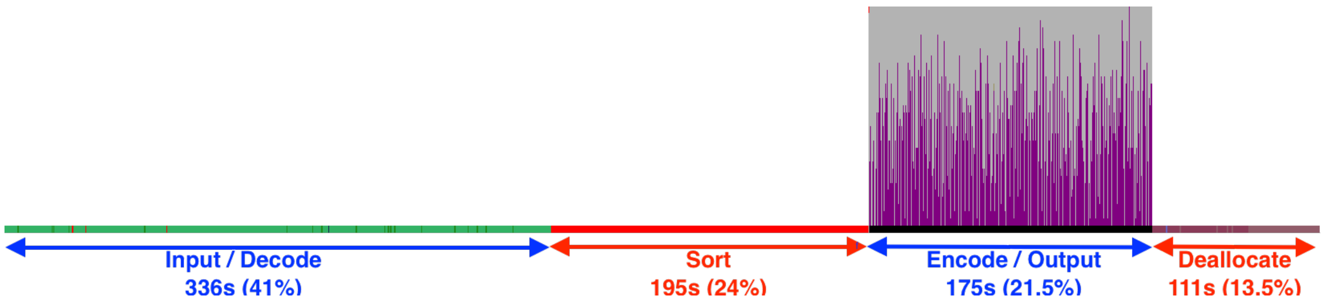


Fig. 1 HPCToolkit performance summary of SAMtools 1.3.1 for an internal sort of a 24.6 GiB BAM file (102.6 GiB uncompressed) using 32 threads. The different colors represent the procedure that a thread is executing: *green* – read/decode, *red* – sort, *purple* – encode (primarily zlib `deflate()`, *gray* – waiting, *white* – no thread exists. Much of the execution is serial, and there is significant thread waiting time in the encode/output phase.

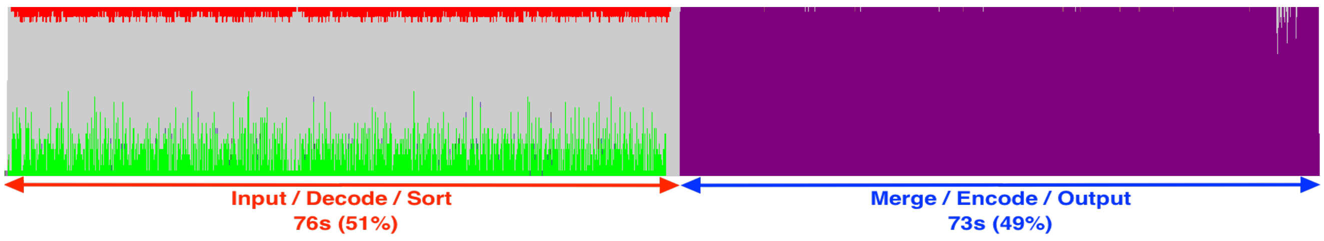


Fig. 2 The performance optimizations described in Section 4 are reflected in the HPCToolkit performance summary of SAMtools OpenMP for the internal sort of a 24.6 GiB BAM file with 32 threads. The different colors represent the procedure that a thread is executing: *green* – read/decode (primarily zlib `inflate()`), *red* – sort, *purple* – encode/output (primarily zlib `deflate()`, *gray* – waiting. There is still significant thread waiting time in the input/decode/sort phase.

consists of BGZF blocks. Each BGZF block (essentially an up-to-64-KiB gzip file with a user-defined field in the gzip header containing the uncompressed length) can be decompressed independently (via the zlib `inflate()` function).

Using a prior effort to parallelize the BAM decoding phase with pthreads [5] as a guide for how to safely add concurrency to the relevant routines, HTSlib was modified so that the master thread reads a group of (default 256) BGZF blocks at a time, generating an `inflate()` task for each. While a coarser level of granularity (i.e., coalescing multiple adjacent blocks into a single payload for each task) would reduce synchronization overhead, the finer level of granularity exposes additional concurrency to enhance load balancing. To help gauge the potential performance impact of fine-grained tasking, a microbenchmark was constructed (Online Resource 1). The microbenchmark revealed that when using 64 threads, it took on average less than 1.5 seconds to create, schedule, and execute 505,455 tasks (corresponding to the number of BGZF blocks in the HG00109 BAM file used to benchmark the internal sort). This result suggested that the finer level of granularity could benefit load balancing without introducing excessive overhead that would unduly impact performance. Looking forward, it will become even more important to express a high level of concurrency to effectively utilize current and future many-core architectures, such as the Intel Xeon Phi.

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

4.2 Memory Allocation and Deallocation

Over 10% of the internal sort run time was spent allocating buffers for each of the ∼207 million BAM records. Two allocations were performed for each BAM record: one for a fixed-length (56-byte) data structure (`bam1_t`), and one for the variable-length member of this data structure (`data`). The corresponding ∼414 million calls to `free()` at the end of execution to deallocate these buffers consumed approximately 13.5% of the run time.

The memory allocation and deallocation overhead was effectively eliminated by allocating a single, contiguous buffer of approximately the maximum memory size requested by the user to store the BAM records. The fixed- and variable-length components of a BAM record are placed adjacently within this buffer, so that each BAM record is contiguous in memory. To facilitate this contiguous storage of a BAM record, a flexible array member (`fam[]`) was added to `bam1_t`. For backwards compatibility, the `data` member points to `fam[]`.

