Intel® Threading Building Blocks Celebrates 10 Years!

Intel® TBB and C++: Partners in Parallel

The Future: How Should Intel® TBB Evolve?

Ideal for Heterogeneous Systems
CONTENTS

Letter from the Editor
From Hatching to Soaring: Intel® TBB
by James Reinders

The Genesis and Evolution of Intel® Threading Building Blocks
A Decade after the Introduction of Intel Threading Building Blocks, the Original Architect Shares His Perspective

A Tale of Two High-Performance Libraries
How Intel® Math Kernel Library and Intel® Threading Building Blocks Work Together to Improve Performance

Heterogeneous Programming with Intel® Threading Building Blocks
With New Features, Intel® Threading Building Blocks can Coordinate the Execution of Computations Across Multiple Devices

Preparing for a Many-Core Future
Johns Hopkins University Adds Multicore Parallelism to Increase Performance of Its Bowtie 2* Application

Leading and Following the C++ Standard
Intel® Threading Building Blocks Adheres Tightly to the C++ Standard Where It Can—and Paves the Way for Supporting Parallelism Best

Intel® Threading Building Blocks: Toward the Future
The Architect of Intel® Threading Building Blocks Shares Thoughts on the Opportunities Ahead
LETTER FROM THE EDITOR


From Hatchling to Soaring: Intel® TBB

Intel® TBB Is One of the Most Important Contributions to Modern Parallel Programming—and There's More to Come

This edition of The Parallel Universe celebrates the 10th anniversary of the introduction of Intel® Threading Building Blocks (Intel® TBB). Intel TBB has been called the most important new addition to parallel programming in the last decade, and I would not argue with that. The articles in this issue will help you understand why. If you will be so kind as to indulge me, I will share my own thoughts about Intel TBB. I have four things in mind to touch on as I ramble about TBB.

It Was a Revolution Inside Intel

Intel TBB is our first commercially successful software product to embrace open source. We knew we wanted to open source Intel TBB from the start, but we were not ready when we launched in 2006. Open source projects were new to our small team—and to Intel.

We focused first on creating a strong Intel TBB and launching it as a product in mid-2006. Then we shifted our attention to revising our build system, cleaning up code (commenting!), and a dozen other things that would be inviting to others who would want to understand and contribute to our source code. We had a goal to be open source in mid-2007.

But a new problem arose: Intel TBB became an immediate hit with customers. We did not hide our desire to be open source to our customers, and this only intensified their interest in Intel TBB. Some of our management asked, “Why give away the source code to such a successful product?” and I boldly presented a multitude of reasons, armed with facts and figures from our team, why we should open up. That was a mistake, and I failed to get the needed permissions before 2006 ended. I licked my wounds, and we eventually realized we needed to prove only one thing: Intel TBB would have far greater adoption if we open sourced it than if we did not. After all, developers bet the very future of their code when...
they adopt a programming model. Perhaps openness matters more for programming models than it does for most other software. We had failed to articulate to our management that this was all that really mattered—and that it was all we needed to know to understand that we must open source Intel TBB.

Armed with this perspective, I approached our senior VP, Renee James, who had to approve our proposal. I surprised her by showing up with only a single piece of paper with a simple graph on it, which compared projected Intel TBB adoption with and without open sourcing. We predicted that Intel TBB would vanish and be replaced within five years if we didn't offer this critical programming model via open source. We predicted great success if we did open source (we actually far underestimated the success, as it turns out). Renee listened to my two-minute pitch, looked at me, and asked, “Why didn't you say this the first time? Of course we should do this.” Of course, I could have pointed out that this graph was identical to slide 7 of the original way-too-long presentation from two months earlier, but I settled on “Thank you,” and the rest is history. We chose the most popular open source licensing at the time: GPL v2* with classpath exception (important for C++ template libraries). Ten years later, we are switching Intel TBB to the Apache* license. We have received a great deal of feedback from the community of users and contributors that this is the right license to use for Intel TBB today.

Intel TBB's First Revolution of Parallelism: Embrace Task-Stealing Abstraction, Fully Composable, Fully C++

OpenMP is incredibly important, but it is not composable. This is a mistake of epic proportions, with long-reaching ramifications, and it cannot be changed because OpenMP is so important and committed to compatibility. I am complicit in the OpenMP mistake, along with everyone else who helped pull it together, review it, and promote it starting in 1997. We overlooked the importance that nested parallelism would have as the amount of hardware parallelism grew. It simply was not a concern in 1997.

Being composable is the most amazing feature of Intel TBB. I cannot overstate the importance of never worrying about oversubscription, nested parallelism, etc. Intel TBB is gradually revolutionizing certain communities of developers who demand compositability for their applications. The Intel® Math Kernel Library (Intel® MKL), which has long been based on OpenMP, offers a version built on top of Intel TBB for exactly this reason. And the much newer (and open source) Intel® Data Analytics Acceleration Library (Intel® DAAL) always uses Intel TBB and the Intel TBB-powered Intel MKL. In fact, Intel TBB is finding use in some versions of Python* too.
The Parallel Universe

“Into the Great Wide Open” called attention to the open source nature of Intel® TBB. At OSCON, attendees were invited to experience and explore the new Intel TBB for themselves.

Of course, the task-stealing scheduler at the heart of Intel TBB is the real magic. While HPC customers worry about squeezing out the ultimate performance while running an application on dedicated cores, Intel TBB tackles a problem that HPC users never worry about: How can you make parallelism work well when you share the cores that you run upon? Imagine running on eight cores, but a virus checker happens to run on one core during your application’s run. That would never happen on a supercomputer, but it happens all the time on workstations and laptops. Without the dynamic nature of the Intel TBB task-stealing scheduler, such a program would simply be delayed by the full time that the virus checker stole—because it would effectively delay every thread in the application. When using Intel TBB on eight cores, an interruption of duration \( \text{TIME} \) on one core may delay the application by as little as \( \text{TIME}/8 \). This real-world flexibility matters a lot.

Finally, Intel TBB is a C++ template library that fully embraces bringing parallelism to C++. The dedication of Intel TBB to C++ has helped inspire changes to the C++ standard. Perhaps our biggest dream of all is that Intel TBB will one day only be the scheduler and the algorithms that use it. The many other things in Intel TBB—helping parallelize parts of STL, creating truly portable locks and atomics, addressing shortcomings in memory allocations, and other features to bring parallelism to C++—that can and should eventually be part of the standard language. Maybe even more of Intel TBB? Time will tell.

Intel TBB’s Second Revolution of Parallelism: Offer Superior Alternatives to Bulk Synchronous Programming

As much as we can praise Intel TBB’s task-stealing scheduler, the algorithms most often used in applications are organized with a lot of synchronization happening at runtime. This is a sign of the times in terms of how parallel programming has been done successfully for years. However, as the amount of parallelism has grown, this has become a great obstacle in the pursuit of scaling. A better approach is to express the flow of data and require a minimal level of synchronization. The flow graph addition to Intel TBB is a leader in this critical revolution in parallel programming. This type of thinking is required for any parallel programming model to support the future well.
Intel TBB’s Bird

On a very different note, I do get asked about the bird we’ve used to represent Intel TBB. My original 2007 Intel TBB book was part of the O’Reilly in a Nutshell series, with its iconic animal designs. O’Reilly made it clear to me, as the author, that they would pick the animal (a mysterious process). Undaunted, I conveyed some ideas I had for animals that made sense to me. Still, O’Reilly chose a beautifully drawn canary for the cover. Everyone can have opinions, but soon our cry around Intel was, “Embrace the bird!” led by Belinda Adkisson, the worldwide marketing director for Intel® Software Developer Products. We can thank Belinda also for the popular, noninfringing “Chirp” bird that we use on T-shirts, stickers, and websites. This cheery little bird remains our Intel TBB mascot.

As an aside, I wrote the Intel TBB book in the spring of 2007 with a great deal of help from the Intel TBB team. I would very much like to do a new book in the coming years. I do not personally have the time this year, but if enough people wanted to help...well, I think we could figure something out. I’m open to suggestions!

I hope this special 10th anniversary issue helps expand your understanding of Intel TBB and its success, and that you find it useful for realizing your application’s full capabilities in exploiting parallelism.

James Reinders
June 2016
Even after a decade, Intel® TBB has stuck to its roots as a pure library solution.

THE GENESIS AND EVOLUTION OF INTEL® THREADING BUILDING BLOCKS

A Decade after the Introduction of Intel® Threading Building Blocks, the Original Architect Shares His Perspective

Arch D. Robison, Principal Engineer, Intel Corporation
Intel® Threading Building Blocks (Intel® TBB) was designed to let nonexperts easily exploit multicore processors, using stock C++ compilers, in a scalable way. Starting out as a small library, it has grown over time to accommodate differing platforms, complexities, and styles of parallel programming. Nonetheless, even after a decade, it has stuck to its roots as a pure library solution with an emphasis on efficient composition with minimal surprise.

The idea for Intel TBB 1.0 arose in 2004. Multicore processors were arriving, and parallel software was (as it still is) the key for harnessing them. The two most popular means at the time were OpenMP® and operating system (OS) threads directly via interfaces such as Pthreads®. These were deemed impractical or insufficient for productively developing efficient software.

The problem with using threads directly is concurrency versus parallelism. Concurrency is about many separate activities each making forward progress. OS threads are designed to do this. If the number of OS threads exceeds the number of available hardware threads, the OS time slices execution so that each thread makes forward progress. This is essential for system responsiveness—and good for running many unrelated activities.

But what if the goal is to solve a single problem? Then the ideal is not to time slice, but to somehow divide up work just enough to keep each hardware thread busy, and to coordinate that work to make efficient use of memory resources. This is what parallelism is about.

