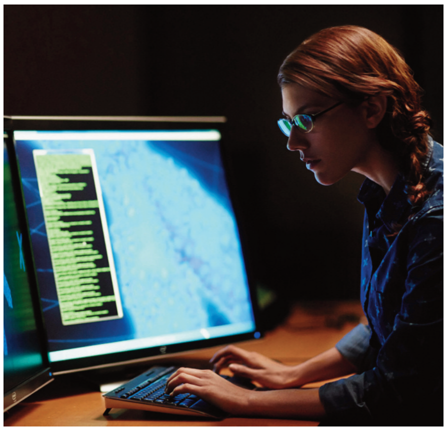
CASE STUDY

Software

Preparing for a Many-Core Future

Intel® Threading Building Blocks

High-Performance Computing



Johns Hopkins University Increases the Performance of its Open-Source Bowtie 2* Application by Adding Multi-Core Parallelism

Modern DNA sequencing provides an inexpensive and high-resolution window into diverse aspects of biology, genetics, and disease. Like a microscope, a sequencer produces a snapshot of a collection of cells. Unlike a microscope, a sequencer does not provide a finished, ready-to-interpret image. Rather, it produces billions of tiny snippets (reads) of DNA that must first be composed into longer, interpretable units such as genes or chromosomes.

Bowtie* and Bowtie 2* are widely used software tools produced in the University's Langmead Lab that allow biologists to piece together the fragmentary evidence generated by DNA sequencers. They do so with respect to a reference genome—a strategy not unlike putting together a puzzle while peeking at the picture of the completed puzzle on the box lid.

Bowtie 2 (Figure 1) is the preferred tool for contemporary datasets. It is particularly good at aligning fragments of DNA (reads) of about 50 up to hundreds or thousands of characters, and at aligning to long genomes like the human genome. Bowtie 2 uses a concise text index called the FM (full text, minute space) Index* to keep its memory footprint small. For the human genome, which consists of about 3 billion DNA bases, its memory footprint is typically around 3.2 GB. The alignment for one read does not depend on the alignment for any other read, so the problem is broadly data parallel. For each read, the computational workload in

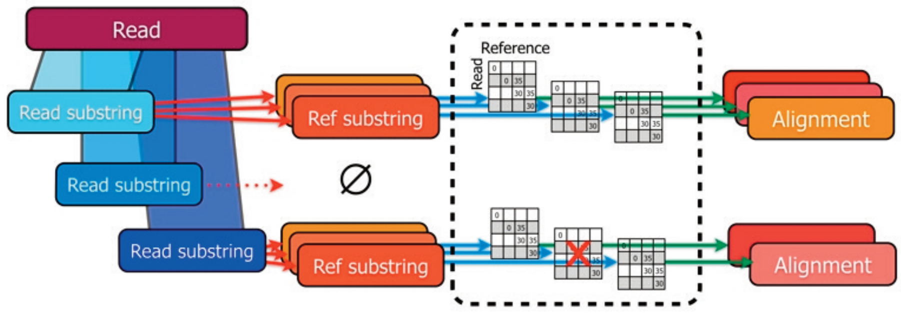


Figure 1. Bowtie 2

Collaborating to Improve Performance and Reliability on Many-Core Systems

Bowtie 2 is roughly evenly divided between index querying—to find alignments for short "seed" substrings extracted from the read—and dynamic programming—to extend the short seed hits into longer gapped alignments. The dynamic programming step has a great deal of instruction-level parallelism and its inner loops can be implemented with SIMD instructions.

