OpenMP Task Parallelism for Faster Genomic Data Processing

Nathan T. Weeks

Introduction: The Genomics Big Data Problem

Genomic sequencing is being leveraged for a wide range of applications such as helping plant breeders select for traits for higher-yielding, more nutritious, and disease-resistant crops; diagnosing and treating diseases; and tracing the ancestry of organisms in the tree of life. With the cost of this technology dropping precipitously since the advent of High-Throughput Sequencing (HTS)[3], sequence data are being generated at an accelerating pace, challenging the ability to store, process, and analyze it.

Much performance-critical software in this space is coded in C or C++, with POSIX threads (pthreads) being the most commonly-used shared-memory parallel programming API. However, this common choice of pthreads by application programmers should be reevaluated: OpenMP support is nearly ubiquitous in C/C++ compilers; OpenMP tasks can succinctly express task parallelism that previously required a non-standard library or bespoke implementation in the lower-level pthreads API; and OpenMP runtimes provide a well-tested means of controlling thread affinity and task execution that would be challenging to replicate in bespoke pthreads libraries.

In this case study, OpenMP task parallelism is retrofitted into an existing nontrivial pthreads-based bioinformatics application to improve the parallel efficiency and take advantage of additional opportunities for concurrency. Replacing separate groups of pthreads utilized in application and library code with OpenMP tasks generated in both allows concurrent execution of application and library code by the same OpenMP team of threads, facilitating higher processor utilization while elegantly avoiding the load-balancing challenges that nested thread teams would pose.

Sequence Alignment Sorting

An HTS workflow can involve many software applications (see Figure 1), each with its own performance characteristics and challenges. Many HTS workflows use an application called a sequence aligner to align sequence reads (short segments of DNA generated from the organism(s) of interest) to a reference genome assembly. The resulting sequence alignments are typically then sorted by their position on the reference genome to make downstream analysis much more efficient.

SAMtools is a command-line utility that is commonly used to sort (and perform other operations on) sequence alignment data. SAMtools leverages the co-developed library HTSlib for I/O of supported sequence alignment file formats. Both are coded in C.

A Binary Alignment/Map (BAM) file is one such sequence alignment file format supported by HTSlib[1]. A BAM file partitions an ordering of sequence alignments into a sequence of $\leq$ 64 KiB blocks, each of which is compressed using a "gunzip-compatible" format called *BGZF* and then concatenated into a single file or data stream.

Reading, decoding, sorting, encoding, and writing large sequence alignment files (tens or hundreds of GBs) can be time-consuming and resource intensive. Initially developed as a single-threaded application, SAMtools has added pthreads-based multithreading to handle increasingly-larger sequence alignment data sets. However, performance analysis (using HPCToolkit[4]) of SAMtools 1.3.1 on an internal (in-memory) sort of a 24.6 GiB BAM file (102.6 GiB uncompressed) revealed substantial room for improvement: almost 80% of the total execution time was single-threaded, and there was much processor idle time during the multithreaded phase (see Table 1).

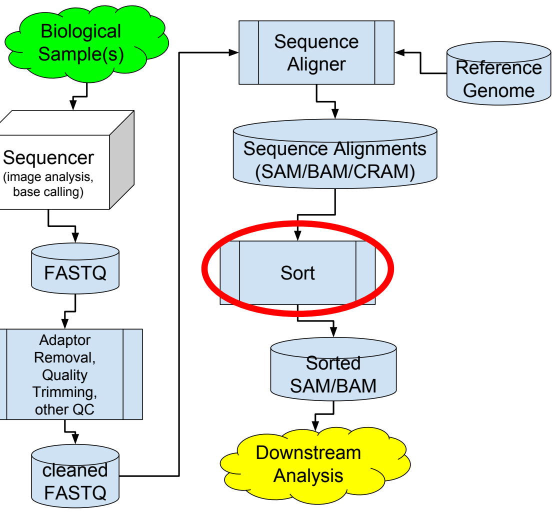
Biological Sample(s)
Sequencer (image analysis, base calling)
FASTQ
Adaptor Removal, Quality Trimming, other QC
cleaned FASTQ
Sequence Aligner
Reference Genome
Sequence Alignments (SAM/BAM/CRAM)
Sort
Sorted SAM/BAM
Downstream Analysis

