Scaling read aligners to hundreds of threads on general-purpose processors

Ben Langmead[1,2,*], Christopher Wilks[1,2], Valentin Antonescu[1], and Rone Charles[1]

[1]Department of Computer Science, Johns Hopkins University
[2]Center for Computational Biology, Johns Hopkins University
[*]Correspondence to: langmea@cs.jhu.edu

October 24, 2017

Abstract

General-purpose processors can now contain many dozens of processor cores and support hundreds of simultaneous threads of execution. To make best use of these threads, genomics software must contend with new and subtle computer architecture issues. We discuss some of these and propose methods for improving thread scaling in tools that analyze each read independently, such as read aligners. We implement these methods in new versions of Bowtie, Bowtie 2 and HISAT. We greatly improve thread scaling in many scenarios, including on the recent Intel Xeon Phi architecture. We also highlight how bottlenecks are exacerbated by variable-record-length file formats like FASTQ and suggest changes that enable superior scaling.

1 Introduction

General-purpose processors are now capable of running hundreds of threads of execution simultaneously in parallel. Intel's Xeon Phi "Knight's Landing" architecture supports 256–288 simultaneous threads across 64–72 physical processor cores [1, 2]. With severe physical limits on clock speed [3], future architectures will likely support more simultaneous threads rather than faster individual cores [4]. In fact, clock speed on many-core Xeon Phi processors (1.3-1.5 Ghz) is about half that of the 10-core processors driving compute-intensive servers available from Amazon Web Services [5]. While specialized (e.g. graphics) processors have been highly multithreaded for some time, this only recently became true for the general-purpose processors that can boot standard operating systems and that typically power servers and desktops.

With these advances come new computer-architecture considerations for programmers. Simply adding multi-threading to a software tool does not guarantee it will use threads well. In fact, it is distressingly common for a software tool's overall throughput to *decrease* with when thread count grows large enough [6]. So whereas past efforts in computational genomics software

for general-purpose processors have focused on speed on a fixed (and usually small) number of threads, it will be important for future efforts to examine scaling to much higher thread counts.

Here we tackle the problem of scaling read aligners to hundreds of threads on general-purpose processors. We concentrate on the Bowtie [7], Bowtie 2 [8] and HISAT [9] tools because they are widely used and representative of a wider group of *embarrassingly parallel* tools, where the computation is readily separable into independent tasks, one per sequencing read. That is, the alignment of one read does not depend on the alignment of any other. [1] Many other sequencing analysis tools are also embarrassingly parallel, including tools for error correction [10, 11], quality assessment and trimming [12], and taxonomic assignment [13, 14]. This is true whether the input data consists of short (e.g. Illumina) reads or longer (e.g. nanopore) reads.

We propose strategies that scale to hundreds of threads better than alternative approaches like multiprocessing (several possibly-multithreaded processes running simultaneously) and the pipelined approach taken by BWA-MEM [15]. We also explore how the FASTQ file format [16], its unpredictable record boundaries in particular, can impede thread scaling. We suggest a way to change FASTQ files and similar formats that enable further improvements in thread scaling while maintaining essentially the same compressed file size.

Synchronization and locking For embarrassingly parallel genomics tools, threads typically proceed by repeatedly (a) obtaining the next read from the input file, (b) aligning the read, and (c) writing the resulting alignment to the output file. Threads' interactions with input and output files must be synchronized. That is, some portions of code must be protected to allow only one thread to execute at a time. Figure 1 illustrates threads operating in parallel while periodically reading input in a synchronized fashion. Failure to properly synchronize can lead to software crashes and corrupt data. Synchronization is achieved with locks. There are various lock types, which incur different types and amounts of overhead. We confirm here that for many-core architectures with non-uniform memory access (NUMA), choice of lock type has a major impact on thread scaling [17]. We explore several lock types, demonstrate their relative merits, and propose the queueing lock from the Thread Building Blocks (TBB) library [18, 19] as a choice that scales well to hundreds of threads.

Multithreading versus multiprocessing Our focus is on making the best use of many threads in a single process. But another strategy is to run multiple simultaneous processes, possibly with many threads each. For example, a user with several FASTQ files might align all the files using Bowtie 2 with the `-p 100` argument, causing Bowtie 2 to use 100 simultaneous threads of execution. Alternately, the user might divide the input into 10 batches and run 10 simultaneous Bowtie 2 processes using `-p 10`. In both cases, 100 threads run in parallel. In the latter case, threads are spread across many simultaneous processes.

[1] There can be minor exceptions to this principle. For instance, HISAT uses alignments from a preliminary batch of reads to detect putative junctions that are then used to inform spliced alignment of subsequent reads.

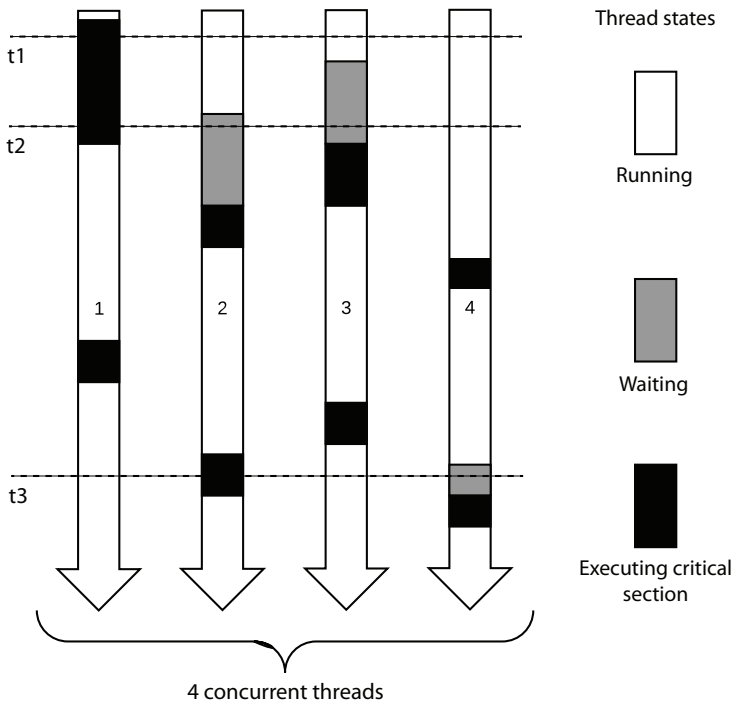

Figure 1: Four threads running simultaneously in an embarrassingly parallel setting. Time progresses from top to bottom. Gray boxes show time spent waiting to enter the critical section. Black boxes show time spent in the critical section, which can be occupied by at most one thread at a time. At time t1 (dashed line), thread 1 is executing the critical section and all the other threads are running. At time t2, thread 1 is still in the critical section and threads 2 and 3 are waiting to enter. At time t3, thread 2 occupies the critical section and thread 4 is waiting.

A trade-off exists between multithreading (MT) and multiprocessing (MP) approaches. MP can suffer from *load imbalance*: when the input is divided into batches, some batches take longer to align than others. This negatively impacts scaling since the overall job does not complete until the longest-running batch completes.

Imbalance can be mitigated by *dynamic* load balancing. Such a scheme might divide the input into many batches, more than there are processes. The load balancer then launches the processes and continually feed each process new input batches upon completion of the previous batch, until all batches are processed. As batch count increases, per-batch running times tend to average out, reducing the disparity among per-process running times. This incurs overhead; the dynamic load balancer must split the inputs, launch and feed the processes, and finally combine outputs.

Another disadvantage of MP is that many data structures, such as the genome index, are copied across processes, leading to a higher memory footprint compared to MT. Since the genome index us identical from process to process, this is a wasteful use of memory and cache space. It can also exceed available memory for aligners with larger data structures, such as GEM [20] and SNAP [21], as shown previously [6]. This can be mitigated using the memory mapping or shared-memory approaches already implemented in Bowtie, Bowtie 2 and HISAT.