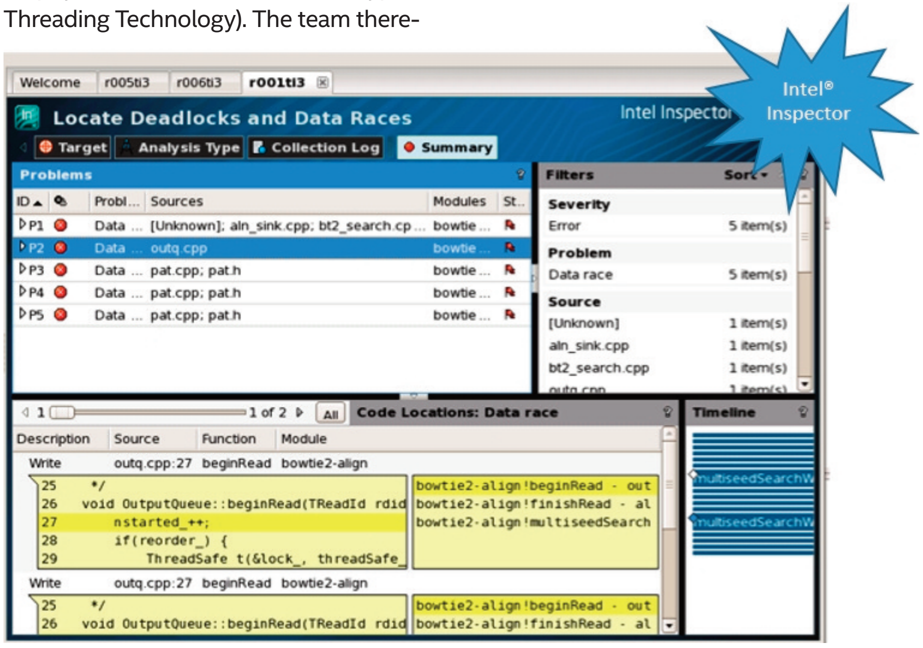
Adding Intel TBB to Bowtie 2

Before its collaboration with Intel, Bowtie 2 had been using a small library called tinythreads for synchronization. Tinythreads supported simple mutual exclusion using a combination of spinlocks and pthreads. But despite the problem being "embarrassingly" parallel, initial benchmarking on non-uniform memory access (NUMA) Intel® Xeon® processor-based systems revealed that Bowtie 2 scaled very poorly to large numbers of threads. For example, per-thread throughput decreased by a factor of about six when moving from a single thread to 120 threads on an Intel® Core™ i7 processor-based system with four NUMA nodes and 60 physical cores (120 with Intel® Hyper-Threading Technology). The team therefore decided to investigate whether Intel's tools—including Intel® VTune™ Amplifier, Intel TBB, and Intel® Inspector—could help to improve Bowtie 2's performance and reliability on many-core systems.

Intel Collaboration on Bowtie 2

Intel engineers have been collaborating with Professor Ben Langmead's team to improve Bowtie 2 code quality and performance. Intel Inspector is a dynamic memory and threading error checking tool that identified several threading errors, which were corrected in the Bowtie 2 release 2.0.3 released in 2012. Figure 2 is a screen shot of Intel Inspector XE, showing the thread data race issues that were identified in Bowtie 2. Performance improvements to Bowtie 2 include:

- **Optimizations** to use hardware population count instructions
- **Improvements** to thread scalability
- **The ability to use** multiple threads to take advantage of the increasing number of processor cores available on Intel® processors



ID	Probl...	Sources	Modules	St..
P1	Data ...	[Unknown]; aln_sink.cpp; bt2_search.cp ...	bowtie ...	
P2	Data ...	outq.cpp	bowtie ...	
P3	Data ...	pat.cpp; pat.h	bowtie ...	
P4	Data ...	pat.cpp; pat.h	bowtie ...	
P5	Data ...	pat.cpp; pat.h	bowtie ...	

Severity	
Error	5 item(s)
Problem	
Data race	5 item(s)
Source	
[Unknown]	1 item(s)
aln_sink.cpp	1 item(s)
bt2_search.cpp	1 item(s)
outq.cpp	1 item(s)

Code Locations: Data race

```
Description   Source        Function   Module
Write         outq.cpp:27   beginRead  bowtie2-align
     */                                           bowtie2-align!beginRead - out
  void OutputQueue::beginRead(TReadId rdid        bowtie2-align!finishRead - al
     nstarted_++;                                 bowtie2-align!multiseedSearch
     if(reorder_) {
        ThreadSafe t(&lock_, threadSafe_
Write         outq.cpp:27   beginRead  bowtie2-align
     */                                           bowtie2-align!beginRead - out
  void OutputQueue::beginRead(TReadId rdid        bowtie2-align!finishRead - al
```

Figure 2. Screenshot of Intel® Inspector identifying threading data race conditions in Bowtie 2 that were fixed in Bowtie 2.0.3 release

The querying of the reference index was optimized by using the population count instruction (popcnt) available in the Intel SSE 4.2 instruction set extension. Intel TBB support was added to Bowtie 2 starting with version 2.2.6, which provides better performance in most situations. Intel TBB simplified experimenting with additional locks to increase the thread scaling of Bowtie 2.

Intel Inspector revealed a number of issues that led to improvements in stability and thread scalability for Bowtie 2. Those improvements are in the Bowtie 2 software today, improving the experience for scientific users.

Performance Issues Encountered

Issue No. 1: Locking

Bowtie 2 threads repeatedly follow a cycle:

1. **Obtain** the next read from the input file.
2. **Align** it to the reference genome using the genome index and dynamic programming.
3. **Print** the resulting alignment(s) to the output file, then repeat.

Step 2 is by far the most work-intensive. Steps 1 and 3 require synchronization among the threads to ensure input records are consumed and output records are written without corruption due to data races.