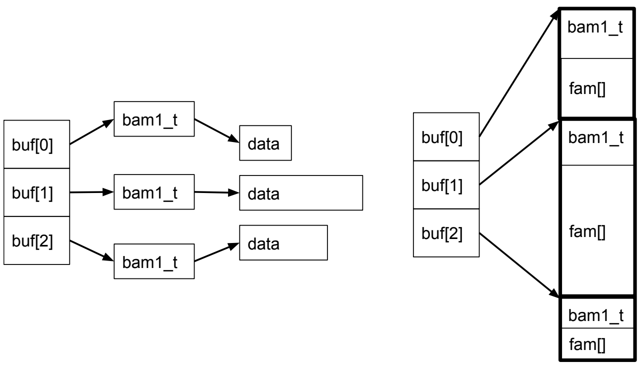


Fig. 3 (*Left*) SAMtools 1.3.1 stores each BAM record in a dynamically-allocated `bam1_t` struct, each containing a separately-allocated `data` member for variable-length data. The array of pointers (`buf[]`) to these `bam1_t` structs is sorted during the sort phase. (*Right*) A flexible array member (`fam[]`) facilitates contiguous storage of the fixed- and variable-length data in a BAM record (`bam1_t`). Each element of `buf[]` points to an 8-byte-aligned `bam1_t` record within a single contiguous memory region.

Reworking other parts of the code to remove dependence on the `data` member, allowing for its removal, could save 8 bytes per `bam_t` BAM record (unlike a pointer, a flexible array member consumes no storage itself).

BAM records begin at the first 8-byte-aligned offset after the end of the previous BAM record to ensure proper memory alignment of all structure members. Each element of the ancillary `buf[]` array points to the start of a BAM record within the contiguous buffer (Fig. 3).

By default, SAMtools OpenMP first stages an input BAM record into a separate buffer using an HTSlib routine that, if necessary, accommodates a larger variable-length `data` member via `realloc()`. If the BAM record (56-byte fixed-length `bam1_t` + variable-length `data`) will fit in the remaining unused part of the contiguous buffer, it is copied into the beginning of the unused part of the contiguous buffer. If the BAM record will not fit, the `buf[]` pointers to the BAM records are sorted, the sorted BAM records are output, and the staged BAM record is copied to the beginning of the contiguous buffer.

SAMtools OpenMP supports an environment variable (`SAMTOOLS_BAM_MAX`) that, when set, assumes BAM records cannot exceed the specified length in bytes, and reads the input data directly into the contiguous buffer if there are at least that many bytes remaining. This avoids the `memcpy()` that is otherwise required copy the BAM record from the "staging" buffer.

An HTSlib API change that could obviate the use of the `SAMTOOLS_BAM_MAX` environment variable would be to add a routine that reads the initial fixed-length part of a BAM record (`bam1_t`), and a routine that reads the variable-length `data` given its length (as recorded in the (`bam1_t l_data` member) as a parameter. This would allow the determination if there is enough space left in the contiguous buffer for the variable-length `data`; if not, the BAM records in the buffer could be sorted and output (and the already-read `bam1_t` copied from its current location near the end of the contiguous buffer to the beginning) before reading the variable-length data.

Finally, the allocation of the `buf[]` array was modified to ensure that SAMtools OpenMP sort tasks (described in Section 4.3), which execute concurrently with subsequent BAM-record input by the master thread, may safely reference segments of the `buf[]` array. SAMtools 1.3.1 initially sizes the `buf[]` array to store 65,536 (2^{16}) `bam1_t` pointers. If the number of BAM records read exceeds this limit, SAMtools 1.3.1 will `realloc()` `buf[]` to the next larger power of 2. However, if there is not enough contiguous space in virtual memory to accommodate the larger size, `realloc()` will cause the array to be moved to another location in virtual memory, invalidating any pointers into the array. SAMtools OpenMP fixes the size of the `buf[]` array so that any pointers into it will not become invalidated during its lifetime. A trade-off is that run-time inflexibility of the `buf[]` array size imposes a second constraint on the number of BAM records that may be sorted in-memory, in addition to the size of the contiguous buffer that stores the BAM records. The `buf[]` array is sized to accommodate one pointer-to-BAM-record for every 300 bytes (loosely corresponding to the size of a BAM record for a 100bp read) allocated to the contiguous buffer for BAM records,

4.3 Sort

(*Internal*) A major shortcoming of the SAMtools 1.3.1 internal sort is that it is performed by only a single thread. To address this issue, SAMtools OpenMP generates a sort task for every (empirically-chosen) 2^{20} BAM records read. Sort tasks can execute concurrently with subsequent `inflate()` tasks, allowing the input/decode and sort phases to overlap, and thus taking advantage of spare computing capacity that may be available (see Fig. 2). The resulting sorted BAM sublists are merged in-memory into a single output stream.

(*External*) If the input BAM records exceed the user-specified memory limit (SAMtools sort `-m` option-argument), then the resulting SAMtools 1.3.1 sort is parallelized. The BAM records in memory are partitioned into N approximately-equal sized sublists, where N is the number of threads. Each thread sorts its sublist, then writes it to a temporary file. This process repeats until all input has been processed in this manner. Finally, the initial thread merges the sorted temporary BAM files (using a min-heap data structure) to produce the final sorted BAM output stream.

A drawback of the aforementioned approach is that for each "chunk" of input, another N files are created—so if M input chunks are processed, $N \times M$ files must be merged in the end. In general, fewer, larger I/O streams to a single process are more efficient than many, smaller I/O streams. To account for this, two optimizations were implemented in SAMtools OpenMP for the external sort:

1. Instead of writing each sorted sublist (2^{20} BAM records for SAMtools OpenMP) to a separate file, the sublists are merged in-memory to output a single sorted temporary BAM file per chunk (as opposed to N files). This reduces the number of temporary files by a factor of N.
2. The last chunk is not output before the final merge; rather, the merge is done with the data in memory. This allows the final $N-1$-way file (plus in-memory sublists) merge to begin sooner, without having to wait for the sorted sublists of the last input chunk to be written to secondary storage.