Figure 1: Initial stages of a hypothetical HTS workflow

Table 1: SAMtools 1.3.1 internal sort execution profile

SAMtools Sort Execution Phase	Execution time (seconds)	Comments
Reading / Decoding	336	single-threaded
Sorting	195	single-threaded for internal sort
Encoding	175	multi-threaded (pthreads) but with low parallel efficiency
Deallocating Data Structures	111	$> 4 \times 10^{11}$ calls to free() (2 for each BAM record)

OpenMP Task Parallelism

SAMtools and HTSlib were augmented with OpenMP task parallelism as described below to:

1. Overlap decoding of BGZF blocks (HTSlib) and sorting of groups of uncompressed sequence alignment records (SAMtools)
2. Improve the parallel efficiency of BGZF encoding the resulting sorted sequence alignments (HTSlib)

A high-level overview of how this was achieved for internal sorting is described below. Additional non-OpenMP-related optimizations, as well as optimizations for external (out-of-core) sorting, are presented the associated journal article[2].

The master thread repeatedly calls an HTSlib-OpenMP routine (`bam_read1()`, described in Listing 2) to return the next sequence alignment record in the input (unsorted) BAM file. After collecting the next 220 records (an empirically chosen number), or upon reaching the end of the input file, the master thread generates a sort task to sort those $\leq 2^{20}$ records (see Listing 1).

Listing 1: Abstraction of SAMtools-OpenMP **parallel** regions for BAM file sorting

```
// Define a threadprivate merge-sort buffer for up to 2^20 pointers to BAM records
// to avoid overhead from repeated memory allocation/deallocation.
// (pthreads version mallocs and frees a buffer for each chunk sorted by each thread)
#define NBAM_PER_SORT 1048576
static bam1_t *sort_tmp[NBAM_PER_SORT];
#pragma omp threadprivate(sort_tmp)

#pragma omp parallel
#pragma omp master
while (bam_read1(...) != EOF) {
    if (/*global sort buffer full*/) {
        sort(/*current partition of <= NBAM_PER_SORT records*/, sort_tmp)
        // wait for any sort tasks to finish
        #pragma omp taskwait
        while(/*in-memory merge of sorted partitions*/)
            // generates encode() OpenMP tasks
            bgzf_write(/*output to a temporary BAM file*/)
    } else {
        if (/*NBAM_PER_SORT sequence alignment records read since last sort task*/) {
            #pragma omp task
            sort(/*current partition of NBAM_PER_SORT records*/, sort_tmp)
        }
    }
}

sort(/*last partition of <= NBAM_PER_SORT records*/, sort_tmp) // in master thread

#pragma omp parallel
#pragma omp master
while (/*merge in-memory partitions with any written to temporary file*/)
    bgzf_write(/*output to final BAM file*/) // generates encode() OpenMP tasks
```

Inside `bam_read1()`, the master thread immediately returns the next sequence alignment record if the BGZF block(s) containing it have already been read, decoded (decompressed), and cached.

Otherwise, the master thread reads ahead a fixed number of BGZF blocks, creating tasks to decode them. While a dedicated read-ahead thread could avoid input "stalls", it was not attempted due to the difficulty of implementing in OpenMP in a manner that preserved backwards-compatibility in the existing HTSlib `bam_read1()` routine.

Decoding is less computationally-intensive than encoding. How much work should each task do (i.e., how many BGZF blocks should each task decode)? Too coarse-grained, and load imbalance can result. Too fine-grained, and the overhead of task creation, task scheduling, and task switching can outweigh performance benefits from the additional exposed concurrency.