While the input and output critical sections are quite short, the input critical section was often implicated as the scalability bottleneck, as described in more detail below. When the team began the project, this critical section would do essentially two things:

1. **Read** the input file in a buffered fashion using C I/O functions
2. **Parse** one additional record of input

Input records are typically represented in the FASTQ text format, where each fragment of DNA (i.e., sequencing read) is represented on a set of four consecutive lines. An important fact about this format is that the individual records do not have a predictable length. Only by parsing the record—at least minimally—can we know exactly where the next record boundary lies.

Also, some DNA sequencing datasets consist of so-called "paired-end reads." For these datasets, the input consists of two FASTQ files, where successive records in the two files match up to form pairs of DNA fragments. In this case, synchronization is required both to parse reads in a given FASTQ file and to ensure that the reads in the two FASTQ files are parsed concurrently as expected.

Intel TBB supplies several different types of mutexes, each with different properties.

To understand some of the behavior the team was seeing, it was important to know these properties.

The Intel TBB documentation describes the following qualities for mutexes:

- **Scalability:** Ability to keep the synchronization overhead constant with a growing number of contending threads
- **Fairness:** Ability to give threads access to the critical section in the order of arrival
- **Recursiveness:** Allowing a thread that is holding a lock on a mutex to acquire another lock on the same mutex
- **Waiting policy:** Whether the thread actively polls the mutex state while waiting, or blocks until signaled that the mutex is free

The team was also interested in the following properties:

- **Thread awareness:** Whether the lock can only be freed by the same thread that acquired it (thread-aware), or by any thread (thread-oblivious)
- **Cohort detection:** Ability for the thread holding the lock to detect if there are threads waiting on the lock

Table 1 summarizes the properties for Intel TBB mutexes used.

As follows from the name, tbb::spin_mutex spins in user space while waiting. It is non-scalable, unfair, non-recursive, thread oblivious, and does not allow detecting contending threads. It is very fast for lightly contended, short, critical sections. This mutex is recommended when contention is low or can be spread out among many spin_mutex objects.

The tbb::queuing_mutex also spins in user space but, unlike spin_mutex, it scales because waiting threads do not access a single shared state. Instead, each polls its own flag. This mutex is fair, non-recursive, and, though not really thread-aware, its scope-based API complicates lock releasing in another thread. It is recommended when scalability and fairness are important. Note that fairness can negatively impact performance due to loss of cache locality for protected shared data.

The tbb::mutex (also called normal mutex) is a wrapper around the OS-specific mutual exclusion primitives:

- CRITICAL_SECTION on the Windows* OS
- pthread_mutex on Linux* OS and OS X*

The most important difference in this mutex is that a waiting thread does not spend CPU cycles but is blocked by the OS until it can take the lock. Because of this, it typically has higher overhead than spinning locks. It is recommended for cases where waiting time is long or unpredictable.

Table 1. Traits and behaviors of mutexes

Mutex	Scalable	Fair	Recursive	Waiting	Thread-Aware	Cohort Detection
spin_mutex	No	No	No	Polls	Oblivious	No
queuing_mutex	Yes	Yes	No	Polls	See below	Yes
mutex	OS Dependent	OS Dependent	No	Blocks	OS Dependent	No

The team experimented with different mutex types to see if the choice of mutex might improve performance. Figure 3 shows performance differences in Bowtie 2 with the various mutexes.

As the figure shows, the worst performance was with the spin_mutex. This was an unexpected result. Since the system in this case supported up to 120 threads (60 physical cores, 120 with Intel HyperThreading Technology), the team expected a spin mutex to be more beneficial to aggregate throughput than, for example, a normal mutex. The fact that the spin mutex was actually the worst performing of the lock types caused them to investigate the cache coherence properties of the spin mutex on NUMA architectures. The team ran some performance tests using Intel VTune Amplifier, which showed a hotspot in the Intel TBB primitive for spin lock acquisition. This supported the theory that cache coherence due to contended reads and writes to the shared lock state was an issue.