4.4 Encoding

SAMtools 1.3.1 supports multi-threaded compression of BAM records. The initial thread `memcpy()`s up to 256 blocks of ≤ 64 KiB uncompressed BAM records into a buffer, while other threads are blocked on a condition variable. Once the buffer has been filled, the initial thread then issues a `pthread_cond_broadcast()` to unblock the other threads. Blocks are assigned to workers in a cyclic fashion. Each thread compresses its block into a temporary buffer, copies the compressed block back to the original buffer, and proceeds with its next assigned block. After the initial thread finishes compressing its assigned blocks, it spin waits until all other threads have finished, then outputs the blocks in order, copies the next up-to-256 uncompressed blocks into the buffer, and the process repeats.

The aforementioned method, which essentially relies on a barrier and serial execution (output of the BGZF blocks by the master thread) in between rounds of parallel work (compression), fails to keep threads busy for much of the encode phase (as illustrated in Fig. 1). SAMtools OpenMP uses an algorithm that expresses more concurrency, providing the OpenMP runtime an opportunity to load balance and decrease the likelihood that a thread will be waiting for work.

Listing 1 C pseudocode routine to concurrently compress ≤ 64 KiB blocks of BAM records and output the resulting compressed BGZF blocks in input order

```
void encode(U,C,U_len,C_len,ul,ticket){
  i = ticket % NBLOCKS
  i_next = (i + 1) % NBLOCKS
  do {
    #pragma omp atomic read
    u_len_priv = U_len[i_next];
    if (u_len_priv == 0) break;
    #pragma omp taskyield
  } while(true);
  U_len[i] = ul; // slot in use
#pragma omp task firstprivate(ticket)
{ i = ticket % NBLOCKS;
  off = i * BLOCK_SIZE
  compress(&C[off],&cl,&U[off],U_len[i]);
  #pragma omp atomic write seq_cst
  C_len[i] = cl; // compressed data ready
  #pragma omp atomic read
  ticket_private = ticket_shared;
  if (ticket_private == ticket)
  #pragma omp critical
  {
    #pragma omp atomic read
    ticket_private = ticket_shared;
    if (ticket_private == ticket)
      do {
        output(&C[off]), cl)
        C_len[i] = 0;
        #pragma omp atomic write
        U_len[i] = 0; // slot unused
        i = (i + 1) % NBLOCKS;
        #pragma omp atomic update
        ticket_shared++;
        #pragma omp atomic read seq_cst
        cl = C_len[i];  // data ready in
      } while (cl > 0); // next slot?
  } // critical
} // task
} // encode()
```

The general idea of the approach is that separate circular buffers are used for input (to the compression tasks) and output. The master thread generates tasks that each compress a block of BAM reads from one circular buffer (U), writing the result directly into the corresponding slot of the other circular buffer (C)—this differs from SAMtools 1.3.1, which compresses the data into a per-thread buffer and then `memcpy()`s it back to the original buffer, overwriting the uncompressed data.

Output is serialized and ordered. A task that has BGZF compressed its block, but whose turn it is not to output, can leave the block for the task that outputs the immediately-preceding block in the circular buffer. The task whose turn it is to output its BGZF block can also output subsequent consecutive BGZF blocks that are ready.

The resulting routine, which is syntactically-amenable to an OpenMP implementation (Listing 1), is described as follows. To avoid a data race, the master thread verifies (lines 5-7) that the next block slot in U is free (i.e., the length of the uncompressed block, stored in $U\_len[next]$, is 0) so that upon return from the routine, it may be safely written to. If the next slot is not open, the master thread invokes a `taskyield` so that it may process another pending task[6], then checks again in a loop until the slot is available.[7] After verifying that the next slot is free, the master thread marks the current slot in use by assigning it the length of the uncompressed block (ul) in bytes (line 10).

Next, the master thread generates a task to compress and (possibly) output the block of BAM records in the current slot. The data that is copied into the task must at least comprise the ordinal-numbered *ticket* used for output ordering, from which the slot could be reconstituted if not part of the `firstprivate` "payload". An offset into C and U is calculated (line 13) based on the block slot number (i) and maximum block size ($BLOCK\_SIZE$). The indexed block in U is compressed into C, saving the resulting compressed block length in cl (line 14). The task then signals that the compressed block is ready for output by assigning its length to $C\_len[i]$ (lines 15-16). This must be done not only atomically, but with sequential consistency (OpenMP `seq_cst` clause) so that if it is not this tasks's turn to output (per the ticket lock check in lines 17-19), changes made to other data (specifically, C) are reflected in memory for the task that will output this task's compressed block.

A variation of the ticket lock [14] is used to enforce output ordering. In lines 17-19, the task checks the ticket that is currently being served (*ticket_shared*) to see if it matches the task's *ticket*. If not, then the task ends, leaving the thread to process another task, and the compressed block beginning at `&C[off]` for another task to output. This pre-`critical`-region ticket check is an important optimization that prevents unnecessary attempts to enter the `critical` region, during which time the executing task would be blocked and not doing useful work. If it is this task's (say, t_j) turn per the ticket value, then t_j enters the `critical` region and checks the ticket value *again* (lines 22-24). This second check is used because the task that last incremented the shared ticket value (say, t_k) at lines 31-32 may have already output t_j's block while it was in the `critical` region.