The MP strategy also has a major advantage: by allowing each processes to deal only with its own private input and output files, the level of thread contention is reduced in each process. That is, a single process has fewer threads to synchronize. There are also NUMA-related reasons why running multiple processes can aid thread scaling, e.g. by allowing each process to have a copy of the genome index that is local to its home NUMA node [6].

Here we focus on improving MT thread scaling since an efficient MT scaling strategy effectively combines the advantages of MT and MP approaches. MT inherently achieves dynamic load balancing without requiring extra software beyond the aligner itself. It also achieves a low memory footprint (no duplicated data structures) without the need for memory mapping or similar mechanism. It is also applicable regardless of how many input files the user starts with, a significant advantage since users typically start with a single input file and desire a single output file. This also makes MT approaches more amenable to streaming contexts, where inputs arrive and outputs leave via individual data streams such as operating-system pipes or socket connections.

Related work Two prior studies [22, 23], examined Bowtie 2 thread scaling with synchronization and Non-Uniform Memory Access (NUMA) as primary concerns. By adapting Bowtie 2 to the FastFlow [24] parallel framework and by making effective use of (a) thread pinning and (b) interleaving of memory pages across NUMA sockets, the modifications improved Bowtie 2's thread scaling. Our suggestions for improving thread scaling are complementary to these proposals.

Herzeel et al [25] re-parallelized sections of the BWA code using the Cilk [26] programming language. They noted a 2-fold improvement in multi-threaded speedup, highlighting the importance of NUMA and load balance issues. Lenis and Senar examined performance of four read aligners, including Bowtie 2 and BWA-MEM on NUMA architectures [6] without modifications,

and noted that a multiprocessing approach that replicated the index data structure across NUMA nodes performed the best. Our goal is to achieve similar improvements with a purely multi-threaded approach on modern hardware.

2 Methods

2.1 Lock types

We begin by examining how lock types affect thread scaling. Different lock types are appropriate for different situations. A *spinlock* uses a loop to repeatedly check if a lock is held. As soon as a check indicates the lock is free, ownership is transferred to the inquiring thread. The check and the transfer can happen simultaneously using an atomic operation [27]. In practice, a spinlock that fails to obtain a lock in a prescribed time interval will simply go to sleep, allowing the operating system (OS) to revive it when the lock becomes free. This avoids *starvation*, where the lock-holding thread is slow to finish its work (and release the lock) because other threads are using the processor to probe the very same lock.

Spinlocks are *optimistic*: they work best when the lock can be obtained quickly. On the other hand, another common lock type is a *standard lock*, which the thread to sleep immediately, allowing the OS to revive it when the lock is free. While pausing and reviving a thread incurs overhead, a standard lock cannot starve other threads. Thus, a standard lock is *pessimistic*, working best when the lock is unlikely to be available soon.

We might suppose that when active thread count is less than or equal to the number of physical cores — a typical situation when a user has dedicated access to a computer and desires speed — starvation is not an issue and spinlocks are ideal. However, this supposition fails on modern many-core systems for reasons relevant to our choice of lock type. One concern is that modern architectures have many cores and caches connected in a NUMA architecture. That is, there is a single addressable memory space visible to all threads, but it is physically divided into partitions that might be attached to separate cores in a multi-socket system, as for the 2-socket Broadwell system used in our evaluations, or both the partitions and the cores might be connected via an interconnection network, as on the Xeon Phi. Either way, thread scaling is impacted in at least two ways: (a) threads running on different cores but accessing the same memory location will incur different access latencies depending on the distance to the memory, and (b) when several threads read and write the same memory location simultaneously, the system's *cache coherence protocol* must step in to ensure all threads have a coherent view of memory. In short, thread scaling is harmed when the added threads must access distant memories or when they compete for the same memory locations as existing threads. This is a general issue but it affects locking in specific ways that we return to this when discussing the queueing lock.

Another issue arises when threads can co-exist on the same physical processor. On Xeon Phi, up to four threads can run simultaneously on one processor, competing for its resources like its arithmetic units and cache. A thread operating by itself on a processor moves at one speed, but

slows when joined by a second thread, slows still further when joined by third, etc. Thus, increasing thread count incurs a mild but increasing starvation penalty even when free thread "slots" remain. This puts optimistic locks at a disadvantage, since their spinning behavior can needlessly starve productive threads on the same processor.

In past versions, Bowtie, Bowtie 2 and HISAT used a spinlock from the `TinyThread++` [28] library. Observing that this scaled poorly (see Results), we extended the three tools to use the open source Intel Thread Building Blocks (TBB) library [18]. TBB provides various lock types [29], including a queuing lock [30] particularly appropriate for NUMA systems like the Knight's Landing and Broadwell systems used here. TBB also provides scalable replacements for standard heap memory allocation functions (e.g. `malloc/free`, `new/delete`). This aids thread scaling, since memory allocations require synchronization.

The TBB queuing lock implements an MCS lock [30], which uses an in-memory queue to organize waiting threads. Like a spinlock, a waiting thread repeatedly probes a variable in memory to learn when it has obtained the lock. Unlike a spinlock, each waiting thread probes a separate queue entry, each entry occupying a separate cache line. This greatly reduces overhead. To elaborate, consider that an atomic operation (e.g. atomic compare-and-swap) might modify a variable in memory, depending on the condition. Consequently, it is treated as a memory write by the cache coherence infrastructure. A write modifies a cache line, causing cache coherence messages to travel between caches for threads that recently accessed the line. When this happens in a loop, new messages are generated each iteration. When many threads spin simultaneously, messages multiply, eventually reaching a point where the messages flood the system bus and starve other threads, including lock holder. This is called *cache-line* or *hotspot* contention [30] and it is a major concern on many-core and NUMA systems [17]. The queuing lock reduces contention in two ways. First, since each thread spins on a variable in a thread-specific cache line, the loop condition can be a simple memory read rather than an atomic operation. This reduces cache coherence messaging. Second, while a memory write is still needed to hand the lock from one thread to another, only two threads are involved in the hand-off, reducing the coherence messages exchanged.

We adapted the three tools to use four lock types: the (original) `TinyThread++` lock, standard TBB lock, TBB spinlock, and TBB queuing lock. On the Linux systems we used for evaluation, the standard TBB lock works by calling `pthread_mutex_unlock`, which in turn uses the Linux `futex` (fast mutex) strategy. This strategy first attempts to obtain the lock using a fast atomic operation then, if unsuccessful (i.e. if the lock is held by another thread), places it on a queue of paused threads until the lock is released.

The lock type is selected at compile time via preprocessing macros. These extensions are available as of the Bowtie v1.1.2, Bowtie 2 v2.2.9 and HISAT v0.1.6-beta software versions. Supplementary Note 1 gives build instructions for the exact software versions tested here.

2.2 Parsing strategies

We also examined how threads coordinate when reading input (e.g. FASTQ reads [16]) or writing output (SAM [31] alignments). These interactions are synchronized, i.e. protected by locks. The name *critical section* is given to a portion of the software that only one thread may execute at a time. The critical section for handling input is called the *input critical section* and is protected by the *input lock*. Likewise for the *output critical section* and the *output lock*.

We hypothesized that to improve thread scaling we should restructure the input and output critical sections. Our first goal was to reduce the time spent in the critical section by deferring as much computation until after the critical section as possible. Our second goal was to reduce the total number of times the critical section was entered. This reduces overhead incurred by locking and unlocking upon entering and exiting.

The original strategy (O-parsing) both reads and parses a sequencing read in the critical section (CS). We developed three variants on this approach (Table 1). In deferred (D) parsing, the CS reads a single input record into a buffer. After the CS, the buffer is parsed into the sequencing read data object. Batch deferred (B) parsing is like D-parsing but handles batches of N reads at a time. The B-parsing critical section has a loop that iterates N times, reading each record into a separate buffer. After the CS, another loop parses each buffer into a sequencing read object. This reduces by a factor of N the total number of times the CS is entered. A similar change is made to the output CS: alignment records are written to the output stream in batches of N reads.