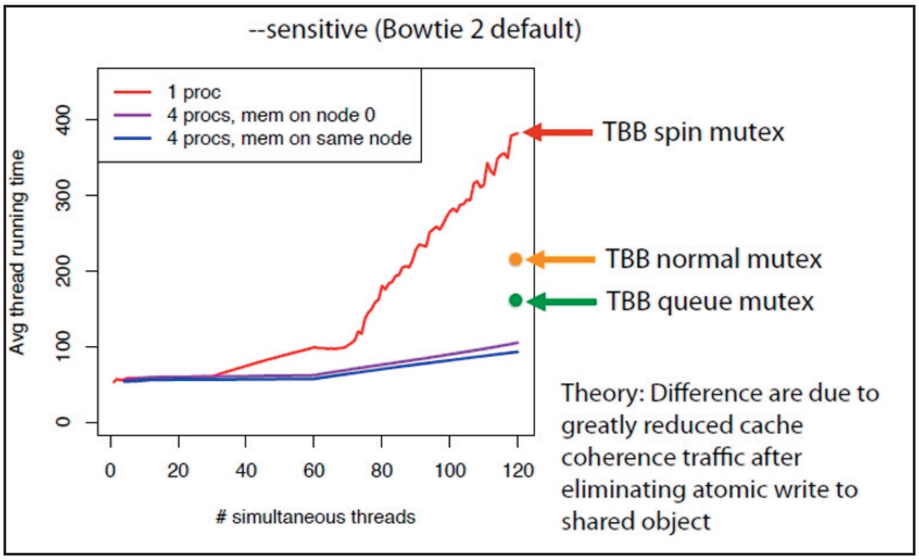
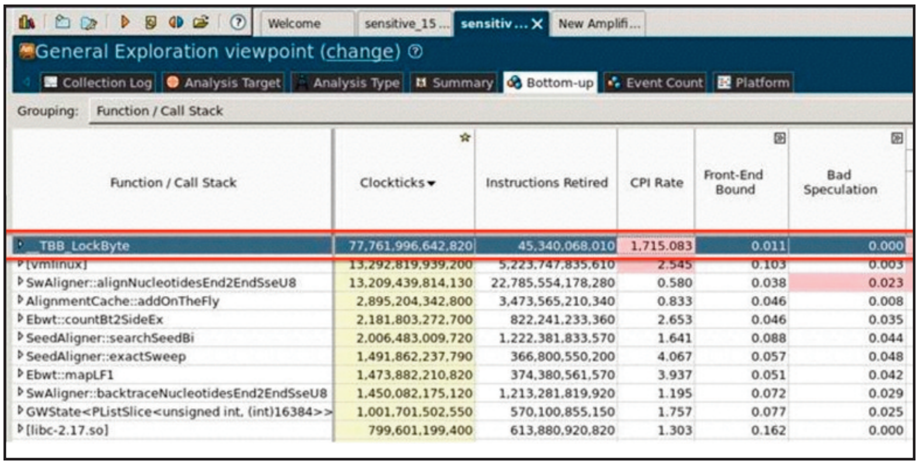


Figure 3. Performance differences in Bowtie 2 with different mutexes



Grouping: Function / Call Stack

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation
__TBB_LockByte	77,761,996,642,820	45,340,068,010	1,715.083	0.011	0.000
[vmlinux]	13,292,819,939,200	5,223,747,835,610	2.545	0.103	0.003
SwAligner::alignNucleotidesEnd2EndSseU8	13,209,439,814,130	22,785,554,178,280	0.580	0.038	0.023
AlignmentCache::addOnTheFly	2,895,204,342,800	3,473,565,210,340	0.833	0.046	0.008
Ebwt::countBt2SideEx	2,181,803,272,700	822,241,233,360	2.653	0.046	0.035
SeedAligner::searchSeedBi	2,006,483,009,720	1,222,381,833,570	1.641	0.088	0.044
SeedAligner::exactSweep	1,491,862,237,790	366,800,550,200	4.067	0.057	0.048
Ebwt::mapLF1	1,473,882,210,820	374,380,561,570	3.937	0.051	0.042
SwAligner::backtraceNucleotidesEnd2EndSseU8	1,450,082,175,120	1,213,281,819,920	1.195	0.072	0.029
GWState<PListSlice<unsigned int, (int)16384>>	1,001,701,502,550	570,100,855,150	1.757	0.077	0.025
[libc-2.17.so]	799,601,199,400	613,880,920,820	1.303	0.162	0.000

Figure 4. Performance testing

Issue No. 2: NUMA Issues

In a processor that supports NUMA, it is not enough to know that you missed a cache on the CPU where you are running. In NUMA architectures, you could also be referencing the cache and DRAM on another socket. The latencies for this type of access are an order of magnitude greater than for the local case. You need the ability to identify and optimize these remote memory accesses.

The Intel TBB queue mutex resulted in better performance than the spin mutex. However, when it was run on multi-socket systems, the team noticed degradation in performance. Figure 5 shows the results of these experiments.

The type of locks used in Bowtie 2 are shared among all the cores and among the sockets. The memory the lock uses is a potential performance bottleneck due to cache coherence issues. The theory was that the lock migration between the NUMA nodes caused the observed performance difference. To check this, the team needed a way for its locks to be NUMA aware.

To analyze the performance and the memory bandwidth and cache coherence issues, tools such as Intel VTune Amplifier have added some new analysis types that make this much easier.