To answer this question, a task microbenchmark[5] was created to simulate the creation, scheduling, and execution (with a very lightweight kernel) of over a half million tasks (corresponding to the number of BGZF blocks in the benchmark BAM file). Compiled with GCC 6.2.0 and run with 32 threads on a NERSC Cori Haswell node, the microbenchmark completed in under 1.5 seconds. This suggested that the finest level of task granularity possible in this case—one BGZF block per task—could maximize concurrency without substantially impacting performance.

A **taskgroup** construct awaits the completion of the read-ahead tasks before the immediate next sequence alignment record is returned. A **taskgroup** is preferred over a **taskwait** construct, as the latter would also (unnecessarily) wait for the completion of concurrently-executing sort tasks generated by the master thread in the SAMtools-OpenMP code.

The resulting algorithm is presented in Listing 2.

Listing 2: Simplification of OpenMP-related code invoked during HTSlib-OpenMP `bam_read1()`

```
// Return next decoded BAM record in BGZF stream
bam_read1(...) {
    if (/*next BAM record in current decoded BGZF block*/)
        return /* next BAM record in current block */;
    else {
        // master thread checks if next BAM record is in a cached decoded BGZF block.
        // not necessary to call within an OpenMP critical section, as updates to the
        // cache are done only from the taskgroup section below
        if (load_block_from_cache(...))
            return /* next BAM record in cached decoded BGZF block */;
        else {
            // next record not in a cached decoded BGZF block.
            // In the master thread, read ahead a fixed number of BGZF blocks.
            // Decode and cache each one in a separate OpenMP task.
            #pragma omp taskgroup
            {
                while (/*read ahead a fixed number of BGZF blocks*/) {
                    #pragma omp task
                    {
                        decode(/*BGZF block*/);
                        #pragma omp critical (cache)
                            // evicts another cached block if cache buffer full
                            cache(/*decoded block*/);
                    } // task
                } // while
            } // taskgroup
            load_block_from_cache(...); // now the decoded BGZF block should be cached
            return /* next BAM record in cached decoded BGZF block */;
        } // else
    } // else
} // bam_read1()
```

After all partitions of $\leq 2^{20}$ sequence alignment records have been sorted, the master thread merges these into a circular buffer, and calls an HTSlib routine `bgzf_write()`, which buffers input data into 64 KiB blocks and (possibly repeatedly) calls an HTSlib-OpenMP routine (described here as `encode()`) that generates an OpenMP task to BGZF encode (compress) and output its input block.

The `encode()` routine (see Listing 3) coordinates the BGZF compression of blocks from an input circular buffer to an intermediate circular buffer before output occurs. If the input circular buffer is almost full (i.e., only one unused 64 KiB slot remains), the master thread spin-waits in a loop until the subsequent slot is available, blocking in `encode()`. In this case, OpenMP **taskyield** construct allows the master thread to participate in the execution of compression tasks to help drain the input circular buffer[1].

[1]At least in principle: **taskyield** is a no-op in GCC 6.2.0 (https://gcc.gnu.org/ml/gcc-patches/2011-08/msg00080.html), so the effective behavior of this loop is to spin-wait

Instead of using a dedicated writer thread/task, which would be syntactically-difficult to implement while maintaining API compatibility, the OpenMP tasks generated in `encode()` that perform compression also perform the output. A ticket-lock-based algorithm is used to serialize and order the output. A notable modification to the usual ticket lock algorithm is that instead of spin-waiting on the ticket lock when a task cannot immediately output its compressed block due to ordering constraints, the task "abandons" its compressed block for another task to output (specifically, the task that outputs the preceding block) so the executing thread can execute another task. Correspondingly, a task that has acquired the ticket lock (and entered the following **critical** section) can output not only the block it compressed, but also all subsequent consecutive "abandoned" compressed blocks, updating the ticket lock value as it proceeds.