Blocked deferred (L) parsing reads a chunk of exactly B input bytes into a buffer, assuming that (a) no read spans a B-byte boundary in the input file, such that no B-byte chunk contains a partial input record, and (b) the number of reads (N) per B-byte chunk is the same for all chunks (except perhaps the last) and known ahead of time. These assumptions do not hold for real FASTQ files, but we can easily modify a FASTQ file to comply. We accomplish this by appending extra space characters to every Nth read until the following read begins at an B-byte boundary (Figure 2). (Trailing spaces are later removed by the aligner.) This has the effect both of enforcing the L-parsing assumptions and of making it easier to parse paired-end files in a synchronized manner, since a B-sized block taken from the same offset in both files will be guaranteed to contain N matching ends. As with B-parsing, the L-parsing output critical section writes alignments in batches of B reads at a time.

2.3 Other aligner modifications

We also modified the aligners to minimize the incidence of heap memory allocations wherever possible. This is because heap memory allocations also require synchronization, thus negatively impacting thread scaling. We also modified each of Bowtie, Bowtie 2 and HISAT to cause each thread to report how much wall-clock time it spends aligning reads, with microsecond accuracy.

(Original) O-parsing

```
obtain lock
read = read_and_parse()
release lock
```

(Deferred) D-parsing

```
buf = new empty buffer
obtain lock
nl = 0
while nl < 4:
    c = read_char()
    if c is newline:
        nl = nl + 1
    append c to buf
release lock

read = parse_fastq(buf)
```

(Batch deferred) B-parsing

```
bufs = array of N empty buffers
reads = array of N empty reads
obtain lock
for i = 1 to N:
    nl = 0
    while nl < 4:
        c = read_char()
        if c is newline:
            nl = nl + 1
        append c to bufs[i]
release lock

for i = 1 to N:
    reads[i] = parse_fastq(bufs[i])
```

(Block deferred) L-parsing

```
reads = array of N empty reads
obtain lock
buf = read B bytes
release lock

for i = 1 to N:
    reads[i] = parse_fastq(buf)
    advance buf to next read
```

Table 1: Pseudocode for four synchronized parsing strategies. Red code is inside the critical section (CS). Original (O) parsing both reads and parses in the CS. Deferred parsing (D) uses the CS to read the next record into a buffer, counting four newlines to find the record boundary, but defers parsing until after the CS. Batch deferred parsing (B) is like (D) but reads N reads at a time. Block deferred parsing (L) reads a fixed-sized chunk of data (B bytes), assuming that no record spans a B-byte boundary. While the assumption for (L) is violated in practice for formats like FASTQ, it suggests a strategy for making formats more amenable to multithreaded parsing.

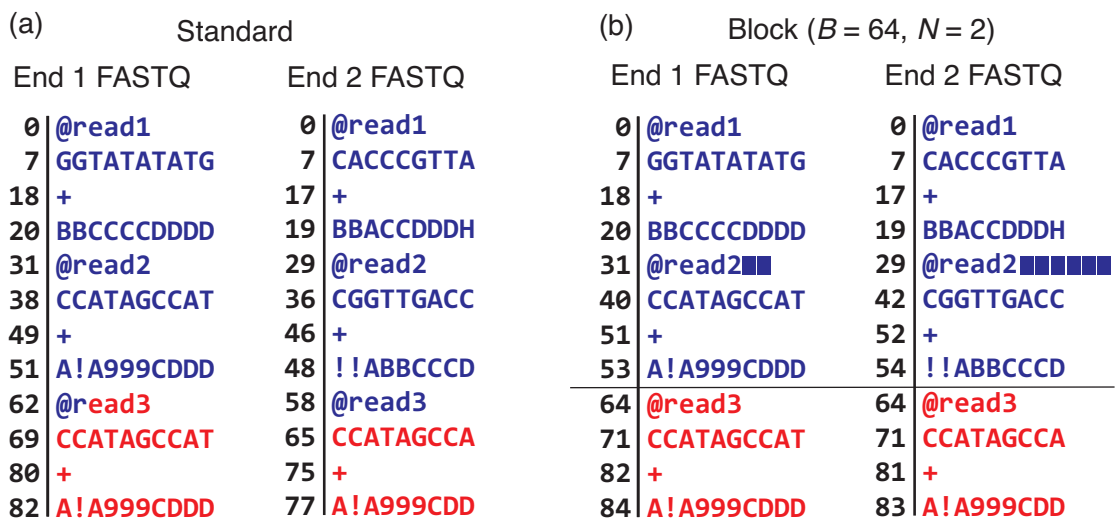

Figure 2: Converting a standard pair of FASTQ files (a) to blocked FASTQ files (b), where the number of bytes (B) and number of input read per block (N) are 64 and 2 respectively. Numbers left of vertical lines indicate byte offsets for FASTQ lines, assuming newline characters (not shown) are one byte. For (b), padding spaces are represented by solid blue rectangles. The first 64 bytes of each file are colored blue and subsequent bytes are colored red. Note that the two ends differ in length; end 1 is 10 bases long and end 2 is 9 bases long. This necessitates differing amounts of padding in the two FASTQ files. But after padding, we are guaranteed that corresponding 64-byte blocks from the files contain N corresponding reads.

2.4 Multiprocessing

While our focus is on single-process multithreaded (MT) approaches, multiprocessing (MP) is another avenue for improving thread scaling. For this reason, our comparisons include an MP-based "baseline" strategy. The MP baseline is measured for every thread count T that is multiple of 16 by running $T/16$ processes, each with 16 threads. For MP experiments involving Bowtie, Bowtie 2 or HISAT, we use memory mapping (`--mm` option) to limit overall memory footprint.

3 Results

Configurations & jobs We evaluate various read aligners and synchronization schemes by running each "configuration" (combination of aligner and synchronization scheme) using the same input data. For each configuration, we perform a series of alignment *jobs* varying the number of input reads and the number of simultaneous threads of execution in direct proportion, thus keeping the number of reads per thread constant. In each case we align to an index of the GRCh38 human genome reference assembly [32]. We measure wall-clock running time of each job, omitting time required for one-time setup tasks such as index loading since these influence thread scaling only slightly when aligning large datasets. The number of reads per thread (Supplementary Table 1) was chosen for each configuration and system so that most jobs take 1 minute or longer.

Any job taking longer than 20 minutes was aborted and omitted from the results. The Linux `top` utility was run in the background to periodically measure system load, processor utilization and memory footprint.

We evaluate Bowtie [7], Bowtie 2 [8] and HISAT [9] because they are widely used. We include a comparison to BWA-MEM [15] for the same reason. While we did not modify the newer HISAT2 [33], we expect the same modifications to benefit that software as well. We ran HISAT with the `--no-spliced-alignment --no-temp-splicesite` options to disable gathering of splice-site evidence because our input reads were from DNA sequencing experiments.

All software used to run the experiments and produce the figures and tables are located at https://github.com/BenLangmead/bowtie-scaling.

Reads We obtained sequencing reads from accessions ERR194147 (Platinum Genomes Project [34]), SRR069520 (1000 Genomes Project [35]) and SRR3947551 (a low coverage whole genome sequencing project [36]). All reads are 100 x 100 nt (paired-end) from the Illumina HiSeq 2000 instrument. We downloaded the reads in FASTQ format, selected a random subset of 100M from each of the 3 accessions, then randomized the order of the resulting set of 300M reads to avoid clustering of reads with similar properties. These constitute the *human_100_300M* input read set. Bowtie is designed to align shorter reads, so we also created set *human_50_300M* consisting of the *human_100_300M* reads truncated to 50 nt at the 3′ end. Unpaired alignment experiments use just the first-end FASTQ files. Download links for reads are in Supplementary Note 2.