For a dramatic demonstration of the difference, consider computing the nth Fibonacci number using a naive recursive algorithm, which, though grossly inefficient, is easy to understand and representative of a parallel, recursive fork-join pattern found in realistic computations such as sorting, searching, and recursive linear algebra. A highly threaded solution, using modern C++ notation, is:

```cpp
double fib_thread(int n) {
    if (n<2)
        return n;
    else {
        double x;
        auto t = std::thread( [&]{x=fib_thread(n-2);} );
        double y = fib_thread(n-1);
        t.join();
        return x+y;
    }
}
```

The recursive concurrency creates exponentially explosive concurrency. I ran `fib(25)` on a 16-core Linux® system, and it consumed 0.4 terabytes of virtual memory before dying. The program is, of course, silly. But real software nonetheless involves calls with potential parallelism at multiple levels—and recursive algorithms can be surprisingly effective. Having each level figure out the right number of threads to use is difficult—and practically impossible if each level is written independently.
What is needed is a way to specify logical work that can be run in parallel (tasks) and have a task scheduler deal with spreading work across threads. The Chare Kernel (now Charm++) strongly influenced this design point. Furthermore, the scheduler should run the tasks in an order that uses memory efficiently, and not have any centralized control that would become a bottleneck. Cilk™ (now Intel® Cilk™) solved this problem in the 1990s by developing the notion of efficient *work-stealing* schedulers. But, alas, it required a special compiler. (Intel's acquisition of Cilk Arts and integration of Cilk into the Intel® compiler was still in the future.)

What about OpenMP? It too requires special compiler support. And tasking support did not arrive until 2008 with OpenMP 3.0. Even today, Visual Studio® 2015 does not implement it. Furthermore, nesting parallel regions in OpenMP still suffer from the “guess the number of threads to use” problem.

To avoid the need for a special compiler, we borrowed the work-stealing idea from Cilk, but wrapped it in a C++ library interface. In doing so, we had to give up some theoretical guarantees (see the article *A Primer on Scheduling Fork-Join Parallelism with Work Stealing*) in exchange for compiler independence. Here is the Fibonacci example rendered in modern Intel TBB:

```cpp
double fib_task(int n) {
  if (n<2)
    return n;
  else {
    double x, y;
    tbb::task_group g;
    g.run([&]{x=fib_task(n-2);});
    y=fib_task(n-1);
    g.wait();
    return x+y;
  }
}
```

This code is structurally similar to the previous example but launches tasks, not threads. The difference in resource usage is dramatic. A program running `fib_task(50)` uses a mere 2.2 MB of virtual space on the same system—100,000x less space than what the threaded version used for n=25. We used n=50 instead of n=25 because `fib_task(25)` runs in such a tiny fraction of a second that it finishes before we can read its entry in the process list. That points out the other advantage of task-based programming: tasks are much cheaper to run than threads.1

---

1. What about using C++11’s `std::async`? It occupies an odd space between tasking and threading. Using `std::async` with `std::launch::async|std::launch::deferred` enables parallelism without oversubscribing the system, but because of the way `std::async` is specified, launching incurs most of the weight of launching a thread.
Programming at the task level can be tedious for codes that are more naturally expressed as loops. So we provided parallel "algorithms," C++ templates for common loop patterns: `parallel_for`, `parallel_reduce`, `parallel_scan`, `parallel_while`, and `pipeline`. (Template `parallel_while` has been superseded by `parallel_do`, and `pipeline` by `parallel_pipeline`). These are a way to specify logical work at a higher level than tasks—"map this functor over this index space." Furthermore, these templates make parallel programming easier for nonexperts precisely because they are restricted to a limited set of patterns that are known to be generally effective. The C++ template feature was critical to their success, because it made the code generic yet efficient. The introduction of C++11 lambda expressions further simplified usage.

A key feature of the algorithm templates is they are explicitly blocked. For example, the `parallel_for` invokes a functor not on a single iteration index, but over a range of indices, typically specified by `blocked_range`. This lets users use all the normal serial (and don't forget SIMD) tricks to write the most efficient "serial brick" of code, which `parallel_for` will compose into an efficient parallel structure.

Providing the higher-level templates also let us get back much of the theoretical guarantees that we had lost. It turns out that continuation-stealing and greedy (see WG21 reference) are doable inside the template with sufficient effort. (Look up "continuation passing style" in the Intel TBB documentation to learn more about how the templates do this.)

Effective parallel programming is more than just a matter of tasking. A single serial bottleneck can strangle parallelism. Notably, many memory allocators of the time had single-threaded origins and did not scale well to multiple threads. Indeed, the malloc of a popular system had a mutex protecting it. So Intel TBB 1.0 provided a scalable memory allocator, based on work by Intel Labs.

Similarly, Intel provided some container classes for common situations that might otherwise require serializing accesses or updates. These containers were `concurrent_hash_map`, `concurrent_queue`, and `concurrent_vector`. The latter has an interface similar to `std::vector`, but does not move elements when a task grows it, so concurrently running tasks working on other elements are not bothered. This feature is critical when multiple tasks are appending to the vector.

Sometimes programmers do need low-level threading tools, so Intel TBB also provided a variety of mutexes and atomic operations. The atomic operations seem minor now with C++11, but back in the mid-2000s, there was no portable interface. The various mutexes in Intel TBB make different tradeoffs on speed in uncontended situations versus fairness in contended ones. The best kind to use: none at all. Design your program to rely on the implicit sequencing of the task model. For example, `parallel_reduce` is a far more effective way to sum a billion numbers than protecting a single shared accumulator with a mutex.
As someone involved in developing Intel TBB 1.0, I have two regrets. One was not providing a better parallel sort. I chose a parallel quicksort because it is an in-place sort that worked well for the small core counts and small caches of the time, but scales poorly. (See the article A Parallel Stable Sort Using C++11 for TBB, Cilk Plus, and OpenMP for more history on this issue and code for a more scalable sort.)

The other regret was making users specify the grain size for parallel_for. I was unconcerned because it was a hint that could be off by an order of magnitude and still work well. But users were mystified. Intel TBB 1.1 fixed the problem by adding the auto_partitioner feature (now the default), which uses an introspection heuristic to determine a good grain size.

The big feature for Intel TBB 2.0 was not technical, but going open source. It now had a dual-license model: the proprietary license or GPLv2 with the (libstdc++) runtime exception. Bits are the same, only license and support differ. At the same time, James Reinders published Intel Threading Building Blocks with O'Reilly, and the front-cover image of a canary was linked with Intel TBB. This was the inflection point for Intel TBB. Now it could be used in other open-source projects such as OpenCV*, and programmers could dig into the workings and contribute improvements.

Intel TBB 2.1 brought a coherent approach to handling exceptions thrown out of tasks (task_group_context) and addressed a performance issue of back-to-back loops (affinity_partitioner). There were also less visible improvements, mostly performance boosters, that continue to this day.

During the early releases, Intel TBB established a strong tradition of link compatibility. Binaries dynamically linked against an old version of Intel TBB continue to work if linked against a newer version. This is important to users—which is why, after our one major slip on this point, we heard about it. One trick we use to keep this compatibility is how the tbb::task structure is laid out in memory. Given a pointer to a task, the user-defined portion has positive offsets, and the library-defined portion has negative offsets. Tasks are always allocated inside the dynamic library, so we are free to establish new fields at further negative offsets or, rarely and carefully, to redefine old fields. What’s at zero offset? The pointer to the C++ vtable. We do occasionally break source compatibility (typically controlled by a macro) and may eventually break binary compatibility—but only if we judge the gain to exceed the cost for our user base.

Intel TBB has grown since then. Along the way, we worked with Microsoft to develop some common functionality. The major results are as follows:
• **task_group** and **parallel_invoke** simplified interfaces to the task scheduler that became easy to use when C++11 added lambda expressions.

• **concurrent_unordered_map** supports concurrent insertion and traversal and is lock-free, but does not support concurrent erasure like the original, **concurrent_hash_map**. There are some hard issues related to C++’s lack of garbage collection for building an efficient, all-purpose, concurrent hash table.

• **combinable** is an interface to thread-local memory.

• **concurrent_queue** was simplified to be an unbounded queue. Bounded queue functionality was moved to **tbb::concurrent_bounded_queue**.

Related to **combinable** was our own **enumerable_thread_specific**, which has some additional features. Having two interfaces for thread-local memory was ironic, because we had resisted adding any support out of fear that it would lead programmers to use threading instead of tasking. But even in task-based programming, thread-local storage is still useful because multiple tasks can operate on it knowing there’s zero possibility of a race condition—yet there’s no need to create separate storage for every task. The thread-local storage support has an unusually simple interface: It treats such storage as a special container that dynamically creates an element for each thread that accesses it. The container can be walked with an iterator to see all the elements, or a reduction operation applied over the elements.

Cilk-style parallelism, though powerful, is not a panacea. Some use cases need FIFO-like processing of tasks, so Intel TBB 3.0 added **task::enqueue**. Some other programs more naturally follow a dataflow model, so Intel TBB 4.0 introduced the Flow Graph interface. (See the sidebar on p. 13, “Overview of Intel Threading Building Blocks.”)

Like Intel TBB 2.0, the Intel TBB coming in 2017 brings both technical improvements and becomes more open with the switch to an Apache* 2.0 license, which should enable it to take root in more environments while continuing to simplify effective use of multicore hardware.
Overview of Intel® Threading Building Blocks

Intel® Threading Building Blocks (Intel® TBB) is designed to let programmers easily leverage multicore processors using standard C++ compilers. It has several high-level components:

- Parallel algorithm templates
- Concurrent containers
- High-level tasking interface (tbb::task_group)
- A flow graph

It also has several lower-level components:

- Low-level tasking interface (tbb::task)
- Mutexes
- Atomic operations
- Memory allocator
- Thread-local storage

The parallel algorithm templates cover common high-level parallel patterns. For example, parallel_reduce does an element-wise operation on a sequence and collects a reduction value.

The concurrent containers are STL-like containers that enable safe parallel operations without locking. For example, concurrent_hash_map and concurrent_unordered_map allow multiple threads to insert items at the same time.

The high-level tasking interface provides easy fork-join parallelism. It loses some theoretical guarantees that can be recovered via the low-level interface; hence, the parallel algorithms internally use the low-level tasking interface.

The flow graph component covers a very different style of parallel programming, where a program is composed as a graph of nodes that send messages between each other. (See the following “Introduction to Intel® TBB Flow Graph” section.)