Listing 3: Algorithm for HTSlib-OpenMP routine invoked by the master thread that generates OpenMP tasks for concurrent block compression and ordered output without a dedicated writer thread/task.

```
// Generate an OpenMP task to compress and ouptut data of length Ul
// starting at U[(*next_ticket) % NSLOTS]
encode(char U[NSLOTS][SLOT_SIZE], // (input) circular buffer for uncompressed blocks
       char C[NSLOTS][SLOT_SIZE], // circular buffer for compressed blocks before output
       int  U_len[NSLOTS],        // if U_len[slot] > 0, U[slot] is in use
       int  C_len[NSLOTS],        // if C_len[slot] > 0, C[slot] is in use
       const int Ul,              // length in bytes of input uncompressed block in
                                  // U[(*next_ticket) % NSLOTS]
       int  *next_ticket,         // sequentially-increasing ticket-lock value to order
                                  // output (initially 0)
       int  *now_serving)         // ticket-lock value of next slot in C[] to be output
{                                 // (initially 0)
    int my_ticket = (*next_ticket)++;
    int slot = my_ticket % NSLOTS; // U[slot] contains input uncompressed data
    int next_slot = (slot + 1) % NSLOTS;
    do {
        int u_len_private;
        #pragma omp atomic read
        u_len_private = U_len[next_slot];
        if (u_len_private == 0) break; // U[next_slot] is unused
        // else U[next_slot] is still in use (this algorithm leaves it unused so encode()
        // may subsequently be safely invoked). taskyield pauses task generation & allows
        // the master thread to execute other (compress) tasks instead of spin-waiting
        // (if supported by the OpenMP runtime; otherwise this may be just a spin wait)
        #pragma omp taskyield
    } while(true);
    U_len[slot] = Ul; // mark U[slot] as in use
    #pragma omp task firstprivate(U, C, U_len, C_len, now_serving, my_ticket)
    {
        size_t compressed_length;
        // BGZF encode U[slot] into C[slot]
        compress(C[slot], &compressed_length, U[slot], U_len[slot]);
        // Store compressed data length in C_len[slot] to indicate C[slot] is ready for
        // (possibly another task to) output. Need sequential consistency so another
        // thread can see updated data in C[slot].
        #pragma omp atomic write seq_cst
        C_len[slot] = compressed_length;
        int now_serving_private;
        #pragma omp atomic read
        now_serving_private = *now_serving;
        // if it is this task's C[slot] turn to be output, then enter critical section.
        // otherwise leave C[slot] for the task that outputs the previous slot in C[].
        if (now_serving_private == my_ticket)
        #pragma omp critical (encode)
        {
            // need to check the value of now_serving *again* in case another task
            // previously in the critical section already output C[slot]
            #pragma omp atomic read
            now_serving_private = now_serving;
            // if another task has not output our C[slot], then now_serving will not have
            // changed; otherwise, it is some other C[slot]'s turn, and there will
            // (eventually) be another task whose turn it is (according to now_serving)
            // waiting to enter the critical region to output it, so we exit the task region
            if (now_serving_private == my_ticket)
            // output C[slot] and any subsequent consecutive slots in C[] that were
            // "abandoned" by other tasks
            do {
                output(C[slot], compressed_length);
                C_len[slot] = 0; // mark C[slot] as having no compressed data to output
                // Mark U[slot] as unused so the master thread can write new uncompressed
                //data into U[slot] and generate a task to compress it into C[slot]
                #pragma omp atomic write
                U_len[slot] = 0;
                slot = (slot + 1) % NBLOCKS;
                #pragma omp atomic update
                (*now_serving)++; // next slot in C[]'s turn to be output
                // Get the length (in bytes) of compressed data in the next slot of C[].
                // Need sequentially-consistent atomic read so the thread executing this
                // task can see data in C[slot] that was flushed during another task's
                // sequentially-consistent atomic write to C_len[slot]
                #pragma omp atomic read seq_cst
                compressed_length = C_len[slot];
            } while (compressed_length > 0); // compressed data to output in next C[] slot
        } // critical
    } // task
} // encode()
```

Thread Affinity

A node of the benchmark system, the NERSC Cori supercomputer[6] (data partition), features two 16-core / 32-thread Intel Xeon "Haswell" processors.