If it is the current task's turn to output (i.e., line 24 evaluates to true), then the task output's its compressed BGZF block (line 26) and sets the $C\_len[i]$ value to 0 (line 27) to indicate that there is now no compressed data in the corresponding slot of C to output. Next, the subsequent slot of the C_len circular buffer is checked to see if there is compressed data in the corresponding slot of C (lines 35-37). The atomic read of C_len must be sequentially consistent to ensure the executing thread's view of the corresponding data in C is consistent with the data flushed to memory by the sequentially-consistent write to C_len at lines 15-16. The `task` and `critical` directives, which "flush" the executing thread's memory on entrance and exit, provide sufficient memory consistency for other data accesses.

To verify correctness, we must show that every block is output once in the correct order. Intuitively, the described ticket lock mechanism enforces output ordering, and the fact that the check is executed inside the `critical` region ensures no block is output more than once. To informally inductively show that every block is output, note that the task (say, t_0) with the first ticket value ($ticket$ == 0) will enter the `critical` region and output its block. If t_0 atomically executes line 34 ($cl = C\_len[i]$) before another task (say, t_1) atomically executes line 16 ($C\_len[i] = cl$), then t_0 will not output t_1's block—however, as t_0 previously incremented the shared ticket value (line 32), and t_1 subsequently checks the shared ticket value (lines 18-19), then t_1 will output its own block. Otherwise, if t_0 executes line 34 after t_1 executes line 16, then t_0 will output t_1's block, and so on. The fact that the block is marked ready for output (line 16) by a task outside the `critical` region before it checks the ticket value at line 17 (and possibly does not enter the critical region), combined with the opposite ordering inside the critical region (atomically increment the ticket at line 32, then check if the subsequent block in the circular buffer is ready for output) ensures that all blocks will eventually be output.

An unexpected overhead from using fine-grained tasking was uncovered during subsequent performance analysis. By default, the GNU OpenMP runtime library (libgomp) causes a waiting thread (e.g., for the next task to execute, or at the entrance to a `critical` region) to initially wait actively by spin-waiting for 300,000 spins before passively waiting (e.g., with the `futex_wait()` system call on Linux). Actively waiting reduces the latency for the thread to continue making progress once it is able to, while backing off to a passive waiting state reduces contention for CPU resources.

[6] Note that `taskyield` is a no-op as of gcc 6.2.0

[7] This check could occur after task generation and before returning from the routine; however, the implementation did not consistently perform as well in practice, possibly due to an undetermined effect on task scheduling.

During the external merge, the master thread reads the temporary sorted BAM files, calling the zlib `inflate()` routine to decompress compressed BGZF blocks, and placing the uncompressed BAM records in a min-heap data structure for merging. Other threads execute computationally-intensive tasks that compress and output the resulting merged BAM. The fine-grained approach used by SAMtools OpenMP of compressing one BGZF block per task results in "smaller" tasks to facilitate better load balancing; however, a side effect is that threads are more likely to be actively waiting (i.e., in a spin-wait) when they are not actively executing tasks. This extra contention caused the unanticipated side effect of making the zlib `inflate()` routine, which during merging is executed by only the master thread (concurrently with any encoding tasks), take significantly longer ($> 33\%$). The slow down of this serial task caused a corresponding increase in total run time.

Setting the `OMP_WAIT_POLICY` environment variable to `passive` (which causes libgomp to set the spin count to 0, resulting in a thread skipping active waiting and instead only wait passively when unable to immediately execute a task) alleviated the contention, and restored the performance of `inflate` in the master thread.

More granular control over the spin count could be achieved with the libgomp-specific `GOMP_SPINCOUNT` environment variable. An alternative approach could be to leverage multi-threaded decoding during the external-sort merge phase (as is used during the initial decoding phase).

As an aside, the encoding/decoding optimizations benefited the standalone HTSlib bgzip utility, though the performance improvements are not quantified in this paper.

5 Benchmarks

The effects of the optimizations to SAMtools OpenMP were characterized using both an internal sort of the BAM file described in Section 3, and an external sort of a much larger BAM file (HG01565.wgs.ILLUMINA.-bwa.PEL.high_cov_pcr_free.20140203.bam from the 1000 Genomes Project), representing alignments of almost 570 million 250 bp Illumina reads. As with the smaller BAM file used to benchmark the internal sort, the file size of the larger BAM used to benchmark the external sort (180.2 GiB, sorted by position) increased dramatically (to 271.4 GiB) when "shuffled" via sorting by query name. Multi-threaded sorting was performed with 1, 2, 4, 8, 16, 32, and 63 or 64 threads. It was assumed that consistent file sizes of the resulting BAM file indicated correct output.

With $\leq$ 32 threads, threads were pinned to processor cores to maximize cache reuse. For SAMtools 1.3.1 (and SAMtools OpenMP with only a single thread, resulting in an inactive OpenMP parallel region), this was accomplished using the SLURM environment variable `SLURM_CPU_BIND=cores`. While the NERSC-recommended environment variables for For SAMtools OpenMP, the NERSC-recommended OpenMP environment variables were used (`OMP_PLACES=threads OMP_PROC_BIND=spread`). For SAMtools OpenMP with between 2 and 32 threads, the NERSC-recommended environment variables `OMP_PLACES=threads OMP_PROC_BIND=spread` were initially used in an attempt to bind OpenMP threads to hardware threads distributed evenly distributed evenly between both sockets to ensure the memory bandwidth and last-level cache of both sockets was utilized. However, it was discovered that due to an apparent bug in the GNU OpenMP runtime (verified with NERSC support staff), this achieved the opposite effect: OpenMP threads were packed in consecutive hardware threads on the same core/socket as if `OMP_PROC_BIND=close` were specified. Substituting `OMP_PLACES=cores` for `OMP_PLACES=threads` was observed to result in the intended `spread` thread affinity to processor cores.