Intel VTune Amplifier has a new analysis type called HPC Performance Characterization. Using it, you can quickly see your CPU utilization. If your application is memory bound, the Memory Bound metric will be highlighted in pink (Figure 6).

Intel VTune Amplifier also includes a much deeper type of memory analysis. Using the Memory Access analysis type, you can see the DRAM bandwidth and also the inter-socket bandwidth known as Intel® QuickPath Interconnect (Intel® QPI) bandwidth. Figure 7 shows the result of a Memory Access analysis collection. The summary view clearly shows we are memory bound.

In the summary view (Figure 8) you can also see a histogram of DRAM bandwidth as well as QPI bandwidth.

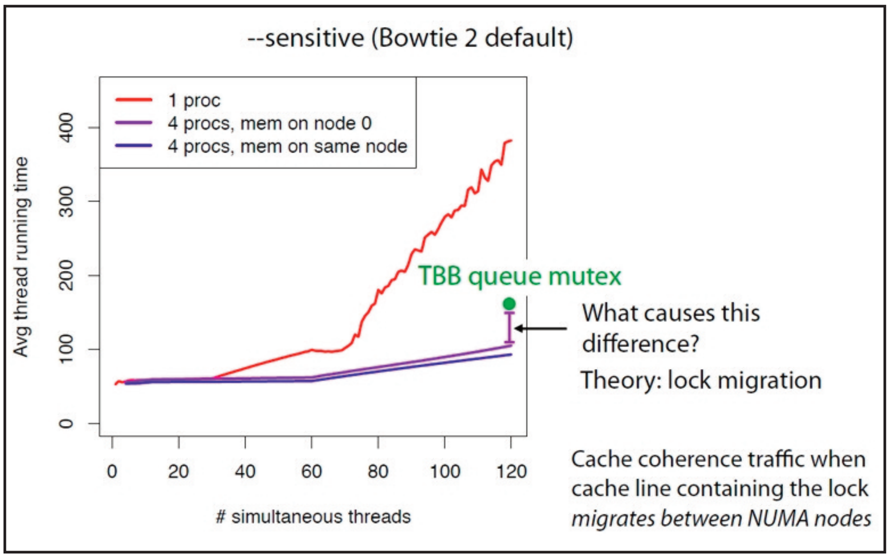
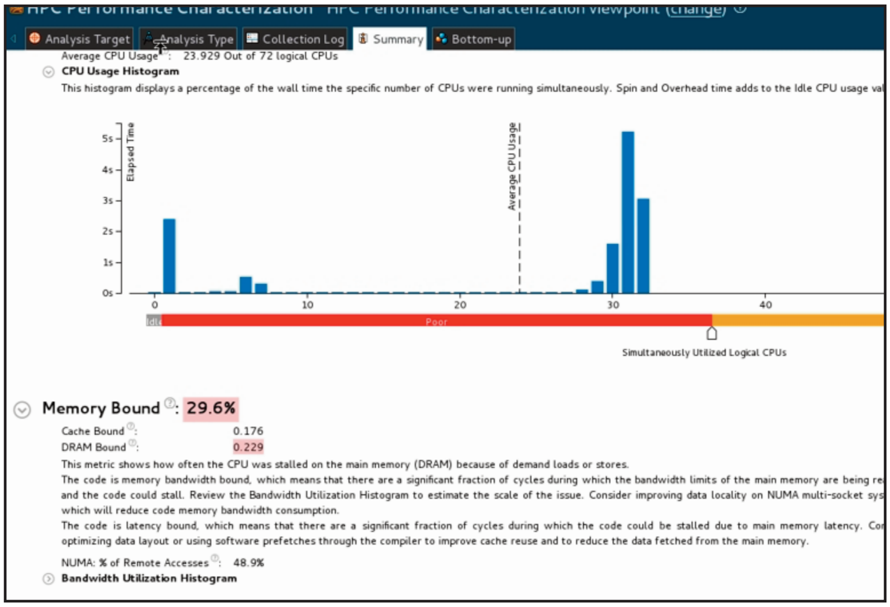


Figure 5. Lock migration experiments


HPC Performance Characterization HPC Performance Characterization viewpoint (change)
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
Elapsed Time
Average CPU Usage
Simultaneously Utilized Logical CPUs
Idle
Poor

Average CPU Usage : 23.929 Out of 72 logical CPUs

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage val

Memory Bound : 29.6%

Cache Bound : 0.176

DRAM Bound : 0.229

This metric shows how often the CPU was stalled on the main memory (DRAM) because of demand loads or stores.
The code is memory bandwidth bound, which means that there are a significant fraction of cycles during which the bandwidth limits of the main memory are being re
and the code could stall. Review the Bandwidth Utilization Histogram to estimate the scale of the issue. Consider improving data locality on NUMA multi-socket sys
which will reduce code memory bandwidth consumption.
The code is latency bound, which means that there are a significant fraction of cycles during which the code could be stalled due to main memory latency. Co
optimizing data layout or using software prefetches through the compiler to improve cache reuse and to reduce the data fetched from the main memory.