The atomic operations are for parallel programming experts writing lock-free algorithms. These operations have direct hardware support on many platforms.

The memory allocator has the typical C++ interface for an allocator. What sets it apart is that it scales well to a large number of threads.

The thread-local storage interface is like an STL container where each thread gets its own element.

Overall, Intel TBB provides a coherent set of components for programming multicore systems using ordinary C++ compilers.
Introduction to Intel TBB Flow Graph

Flow Graph is a subframework within Intel TBB that targets scenarios where parallelism is represented more naturally by decomposing program logic into a set of nodes of a graph, which communicate over edges of the graph. Some of the design forces that motivate use of Flow Graph include the following:

- There are dependencies between operations that do not fit a fork-join model.
- There is a large amount of persistent state associated with some nodes in the graph.
- First-in/first-out behavior is desired, as opposed to Intel TBB's tasking that tends to be last-in/first-out.

For example, suppose you have a stream of images and want to determine which ones are faces. Some face detectors run multiple tests and make a final determination, depending on the test results. Since such a program would be complicated to show in its entirety, I've simplified it to a Fibonacci face detector example (Figure 1). It takes a stream of integers from standard input and outputs the numbers that are Fibonacci numbers. For each input integer \( m \), it runs two tests:

- \( 5m^2 - 4 \) a perfect square?
- \( 5m^2 + 4 \) a perfect square?

An integer \( m \) is a Fibonacci number if, and only if, at least one test passes.

Figure 1 shows a flow graph for this problem, with nodes \( p \) through \( v \). The annotations on the edges show what data flows through each edge in response to an input value of 89:

It works like this:

- \( p \) reads a value from standard input and sends each value to \( q \).
- \( q \) broadcasts each value to \( r \) and \( s \).
- \( r \) and \( s \) each run one of the two tests and send their results to \( t \). The result is a pair of the original value and a Boolean flag indicating whether the value passed the test.
- \( t \) joins the two results into a single tuple of pairs.
- \( u \) checks if both flag values are true and, if so, passes the result on to \( v \).
- \( v \) writes values to standard output.

Figure 2 shows the complete program, which exemplifies several kinds of nodes.
```cpp
#include <cmath>
#include <tuple>
#include <cassert>
#include <iostream>
#include "tbb/flow_graph.h"

bool is_perfect_square(long x) {
    long y = sqrt(x);
    return y*y==x;
}

int main() {
    using namespace tbb::flow;
    graph g;
    source_node<int> p(g,
        [&](int& m) {return bool(std::cin >> m);});
    broadcast_node<int> q( g );
typedef std::pair<int,bool> rec;
    function_node<int,rec> r(g, 1, [](int x) {
        return std::make_pair(x,is_perfect_square(5L*x*x-4));
    });
    function_node<int,rec> s(g, 1, [](int x) {
        return std::make_pair(x,is_perfect_square(5L*x*x+4));
    });
    join_node<std::tuple<rec,rec>> t(g);
    multifunction_node<std::tuple<rec,rec>, std::tuple<int>> u(g, 1,
        [] (const std::tuple<rec,rec>& t, auto& outputs){  // C++14 polymorphic lambda
            rec rx = std::get<0>(t);
            rec ry = std::get<1>(t);
            assert( rx.first==ry.first );
            if (rx.second || ry.second)
                std::get<0>(outputs).try_put(rx.first);
        });
    function_node<int> v(g, 1, [](int z) {
        std::cout << z << std::endl;
    });
    // Build graph backwards, because nodes start immediately once connected.
    make_edge(u, v);
    make_edge(t, u);
    make_edge(q, r); make_edge(r, input_port<0>(t));
    make_edge(q, s); make_edge(s, input_port<1>(t));
    make_edge(p, q);
    g.wait_for_all();
    return 0;
}
```

Fibonacci face detector
For simplicity, the code presumes that $5m^2 + 4$ fits without overflow in a long and a double. The code constructs the nodes first, and then wires the edges, working backward through the graph to avoid losing any messages when the source $p$ starts sending messages immediately after it is connected.

- $p$ evaluates its functor repeatedly, once per message to be sent to $q$. Each invocation tries to read a value from `std::cin` and write it to $m$. The boolean return value indicates whether a value was written.
- $q$ broadcasts its input values to each of its two outputs.
- $r$ and $s$ apply their respective tests to each input item and each output a pair `<int, bool>`.
- $t$ joins its inputs into tuples. Because there are two inputs, the `make_edge` calls further down in the listing have to distinguish which input to connect to via function `input_port`.
- $u$ does the final filtering. The `outputs` parameter to the functor is a tuple of output ports, a `multifunction_node` is allowed to output any number of values to any of the output ports. This example needs only one output port (`std::get<0>(outputs)`) and always sends zero or one value per invocation.
- $v$ writes the final value to `std::cout`.

Except for the `multifunction_node u`, the library knows exactly how many messages each invocation of a functor generates, and can use this information to optimize for locality and optimize away message objects.

The constructors for $r$, $s$, $u$, and $v$ take a parameter, set to 1 in this example, which indicates how many input values can be processed in parallel. When set to $k$, the runtime is allowed to have up to $k$ invocations of the functor running in parallel. When $k > 1$, outputs are not necessarily delivered in input order, so care must be taken. In this example, that could be done by tagging values coming out of node $p$ with sequence numbers, and adding `tbb::flow::sequencer_node` objects to the graph to re-sequence out-of-order messages.

A node can have a nested flow graph or call Intel TBB parallel algorithms. For example, if we were dealing with huge numbers (millions of bits), then the tests in nodes $r$ and $s$ involve some FFT operations (just like image processing). Those operations could be parallelized. Thus, in the flow graph model, scalable parallelism can be achieved in any of three ways:

- Have nodes process multiple items in parallel.
- Have nodes call parallel algorithms or nested flow graphs.
- Make the number of nodes scale with the number of hardware threads.

The Flow Graph interface also supports more kinds of nodes than mentioned so far, such as ones for buffering (`buffer_node`), one-of-many routing (`split_node`), priority queues (`priority_queue_node`), dynamic throttling (`limiter_node`), etc. The overall goal is to support a variety of dataflow styles, from small static graphs to large dynamic ones.

(For more about the Flow Graph interface, see Intel® TBB Tutorial: Flow Graph.)
A TALE OF TWO HIGH-PERFORMANCE LIBRARIES

How Intel® Math Kernel Library and Intel® Threading Building Blocks Work Together to Improve Performance

Vipin Kumar E.K., Technical Consulting Engineer, Intel Corporation
Intel® Math Kernel Library (Intel® MKL) is more than twice as old as Intel® Threading Building Blocks (Intel® TBB). Historically, Intel MKL has used OpenMP* for parallelism. However, recent versions of Intel MKL (starting with Intel MKL 11.3) use Intel TBB instead of OpenMP. That means programs that use Intel TBB for parallelism benefit from using Intel MKL, which also uses Intel TBB.

This article explains:
- How to enable Intel TBB support
- A list of functions currently enabled with Intel TBB
- Usages, tips, and techniques to further improve performance

We also showcase performance advantages in the Intel TBB version.

**Intel MKL 11.3 Includes Intel TBB Support**

Intel MKL, Intel's high-performance math library, has included Intel TBB parallelism support beginning with version 11.3. With this threading layer for Intel TBB, if your application is already using Intel TBB, you can get the advantages of Intel MKL with Intel TBB parallelization.

As a first step, we enabled Intel TBB support for some of our users' most compute-intensive and heavily used functions. Depending on users' feedback, we will support more and more functions in upcoming versions.

**Table 1** shows the functions that were enabled with Intel TBB threading support in version 11.3.

<table>
<thead>
<tr>
<th>BLAS</th>
<th>All BLAS level 3 functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>?axpy, ?dot, ?gemv</td>
</tr>
<tr>
<td>Sparse BLAS</td>
<td>mkl_sparse_mv</td>
</tr>
<tr>
<td>Sparse Solvers</td>
<td>Intel® MKL PARDISO</td>
</tr>
<tr>
<td></td>
<td>Fast Poisson Solver</td>
</tr>
</tbody>
</table>

**Table 1.** Intel® TBB parallelism support in Intel® MKL
Table 1 shows the data types:

- `s` - real, single precision
- `c` - complex, single precision
- `d` - real, double precision
- `z` - complex, double precision

Therefore, `dgemv` means a double precision general matrix vector multiplication function.

Other Intel MKL 11.3 functions execute sequential code. These may benefit as well from internally calling a function from the list above. Depending on feedback from users, future versions of Intel MKL may support Intel TBB in more functions.

Using Intel TBB Support in Intel MKL

We made things very easy for users. You only need to link applications to Intel TBB and Intel MKL using Intel® C/C++ Compiler. Please note, only dynamic linking to the Intel TBB library is available, although Intel MKL supports both static and dynamic linking.

Let’s assume your application is programmed in C. For Linux* operating systems, you can build your application with the Intel C/C++ Compiler and link it with Intel TBB and Intel MKL by using the following commands:

- If you are using both dynamic libraries of Intel TBB and Intel MKL:
  ```bash
  icc program.c -mkl -tbb
  ```
- If you are linking with dynamic Intel TBB and static Intel MKL:
  ```bash
  icc program.c -static -mkl -tbb
  ```

If you are using Windows*, use the following command to compile your application program and link it to dynamic Intel TBB and Intel MKL:

- With dynamic Intel TBB and dynamic Intel MKL:
  ```bash
  icl.exe program.c -mkl -tbb
  ```

Fine-Tuning Techniques

You can further improve the performance of Intel MKL by making sure Intel TBB pins threads to the processor cores. This can be done using the `tbb::affinity_partitioner` class.

If the input data in your application is small, you can improve performance by reducing the number of threads of Intel MKL for small input data. You can use the `tbb::task_scheduler_init` class to achieve this.

Find more information on controlling the threading behavior of Intel TBB in the Intel TBB documentation.
Performance Comparison with OpenMP

Figure 1 shows the performance comparison of sequential Intel MKL, OpenMP-enabled Intel MKL, and Intel TBB-enabled Intel MKL for 10 simultaneous calls for LU, Cholesky, and QR factorization functions from LAPACK.