Evaluation systems Each job was run on two servers. The first (*Broadwell*) is a dual-socket system with two Intel Xeon E7-4830 v4 2.00GHz CPUs and 1 TB of DDR4 memory. Both CPUs have 28 physical processor cores, enabling up to 112 threads of execution since each core supports 2 simultaneous "hyperthreaded" threads. The system runs CentOS 6.8 Linux, kernel v2.6.32, and is located at the Maryland Advanced Research Computing Center (MARCC). The second system (*KNL*) is an Intel Xeon Phi 7250 (Knight's Landing) system with 96GB DDR4 memory (as well as a 16GB high-speed MCDRAM). The system has 68 physical processor cores, enabling up to 272 threads of execution, 4 per core. This system runs Red Hat Enterprise Linux 7, kernel v3.10.0, and is located in the Stampede 2 cluster at the Texas Advanced Compute Center (TACC) accessible via the XSEDE network.

3.1 Varying lock type

As discussed in Methods, we extended Bowtie, Bowtie 2 and HISAT to use one of four lock types: a `TinyThread++` spinlock, TBB standard lock, TBB spinlock, or TBB queueing lock. We tested each in a pure multithreading (MT) context; for each run, we launched a single aligner process configured via the `-p` option to use the desired number of simultaneous threads, T. At multiples of 16, we also compared to a multiprocessing (MP) baseline that launched $T/16$ simultaneous processes, each using 16 threads. The MP baseline uses the TBB queueing lock.

When plotting results, we arranged thread count on the horizontal axis and maximum per-thread wall-clock time (i.e. time required to align all reads) on the vertical axis. Because we vary the number of input reads in direct proportion to thread count, a flat horizontal line would indicate perfect scaling. Because we omit runs with a running time of over 20 minutes, lines for poorly-scaling configurations can "fall off" the top of the plot.

Figure 3 shows how thread count affects running time for unpaired alignment. Supplementary Figure 1 shows the same for paired-end alignment. We observe that the MP baseline generally outperformed all multithreading modes (MT). We also see the choice of lock type can have a major impact on scaling, seen clearly for the Broadwell+HISAT configurations. While no lock type performed best in all cases, the TBB queueing lock tended to eventually outperform the other MT configurations at high thread count. This is clearest for HISAT and Bowtie. There were also cases where the TBB standard lock or spinlock outperformed the queueing lock at high thread counts. This is clearest for Broadwell+Bowtie, where the TBB spinlock performed best even at the highest thread counts, and for KNL+Bowtie 2, where the standard lock outperformed.

Table 2 shows peak throughputs (also represented by squares in Figure 3) for each lock type and the MP baseline. For 6 out of 12 combinations of aligner, test system and paired-end status, the queueing lock has the second-highest peak throughput after the MP baseline. In the remaining six, queueing lock throughput is close to (never $> 10\%$ lower than) the best MT configuration.

In some cases, the queueing lock took a steep turn upwards at the highest thread counts, e.g. for Broadwell+Bowtie and Broadwell+Bowtie 2. This contrasts with the TBB standard lock, which also tended to rise at higher thread counts, but more slowly, almost linearly. This is likely due to starvation; at high thread counts, threads share cores (up to 2 threads per cores on Broadwell, 4 on KNL), so optimistic lock types like the queueing lock will tend to spin fruitlessly on contended locks, starving other threads, including the lock holder. The problem is not shared by the (pessimistic) standard lock, which avoids starvation and so deteriorates more slowly.

It is not surprising that even the best-scaling configuration — the MP baseline — had an upward trend in running time. Increasing thread count increases contention for shared resources, slowing the threads on average. For example, higher thread count leads to greater contention for shared memory, e.g. L1 and L2 caches, translation look-aside buffer, and computational hardware, e.g. arithmetic and vector processing units. This was more pronounced on the KNL system where up to 4 threads can share a processor.

Divergence between the lock-type scaling behaviors was lower for Bowtie 2 than for the other tools. This is likely because Bowtie 2 requires more time to align a single read. This spreads locking attempts out over time and thereby reduces contention. Thus, Bowtie 2's lower divergence is consistent with the theory that differences come primarily from contention overhead.

In summary: while the MP baseline outperformed all MT configurations, the TBB queueing lock often scaled best, with the TBB spin and standard locks doing well or better in some situations. We use the queueing lock in subsequent Bowtie, Bowtie 2 and HISAT experiments.

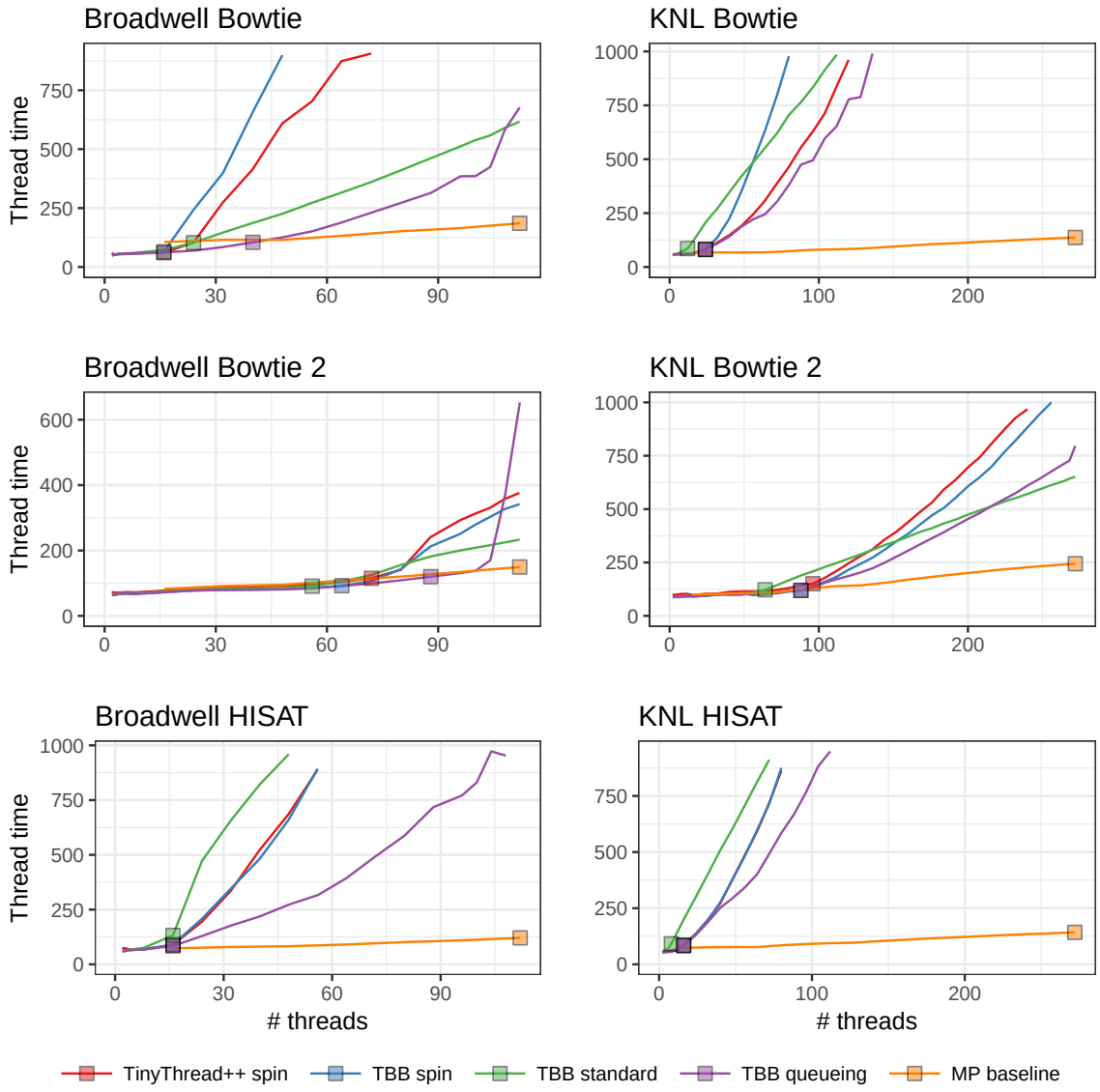

Figure 3: Comparison of 4 lock types and multiprocessing baseline. Reads are unpaired. Results are shown for three aligners (rows) and two systems (columns). Jobs that ran for over 20 minutes are omitted. Squares indicate the point on each line yielding maximal total alignment throughput. These points are summarized in Table 2.