At $\leq$ 32 OpenMP threads, threads were evenly distributed between processor cores on different sockets by specifying environment variables **OMP_PROC_BIND=spread** and **OMP_PLACES=cores** (**OMP_PLACES=threads** was not used due to an issue—reported to and subsequently fixed by GCC developers—concerning thread affinity on SMT processors[7]).

A ~15% slowdown was observed at 64 OpenMP threads (two per processor core). Performance analysis suggested this was due to contention between the master thread, which executes all code outside of OpenMP tasks, and the other thread sharing the same processor core. This was resolved by using 63 threads per processor core (**OMP_PLACES={0,32},{1}:31,{33}:31**), granting the master thread its own core, resulting in a modest (< 5%) performance gain vs. 32 threads.

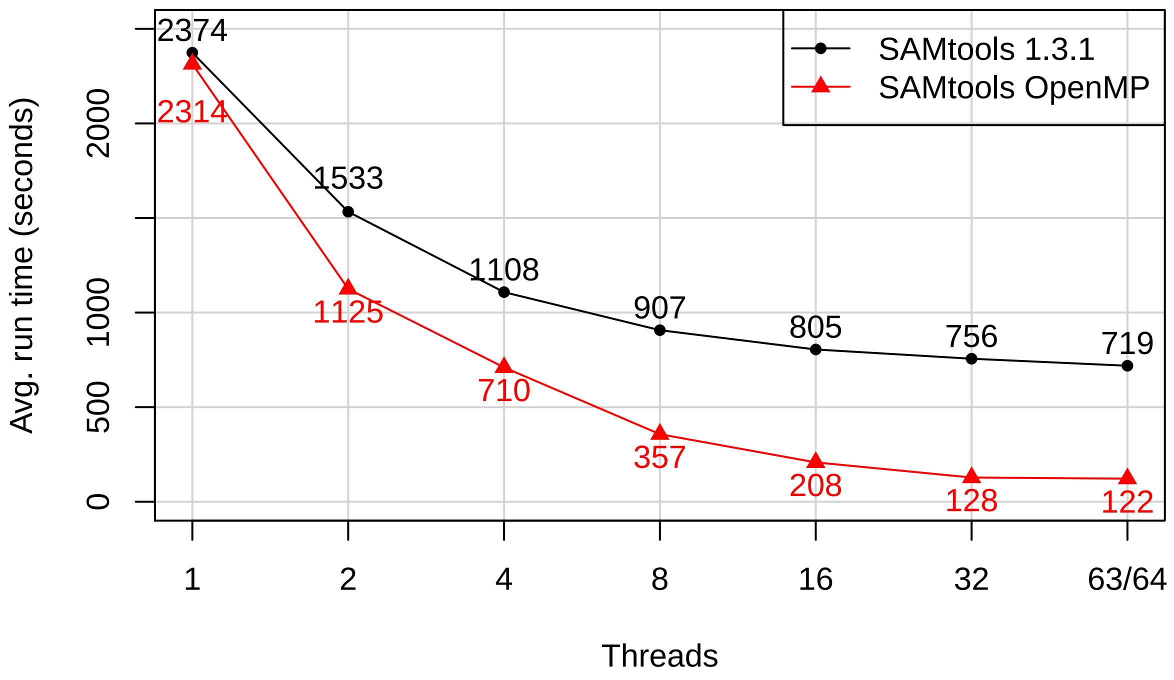

Figure 2: Internal sort of a 24.6 GiB BAM file containing alignments for approximately 207 million sequence reads

Results

BAM sorting with SAMtools

On a single node of the NERSC Cori supercomputer, SAMtools OpenMP was almost 6X faster than SAMtools 1.3.1 for the internal sort at the largest thread count (Figure 2).