When Intel TBB threading is used with Intel MKL, the OpenMP-related thread control environment variables—like `OMP_NUM_THREADS`, `MKL_NUM_THREADS`, `MKL_DOMAIN_*`, `MKL_DYNAMIC`—and related API functions will not have any effect.

If there is light threading in the Intel TBB application (a single call to Intel MKL at a given moment, with no other heavy work in parallel in that moment), then OpenMP Intel MKL generally performs better.

If there is heavy threading in the Intel TBB application, with a lot of work happening in parallel to an Intel MKL call, then Intel TBB-enabled Intel MKL is ideal. The Intel TBB-enabled Intel MKL shows solid performance improvements through better interoperability with other parts of the workload.

For OpenMP, pthread, or sequential applications, OpenMP-enabled Intel MKL is preferable.

For More Information

To learn more, see the article [Using Intel® MKL with Threaded Applications](#) or the Intel MKL User’s Guide.
HETEROGENEOUS PROGRAMMING WITH INTEL® THREADING BUILDING BLOCKS

With New Features, Intel® Threading Building Blocks Can Coordinate the Execution of Computations across Devices

Alexei Katranov, Senior Software Development Engineer; Oleg Loginov, Senior Software Development Engineer; and Michael Voss, Software Architect, Intel Corporation
Due to energy constraints, it has not been practical for over a decade for chip designers to provide the levels of performance demanded by developers through increased CPU frequency. Computing systems have emerged that are increasingly heterogeneous in the quest for greater performance per watt through hardware that is specifically optimized for important computational kernels and application domains. As shown in Figure 1, the compute resources available in these systems may include general purpose application cores, integrated or discrete graphics processing units (GPUs), many-core coprocessors, field-programmable gate arrays, domain-specific integrated compute engines, and fixed-function IP blocks.

To reap the benefits of heterogeneous systems, though, developers generally identify the critical kernels in their applications and run these kernels on high-performance, low-power hardware accelerators. A variety of software libraries and programming models exist to manage computation on heterogeneous systems, such as those shown in Figure 2. These models vary widely in their level of abstraction, from low-level APIs like OpenCL™ to domain-specific libraries such as Intel® Math Kernel Library, with relatively high-level general interfaces such as OpenMP* in between. If high-level, domain-specific libraries exist that can take advantage of the available hardware resources, developers may be shielded from much, if not all, of the heterogeneity in their system. However, if no such libraries are available, developers may be forced to offload handwritten kernels using low-level or even device-specific programming APIs. Developers must also coordinate the scheduling of computations on the specialized hardware units with the computations running on the general application cores, and the communications between these compute resources.
Given the range of available models and the tradeoffs between them, it should be expected that in large, complex applications, there will not be a one-size-fits-all approach that works for all components and devices, but instead, that different approaches and models will be intermixed within the same application. Achieving best performance in a single application while coordinating these heterogeneous resources and this mix of software components will be challenging.

The Intel® Threading Building Blocks (Intel® TBB) library is evolving to meet this challenge with new features that allow it to serve as a coordination layer for applications that make use of a mix of offload models and high-level libraries. The Intel TBB library is particularly well suited for this task because it exposes both a graph API that can be used to express the high-level dependencies between large computations and an efficient work-stealing task scheduler that can be used to implement parallelism that will nest compositely within this higher-level graph.

Figure 3 shows how this combination of features can be leveraged to provide a highly composable coordination layer for heterogeneity. An Intel TBB flow graph can contain nodes that execute on the host (the green nodes) and other nodes that execute on different accelerators (the red and orange nodes). By using an Intel TBB flow graph to schedule the execution of these high-level nodes, a single task scheduler and thread pool will execute both the high-level tasks for coordination and the lower-level tasks for implementing any nested parallelism in the host nodes. This single point of scheduling can maximize composability, prevent oversubscription, and yield the best utilization of resources.

The Intel® TBB library can serve as both a composability layer and a coordination layer. Of course, maximal benefit from this approach is achieved when the computational nodes that are executed on the host use Intel TBB for their own nested parallelism. This allows tasks from all nodes, as well as for coordinating the graph layer, to be seen and scheduled by Intel TBB.

Intel TBB as it executes on the host has been described in other articles in this issue of The Parallel Universe; the remainder of this article describes the extensions made to the flow graph API to support coordination of computation when executed on other devices attached to the main host.
New Flow Graph Features for Heterogeneous Computing

The goal of the Intel TBB extensions described in this article is to support the coordination of heterogeneous hardware and software. Figure 4 shows the extended flow graph architecture that is being developed to support this effort. At the bottom of this software stack is the Intel TBB task scheduler. The flow graph API is built on top of this scheduler, and all parallel computations launched on the host by a flow graph are executed within TBB tasks.

![Intel® TBB flow graph architecture for heterogeneity](image)

To enable coordination of asynchronous accelerators, we have added an additional layer (the green layer in Figure 4) to support communication with asynchronous activities. This layer is necessary since Intel TBB tasks are executed by the Intel TBB scheduler nonpreemptively; that is, once an Intel TBB task begins executing on a worker thread, it will execute to completion on that thread, holding the thread the whole time. By default, there is only one Intel TBB worker thread per hardware thread, so when working with asynchronous accelerators, we cannot allow Intel TBB worker threads to idly spin while waiting for work to complete on a device. The green layer in Figure 4 adds facilities for asynchronous communication from within an Intel TBB flow graph, keeping worker threads free to do useful work.

On top of this asynchronous communication layer we provide additional support for streaming models, like OpenCL and DirectCompute*, represented by the yellow layer in Figure 4. And finally, at the very top level of our software stack, we provide specializations for particular models of customer interest, such as support for OpenCL. One can note that a user application may interface to Intel TBB through more than one layer of this stack.
Table 1 provides a brief overview of some of the specific features within these levels of our software stack. At the time of publication, most of these features are available as preview features in the library. Some features have not yet been released. It should be noted that since these features are still being actively developed and feedback is being solicited from our developer community, the interfaces, names, and usage models described in the following sections may change considerably in the future.

<table>
<thead>
<tr>
<th>async_node&lt;Input, Output&gt;</th>
<th>Basic building block. Enables asynchronous communication from a single/isolated node to an asynchronous activity. Available as preview feature since Intel TBB 4.3 U6</th>
</tr>
</thead>
<tbody>
<tr>
<td>async_msg&lt;T&gt;</td>
<td>Basic building block. Enables asynchronous communication with chaining across graph nodes. Available as preview feature since Intel TBB 4.4 U4</td>
</tr>
<tr>
<td>heterogeneous_node</td>
<td>Higher level abstraction for streaming models such as OpenCL™ or DirectCompute*. Must be customized for a specific API through a user-supplied factory. Tentatively planned for 2nd half of 2016</td>
</tr>
<tr>
<td>opencl_node</td>
<td>A customization of heterogeneous_node for OpenCL. Available as preview feature since Intel TBB 4.4 U2</td>
</tr>
</tbody>
</table>

Table 1. New features in Intel® TBB for supporting heterogeneous computing

Basic Building Blocks for Asynchronous Communication

As shown in Table 1, we have added two low-level interfaces for wrapping communication with asynchronous activities: async_node and async_msg. Both approaches require that developers provide the code to do the communication; these classes simply provide an interface that allows the user's code to plug in seamlessly to an Intel TBB flow graph application.

async_node

The main purpose of an async_node is to provide a simple, transparent interface between a flow graph and an external activity managed by the user or another runtime. This external activity may be an external thread pool, an integrated GPU, or some other accelerator.

As shown in Figure 5, an async_node invokes a user-provided function that submits a job to an external activity to start asynchronous calculations. The external activity then uses a gateway interface provided by async_node to return its result back to the next flow graph node(s). Since this is a basic building block, managing the external activity itself is the user's responsibility. However, this simple wrapper allows the rest of the flow graph to treat this node like any other flow graph node. The only difference is that much of the node's computational work happens outside of an Intel TBB task and therefore outside of the control of the Intel TBB runtime library.
An **async_node** provides an interface through which developers can hide asynchronous communication.

**async_msg**

Like **async_node**, the purpose of **async_msg** is to provide an interface between a flow graph and an external activity managed by the user or another runtime. **async_msg** does this by wrapping the messages that are sent between nodes instead of providing a new node type. The **async_msg** wrapper provides opportunities for interesting optimizations at the cost of a bit more complexity in the interface.

An **async_msg**, or a user type derived from **async_msg**, acts like a future that is specially handled by flow graph nodes. An **async_msg<T>** represents an asynchronous job result of type **T** that can be processed by the flow graph while the result may not be ready yet. The final result is (eventually) provided by an asynchronous activity with a call to the **async_msg<T>::set(T result)** method.

As shown in **Figure 6**, an **async_msg** can be used to implement an external calculation chain (or a graph). The first node in a processing chain (**n1**) can upload some data to an external asynchronous activity. On receiving an **async_msg**, the next node (**n2**) can request that additional operations be performed by the asynchronous activity on the result of the preceding node’s request. The last node in the calculation chain (**n3**) requires the final output result, and will receive the final result only when the asynchronous activity calls the set method.
A chain of nodes can communicate through async_msgs, enabling optimizations.

The TBB flow graph has been extended to respond to async_msgs in two different ways, allowing users to optimize asynchronous communications.

First, if an async_msg<T> object reaches a node or port in a flow graph that expects a plain old message of type T, the Intel TBB runtime intercepts any incoming async_msg<T> and forwards the final type T result to the port only when the async_msg has been finalized and its result is available. This allows async_msg objects to be sent to any existing flow graph node as a final node in a chain.

Second, developers can define their nodes to accept async_msg<T> objects directly. The Intel TBB runtime will not intercept these messages, but instead delivers them as soon as they are received. A user node that accepts async_msg types may communicate with the asynchronous activity from its body and/or update states or flags kept in the async_msg. Typically, developers will create model-specific message types that inherit from async_msg<T> so they can store additional states or flags that are updated while the data processing chain is working. Our OpenCL support, for example, embeds opencl_event objects in the messages that it sends between nodes.