To test the effect of Hyper-Threading (utilizing both hardware threads on each processor core), SAMtools 1.3.1 was run with 64 threads, setting the environment variable `SLURM_CPU_BIND=threads` to pin application threads to hardware threads. With SAMtools OpenMP, a performance regression ($\sim$15% slowdown) was observed when increasing the number of threads from 32 to 64 and controlling OpenMP thread affinity using `OMP_PLACES=threads OMP_PROC_BIND=spread`. Performance profiling revealed that much of the performance regression was due to an increase in the amount of time the master thread spent performing the newly-implemented in-memory merge of sorted BAM records during output (described in Section 4.3). As this merge was performed solely by the master thread, it was hypothesized that the cause of the reduced performance introduced by using 2 OpenMP threads per processor core was contention from another OpenMP thread executing on the same processor core as the master thread. This hypothesis was tested by using 63 threads and setting `OMP_PLACES={0,32},{1}:31,{33}:31`, causing the master thread to be pinned to the first processor core (logical processors 0 and 32 on Cori), and the remaining OpenMP threads pinned to the hardware threads of the other processor cores. The $\sim$15% slowdown turned into a $\sim$5% speedup, providing evidence to support this hypothesis.

For SAMtools OpenMP, the `OMP_WAIT_POLICY` was set to `passive` to reduce contention with the master thread during decompression (described in Section 4.4), while the `SAMTOOLS_BAM_MAX` environment variable (described in Section 4.2) was set to 65536—a safe maximum BAM record size in bytes for Illumina read alignments—to eliminate a `memcpy()` for each input BGZF block.

5.1 Internal Sort

Ten trials were performed with SAMtools OpenMP and SAMtools 1.3.1 at each thread count. The per-thread memory (`-m`) option-argument was specified so as to keep the total memory allocated for BAM records at $\sim$114 GiB for SAMtools 1.3.1 (the approximate minimum that allowed SAMtools 1.3.1 to perform the sort entirely in memory), and 90 GiB for SAMtools OpenMP (which uses the `-m` option-argument to size only the contiguous-memory buffer for BAM records, rather than as an approximate total limit that also attempts to account for the ancillary `buf[]` array).

5.1.1 Burst Buffer with Internal Sort

In addition to the Lustre parallel file system, another available storage tier on Cori is the *Burst Buffer* [3], a Cray DataWarp I/O accelerator. The Burst Buffer provides high-bandwidth, low-latency I/O via SSDs deployed in special Burst Buffer nodes on the Cray Aries network. To minimize the possibility of I/O performance degradation due to contention from other jobs utilizing the shared Lustre parallel file system, the input BAM file was staged the Burst Buffer before execution, and output BAM file written to the Burst Buffer. The job script directive `DW jobdw capacity=80GiB access_mode=private type=scratch` reserved ample space on a single Burst Buffer SSD.

The I/O transfer size for reads and writes should be at least 512 KiB to get good performance with the Burst Buffer due to the lack of client-side file system caching (as of this writing, this issue is slated to be addressed in a future release of the Cray DataWarp software) [3]. HTSlib uses the `stat()` system call to query the file system for the preferred I/O block size, and as a result performs sufficiently-large 8 MiB writes to the Burst Buffer. However, HTSlib caps reads to 32 KiB, causing a severe performance hit. As a workaround, the Cray IOBUF library was used to prefetch the input BAM file and deliver 32 KiB reads to HTSlib from in-memory IOBUF buffers. Using an empirically-chosen two 32M buffers (by setting the `IOBUF_PARAMS` environment to `count=2:size=32M`) for the input BAM file resulted in greatly improved read performance, increasing the average `read()` rate from almost 70 MB/s to almost 2.4 GB/s (as observed by the IOBUF asynchronous reads from the Burst Buffer, with an effective throughput of almost 5 GB/s delivered to the `read()` system calls in HTSlib).

5.1.2 Internal Sort Results

The resulting run times are presented in Fig. 4. The timings were fairly consistent between trials, with an average relative standard deviation (RSD) of 0.74% for SAMtools 1.3.1, and 0.85% for SAMtools OpenMP.

The single-threaded performance of SAMtools OpenMP was only slightly better than SAMtools 1.3.1, with a 2.5% decrease in average run time. This was a smaller performance improvement than anticipated—the memory allocation/deallocation optimizations alone (Section 4.2) were expected to contribute to a greater reduction in run time. Additionally, an unexpected apparent superlinear speedup of SAMtools OpenMP was observed at 2 threads. The underperformance of single-threaded SAMtools OpenMP could possibly be attributed to the order in which the applications were run: a total of 10 job scripts were submitted, each running all thread counts once for both SAMtools OpenMP and SAMtools 1.3.1. The first application run in this loop was SAMtools OpenMP with 1 thread. Further tests suggested that the first SAMtools sort execution in a job script experienced slightly worse performance than subsequent executions. This could not be attributed to increased read transfer time, as the reported IOBUF performance metrics indicated the aggregate time of all `read()` system calls to the input BAM file ranged 5.7 to 7.2 seconds among single-threaded SAMtools OpenMP executions. Additional analysis is needed to pinpoint the cause.