NUMA: % of Remote Accesses : 48.9%

Bandwidth Utilization Histogram

Figure 6. HPC performance characterization

Lock Cohorting: Implementing Locks that are NUMA Aware

In its final experiments with locking, the team used a technique known as lock cohorting to create a NUMA-aware mutex on top of Intel TBB (a technique described in an excellent paper by Dice et al called "Lock Cohorting: A General Technique for Designing NUMA Locks").

As the paper states, a NUMA-aware cohort lock can be implemented on top of two NUMA-oblivious locks with the following properties:

1. **A thread-oblivious lock** that "allows the acquiring thread to differ from the releasing thread"
2. **A cohort-detecting lock** for which "a thread releasing the lock can detect if it has a nonempty cohort of threads concurrently attempting to acquire the lock"

In a nutshell, the cohort lock consists of a set of cohort-detecting locks, each acquired by threads running on the same NUMA node, and a top-level, thread-oblivious lock which transfers ownership between NUMA nodes. In case of contention, the lock ownership is typically transferred to another thread on the same node, which provides better NUMA locality for both the lock state and the data protected by the lock.

While it does not yet have a cohort lock class, Intel TBB provides suitable mutex classes for building it. As we discussed earlier, tbb::spin_mutex is thread-oblivious and can be used as the top-level lock. The tbb::queuing_mutex can be easily extended to provide cohort detection and used for the node-level locks. The paper referenced above describes the general implementation approach.

Below, we outline specific aspects of an Intel TBB-based implementation for readers interested in doing their own experiments.

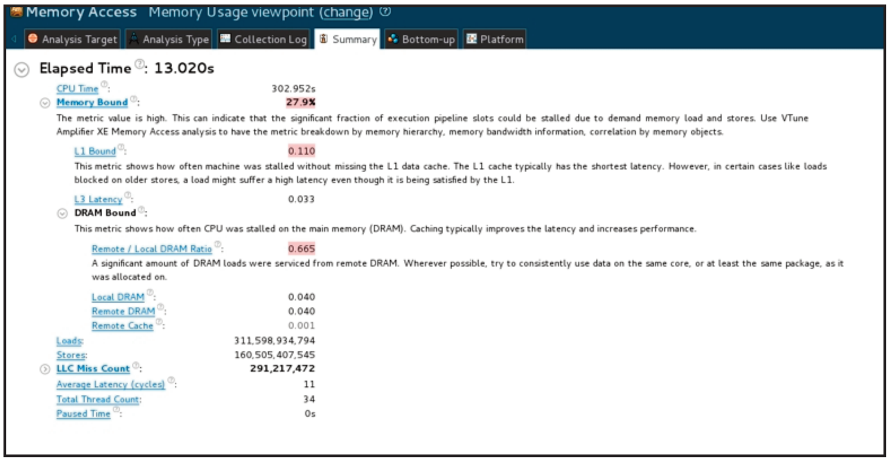
Since Intel TBB is NUMA-agnostic and does not provide any API for querying or managing thread placement, we designed the lock so that users need to specify the desired number of nodes (cohorts) at lock construction and the node/cohort index at lock acquisition.

Other than thread-obliviousness, there are no special requirements for the top-level lock, so the tbb::spin_mutex can be used as-is.

Extending tbb::queuing_mutex to provide cohort detection can be done by adding the following method to the class tbb::queuing_mutex::scoped_lock:

```
bool
tbb::queuing_mutex::scoped_lock::is_alone() {
  return next==NULL;
}
```

The method returns true if the mutex is uncontended and false if there is another thread waiting for it.



Elapsed Time: 13.020s

CPU Time: 302.952s

Memory Bound: 27.9%

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use VTune Amplifier XE Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

L1 Bound: 0.110

This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1.