An external sort of a much larger (271.4 GiB) BAM file, using the Cori Burst Buffer for input, output, and intermediate files, was about 2X faster at the largest thread count.

The replacement pthreads code with OpenMP resulted in a net reduction of about 170 source lines of code—even after the addition of code for OpenMP task parallelism where no parallelism previously existed and other optimizations. Code clarity and maintainability are generally enhanced when less code is used to express an idea.

Parallel Compression on Knights Landing

bgzip-openmp

`bgzip` is a small (< 300 SLOC) command-line utility distributed with HTSlib that performs BGZF (de-)compression of an input file. The OpenMP-enabled version (`bgzip-openmp`) is merely an insertion of two lines of OpenMP directives (see Listing 4) to invoke the OpenMP-enabled HTSlib `bgzf_write()` routine (in which `encode()` tasks are generated) on each up-to-64 KiB input block by the master thread from within an OpenMP parallel region.

Listing 4: Two OpenMP directives were added in `bgzip-openmp` to activate OpenMP-task-optimized compression in HTSlib-OpenMP.

```
#pragma omp parallel
#pragma omp master
while((bytes = read(input_file, buffer, 65536)) > 0)
    if (bgzf_write(output_bgzf_file, buffer, bytes) < 0) error(...)
```

pigz

pigz is a long-standing (1.0 release in January 2007) command-line utility dedicated to providing fast parallel gzip-compatible compression on modern multi-core processors. pigz uses pthreads to coordinate the initial ("main") thread, p compression threads, and a write thread for the compressed data.

Comparing pigz and bgzip-openmp

At their core, *bgzip-openmp* and *pigz* effectively provide a different parallel coordination layer to the same zlib compression library. To validate the scalability of the OpenMP implementation against the current "gold standard" pthreads implementation, both *bgzip-openmp* and *pigz* 2.4 were benchmarked on a NERSC Cori compute node featuring a 68-core Intel Xeon Phi "Knights Landing" (KNL) processor. Each KNL processor core supports simultaneous multithreading of 4 threads.

pigz was invoked with `--independent --blocksize 64` to mimic the BGZF format, while *bgzip-openmp* was invoked with environment variables `OMP_PLACES=threads OMP_PROC_BIND=true` to allow the OpenMP runtime to control thread affinity. To eliminate performance variance due to I/O across a shared network to a shared file system, the input 10 GiB text file staged to memory (/dev/shm), and compressed output written to /dev/null.

The results (Figure 3) show that *pigz* and *bgzip-openmp* provide comparable performance up to 1 thread per KNL processor core. (Note that the *pigz* `--processors p` option causes the creation of p compression threads, in addition to the main thread and write thread. The *bgzip-openmp* `--threads t` option controls the *total* number of OpenMP threads, with $t-1$ threads dedicated to executing compression tasks, and the master thread participating in compression task execution only if determined by the OpenMP runtime. The "advantage" of the extra compression thread in *pigz* disappears at higher thread counts.) The performance of *bgzip-openmp* sees further performance improvement at 2 threads per KNL processor core (136 threads total), whereas pigz suffers a severe performance degradation past 1 thread per core.