As with async_node, if developers use async_msg as a building block to implement their own asynchronous system, it is their responsibility to manage the external activity.

Higher Level Support for Streaming Models

The async_node and async_msg are low-level abstractions. On top of these basic building blocks, the Intel TBB library provides higher-level support for streaming computation models, such as OpenCL or DirectCompute. This higher-level support is implemented in two parts: a part that is common across streaming models, abstracted through a class heterogeneous_node, and a customization part, abstracted by factories. We plan to support a limited set of predefined factories based on customer demand, with OpenCL being our first available target.
**heterogeneous_node**

*heterogeneous_node* provides an interface that the flow graph can use to efficiently communicate with devices that support features common to streaming computation models. The implementation of *heterogeneous_node* does not depend on any third-party APIs or libraries. Instead, it delegates all API and platform-specific operations to a *factory* that handles the specifics for a particular API.

The *heterogeneous_node* class delegates operations such as communication, synchronization, and memory management to the factory. It also invokes a user-provided device selection functor that can be used to tune the factory for a specific device if desired. The implementation of these operations will differ from API to API, but the flow graph can be shielded from these differences and efficiently communicate through the *heterogeneous_node* interface without knowing the details hidden by its factory.

**opencl_node**

*opencl_node* is currently released as a preview feature. *opencl_node* is a *heterogeneous_node* that is customized through the use of an *opencl_factory* and that communicates through *opencl_msg* objects that are derived from *async_msg*. The *opencl_factory* implements the specifics of offloading computations to OpenCL-powered devices and hides the details of kernel compilation, kernel enqueuing, and data transfer.

A listing for a simple “Hello World!” example that uses an *opencl_node* is shown in Figure 7. When instantiating an *opencl_node*, a developer provides a tuple of ports, an *opencl_program*, and a kernel name to the node. Additionally, an *opencl_device* can be provided if necessary. The node hides much of the complexity that is usually involved in initialization, compilation, and offload if OpenCL is used directly. An example of instantiating an *opencl_node* is shown in lines 05 and 06 of *main.cpp* in Figure 7.

```cpp
main.cpp:
01: int main() {
02:   using namespace tbb::flow;
03: 
04:   opencl_graph g;
05:   opencl_node<tuple<opencl_buffer<cl_char>>> clPrint( g, "hello_world.cl", "print" );
06: 
07:   const char str[] = "Hello, World!";
08:   opencl_buffer<cl_char> b( g, sizeof(str) );
09:   std::copy_n( str, sizeof(str), b.begin() );
10: 
11:   clPrint.set_ndranges( { 1 } );
12:   input_port<0>(clPrint).try_put( b );
13: 
14:   g.wait_for_all();
15:   return 0;
16: }
```

**hello_world.cl:**

```plaintext
01: kernel void print( global char *str ) {
02:     printf("OpenCL says ");
03:     for ( ; *str; ++str ) printf("%c", *str);
04: }
```

7 Listing for a "Hello, World!" OpenCL™ example (the kernel code is defined in *hello_world.cl*)
OpenCL does not work with memory allocated with the usual C/C++ allocation routines. To create a memory buffer on a device, special allocation functions must be used. To assist users in making use of our `opencl_node`, we provide an `opencl_buffer` class that hides the details of allocation and deallocation. An `opencl_buffer` accepts the element type of the buffer as a template argument and the number of elements as an argument to the constructor. An example of this is shown on lines 09 and 10 of main.cpp in Figure 7. In this example, an `opencl_buffer` is created at line 09 and the "Hello, World!" string is copied in to it.

OpenCL kernels are executed across a range. In Figure 7, we are processing a single string object, and so line 12 sets the single dimension of the `ndrange` to 1. And finally, we explicitly put the buffer to the `opencl_node` at line 13. This is, of course, a minimalistic example; in a more realistic example, the `opencl_node` will be connected to the rest of the graph through edges.

Because `opencl_node` communicates using `async_msgs`, chaining is supported, and kernels can be enqueued to OpenCL devices even before their inputs are ready. For example, in Figure 8, a diagram of a two-operation example is shown. In this example, two buffers (`b1` and `b2`) are multiplied and then the result is added to a buffer `b3`. Because our `opencl_node` supports chaining, as soon as `cl_mul(b1, b2)` is enqueued, a message is sent to the `cl_add` node. This message contains the id of the OpenCL device that was selected by `cl_mul` and an `opencl_event` that corresponds to the completion of the `cl_mul` operation. When `cl_add` receives this message and `b3`, it can immediately enqueue its kernel to the OpenCL device, even before the `cl_mul` kernel has generated its final output buffer.

![Diagram of a two-operation example](https://example.com/diagram.png)

**Figure 7** and **Figure 8** demonstrate only the most basic ideas of our OpenCL support. Interested readers can find a more detailed overview of the `opencl_node` on the Intel Developer Zone.
Putting It All Together

As was shown in Figure 3, best performance is obtained when Intel TBB is used as both the underlying thread engine for nested work on the host as well as the coordination layer that orchestrates the execution of work on all devices. To assist with the former, many domain-specific and math libraries provided by Intel use Intel TBB as their underlying threading engine. These include such libraries as the Intel® Math Kernel Library and Intel® Data Analytics Acceleration Library. To assist with the latter, we have introduced low-level and high-level support in this article.

Figure 9 shows a Mandelbrot example that puts together some of the concepts outlined in this article. A small flow graph is shown on the left that computes a fractal, assigning some tiles of the image to the CPU and some tiles to the integrated graphics unit. The pu_token buffer is prefilled with two tokens—one that represents the CPU and one that represents the iGPU. The tile generator generates ranges over which the fractal should be computed. These are paired with tokens and sent to the dispatcher. Based on the token received, the dispatcher sends the range to a node that computes the tile on the CPU or one that computes it on the iGPU. This is shown in the computed image on the right side of Figure 9, with the brighter portion computed on the CPU and the darker assigned to the iGPU. Using the extensions described in this article, the GPU node could be implemented as an async_node that wraps calls to a platform-specific GPGPU model, or it could be implemented using an opencl_node. The CPU node could be implemented using a nested Intel TBB parallel_loop. Because of the extensions added to the flow graph and the composability provided by the Intel TBB library, these different models will work seamlessly together.

9 Mandelbrot examples
Conclusion

This article presents some of the new features being added to the Intel TBB library to enable coordination of heterogeneous software and hardware. Many of these features are available as preview features in the most recent Intel TBB releases so that we can get feedback from our developer community. The `async_node` has been available since Intel TBB 4.3 U6. The `opencl_node` has been available since Intel TBB 4.4 U2. The `async_msg` was first released in Intel TBB 4.4 U4. The `heterogeneous_node` has not yet been released, but is tentatively planned for the second half of 2016. To find out more about these features and download the latest support, visit the `threadingbuildingblocks.org` website.

References

1. `heterogeneous_node` is planned as a preview feature in 2016 but is not yet released. These plans may change without notice.
Peeking at the picture of the completed puzzle.

PREPARING FOR A MANY-CORE FUTURE

Johns Hopkins University Adds Multicore Parallelism to Increase the Performance of Its Bowtie 2* Application

Kevin O’Leary, Software Technical Consulting Engineer, Intel Corporation; Ben Langmead, Professor, Johns Hopkins University; John O’Neill, Application Engineer, Intel Corporation; and Alexey Kukanov, Software Architect, Intel Corporation
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 Langmead Lab at Johns Hopkins University 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 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® Threading Building Blocks (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 nonuniform 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 Ivy Bridge system (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 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

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.

---

**Figure 2** Screen shot of Intel® Inspector XE, showing the thread data race issues that were identified in Bowtie 2.

For more complete information about compiler optimizations, see our Optimization Notice.
Performance Issues Encountered

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 our scientific users.

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 we were seeing, it is important to know these properties. The Intel TBB documentation describes the following qualities for mutexes:

- **Scalability**: The ability to keep the synchronization overhead constant with a growing number of contending threads
- **Fairness**: The 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

See [Mutex Flavors](#) in the Intel TBB documentation for details.

We are also interested in the following properties:

- **Thread awareness**: Whether the lock can be freed only 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.

<table>
<thead>
<tr>
<th>Mutex</th>
<th>Scalable</th>
<th>Fair</th>
<th>Recursive</th>
<th>Waiting</th>
<th>Thread-aware</th>
<th>Cohort detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>spin_mutex</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Polls</td>
<td>Oblivious</td>
<td>No</td>
</tr>
<tr>
<td>queuing_mutex</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Polls</td>
<td>See below</td>
<td>Yes</td>
</tr>
<tr>
<td>mutex</td>
<td>OS dependent</td>
<td>OS dependent</td>
<td>No</td>
<td>Blocks</td>
<td>OS dependent</td>
<td>No</td>
</tr>
</tbody>
</table>

**Table 1.** Traits and behaviors of mutexes

As indicated from the name, `tbb::spin_mutex` spins in user space while waiting. It is nonscalable, unfair, nonrecursive, 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, nonrecursive, 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 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.

We 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.

In Figure 3, you clearly see 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 Hyper-Threading Technology), we 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 us to investigate the cache coherence properties of the spin mutex on NUMA architectures. We ran some performance tests (Figure 4) using Intel VTune Amplifier, which showed a hotspot in the Intel TBB primitive for spinlock acquisition. This supported our theory that cache coherence due to contended reads and writes to the shared lock state was an issue.
Issue No. 2: NUMA

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 multisocket systems, we noticed degradation in performance. Figure 5 shows the results of these experiments.
The type of locks we use 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, we needed a way for our locks to be NUMA aware. (We will describe this in the next section.)

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 intersocket 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.