L3 Latency: 0.033

DRAM Bound:

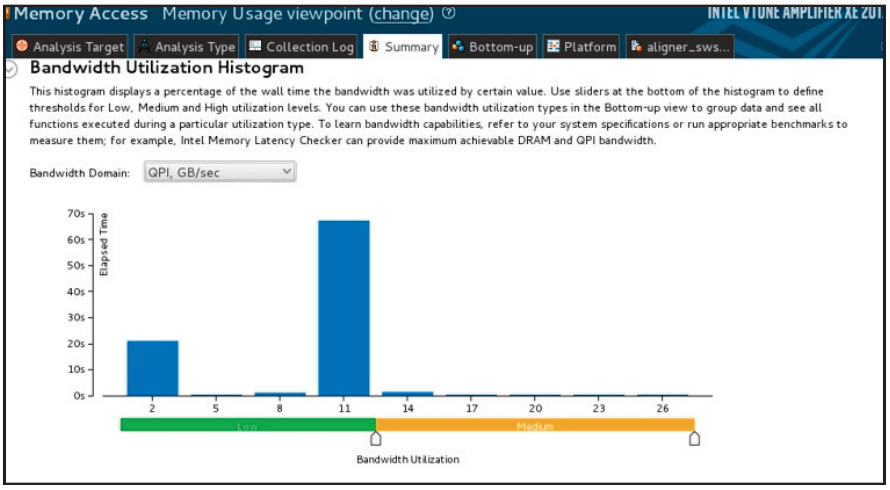
This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.

Remote / Local DRAM Ratio: 0.665

A significant amount of DRAM loads were serviced from remote DRAM. Wherever possible, try to consistently use data on the same core, or at least the same package, as it was allocated on.

Local DRAM:	0.040
Remote DRAM:	0.040
Remote Cache:	0.001
Loads:	311,598,934,794
Stores:	160,505,407,545
LLC Miss Count:	291,217,472
Average Latency (cycles):	11
Total Thread Count:	34
Paused Time:	0s

Figure 7. Memory access analysis

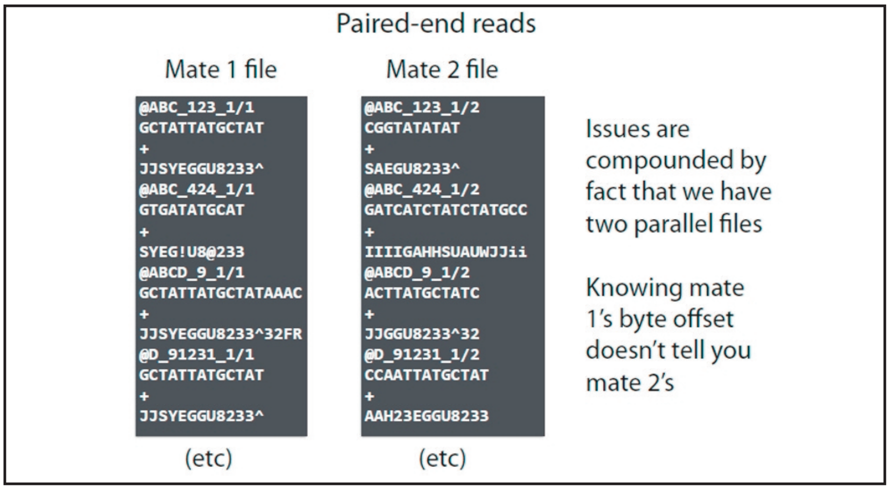


Bandwidth Utilization Histogram

This histogram displays a percentage of the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and QPI bandwidth.

Bandwidth Domain: QPI, GB/sec

Figure 8. Summary analysis view


Paired-end reads
Mate 1 file
Mate 2 file
Issues are compounded by fact that we have two parallel files
Knowing mate 1's byte offset doesn't tell you mate 2's
(etc)
(etc)

```
@ABC_123_1/1
GCTATTATGCTAT
+
JJSYEGGU8233^
@ABC_424_1/1
GTGATATGCAT
+
SYEG!U8@233
@ABCD_9_1/1
GCTATTATGCTATAAAC
+
JJSYEGGU8233^32FR
@D_91231_1/1
GCTATTATGCTAT
+
JJSYEGGU8233^
```

```
@ABC_123_1/2
CGGTATATAT
+
SAEGU8233^
@ABC_424_1/2
GATCATCTATCTATGCC
+
IIIIGAHHSUAUWJJii
@ABCD_9_1/2
ACTTATGCTATC
+
JJGGU8233^32
@D_91231_1/2
CCAATTATGCTAT
+
AAH23EGGU8233
```

Figure 9. Scalable FASTQ parsing

cohort lock should limit the number of node-local acquisitions and decide when to pass ownership to another cohort. The limit can be hard-coded or provided as an argument at lock construction. Each node-level lock needs, besides a queuing_mutex, a counter for the number of local passes and a Boolean flag indicating whether the next thread in the cohort can proceed immediately or should first take the global lock.

The preferred method for lock operations in Intel TBB is to use a special scoped_lock class. The scoped object for the cohort lock should internally hold an instance of tbb::queuing_mutex::scoped_lock or be inherited from it, and also keep the cohort index necessary to release the lock.