Activating the multi-threading code paths in each version at two threads, SAMtools OpenMP was noticeably faster than SAMtools 1.3.1 (1.36X speedup), with the performance gap widening as threads were added. The best average run time of SAMtools OpenMP (with 63 threads) outperformed the best average run time of SAMtools 1.3.1 (with 64 threads) by a factor of 5.9X. Peak memory usage for SAMtools 1.3.1 (expressed in terms of the virtual-memory upper bound measured by the SLURM `MaxVMSize` job accounting field, as the physical-memory `MaxRSS` was not reliably recorded) ranged from approximately 117.46 GiB with 1 thread to 121.5 GiB with 64 threads. For SAMtools OpenMP, `MaxVMSize` ranged from approximately 92.67 GiB (observed with 2 threads) to 96.9 GiB (63 threads). This suggests that in addition to less time overhead, the SAMtools OpenMP approach of allocating a single contiguous-memory buffer to store all BAM records also has less space overhead than the SAMtools 1.3.1 approach of two memory allocations per BAM record, allowing larger BAM files to be sorted within a given physical-memory limit.

5.2 External Sort

Due to the increased wall time required to perform the external sort, only three trials were performed at each thread count. The trials could not be run consecutively on the same node due to the queue wall-time limit, so they were submitted as individual jobs, and thus scheduled to run on any available node.

5.2.1 Burst Buffer with External Sort

As with the internal sort, the Burst Buffer was used to stage the input BAM file and receive the output BAM file. The external sort required additional storage for intermediate sorted BAM files. This additional storage was provisioned and striped across three Burst Buffer SSDs using the job script directive `DW jobdw capacity=639GB access_mode=striped type=scratch` (note that the granularity of the Burst Buffer stripes is 213 GB).

Utilizing the IOBUF library with SAMtools 1.3.1 proved challenging. As previously mentioned, the IOBUF library is necessary for SAMtools to get good read performance from the Burst Buffer, but is not necessary for good write performance. Moreover, the IOBUF library is not thread safe, and concurrent writes by different threads to even different temporary sorted BAM files caused frequent runtime errors. However, IOBUF buffers both reads and writes to files matching user-specified filename patterns, and cannot be configured to buffer only reads, but not writes (or vice versa) to a particular a file. Therefore, a workaround was implemented via a small source-code patch (Online Resource 2) to SAMtools 1.3.1 that caused a `.write` extension to be added to the temporary sorted BAM filenames when written. After the temporary BAM files were closed, and before they were opened again for reading during the final merge, they were renamed without the `.write` suffix. This allowed the filename patterns specified in the `IOBUF_PARAMS` environment variable to match only the input and temporary files when read, excluding the output and temporary files when written.

As described in Section 4.3, the SAMtools 1.3.1 external sort produces an increasing number of temporary sorted BAM files as the number of threads increases (e.g., with 64 threads, 760 temporary files were generated for the benchmark data set). The `IOBUF_MAX_FILES` environment variable was increased from the default 256 to 768 for the 32 and 64 thread trials to allow IOBUF to handle the larger number of file descriptors associated with the additional temporary files. To reduce IOBUF memory requirements, two 8M IOBUF buffers (rather than two 32M buffers, as was used for the input BAM file) were dedicated to each temporary file. The memory limit specified for the SAMtools 1.3.1 external sort (`-m` option-argument) was decreased relative to the internal-sort limit to 100 GiB to accommodate the extra memory used by IOBUF.

SAMtools OpenMP writes only one temporary sorted BAM for each chunk of (uncompressed) BAM records read into memory, rather than one for each thread. This results in fewer, larger intermediate BAM files involved in the final merge (e.g., with the benchmark data set at all thread counts, only 6 temporary BAM files plus almost 71 million BAM records from the last unwritten chunk in memory). Because synchronization ensures that only one thread is writing at a time, SAMtools OpenMP can safely utilize the non-thread-safe IOBUF library. This obviates the need for a workaround (like the modification to SAMtools 1.3.1) that prevents IOBUF from handling writes to the temporary BAM files. While SAMtools OpenMP could dedicate more memory for sorting due to fewer intermediate BAM files (and thus fewer IOBUF buffers), for consistency the authors chose to set the memory allocated to the contiguous-memory BAM buffer (`-m` option-argument) at 100 GiB, corresponding to the amount specified for SAMtools 1.3.1.

5.2.2 External Sort Results

The external sort benchmark results are presented in Fig. 4). Single-threaded performance improvement of SAMtools OpenMP was $\sim$6.6%. This could be attributed to not only memory-allocation-related optimizations (Section 4.2), but also the fact that the last chunk of $\sim$71 million sorted BAM records was merged directly from memory (Section 4.3) with the other 6 sorted intermediate BAM files, rather than being encoded and output to the Burst Buffer before the final merge (note that due to differences in its memory accounting, single-threaded SAMtools 1.3.1 generated 11 intermediate sorted BAM files). At $\geq$ 8 threads, SAMtools OpenMP outperformed SAMtools 1.3.1 by approximately a factor of 2. There was little variation in runtimes between either version at any thread count: the average RSD was 0.39% for SAMtools 1.3.1, and 0.40% for SAMtools OpenMP.