**Memory access analysis**

**Summary analysis view**

**Lock Cohorting: Implementing Locks That Are NUMA Aware**

In our final experiments with locking, we 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.: **Lock Cohorting: A General Technique for Designing NUMA Locks.**
As the paper shows, 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, in which ownership is transferred 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 before, `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 Dice paper 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`:

```cpp
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.

A 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, besides a `queuing_mutex`, also needs 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.
Prepare your software for next-generation Intel® Xeon Phi™ processors (a.k.a. Knights Landing) with the Developer Access Program. The program is open for developers worldwide, and it offers access to Intel Xeon Phi processor-based developer platforms and valuable training, tools, and support resources to help you optimize your code for the future of parallel computing.

Exclusive Developer Platforms
Fully configured, turnkey Intel Xeon Phi processor-based developer platforms—available in pedestal or rack configurations.

Liquid-Cooled Pedestal
- Developer edition of Intel® Xeon Phi™ processor: 16GB MCDRAM, 6 channels of DDR4, AVX 512
- Liquid-cooled system
- 6 DIMM slots
- 2 PCIe® 3.0 x16; 1x PCIe 3.0 x4 (in a x8 mechanical slot)
- 2 Intel® i350 Gigabit Ethernet
- 2 HDDs; 2 SSDs
- CentOS® 7.2
- Intel® Parallel Studio XE Professional Edition named user license

Air-Cooled Pedestal
- Developer edition of Intel® Xeon Phi™ processor: 16GB MCDRAM, 6 channels of DDR4, AVX 512
- Air-cooled system
- 6 DIMM slots
- 1 PCIe® 3.0 x16; 1 PCIe 3.0 x8
- 2 Intel® i210 Gigabit Ethernet
- 2 HDDs; 2 SSDs
- CentOS® 7.2
- Intel® Parallel Studio XE Professional Edition named user license

Rack Developer Platform
- Developer edition of Intel® Xeon Phi™ processor: 16GB MCDRAM, 6 channels of DDR4, AVX 512
- 4 2U hot-swap nodes
- 6 DIMM slots/node
- Riser 1: 1 PCIe® 3.0 x16, Riser 2: 1 PCIe Gen3 x20 (x16 or x4)/node
- 2 Intel® i210 Gigabit Ethernet/Node
- 12 3.5” hot-swap drives
- CentOS® 7.2
- Intel® Parallel Studio XE Cluster Edition named user license

Intel® Parallel Studio XE
Free, one-year license for Intel® Parallel Studio XE including compilers, libraries, and other Intel® software

Training and Resources
Knights Landing training, parallel programming webinars, community forums, how-to and optimization guides, and support from Colfax

Learn more and get started today: dap.xeonphi.com

Copyright © 2016, Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Inside, the Intel Inside logo, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.
The preferred method for lock operations in Intel TBB is to use a special `scoped_lock` class (for more information, see Mutex Concept in the Intel TBB documentation). 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 previously. 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 a `tbb::internal::padded` class template defined in `tbb_stddef.h`:

```cpp
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.

**Mutex Comparison in Bowtie2**

**Time of Slowest Thread on 120 Threads**

(200K reads per thread, slowest to fastest)

<table>
<thead>
<tr>
<th>Run Type/ Lock Type</th>
<th>Spin Mutex</th>
<th>Normal (Heavy) Mutex</th>
<th>Queue Mutex</th>
<th>Cohort Mutex**</th>
<th>Numa &quot;Node Bind&quot;***</th>
</tr>
</thead>
<tbody>
<tr>
<td>very-fast</td>
<td>00:06:38.207</td>
<td>00:03:31.638</td>
<td>00:02:46.012</td>
<td>00:02:29.866</td>
<td>00:00:54.319</td>
</tr>
<tr>
<td>fast</td>
<td>00:06:15.992</td>
<td>00:03:32.426</td>
<td>00:02:44.905</td>
<td>00:02:29.504</td>
<td>00:01:08.890</td>
</tr>
<tr>
<td>sensitive</td>
<td>00:06:29.325</td>
<td>00:03:36.371</td>
<td>00:02:47.158</td>
<td>00:02:31.084</td>
<td>00:01:36.222</td>
</tr>
<tr>
<td>very-sensitive</td>
<td>00:03:18.954</td>
<td>00:03:39.274</td>
<td>00:03:19.625</td>
<td>00:03:16.617</td>
<td>00:03:08.392</td>
</tr>
</tbody>
</table>

**Not a TBB built-in mutex.

***Threads are split up into four independent Bowtie* processes with 30 threads each. 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.

| Table 2. Mutex comparison |
A final strategy we have been pursuing 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” (Figure 9). We find that, in addition to the improvements we obtained by moving to queue locks and NUMA-aware locks, we obtained substantial additional improvement by 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 our initial work, we were able to track down threading data races using Intel Inspector. Once these correctness issues were addressed, we increased the performance by using Intel TBB with different mutexes that were better suited to our application. The NUMA issues we experienced were handled by lock cohorting and pinning threads to specific cores. Finally, by splitting reads from parsing in our critical section, we 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 this collaboration. By working directly with Intel engineers, and by using Intel tools and libraries such as Intel VTune Amplifier, Intel Inspector, and Intel TBB, we were able to effectively prepare these core genomics software tools for the many-core future around the corner.

Learn more

Course on genomics and algorithms for DNA sequencing

Johns Hopkins University
Split the problem into subranges that can be again considered as ranges.

Leading and Following The C++ Standard

Intel® Threading Building Blocks Adheres Tightly to the C++ Standard Where It Can—and Paves the Way for Supporting Parallelism Best

Alexei Katranov, Senior Software Development Engineer, Intel Corporation
From the beginning, the Intel® Threading Building Blocks (Intel® TBB) interfaces and user API were inspired by the C++ language. Intel TBB has improved and extended many C++ ideas to be suitable for parallel programming. Nevertheless, parallel paradigms are different from serial ones. This affects not only the implementation side of interfaces, but also the user API itself.

**STL-Like Idea**

Intel TBB provides a wide set of concurrent containers that are similar to their STL counterparts. Intel TBB modifies some STL interfaces to make it possible to implement them to be thread-safe. The key idea is to follow the spirit of STL, even if the interfaces must change slightly.

For example, one of the widely used containers is `std::vector`. Essentially, it is a dynamically allocated array with a random element access possibility. However, it cannot be safely used in parallel programming. Although C++11 allows using `const` qualified methods concurrently, any modification methods are not thread-safe. Intel TBB provides the `tbb::concurrent_vector` container that is similar to `std::vector` in some aspects.

Consider the following example that adds 10-integer numbers to `std::vector`:

```cpp
std::vector v;
for( int i = 0; i<10; ++i ) {
    v.push_back(i);
}
```

With Intel TBB, it can be easily transformed for parallel execution:

```cpp
tbb::concurrent_vector v;
tbb::parallel_for( 0, 10, [&v]( int ) {
    v.push_back(i);
});
```

We just replaced the usual `for` with `tbb::parallel_for` (powered with the C++11 lambda expression) and `std::vector` with `tbb::concurrent_vector`.

While Intel TBB provides concurrent versions of STL containers, in order to be thread-safe, the `std::vector` and `tbb::concurrent_vector` have slightly different interfaces. For example, `tbb::concurrent_vector` does not have the `insert` method but provides methods that are not present in `std::vector` such as `grow_by` or `grow_to_at_least`. Also, not all methods are thread-safe in `tbb::concurrent_vector` (e.g., constructors or whole container operations). Moreover, the `tbb::concurrent_vector` does not represent the contiguous array of elements. Therefore, `tbb::concurrent_vector` should not be considered to be a full replacement for `std::vector`. 
The basic idea of `std::vector` is that it is a dynamically allocated data structure with the following main properties:

- Random access that takes constant O(1) time
- Insertion of elements at the end that also takes amortized constant O(1) time

Despite the fact that `tbb::concurrent_vector` is different from `std::vector`, it satisfies both of these properties.

Parallel programming breaks some of the usual expectations and guarantees in STL. In a serial application, the order of elements in `std::vector` is the same as the order in which they are added to the container. However, in a parallel application, while one thread is adding elements, another thread may add an element between the elements of the first thread. Therefore, we cannot speak about the global order of elements in a container. However, the order of elements added by each thread is preserved.

Moreover, it becomes difficult to access a just-added element because another thread may add one more. Thus, Intel TBB extends some STL interfaces to return an iterator to a just-added element (e.g., `push_back` in `tbb::concurrent_vector`). However, STL iterators may be invalidated when the container continues growing. In a parallel application, it is difficult to guarantee that the just-obtained iterator will not be invalidated by another operation of a parallel thread.

Intel TBB has chosen another approach: `tbb::concurrent_vector` uses iterators that cannot be invalidated by concurrent growing. It should be noted that we usually tend to limit or reduce STL guarantees to provide an implementation suitable for parallelism. However, the iterator concept restrictions were relaxed; otherwise, it would be almost unusable in a parallel application.

Strictly speaking, it is impossible to implement concurrent replicas of STL containers. One of the reasons is the serial nature of some STL interfaces—for example, accessing elements in `std::queue`:

```cpp
int val = q.front();
q.pop_front();
```

In a serial code, it works as expected. However, there is a race between two threads simultaneously calling the `pop_front` method. As a result, both threads may obtain the same element with the `front` method and pop two different elements. The race is caused by the interfaces and cannot be overcome by any implementation approach. Thus, only an external lock can prevent the race and provide the desired behavior.
To overcome some interface limitations, Intel TBB provides special interfaces that are similar to STL ones but able to perform several operations at once. For example, `tbb::concurrent_queue` has the `try_pop` method that allows checking for the presence of an element and extracting it simultaneously. Though `tbb::concurrent_queue` tries to mimic `std::queue`, the basic idea of a first-in/first-out data structure is fragile in parallel programming.2

Not all STL containers have counterparts in Intel TBB. For example, Intel TBB does not provide an analog of `std::list`. (For a good discussion about this topic, see “Linked Lists: Incompatible with Parallel Programming?”3) In plainer words, the `std::list` concept has not been extended to parallel programming in a way that encourages performance. Therefore, Intel TBB has no concurrent version for `std::list`.

One of the main principles of Intel TBB is to remain as general as possible. Undoubtedly, we can implement a more efficient concurrent single producer/single consumer queue than `tbb::concurrent_queue`. However, it has only a limited number of applications and may lead to misuse when more than one consumer or producer accesses this queue. Therefore, Intel TBB balances many aspects of STL and parallelism, including:

- Efficiency
- Clear and usual concepts
- The variety of functionality
- Avoiding error-prone approaches

The best example of tradeoffs in Intel TBB is two implementations of hash-tables: `tbb::concurrent_unordered_map` and `tbb::concurrent_hash_map`. The former allows concurrent traversal but prohibits concurrent erasure. The latter allows concurrent erasure but replaces the idea of iterators with an accessor approach.