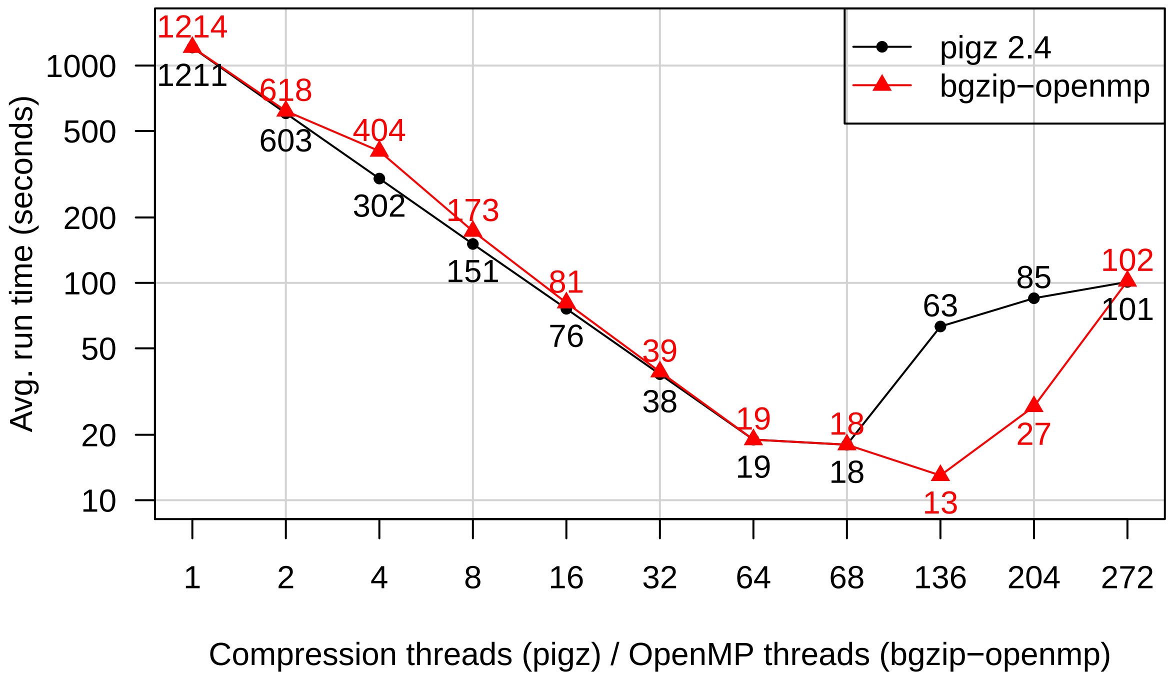

Figure 3: Compression of a 10 GiB text file on a 68-core Intel Xeon Phi (Knights Landing)

Conclusion

This proof of concept exemplifies a parallel library design pattern enabled by OpenMP whereby the library implements otherwise-latent OpenMP *task* regions that are activated by an OpenMP *parallel* region in the calling application. The resulting tasks generated by the library are executed by the same OpenMP team of threads as the calling application, allowing the calling application to create as many threads as there are hardware resources to execute them without the increased likelihood of load imbalance or oversubscription due to the library creating its own threads.

More specifically, the *bgzip-openmp* example, besides illustrating that latent OpenMP task parallelism can be trivially activated by the calling OpenMP application, suggests that an OpenMP-enabled zlib (or other compression library) exposing a high-level interface such as `bgzf_write()` could possibly be beneficial for genomics and other big-data applications that utilize compression.

Acknowledgements

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.

About the Author

Nathan T. Weeks an aspirant HPC application performance engineer and sysadmin with experience supporting computational biologists. This work is part of his ongoing Computer Science PhD dissertation research at Iowa State University.

References

[1] SAM/BAM Format Specification: *http://samtools.github.io/hts-specs/SAMv1.pdf*.

[2] Weeks, N.T. and Luecke, G.R. 2017. Optimization of SAMtools sorting using OpenMP tasks. *Cluster Computing*. 20, (Sep. 2017), 1869–1880. DOI:https://doi.org/10.1007/s10586-017-0874-8.

[3] Wetterstrand KA. DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP): *https://www.genome.gov/sequencingcostsdata*. Accessed: 2017-11-20.

[4] *http://hpctoolkit.org/*.

[5] *https://static-content.springer.com/esm/art%3A10.1007%2Fs10586-017-0874-8/MediaObjects/10586_2017_874_MOESM1_ESM.pdf*.

[6] *http://www.nersc.gov/users/computational-systems/cori/*.

[7] *https://gcc.gnu.org/bugzilla/show_bug.cgi?id=80822*.