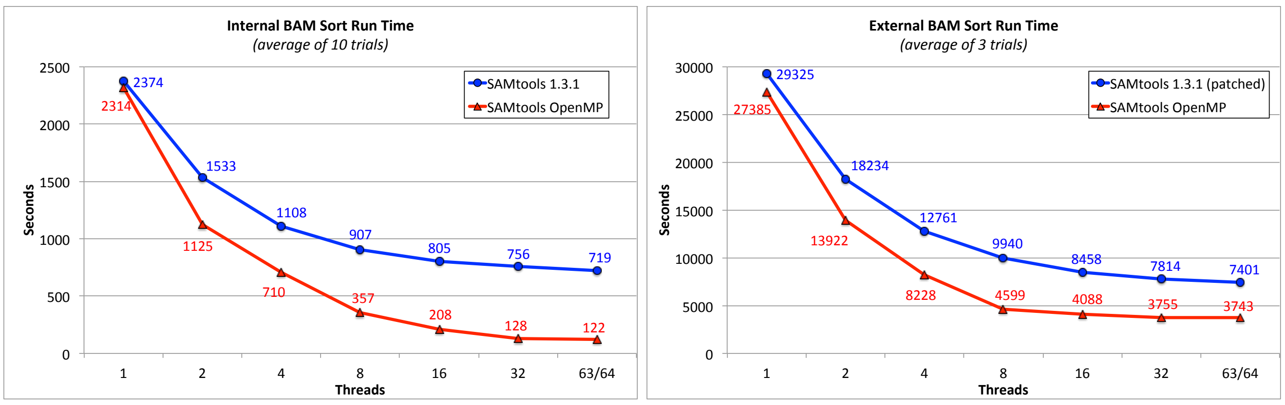


Fig. 4 *(left)* Run times in seconds for the internal sort of a 24.6 GiB BAM file from ~207 million 100 bp reads. *(right)* Run times for the external sort of a 271.4 GiB BAM file from ~570 million 250 bp reads. SAMtools 1.3.1 was patched to avoid significant performance degradation when using the Burst Buffer.

The approximate peak memory usage (`MaxVMSize`) recorded for SAMtools 1.3.1 ranged from ~104 GiB (1 thread) to 122.3 (64 threads). Memory usage scaled with the number of threads primarily due to the increased number of IOBUF buffers required to accommodate the larger number of intermediate sorted BAM files. The corresponding approximate range recorded for SAMtools OpenMP—104 GiB (1 thread) to 105.8 GiB (64 threads)—indicated relatively stable memory usage, with more memory headroom available at all thread counts.

6 Future Work

When decoding sequentially-read compressed BAM files, a dedicated read-ahead thread could increase the rate at which decoding tasks are generated, and reduce the frequency with which threads are idle. During the merge phase of an external sort, multi-threaded decoding (without a per-file read-ahead thread) of the sorted temporary BAM files being merged could potentially improve performance.

Intel's Quick Assist hardware accelerator has been shown to improve the performance of zlib compression by over 20X [10], and provides hardware-accelerated decompression as well. Because a large fraction of the computation time in the SAMtools BAM sort workflow concerns BAM compression and decompression, this hardware accelerator could potentially have a major impact on the performance of this workflow.

A distributed-memory implementation (e.g., utilizing MPI or Unified Parallel C) could allow larger BAM files to be sorted in memory.

The concepts described in this paper could be implemented to support CRAM encoding/decoding in HTSlib.

7 Conclusions

SAMtools is a fundamental component of many HTS workflows, and processes a significant fraction of the HTS alignment data that is generated. Performance improvements to this important tool have the potential to reduce time to solution for a variety of genomics problems facing agriculture, oncology, pathology, and other life sciences.

Performance analysis exposed unexpected bottlenecks in the SAMtools internal sort workflow. Subsequent performance enhancements to the SAMtools BAM sorting code described in this paper improved scalability for both internal (5.9X speedup with 32 processor cores) and external (1.98X speedup) sorting. However, there are still performance challenges to address.

OpenMP facilitated many of the performance gains. Its high-level API allowed task parallelism to be succinctly retrofitted in parts of the SAMtools and HTSlib codebase that could benefit; in some cases supplanting the existing pthreads code, while in others providing parallelism that had not previously been expressed. The many optimization techniques demonstrated in this paper, such as the flexible-array-member-facilitated contiguous-memory optimization and the OpenMP-compatible two-circular-buffer encoding routine, can be applied to other problems.

Genomics data production is accelerating, challenging existing software tools that process and analyze it. Processor roadmaps that resulted in rapidly increasing serial performance in successive generations were abandoned over a decade ago. Software tools must be parallelized to extract performance gains from increasingly-parallel modern multi-core processors. High-level APIs such as OpenMP are needed to effectively express this parallelism. Performance tools are needed to analyze complicated applications to pinpoint bottlenecks and determine where optimization might be productive. But perhaps most difficult is the challenge of assembling an interdisciplinary team of computational genomics experts and HPC software developers to craft performant, functional genomics software tools that can handle increasingly-big genomics data.

8 Acknowledgment

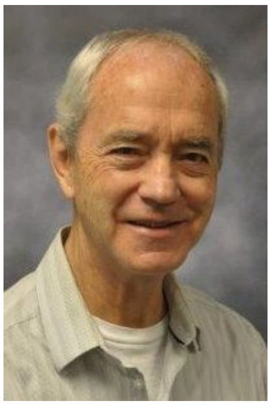
This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