As we discuss STL, we should mention an allocator concept that is widely used in containers. The allocators, initially designed for serial execution, are often not suitable for parallel execution. The allocator market segment has shifted dramatically over the last decade, but Intel TBB was one of the first libraries that suggested an efficient allocator for parallel applications.4 The allocator concept seems to be one of a few STL concepts that turned out to be fully compatible with parallelism. Intel TBB provides its own allocators, `scalable_allocator` and `cache_align_allocator`, to cope with two main issues in parallel programming: scalability and false sharing.5
Leading C++

The iterator concept cannot be easily extended in parallel programming. Indeed, four of the five types of iterators in STL are naturally serial and introduce dependencies between loop iterations, eliminating the efficiency—or even the possibility—of parallelization. Only a random access iterator can be easily used in parallel. Therefore, Intel TBB developed a Range concept for parallel algorithms. The random access iterator can be easily transformed to the Range concept if needed. However, the main advantages of the Range concept are independence from data structures and layouts, and a native idea of divisibility.

The Range concept lets us think generically and not bind the parallel algorithms to the processing data-like iteration over an array. For example, `tbb::parallel_for` is a map higher-order function. It iterates over a given range and does not depend on user data structures. The idea of generic abstractions is essential in parallel programming because it allows the combining of several algorithms into only one parallel region, reducing the relative cost of parallelization.

The idea of divisibility is intensely used by the partitioners, in Intel TBB, to split the problem into subranges that can be again considered as ranges. The recursion idea allows distributing work among the available threads in a logarithmic number of steps (relating to the number of threads).

Long before C++11 introduced `std::atomic`, Intel TBB implemented `tbb::atomic`. Undoubtedly, atomic operations is a must-have feature in parallel programming, especially when we are speaking about lock-free algorithms. Interestingly, `tbb::atomic` is very similar to `std::atomic`; however, given the limitations of C++98, `tbb::atomic` did not originally provide any constructors to ensure correct behavior in some scenarios.

The atomic operations are not the only things that beat the clock. Another one is a mutex. Intel TBB provides a wide range of different mutexes. As mentioned in "Linked Lists: Incompatible with Parallel Programming?", none can be considered the best. Like atomics operations, the mutex is also a must-have feature because not all algorithms can be implemented in a lock-free way. Unlike `tbb::atomic`, which provided fewer constructors than `std::atomic`, `tbb::mutex` provides a copy constructor that is deleted in `std::mutex`. The C++ Standards Committee decided to prevent mistakes that could happen from copying a mutex while it is being locked or unlocked.
One more concept that is initially serial but cannot be avoided in parallel programming is exception handling. The following example just adds elements to an array and guarantees safe behavior in case of `std::bad_alloc`:

```cpp
try {
    std::vector<int> v;
    for ( int i=0; i<N; ++i ) {
        v.push_back(i);
    }
} catch ( std::bad_alloc ) {
    std::cout << "No memory" << std::endl;
}
```

The example can easily be parallelized with Intel TBB:

```cpp
try {
    tbb::concurrent_vector<int> v;
    parallel_for ( 0, N, []( int i ) {
        v.push_back(i);
    });
} catch ( std::bad_alloc ) {
    std::cout << "No memory" << std::endl;
}
```

It looks intuitive and simple. However, what if `tbb::parallel_for` is called from one thread but an exception is thrown on another thread? The Intel TBB runtime cannot know that `tbb::parallel_for` is called from the try block, but it does not mean that it is impossible to support this use case. To cope with exceptions, Intel TBB uses the `tbb::task_group_context` object that is created implicitly in each parallel algorithm. The Intel TBB task scheduling system catches all exceptions thrown from user code, sets the flag to the `tbb::task_group_context` object, and cancels the tasks within the current task group. When the parallel algorithm finishes, it checks whether the exception flag is present and rethrows the exception. Without C++11 support, this mechanism may rethrow only an approximation instead of an original exception.

The cross-thread exception propagation is only one side of the coin. The other side includes guarantees provided by containers when an exception is raised. Intel TBB containers try to offer a practical level of exception safety. For example, many Intel TBB containers remain usable when a constructor of an element throws an exception. However, `std::bad_alloc` can cause the container to be broken and only a limited set of operations is allowed.

Serial performance is often traded for better concurrency. Therefore, STL-like operation may be less efficient than STL originals, which regretfully break the pay-as-you-go principle of C++ that is also an important part of the Intel TBB approach. Theoretically, it is possible to provide dual interfaces for serial and parallel usage. However, this does not make interfaces easier for users, since it is not always clear where and which interface should be used. Although Intel TBB provides these types of interfaces (usually starting with the `unsafe_` prefix), they are mostly intended for limited or debug purposes.
The Era of C++11

It may look as if C++11 and further C++ standard improvements related to parallelism and concurrency make Intel TBB obsolete and make it seem that there is no reason to continue its development. Undoubtedly, some low-level functionality such as \texttt{tbb::mutex} or \texttt{tbb::atomic} are now redundant and can be really replaced with their counterparts from C++11. However, if we consider C++11 in detail, \texttt{std::async} is far from the tasking model. There is no Range concept that makes Intel TBB parallel algorithms highly generic, no analog to Intel TBB flow graph, etc. Conversely, C++11 brought much-desired features and gave a second life to Intel TBB.

A C++11 lambda expression is in a sense just syntactic sugar; however, Intel TBB parallel algorithms benefit highly from it. Consider the old-style parallel algorithm increasing values in an array by one:

```cpp
class IncrementBody {
    const std::vector<int> &myArray;
    public:
        IncrementBody( const std::vector<int> &arr ) : myArray(arr) {}
        void operator() ( int i ) { ++myArray[i]; }
    }
    void increment(const std::vector &arr) {
        tbb::parallel_for( (size_t)0, arr.size(), IncrementBody(arr) );
    }
```

It looks very verbose for a one-increment operation. Let us rework it using a lambda expression:

```cpp
void increment(const std::vector<int> &arr) {
    tbb::parallel_for( (size_t)0, arr.size(), [&arr](int i) {
        arr[i]++;
    });
}
```

We can see that writing code using Intel TBB is much nicer with the C++11 lambda feature!

One more syntax-related feature in C++11 is initializer lists. Previously, only C-style arrays and structures could be initialized with braces. Starting with C++11, all STL containers support initializer lists for construction and some assignment operations. Intel TBB has adopted the new feature and now we can construct concurrent containers with a bunch of values:

```cpp
tbb::concurrent_vector<int> v = { 0, 1, 2 };
```

However, initializer lists can be useful for some sort of optimization and additional guarantees, especially in parallel programming. Let us consider the following example:

```cpp
v.grow_by( { 3, 4, 5 } );
```
In the example, we add three values to the vector. Theoretically, it can be more efficient than three consecutive `push_back` method calls. Moreover, we have a guarantee that our values are not interchanged with values added by other threads.

C++11 introduced not only syntax-related features but also efficient oriented features such as move semantics. Unfortunately, move semantics cannot improve performance automatically without direct support of used data structures. Undoubtedly, STL containers implement move constructors, assignment operators, and other move-semantics-related stuff. Nevertheless, if we implement our own data structures without using STL ones, we need to implement the move semantics support explicitly. Therefore, Intel TBB extended its containers interface and implementation to align with C++11.

In the previous section, we discussed the exception propagation in parallel programming. It was pointed out that only an approximation of the originally thrown exception can be propagated. The C++11 `exception_ptr` feature was used to improve Intel TBB exception propagation functionality. It was one of the first C++11 features that forced Intel TBB to provide two versions of the library. Fortunately, the C++11 version of the Intel TBB library is fully compatible with previous versions of Intel TBB and can be used with most modern compilers.

**Conclusion**

Intel TBB has followed C++ language principles from its inception. However, it has not duplicated the STL interfaces exactly, but adapted them to parallelism. Moreover, Intel TBB does not provide replicas of all STL algorithms and containers. The guiding principle is to extend the spirit of C++ concepts and principles to be suitable for parallel programming. Nevertheless, Intel TBB has not limited itself with STL ideas. New concepts and thread-safe interfaces have been introduced. Some of these became obsolete with the introduction of C++11; others have not. Overall, they have given Intel TBB a new life with new concepts, features, and interfaces.

**References**

1. "tbb::concurrent_vector: Secrets of Memory Organization"
2. "TBB containers vs. STL. Functionality rift"
3. "Linked Lists: Incompatible with Parallel Programming?"
5. "Scalable Memory Allocator"
6. "Map (High-Order Function)"
7. "Why Atomic<T> Has No Constructors in C++03 Mode"
8. "Mutex, Lock, Condition Variable Rationale"
A long and bright future for Intel® TBB

INTEL® THREADING BUILDING BLOCKS: TOWARD THE FUTURE

The Architect of Intel® Threading Building Blocks Shares Thoughts on the Opportunities Ahead

Alexey Kukanov, Software Architect, Intel Corporation

For more complete information about compiler optimizations, see our Optimization Notice.
In the ever-changing industry landscape, **parallel programming** matters more today than it did 10 years ago. Intel® Threading Building Blocks (Intel® TBB) was born at the beginning of the multicore era with the purpose of making parallel programming easier for nonexperts. It achieved that goal, and, overall, it was a successful project. It was used in many applications and libraries over the computing spectrum, ported by the community to a variety of platforms (including non-Intel® architectures such as ARM* and Power*), and influenced third-party solutions for parallel programming. It has received two industry awards and was once named by an independent research company as the most-used threading abstraction library.

After a decade in the field, Intel TBB is alive and well. The recent development of the C++ standard toward adding features for concurrency and parallelism did not make it obsolete, except for low-level functionality such as atomics and threads. The high-level generic parallel patterns, concurrent containers, and the flow graph have no matches in the standard at this time.