		Broadwell (112 threads)				KNL (272 threads)			
		Paired		Unpaired		Paired		Unpaired	
		Threads	K reads/sec	Threads	K reads/sec	Threads	K reads/sec	Threads	K reads/sec
Bowtie	TinyThread++	80	92.91	16	263.38	72	41.99	24	132.20
	TBB spin	88	102.89	16	249.94	72	43.25	24	132.90
	TBB standard	72	84.04	24	232.55	40	22.92	12	63.12
	TBB queueing	88	101.70	40	382.27	64	40.22	24	132.37
	MP baseline	112	109.93	112	603.46	272	73.06	272	893.87
Bowtie 2	TinyThread++	80	51.34	72	125.46	96	14.02	96	41.48
	TBB spin	72	57.51	64	138.69	96	18.15	88	47.70
	TBB standard	56	49.14	56	123.23	72	12.30	64	34.02
	TBB queueing	72	57.68	88	146.90	88	16.62	88	48.12
	MP baseline	112	58.60	112	149.76	272	27.81	272	72.38
HISAT	TinyThread++	16	82.76	16	217.24	12	9.23	16	74.50
	TBB spin	16	85.74	16	211.77	12	29.94	16	77.29
	TBB standard	8	56.56	16	145.17	6	16.23	8	35.09
	TBB queueing	16	92.77	16	227.93	12	28.54	16	76.29
	MP baseline	112	522.95	112	1111.65	272	377.66	272	763.70

Table 2: Peak throughputs for four lock types. For each row, maximal peak throughput (K reads/sec) and number of threads (Threads) that achieved peak throughput are reported. For each combination of aligner, paired-end status and test system, the best and second-best throughputs are highlighted in red and orange respectively.

3.2 Varying parsing method

The gap between MP baseline and MT methods spurred us to examine the highly contended input and output locks. We hypothesized the gap was due to a combination of (a) the length of time spent in these critical section, and (b) the overhead of locking and unlocking before and after these critical section. We tried to close the gap using the strategies discussed in Methods: deferred parsing (D), batch parsing (B). B-parsing used a batch size of 32 in all experiments.

Figure 4 shows running time versus thread count for unpaired alignment using each parsing strategy. Supplementary Figure 2 shows the same for paired-end alignment. A clear ordering exists among the strategies: B-parsing outperformed D-parsing, which outperformed O-parsing. This was true in every scenario tested. B-parsing scaled well enough to be competitive with the MP baseline in multiple scenarios, e.g. for Bowtie 2, for Broadwell+Bowtie, and for all paired-end scenarios apart from KNL+HISAT.

Table 3 shows peak throughputs (also represented by squares in Figure 4) for each lock type and the MP baseline. B-parsing had either the highest or second-highest peak throughput in all scenarios except paired-end KNL+Bowtie, where it performed slightly worse than D-parsing. The MP baseline had the highest peak throughput in 6 of 12 scenarios. Importantly, there was still a wide gap between the MP baseline and the second-best throughput (achieved by B-parsing) in a few scenarios, including KNL+HISAT and unpaired KNL+Bowtie.

As with the lock type investigation, we find that divergence between the parsing strategies was lower for Bowtie 2 than for the other tools. Again, this is likely because Bowtie 2 spends more time on aligning each read compared to the other tools, reducing contention.

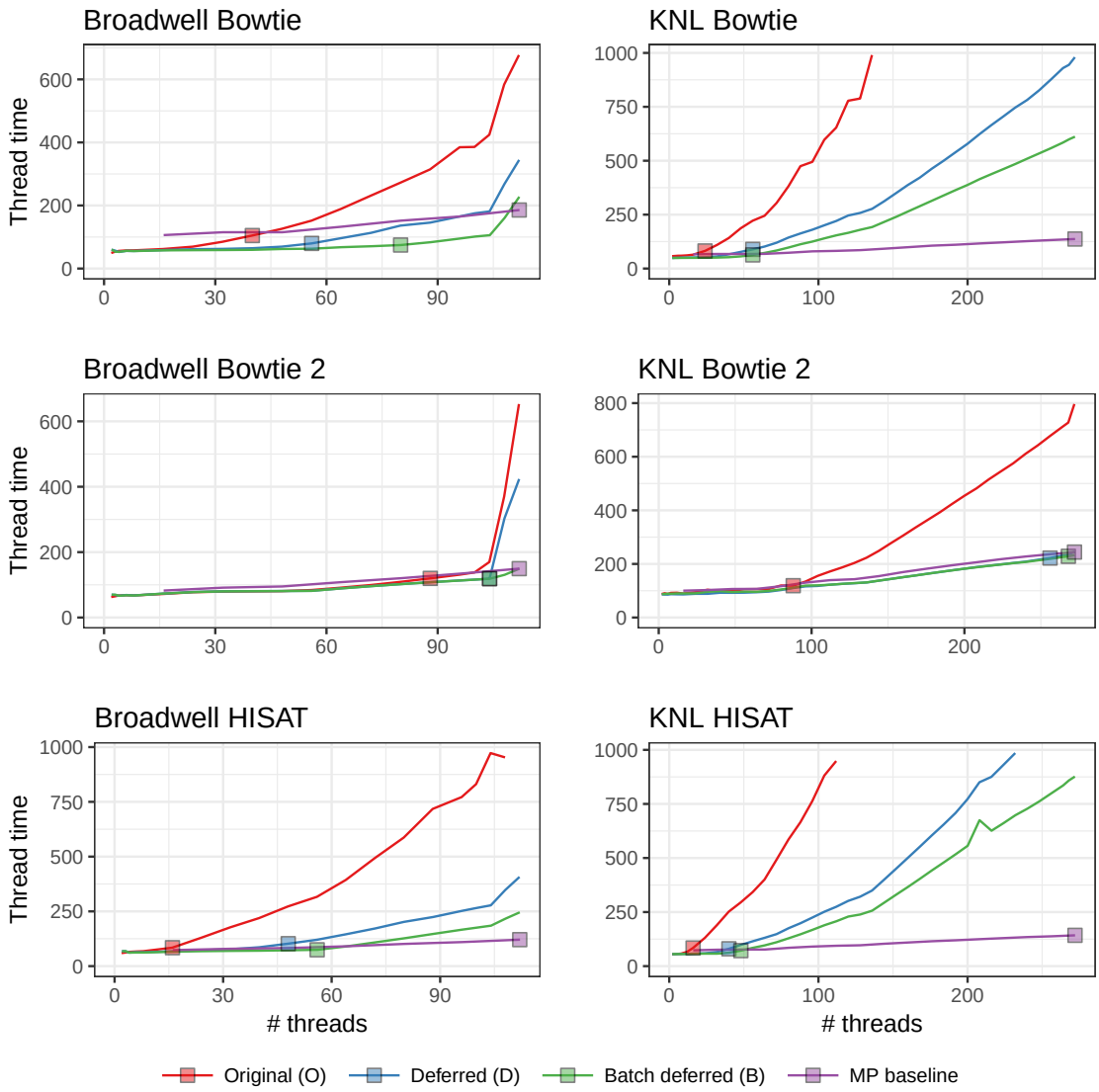

Figure 4: Comparison of 3 parsing strategies and multiprocessing baseline. Reads are unpaired. Jobs that ran for over 20 minutes are omitted. Squares indicate the point on each line yielding maximal total alignment throughput and these points are summarized in Table 3.

