Flow Graphs, Speculative Locks, and Task Arenas in Intel® Threading Building Blocks

By Alexey Kukanov, Software Architect, Vladimir Polin, Software Engineering Manager, and Michael J. Voss, Software Architect, Threading Runtimes, Developer Products Division, Intel Software and Services Group

Introduction

The Intel® Threading Building Blocks (Intel® TBB) library provides software developers with a solution for enabling parallelism in C++ applications and libraries. The well-known advantage of the Intel TBB library is that it makes parallel performance and scalability easily accessible to software developers writing loop- and task-based applications. The library includes (Figure 1) a number of generic parallel algorithms, flow graph interface, a work-stealing-based task scheduler, concurrent containers, scalable memory allocation, thread local storage, and synchronization primitives. By mapping essential parallel patterns to these building blocks, developers build robust parallel applications that abstract platform details and threading mechanisms, while achieving performance that scales with increasing core count.
Structured *Parallel Programming*\(^1\) describes a number of useful parallel patterns. Intel TBB algorithms map to those patterns as follows:

- `parallel_for`: map
- `parallel_reduce`, `parallel_scan`: reduce, scan
- `parallel_do`: workpile
- `parallel_pipeline`: pipeline
- `parallel_invoke`, `task_group`: fork-join
- `flow_graph`: plumbing for reactive and streaming apps

---

\(^1\) Intel® Threading Building Blocks features.
The Intel TBB library can be used with any C++ compliant compiler. It also supports C++11 features, such as lambda expressions, that reduce coding efforts by removing the need to implement special functor classes. Different Intel TBB library components can be used independently, and even mixed with other threading technologies.

Intel TBB version 4.2 introduces new features and improvements that include support for the latest Intel® architecture features, such as the Intel® Transactional Synchronization Extensions technology (Intel® TSX) and Intel® Xeon Phi™ coprocessors for Windows* OS. It also provides support for Windows Store® and Android* applications2,3 and introduces some new functionality, as well as adds a number of improvements to its popular features (for more details, see the library documentation). Intel TBB 4.2 is available as a standalone product and is also a component of software bundles such as Intel® INDE, Intel® Cluster Studio XE, Intel® Parallel Studio XE, Intel® C++ Studio XE, Intel® Composer XE, and Intel® C++ Composer XE.

The open source version of the Intel TBB library supports more architectures, compilers, and operating systems, thanks to contributions from the community. It can work with the Solaris® OS with the Oracle® C++ compiler for Intel architecture and SPARC*-compatible processors, IBM Blue Gene* supercomputers and PowerPC*-compatible architectures, Android OS, Windows Phone* 8 OS and the Windows RT* OS for ARM*-compatible architectures, FreeBSD* OS, Robot* Operating System, and many other platforms where GCC* built-in atomics are supported. This broad cross-platform support means that once an application is written using the Intel TBB library, it can be ported to many different OSes or architectures without rewriting and re-debugging the core algorithms.

In the remainder of this article, we focus on a few specific features of the library. First, we provide an overview of the flow graph interface4 available since Intel TBB 4.0. We then present two new features introduced in Intel TBB 4.2: speculative locks that take advantage of the Intel Transactional Synchronization Extensions (Intel TSX) technology and user-managed task arenas that provide enhanced concurrency control and work isolation.

### Expressing Dependency and Data Flow Graphs

While the Intel TBB library is well known for its loop-based parallel algorithms, the flow graph interface4 extends its capabilities to allow fast, efficient implementations of dependency graph and data flow algorithms, enabling developers to exploit parallelism at higher levels in their application.
For example, consider a simple application that invokes four functions sequentially, as shown in Figure 2(a). A loop-parallel approach would involve examining each of these functions, looking for parallelism that can be exploited with algorithms such as parallel_for and parallel_reduce. In some cases this is sufficient for good performance; but in others, there may not be enough parallelism to exploit at that level. For example, Figure 2(b) shows the application after loop-parallelism is found and exploited in functions B and D. The execution time will improve, but what if a greater level of performance is still desired?

Sometimes, invoking functions one after the other can be overly constrained: a total order must be chosen for invocations, even if only a partial order is required. In Figure 2(a), the functions were ordered by the developer to ensure that each function executes after all of the values it needs have been computed, but what if functions B and C both require the output generated by A, but C does not depend on the output of B? Figure 2(c) shows the graph- and loop-parallel implementation of the example. In this implementation, the loop-level parallelism is exposed and the total ordering is replaced with a partial ordering that would allow B and C to execute concurrently.