Nevertheless, this is no time for complacency. The new era of silicon evolution brings in complex systems with multiple independent and diverse programmable units and different memory types. Programming such systems is more challenging than ever. We foresee Intel TBB as one of the key technologies to address these new challenges. And for that, the library should evolve and expand. There are multiple ideas for improvements and new features, based on the feedback from users as well as observed industry changes.

In this article, I describe some generic and notable ideas we are looking at. But, please be aware that for most of those, there are no well-defined plans yet.

**Heterogeneous Programming**

Hardware and software heterogeneity is a major industry trend, so one of the most important directions for Intel TBB is to facilitate heterogeneous programming, similar to the way it did for multicore programming in the past. Since 2015, Intel has added new functionality to the flow graph API as a coordination layer for heterogeneous programs. (You can read more about the effort in *Heterogeneous Programming with Intel® Threading Building Blocks* on p. 21)

We are considering adding heterogeneous variants of the parallel patterns that would offload execution to other devices. This is a challenging task, however; compiler support seems essential for building a convenient and efficient API, and all existing programming models for heterogeneous parallelism use special compilers, while the library-only approach that Intel TBB sticks to is a disadvantage. Another heterogeneous idea we ponder is how to extend parallel patterns with a possibility to execute across multiple devices (such as the CPU and GPU) with automatic load balancing, provided that there is a sufficient amount of work for it to make sense performance-wise. We are also considering extending the flow graph to support coordination...
between independent compute nodes in a cluster, thus adding capabilities for distributed memory programming with Intel TBB. Of course, this functionality would internally utilize MPI or other communication interfaces to coordinate execution and pass data between graph nodes running on different machines.

Interestingly, Intel TBB is mostly hardware-agnostic internally and so far has very few hardware technology-oriented features, the most notable one being speculative locks implemented on top of Intel® Transactional Synchronization Extensions (see *The Parallel Universe, Issue 18*). We foresee more features of this sort, particularly for supporting new types of memory such as high-bandwidth memory and nonvolatile memory. The users will likely be interested in the Intel TBB memory allocator handling these memory types, so we are working on that. We also see an opportunity in designing persistent containers that utilize nonvolatile memory to preserve the stored data and the internal state after an application has stopped or been aborted, without the need for programmers to explicitly write the data to a file and then read it on the next launch.

One of the major factors for Intel TBB evolution during all these years was the feedback from its users. A number of features, from the *parallel_do* function template in version 2.1 to the flow graph API in 4.0 to the *global_control* class in version 4.4, not to mention many improvements and fixes, were driven by user requests and feedback. Beyond all doubt, this practice will continue. Some recent requests—such as providing alternative mechanisms for work isolation and reducing idle spinning when threads cannot find tasks to execute—are in Intel's near-term plans.

**Thread Affinity**

One recurrent question about Intel TBB is how to set affinity for its threads. Historically, Intel TBB deliberately avoids exposing its worker threads to the application, and in particular playing with thread affinity. The library primarily targets complex modular applications in dynamic, versatile environments when resources need to be shared and where good composability and load balancing are more important for performance than fragile platform-specific optimizations. In such cases, pinning threads to certain hardware resources might hurt more than it helps, as it reduces OS possibilities of thread scheduling and might result in resource contention.

But, for some applications typically having roots in high-performance computing, composability is not a big concern and performance is king. Preventing thread migration between CPU cores/sockets via affinity might improve performance, and affinity is also necessary to colocate data and threads processing these data on NUMA systems. Due to the complexity and risks, support for affinity in Intel TBB is limited to providing the points where it can be set for worker threads. With the help of *task_scheduler_observer* and *task_arena* classes, programmers can group worker threads in accordance with machine topology (or in a different way) and pin them to
desired cores using an OS-specific affinity API. However, some of our users are not quite satisfied with this solution, so we need to think if their experience can be improved without compromising the core design principles and impacting all other users. And for NUMA-sensitive applications, we need to explore if enabling affinity is sufficient and programmers can do the rest themselves, or if more direct support is necessary (e.g., in the form of data placement hints for parallel algorithms).

**Backward Compatibility**

As mentioned by Arch Robison in his article “The Genesis and Evolution of Intel® Threading Building Blocks” on p. 7, Intel TBB has established a strong tradition of backward compatibility, for both its public API and binaries. In fact, Intel TBB 4.4 binaries are still link-compatible with programs written years ago for Intel TBB 2.0. But backward compatibility has downsides, such as increased binary size for all users. There is obsolete functionality that we would like to finally remove, and most of the source code could be simplified and improved if we drop support for C++03. Together, all of it provides a good rationale for breaking backward compatibility once.

However, a compatibility break is not an easy decision, as Intel TBB is used by other libraries, such as Intel® Math Kernel Library (Intel® MKL) and Intel® Data Analytics Acceleration Library (Intel® DAAL), as well as popular open source frameworks such as OpenCV. Our current recommendation for applications that use both Intel TBB and these libraries is to maintain a single, most recent version of Intel TBB, relying on its strong backward compatibility. In case we make incompatible changes, all these libraries and applications will need to update to the new version, which cannot be done in a day. So, for a period of time, incompatible versions of the library will need to coexist in the field, not conflicting with each other and avoiding competition for CPU cores. For this and other reasons, we plan to invest in improved coexistence and resource sharing between multiple instances of Intel TBB used within an application.

Once we have a solution for multiple instances of Intel TBB to coexist well, and can make incompatible modifications, we plan to switch the code to C++11. Actually, this shift already has started with adding C++11 specific extensions to existing classes (see Alexei Katranov’s article, Leading and Following the C++ Standard, on p. 46). Some of the new functionality is so much better with modern language features such as variadic templates that we do not make it available for old C++03 compilers, and this trend will continue. Eventually, support for pre-C++11 compilers will be fully dropped, and so will APIs that have become obsolete over time. The following table lists functionality that will likely be removed in later versions, along with the recommended alternatives. Code modification will be required but, in most cases, it should be rather straightforward thanks to similar API semantics.
### Table 1. Functionalities that may be removed in future versions of Intel® TBB

<table>
<thead>
<tr>
<th>Class/Template</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>parallel_while class template</code></td>
<td>Deprecated long ago. Use <code>parallel_do</code> or <code>parallel_pipeline</code> instead.</td>
</tr>
<tr>
<td><code>pipeline class</code></td>
<td>Use <code>parallel_pipeline</code> instead. For thread bound filters, use <code>async_node</code> in the flow graph.</td>
</tr>
<tr>
<td><code>tbb_thread class, std::thread (compat/thread), std::condition_variable class (compat/condition_variable)</code></td>
<td>These classes were developed to ease migration to C++11. Use the corresponding standard classes instead.</td>
</tr>
<tr>
<td><code>atomic class template</code></td>
<td>Use <code>std::atomic</code> instead.</td>
</tr>
<tr>
<td><code>mutex class, recursive_mutex class</code></td>
<td>Use corresponding standard classes instead.</td>
</tr>
<tr>
<td><code>tick_count class</code></td>
<td>Use <code>std::high_resolution_clock</code> instead.</td>
</tr>
<tr>
<td><code>captured_exception, movable_exception</code></td>
<td>Classes for wrapping exceptions propagated from parallel code. In C++11, exceptions can be propagated “as is,” so use proper exception types instead.</td>
</tr>
</tbody>
</table>

## Extensions for Parallel Execution

Another direction of development driven by C++ evolution is implementing the so-called “Parallel STL” —the extension for standard library algorithms such as `std::copy, std::transform`, etc., to be possibly executed in parallel. Currently it is documented in the “Technical Specification for C++ Extensions for Parallelism,” and is on the way to the next version of the standard, C++17. The specification does not mandate in which way parallelism is achieved, leaving it to implementers. For applications that use Intel TBB (or another library based on it), parallel versions of STL algorithms are best implemented on top of Intel TBB as well, to avoid performance loss due to CPU oversubscription by multiple thread pools. So we are going to provide our own implementation of this part of a future C++ standard. Our implementation will use vectorization capabilities of Intel processors, and also provide an additional policy for serial vectorized execution of algorithms, which has been proposed to the C++ standards committee, but is not yet in the specification.

There are many other proposals for C++ language and library features related to concurrency and parallelism that are being considered for standardization. We will keep looking at these and continue making adjustments in our library to follow the standard. Meanwhile, we are looking for opportunities to enable parallelism in other ecosystems as well. Intel® Distribution for Python® provides an Intel TBB-based task pool that can substitute the standard implementation used by Python libraries, and provide better performance as well as composability with other modules utilizing Intel MKL (see the blog post [Unleash Parallel Performance of Python Programs](#)). In the future, we plan to cooperate with the Python community to enable concurrency and parallelism in more libraries.
More to Come

With all these opportunities, I see a long and bright future for Intel TBB. Multicore processors are now ubiquitous, and enabling parallelism is more important than ever. The new trend on diversity and heterogeneity of computing poses new programmability challenges to tackle. The C++ language keeps evolving, and so does our library, both using the new language features and providing functionality that complement the standard in various ways. And, of course, we keep listening to our users and making changes in response to their needs and feedback.

One important change we are implementing this year is releasing the Intel TBB source code under an Apache* 2.0 license. Ten years ago, early adopters requested that we open the source code; looking back now, we realize that it opened doors for Intel TBB adoption by the industry. At the time, GPL seemed the most appropriate license for the purpose. Since then, Intel has opened the sources for other software, including Intel® OpenMP* runtime and the brand-new Intel DAAL, both with less restrictive licenses—again, based on input from users. Now it is time for Intel TBB to realign with changing needs and with other open source projects, and we expect this to open doors into more environments and applications.

There is a lot of work ahead—but also a lot of fun and excitement. Let’s move on!
BIRDS OF A FEATHER CODE WELL TOGETHER

Tools for an Agile World
Get Intel® Threading Building Blocks along with other fine-feathered friends who are part of the Intel® Performance Libraries:

Intel® Math Kernel Library
Intel® Integrated Performance Primitives
Intel® Data Analytics Acceleration Library

These prebuilt, tested, and performance-optimized software libraries are all available free under community licensing. So you can create better, faster applications with no fees, no royalties, and no restrictions.

Fly Me to the Nest >