		Broadwell (112 threads)				KNL (272 threads)			
		Paired		Unpaired		Paired		Unpaired	
		Threads	K reads/sec	Threads	K reads/sec	Threads	K reads/sec	Threads	K reads/sec
Bowtie	O-parsing	88	101.70	40	382.27	64	40.22	24	132.37
	D-parsing	104	119.77	56	702.24	168	71.85	56	283.60
	B-parsing	104	119.64	80	1073.21	168	71.63	56	398.26
	MP baseline	112	109.93	112	603.46	272	73.06	272	893.87
Bowtie 2	O-parsing	72	57.68	88	146.90	88	16.62	88	48.12
	D-parsing	104	64.93	104	174.77	272	29.22	256	74.99
	B-parsing	104	64.89	104	175.11	272	29.06	268	76.22
	MP baseline	112	58.60	112	149.76	272	27.81	272	72.38
HISAT	O-parsing	16	92.77	16	227.93	12	28.54	16	76.29
	D-parsing	56	320.01	48	561.67	56	103.65	40	199.25
	B-parsing	56	441.95	56	900.78	64	133.53	48	267.50
	MP baseline	112	522.95	112	1111.65	272	377.66	272	763.70

Table 3: Peak throughputs for three parsing strategies. For each of combination of aligner, paired-end status and test machine, the best and second-best throughputs are highlighted in red and orange respectively. In all cases the TBB queueing lock type is used.

3.3 Final evaluation

Finally we compared B-parsing to L-parsing, i.e. block deferred parsing. Because L-parsing's critical section is the most streamlined, we hypothesized that it would outperform B-parsing. But recall that L-parsing requires the input to be padded, requiring an initial pass over the input reads that might itself become the throughput bottleneck depending on the scenario. We return to the important question of L-parsing practicality in the Discussion section below.

To test block parsing (L) we created new input sets with appropriate padding (example in Figure 2). Specifically, we created a new input set called *human_100_block_300M* with the same reads as *human_100_300M* but with the FASTQ for both ends padded to achieve 12 KB blocks ($B = 12288$) and 44 reads per block ($N = 44$). Similarly, we created an input set called *human_50_block_300M* with the reads from *human_50_300M* padded to achieve 12 KB blocks ($B = 12288$) and 70 reads per block ($N = 70$). N and B are provided to the read aligner via command-line options (`--block-bytes` and `--reads-per-block`) added for these experiments.

In the case of Bowtie 2, we also compared to BWA-MEM v0.7.16a [15] with default arguments. BWA-MEM uses a pipelined strategy. Two master threads run simultaneously, each cycling through three steps: (a) parsing a batch of input reads, (b) aligning the batch, and (c) writing the output alignments for the batch. The two master threads cannot be in the aligning step at the same time, creating a degree of interleaving; one thread can be aligning while the other is writing output or reading input. This synchronization is enforced by a pthreads [37] lock and condition variable. When a master thread enters the alignment step, it spawns T alignment threads with T specified via the `-t` option. A work stealing mechanism enables dynamic load balancing among alignment threads, with synchronization facilitated by atomic operations. Batch size is determined by multiplying a constant number of input bases (10 million) by T. We note that (a) these are relatively large batches compared to Bowtie 2, which uses a batch size of 32 reads for B-parsing and

		Broadwell (112 threads)				KNL (272 threads)			
		Paired		Unpaired		Paired		Unpaired	
		Threads	K reads/sec	Threads	K reads/sec	Threads	K reads/sec	Threads	K reads/sec
Bowtie	B-parsing	104	119.64	80	1073.21	168	71.63	56	398.26
	MP baseline	112	109.93	112	603.46	272	73.06	272	893.87
	L-parsing	104	119.70	104	1239.35	168	72.02	268	1132.59
Bowtie 2	B-parsing	104	64.89	104	175.11	272	29.06	268	76.22
	MP baseline	112	58.60	112	149.76	272	27.81	272	72.38
	L-parsing	108	66.74	104	178.86	272	29.22	272	78.71
	BWA-MEM	108	50.51	104	134.04	192	21.31	200	52.82
HISAT	B-parsing	56	441.95	56	900.78	64	133.53	48	267.50
	MP baseline	112	522.95	112	1111.65	272	377.66	272	763.70
	L-parsing	104	661.59	104	1373.97	268	489.03	268	1000.84

Table 4: Peak throughputs for three parsing strategies. For each of combination of aligner, paired-end status and test machine, the best and second-best throughputs are highlighted in red and orange respectively. In all cases the TBB queueing lock type is used.

of either 44 or 70 reads for L-parsing depending on read length, and (b) that, while the batch size is independent of thread count for Bowtie 2, it grows linearly with thread count in BWA-MEM.

For our BWA-MEM experiments, we used the same number of input reads as for the Bowtie 2 experiments (Supplementary Table 1). We also corrected an issue in the BWA-MEM code that caused failures for thread counts over 214, a limit we exceed on KNL (Supplementary Note 3).

Figure 5 shows the comparison for unpaired alignment and Supplementary Figure 3 shows the same for paired-end alignment. Table 4 gives the maximal peak throughput for each configuration. While L- and B-parsing scaled similarly at low thread counts, L-parsing maintained excellent scaling through higher thread counts in all configurations. B-parsing scaled substantially worse than L-parsing for HISAT and for unpaired Bowtie.

Remarkably, L-parsing scaled better than the MP baseline in most cases. Additionally, L-parsing had the highest peak throughput (Table 4) in 11 out of the 12 scenarios. Thus, L-parsing is the only approach we have found that consistently scales better than the MP baseline.

While BWA-MEM scales well, both the B-parsing and L-parsing Bowtie 2 configurations scaled better. This is particularly true on the KNL system and is also supported by the peak throughputs (Table 4), where B-parsing achieved 28–44% higher throughput and L-parsing achieved 32–49% higher throughput than BWA-MEM across the four scenarios. BWA-MEM's larger input chunk size, together with the fact that chunk size scales linearly with thread count, caused BWA-MEM's memory footprint to grow much faster than Bowtie 2's (Supplementary Figure 4).

4 Discussion

General-purpose processors now support hundreds of simultaneous threads of execution and future architectures will likely continue the trend of squeezing more relatively slow threads onto a single chip. Genomics software must adapt to high thread counts, slow individual threads, and