1. (2009) Picard. URL `https://broadinstitute.github.io/picard/`
2. Adhianto L, Banerjee S, Fagan M, Krentel M, Marin G, Mellor-Crummey J, Tallent NR (2010) HPCTOOLKIT: tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience 22(6):685–701, DOI 10.1002/cpe.1553
3. Bhimji W, Bard D, Romanus M, Paul D, Ovsyannikov A, Friesen B, Bryson M, Correa J, Lockwood GK, Tsulaia V, et al (2016) Accelerating Science with the NERSC Burst Buffer Early User Program. In: 2016 Cray User Group (CUG 2016), URL `https://cug.org/proceedings/cug2016_proceedings/includes/files/pap162.pdf`
4. Bonfield JK (2014) The Scramble conversion tool. Bioinformatics 30(19):2818–2819, DOI 10.1093/bioinformatics/btu390
5. Consortium TGP (2015) A global reference for human genetic variation. Nature 526(7571):68–74, DOI 10.1038/nature15393
6. Declerck T, Antypas K, Bard D, Bhimji W, Canon S, Cholia S, He HY, Jacobsen D, Prabhat NJW (2016) Cori-A System to Support Data-Intensive Computing. In: 2016 Cray User Group (CUG 2016), URL `https://cug.org/proceedings/cug2016_proceedings/includes/files/pap171.pdf`
7. Diekmann R, Gehring J, Luling R, Monien B, Nubel M, Wanka R (1994) Sorting large data sets on a massively parallel system. In: Proceedings of 1994 6th IEEE Symposium on Parallel and Distributed Processing, pp 2–9, DOI 10.1109/SPDP.1994.346188
8. Faust GG, Hall IM (2014) SAMBLASTER: fast duplicate marking and structural variant read extraction. Bioinformatics 30(17):2503–250, DOI 10.1093/bioinformatics/btu314
9. Herzeel C, Costanza P, Decap D, Fostier J, Reumers J (2015) elPrep: High-Performance Preparation of Sequence Alignment/Map Files for Variant Calling. PLoS ONE 10(7):1–16, DOI 10.1371/journal.pone.0132868
10. Intel Corporation (2015) Programming Intel QuickAssist Technology Hardware Accelerators for Optimal Performance. Tech. rep., URL `https://01.org/sites/default/files/page/332125_002_0.pdf`
11. Kelly BJ, Fitch JR, Hu Y, Corsmeier DJ, Zhong H, Wetzel AN, Nordquist RD, Newsom DL, White P (2015) Churchill: an ultra-fast, deterministic, highly scalable and balanced parallelization strat-

egy for the discovery of human genetic variation in clinical and population-scale genomics. Genome Biology 16(1):6, DOI 10.1186/s13059-014-0577-x
12. Li H, Handsaker B, Wysoker A, Fennell T, Ruan J, Homer N, Marth G, Abecasis G, Durbin R, Subgroup GPDP (2009) The Sequence Alignment/Map format and SAMtools. Bioinformatics 25(16):2078–2079, DOI 10.1093/bioinformatics/btp352
13. Lin M (2014) Faster BAM sorting with SAMtools and RocksDB. URL `http://devblog.dnanexus.com/faster-bam-sorting-with-samtools-and-rocksdb/`
14. Mellor-Crummey JM, Scott ML (1991) Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. ACM Trans Comput Syst 9(1):21–65, DOI 10.1145/103727.103729, URL `http://doi.acm.org/10.1145/103727.103729`
15. OpenMP Architecture Review Board (2013) OpenMP Application Program Interface, Version 4.0. URL `http://www.openmp.org/resources/openmp-compilers/`
16. Puckelwartz MJ, Pesce LL, Nelakuditi V, Dellefave-Castillo L, Golbus JR, Day SM, Cappola TP, Dorn GW II, Foster IT, McNally EM (2014) Supercomputing for the parallelization of whole genome analysis. Bioinformatics 30(11):1508, DOI 10.1093/bioinformatics/btu071
17. Raczy C, Petrovski R, Saunders CT, Chorny I, Kruglyak S, Margulies EH, Chuang HY, Kllberg M, Kumar SA, Liao A, Little KM, Strmberg MP, Tanner SW (2013) Isaac: ultra-fast whole-genome secondary analysis on illumina sequencing platforms. Bioinformatics 29(16):2041, DOI 10.1093/bioinformatics/btt314
18. Rengasamy V, Madduri K (2016) SPRITE: A Fast Parallel SNP Detection Pipeline, Springer International Publishing, Cham, pp 159–177. DOI 10.1007/978-3-319-41321-1_9
19. Stephens ZD, Lee SY, Faghri F, Campbell RH, Zhai C, Efron MJ, Iyer R, Schatz MC, Sinha S, Robinson GE (2015) Big Data: Astronomical or Genomical? PLoS Biol 13(7):1–11, DOI 10.1371/journal.pbio.1002195
20. Tarasov A, Vilella AJ, Cuppen E, Nijman IJ, Prins P (2015) Sambamba: fast processing of NGS alignment formats. Bioinformatics 31(12):2032–2034, DOI 10.1093/bioinformatics/btv098
21. Tischler G (2017) biobambam2. URL `https://github.com/gt1/biobambam2`
22. Weeks NT, Luecke GR (in press) Performance Analysis and Optimization of SAMtools Sorting. In: 4th International Workshop on Parallelism in Bioinformatics (PBio2016)
23. Wetterstrand K (2016) DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP). URL `http://www.genome.gov/sequencingcostsdata`

Nathan T. Weeks received B.S. degrees in Computer Science (2002) Mathematics (2003) from South Dakota State University, and an M.S. (2012) in Computer Science from Iowa State University. He is currently pursuing a Ph.D. in Computer Science at Iowa State University. His research interests include parallel computing, software application optimization, and bioinformatics.



Glenn R. Luecke received his B.S. degree from Michigan State University in Mathematics and his Ph.D. in Mathematics from the California Institute of Technology. He is currently Professor of Mathematics, adjunct Professor of Computer Engineering, Senior Member of the ACM, Director of an HPC group and in charge of HPC education and training at Iowa State University. Professor Luecke's HPC group is involved in research in the areas of parallel algorithms, parallel tools, and the evaluation of high performance computing systems. He has had over 60 graduate students, visiting scholars, and post-doctoral students.