Combining everything for a cohort lock needs to contain an instance of tbb::spin_mutex and an array of node-level lock structures described above. Since the number of cohorts is an argument to the constructor, the array should be allocated from heap at the lock creation. To avoid false sharing, node-level locks should better be padded. An easy way to add padding is via tbb::internal::padded class template defined in tbb_stddef.h:

using tbb::internal::padded;

padded<Node_Lock> * cohorts = new padded<Node_Lock>[n_of_cohorts];

Since padded<Node_Lock> publicly inherits Node_Lock, all fields and methods of the latter can be used without problems after padding.

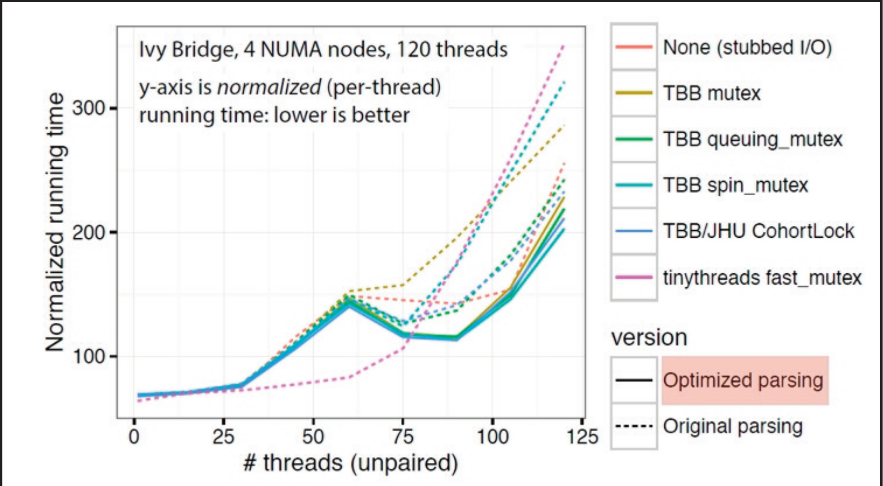
Final Locking Results

As Table 2 shows, Queue and Cohort mutexes reduce cache coherent traffic between NUMA nodes by minimizing the scope of the shared locking variable and by not spinning on it. The fastest mutex is still substantially slower than close to optimal performance (Cohort versus Node-Bind). The length of the critical section might explain most of this difference.

Table 2. Mutex comparison

Run Type/ Lock Type	Spin Mutex	Normal (Heavy) Mutex	Queue Mutex	Cohort Mutex[1]	Numa Node Bind[2]
very_fast	00:06:38.207	00:03:31.638	00:02:46.012	00:02:29.866	00:00:54.319
fast	00:06:15.992	00:03:32.426	00:02:44.905	00:02:29.504	00:01:08.890
sensitive	00:06:29.345	00:03:36.371	00:02:47.158	00:02:31.084	00:01:36.222
very_sensitive	00:03:18.954	00:03:39.274	00:03:19.625	00:03:15.617	00:03:08.392

[1]Not a TBB built-in mutex
[2]Threads are split up into four independent Bowtie processes with 30 threads eachl Each of the four Bowtie processes is run pinned to a numa node with its own copy of the genome index pinned to its node's memory.



Optimized parsing yields *very* substantial thread-scaling boost.
Scaling essentially ideal up to 120 threads, regardless of lock.

Figure 10. Thread scaling: Bowtie 2 paired-end

A final strategy the team has used to improve thread scalability in Bowtie 2 is to simplify the input critical section to include only the minimal amount of parsing required to detect record boundaries. The more work-intensive task of fully parsing a FASTQ record is then deferred to a routine that runs after the critical section. We call this "light parsing." The team found that, in addition to the improvements obtained by moving to queue locks and NUMA-aware locks, there was a substantial additional improvement from switching to light parsing, as shown in Figure 10.

Conclusion

Johns Hopkins and Intel have been collaborating on the Bowtie 2 application. Adding parallelism via Intel TBB resulted in a substantial speedup of the application. In its initial work, the team was able to track down threading data races using Intel Inspector. Once these correctness issue were addressed, the team increased the performance by using Intel TBB with different mutexes that were better suited to the application. The NUMA issues the team experienced were handled by lock cohorting and pinning threads to specific cores. Finally, by splitting reads from parsing in a critical section, the team saw essentially ideal scaling up to 120 threads.

Overall, the team at Johns Hopkins—and users of the Bowtie and Bowtie 2 software tools—have benefitted greatly from the collaboration with Intel. By working directly with Intel engineers, and by using Intel tools and libraries such as Intel VTune Amplifier, Intel Inspector, and Intel TBB, the team was able to effectively prepare these core genomics software tools for the many-core future around the corner.

Learn More

Genomics and Algorithms for DNA Sequencing >

Johns Hopkins University >

Intel® Threading Building Blocks >