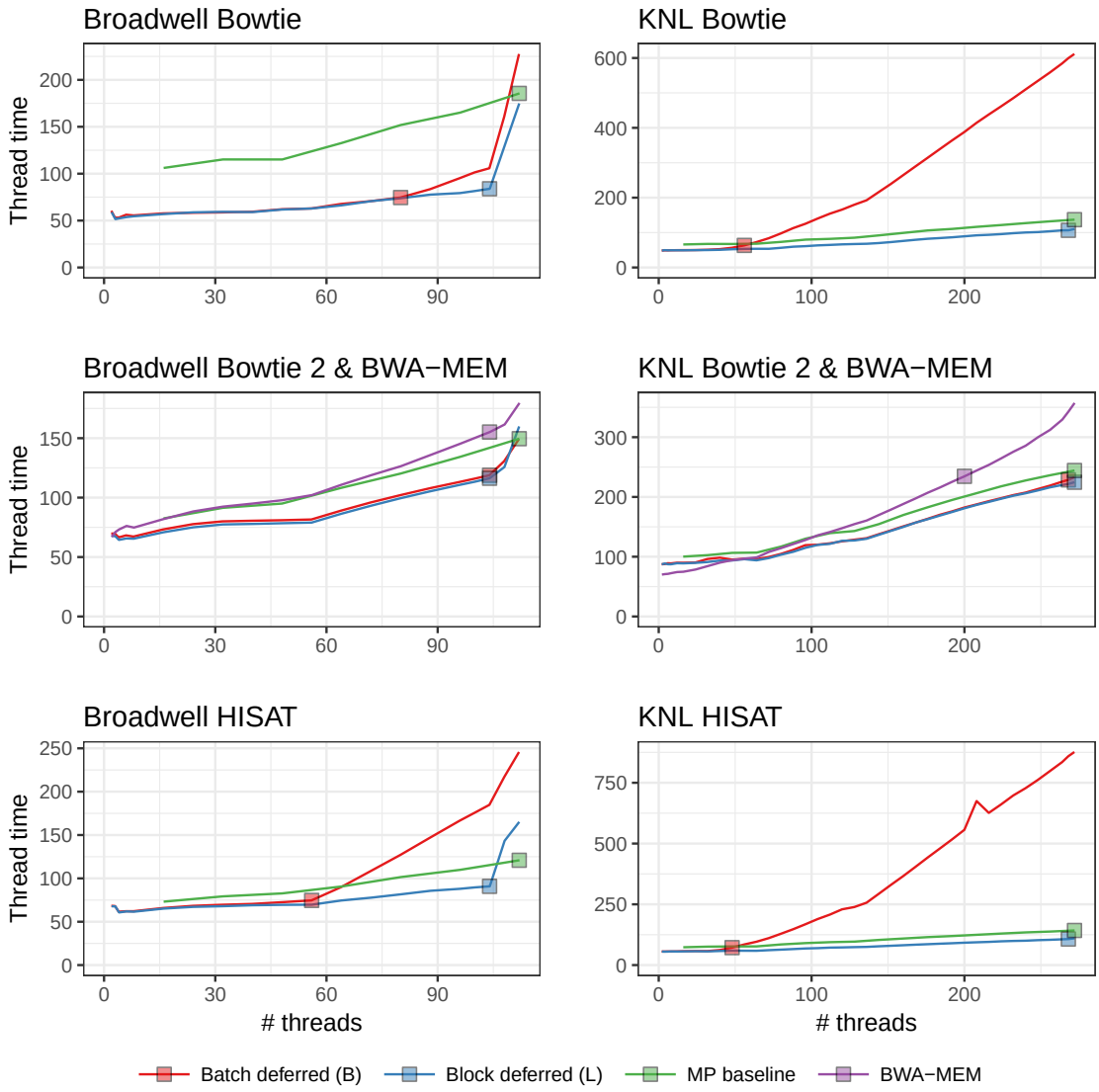

Figure 5: Comparison of BWA-MEM, the multiprocessing baseline, and the 2 most advantageous multithreading configurations tested. Unpaired alignment was performed. Jobs that ran for over 20 minutes are omitted. Squares indicate the run for each configuration that yields the greatest overall alignment throughput. These points are summarized in Table 4.

system architectures that more closely resemble small computer clusters — complete with interconnection network and distributed storage — than simpler processors of the past.

We addressed how lock types, design of critical sections, NUMA, starvation and other issues can impact thread scaling on two Intel systems, including one based on the many-core Knight's Landing architecture. We greatly improved thread scaling for three commonly used alignment tools: Bowtie, Bowtie 2 and HISAT. We measured the effect of each candidate improvement, and also showed that the improvements to Bowtie 2 allow it to scale more favorably than BWA-MEM with respect to both time and peak memory footprint. The TBB queueing lock and the B-parsing method are the default as of Bowtie v1.2.0 and Bowtie 2 v2.3.0.

Bowtie and HISAT align reads more quickly than Bowtie 2 or BWA-MEM (Table 4), making their locks more contended and thread scaling more difficult. This is reinforced by how little L-parsing improved Bowtie 2's scaling, compared to how much it improved scaling for Bowtie and HISAT. So while we improved a range of tools in this study, greater gains may be possible by adapting these methods to even faster tools, such as pseudoaligners [38], quasi-mappers [39] and tools that analyze reads based on k-mer content [14].

Despite the gains shows here, there is further room for improvement. For example, the best-scaling MT method that works with standard unpadded sequencing reads is B-parsing, which is why B-parsing is now the default strategy (coupled with the TBB queueing lock) in Bowtie and Bowtie 2. But the MP baseline outperformed B-parsing in some scenarios, and L-parsing outperformed it in all scenarios. We are still seeking MT methods that scale as well as the MP baseline or L-parsing but that also work with unpadded inputs.

To improve MT scaling further, it will be important to investigate more lock types. The queueing (MCS [30]) lock scaled best at high thread counts, likely because of reduced cache-coherence communication. But other lock types could improve on this in two ways. First: like the spinlock, the queueing lock is optimistic. But at high thread counts, locks become more contended and pessimism is more appropriate. It will be important to investigate lock types that adapt their degree of optimism in inverse proportion to the lock contention, e.g. the hierarchical backoff lock [40]. Secondly, while the queueing lock successfully reduces cache-coherence communication, other locks go further in this regard. The cohort lock [41], for example, further reduces communication by maximizing the chance that consecutive holders of the lock are physically proximate (i.e. on the same NUMA node), avoiding expensive longer-distance communication.

We argued that genomics file formats, notably FASTQ and FASTA, presented obstacles to further thread scaling improvements. Because records do not have a predictable length, boundaries between records must be identified in a synchronized manner, i.e. inside a critical section. By adapting FASTQ to have predictable record boundaries and combining that with our L-parsing strategy, we achieve the best scaling of any method evaluated here, including the MP baseline.

The padding needed for L-parsing is easy to add and works regardless of the reads' paired-end status or length (including mixed lengths within a file and/or between paired ends), as long as the block size B is large enough to accommodate the longest read. This suggests a strategy of pre-

padding FASTQ files before feeding them to L-parsing aligners. But this might simply move the synchronization bottleneck into the padding software. Software that adds padding must detect (and manipulate) record boundaries, similarly to the D-parsing and B-parsing critical sections. As we established, that makes it difficult to scale to a large number of threads. If the padding software and the downstream read alignment software are connected in a pipeline, the aligner can only be fed at the rate at which padding is added. It may still be worth the cost, though, if we expect the same input file to be re-used across multiple L-parsing alignment jobs. This effectively amortizes the computational cost of the padding over many downstream analyses, each benefiting from L-parsing's superior thread scaling.

Compression is also a key issue for future work. Because the padding added for L-parsing is repetitive (strings of space characters), compression can greatly reduce the padding's added cost. For instance, the padding added to our input sets increased uncompressed FASTQ file size by 9–14%, but the compressed FASTQ file size increased just 1.0–1.5% (Supplementary Table 2). More generally, it is common to store sequencing reads in a compressed form, then decompress — e.g. with `gzip` or the `libz` library — prior to read alignment. But if decompression must be performed either upstream of the read aligner or in the aligner's input critical section, decompression is liable to become a new thread-scaling bottleneck. A possible workaround is similar to the idea behind D-parsing: instead of both reading and decompressing in the critical section, decompression could be deferred until after the critical section. A question for future work is whether compressed inputs can be used to reduce the space overhead of padding while still providing thread scaling similar to what we achieved here with L-parsing and uncompressed inputs.

5 Acknowledgments

We are grateful to many individuals at Intel, and the Intel Parallel Computing Center program, for their technical and administrative assistance, including John Oneill, Ram Ramanujam, Kevin O'leary, Lisa Smith and Brian Napier. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562.

6 Funding

BL, VA and CW were partly supported by an Intel Parallel Computing Center grant to BL. BL, CW and RC were supported by National Institutes of Health/National Institute of General Medical Sciences grant R01GM118568 to BL. Experiments on the KNL system used the XSEDE Stampede 2 resource at the Texas Advanced Computing Center (TACC), which we accessed using allocation TG-CIE170020 to BL.

References

[1] Avinash Sodani. "Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor". In: *Hot Chips 27 Symposium (HCS), 2015 IEEE*. IEEE. 2015, pp. 1–24.

[2] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[3] M Mitchell Waldrop. "The chips are down for Moore's law". In: *Nature News* 530.7589 (2016), p. 144.

[4] Pedro Valero-Lara, Abel Paz-Gallardo, Manuel Prieto-Matías, Alfredo Pinelli, Erich L Foster, and Johan Jansson. "Multicore and Manycore: Hybrid Computing Architectures". In: *Innovative Research and Applications in Next-Generation High Performance Computing* (2016), p. 107.

[5] Jeff Barr. *New Compute-Optimized EC2 Instances*. https://aws.amazon.com/blogs/aws/new-c4-instances/. [Online; accessed 20-Oct-2017]. 2014.

[6] Josefina Lenis and Miquel Angel Senar. "A performance comparison of data and memory allocation strategies for sequence aligners on NUMA architectures". In: *Cluster Computing* 20.3 (2017), pp. 1909–1924.

[7] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome". In: *Genome biology* 10.3 (2009), p. 1.

[8] Ben Langmead and Steven L Salzberg. "Fast gapped-read alignment with Bowtie 2". In: *Nature methods* 9.4 (2012), pp. 357–359.

[9] Daehwan Kim, Ben Langmead, and Steven L Salzberg. "HISAT: a fast spliced aligner with low memory requirements". In: *Nature methods* 12.4 (2015), pp. 357–360.

[10] David R Kelley, Michael C Schatz, Steven L Salzberg, et al. "Quake: quality-aware detection and correction of sequencing errors". In: *Genome Biol* 11.11 (2010), R116.

[11] Li Song, Liliana Florea, and Ben Langmead. "Lighter: fast and memory-efficient sequencing error correction without counting". In: *Genome Biol* 15 (2014), p. 509.

[12] Anthony M Bolger, Marc Lohse, and Bjoern Usadel. "Trimmomatic: a flexible trimmer for Illumina sequence data". In: *Bioinformatics* (2014), btu170.

[13] Nicola Segata, Levi Waldron, Annalisa Ballarini, Vagheesh Narasimhan, Olivier Jousson, and Curtis Huttenhower. "Metagenomic microbial community profiling using unique clade-specific marker genes". In: *Nature methods* 9.8 (2012), pp. 811–814.

[14] Derrick E Wood and Steven L Salzberg. "Kraken: ultrafast metagenomic sequence classification using exact alignments". In: *Genome Biol* 15.3 (2014), R46.

[15] Heng Li. "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM". In: *arXiv preprint arXiv:1303.3997* (2013).

[16] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice. "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants". In: *Nucleic Acids Res.* 38.6 (Apr. 2010), pp. 1767–1771.

[17] Davidlohr Bueso. "Scalability techniques for practical synchronization primitives". In: *Queue* 12.11 (2014), p. 40.

[18] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[19] Thomas Willhalm and Nicolae Popovici. "Putting intel® threading building blocks to work". In: *Proceedings of the 1st international workshop on Multicore software engineering*. ACM. 2008, pp. 3–4.

[20] Santiago Marco-Sola, Michael Sammeth, Roderic Guigó, and Paolo Ribeca. "The GEM mapper: fast, accurate and versatile alignment by filtration". In: *Nature methods* 9.12 (2012), pp. 1185–1188.

[21] Matei Zaharia, William J Bolosky, Kristal Curtis, Armando Fox, David Patterson, Scott Shenker, Ion Stoica, Richard M Karp, and Taylor Sittler. "Faster and more accurate sequence alignment with SNAP". In: *arXiv preprint arXiv:1111.5572* (2011).

[22] Claudia Misale. "Accelerating Bowtie2 with a lock-less concurrency approach and memory affinity". In: *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*. IEEE. 2014, pp. 578–585.

[23] Claudia Misale, Giulio Ferrero, Massimo Torquati, and Marco Aldinucci. "Sequence alignment tools: one parallel pattern to rule them all?" In: *BioMed research international* 2014 (2014).

[24] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. "Fastflow: high-level and efficient streaming on multi-core". In: *Programming multi-core and many-core computing systems, parallel and distributed computing* (2014).

[25] Charlotte Herzeel, Thomas J Ashby, Pascal Costanza, and Wolfgang De Meuter. "Resolving Load Balancing Issues in BWA on NUMA multicore architectures". In: *Parallel Processing and Applied Mathematics*. Springer, 2014, pp. 227–236.

[26] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. *Cilk: An efficient multithreaded runtime system*. Vol. 30. 8. ACM, 1995.

[27] Thomas E. Anderson. "The performance of spin lock alternatives for shared-money multiprocessors". In: *IEEE Transactions on Parallel and Distributed Systems* 1.1 (1990), pp. 6–16.

[28] TinyThread++ team. *TinyThread++ portable threads*. http://tinythreadpp.bitsnbites.eu. [Online; accessed 4-Oct-2017]. 2012.

[29] Intel. *Intel Thread Building Blocks Developer Guide: Mutex Flavors*. https://software.intel.com/en-us/node/506086. Online; Accessed on 2016-07-27. 2016.

[30] John M Mellor-Crummey and Michael L Scott. "Synchronization without contention". In: *ACM SIGPLAN Notices* 26.4 (1991), pp. 269–278.

[31] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin. "The Sequence Alignment/Map format and SAMtools". In: *Bioinformatics* 25.16 (Aug. 2009), pp. 2078–2079.

[32] Deanna M Church, Valerie A Schneider, Karyn Meltz Steinberg, Michael C Schatz, Aaron R Quinlan, Chen-Shan Chin, Paul A Kitts, Bronwen Aken, Gabor T Marth, Michael M Hoffman, et al. "Extending reference assembly models". In: *Genome biology* 16.1 (2015), p. 13.

[33] Daehwan Kim. *HISAT2: graph-based alignment of next generation sequencing reads to a population of genomes*. `http://ccb.jhu.edu/software/hisat2/index.shtml`. [Online; accessed 4-Oct-2017]. 2017.

[34] M. A. Eberle, E. Fritzilas, P. Krusche, M. Kallberg, B. L. Moore, M. A. Bekritsky, Z. Iqbal, H. Y. Chuang, S. J. Humphray, A. L. Halpern, et al. "A reference data set of 5.4 million phased human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree". In: *Genome Res.* 27.1 (Jan. 2017), pp. 157–164.

[35] A. Auton, L. D. Brooks, R. M. Durbin, E. P. Garrison, H. M. Kang, J. O. Korbel, J. L. Marchini, S. McCarthy, G. A. McVean, G. R. Abecasis, et al. "A global reference for human genetic variation". In: *Nature* 526.7571 (Oct. 2015), pp. 68–74.

[36] N. Rustagi, A. Zhou, W. S. Watkins, E. Gedvilaite, S. Wang, N. Ramesh, D. Muzny, R. A. Gibbs, L. B. Jorde, F. Yu, et al. "Extremely low-coverage whole genome sequencing in South Asians captures population genomics information". In: *BMC Genomics* 18.1 (May 2017), p. 396.

[37] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[38] NL Bray, H Pimentel, P Melsted, and L Pachter. "Near-optimal probabilistic RNA-seq quantification." In: *Nature biotechnology* 34.5 (2016), p. 525.

[39] Avi Srivastava, Hirak Sarkar, Nitish Gupta, and Rob Patro. "RapMap: a rapid, sensitive and accurate tool for mapping RNA-seq reads to transcriptomes". In: *Bioinformatics* 32.12 (2016), pp. i192–i200.

[40] Zoran Radovic and Erik Hagersten. "Hierarchical backoff locks for nonuniform communication architectures". In: *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*. IEEE. 2003, pp. 241–252.

[41] David Dice, Virendra J Marathe, and Nir Shavit. "Lock cohorting: A general technique for designing NUMA locks". In: *ACM Transactions on Parallel Computing* 1.2 (2015), p. 13.