The Intel TBB flow graph interface allows developers to easily express graph parallelism. It provides features to support the dependency, streaming, and data flow graphs that can be found in many domains, such as media, gaming, finance, high performance computing, and healthcare. This interface is a fully supported feature of the library, and has been available since Intel TBB 4.0.
When using a flow graph, computations are represented by nodes and the communication channels between these computations are represented by edges. The user is responsible for using edges to express all dependencies that must be respected when nodes are scheduled for execution, giving the Intel TBB scheduler the flexibility to exploit the parallelism that is explicit in the graph topology. When a node in the graph receives a message, an Intel TBB task is spawned to execute its body object on the incoming message.

The flow graph interface supports several different types of nodes (Figure 3), including functional nodes that execute user-provided body objects, buffering nodes that can be used to order and buffer messages as they flow through the graph, aggregation and deaggregation nodes that join and split messages, and other special purpose nodes. Users connect instances of these node types together with edges to specify the dependencies between them and provide body objects to perform their work.

Below is the source code for a simple “Hello World” flow graph application. This example is very simple and does not contain any parallelism, but it does demonstrate the syntax of the interface. In this code, two nodes are created, hello and world, which are constructed with lambda expressions that print “Hello” and “World” respectively. Each node is of type continue_node, a functional node type provided by the interface. The make_edge call creates an edge between the hello node and the world node. Whenever a task spawned by the hello node completes, it will send a message to the world node, causing it to spawn a task to execute its lambda expression.
In the code above, the call to `hello.try_put(continue_msg())` sends a message to the `hello` node, causing it to spawn a task to execute its body object. When that task completes, it sends a message to the `world` node. Only when all of the tasks spawned by the nodes are complete, does the call to `g.wait_for_all()` return.

The Intel TBB flow graph interface enables the expression of very complex graphs that may include thousands of nodes and edges, cycles, buffering, and more. Figure 4 shows a visual representation of two flow graph implementations of Cholesky Factorization that use an algorithm similar to that described in Performance Evaluation of Concurrent Collections on High-Performance Multicore Computing Systems. In Figure 4(a), each invocation of the Intel® Math Kernel Library (Intel® MKL) functions `dpotf2m`, `dtrsm`, `dgemm` and `dsyrk` is a distinct node in the flow graph. In this case, the graph is large with many nodes per tile of the matrix, but is easily created by modifying the original sequential Cholesky Factorization loop nest, replacing function invocations with node and edge creations. In this graph, the edges are used to communicate before-after relationships; each node must wait until all of its predecessors are complete. It is easy to see the parallelism available in this graph.
Figure 4(b) shows an alternative implementation that is also possible to express with the flow graph interface. This version is a small, compact flow graph, where there is only a single node for all invocations of each Intel MKL function type. In this implementation, the tiles are passed as messages between nodes in the graph. When a node receives a set of matching tiles, it spawns a task to apply its body object to this set of tiles and then forwards the new tile it generates on to other nodes. The **parallelism** in this graph comes from the library's ability to execute multiple instances of each flow graph node concurrently.
While the details of the implementations described in Figure 4 are beyond the scope of this article, this example demonstrates that the Intel TBB flow graph interface is a powerful and flexible abstraction. It can be used to create large, directed-acyclic graphs such as that shown in Figure 4(a), where the developer chose to create a separate node for each Intel MKL call. And, it can also be used to create compact data flow graphs that include cycles, joins, and conditional executions as shown in Figure 4(b).

More information on the flow graph interface is available in the Intel TBB Reference Manual.

Speculative Locks

Intel TBB 4.2 features speculative locks: new synchronization classes based on Intel TSX technology. A speculative lock may allow protected critical sections to proceed simultaneously, under an assumption that data accesses and modifications do not conflict with each other. If in practice a conflict occurs, one or more of the speculative executions should be invalidated, leaving the protected data intact and thus remaining invisible to other threads. Threads involved in a conflict will then repeat their critical sections, possibly taking a real lock for data protection (as Intel TSX technology does not guarantee that speculative execution will eventually succeed).

With the Intel TBB implementation of speculative locks\(^6,7\), all these steps happen under the hood and the programmer can use the customary mutex API. Moreover, on a processor that does not support Intel TSX technology, the implementation will automatically fall back to a regular lock, thus allowing developers to write portable programs that may benefit from transactional synchronization.

The Intel TBB library currently provides two mutex classes that support Intel TSX technology: speculative_spin_mutex and speculative_spin_rw_mutex. The latter was added as a preview feature in Intel TBB 4.2 update 2.

The class speculative_spin_mutex is API-compatible and very similar in nature to the class spin_mutex; both reside in the same header file tbb/spin_mutex.h. The main difference of speculative_spin_mutex, besides the ability to utilize Intel TSX, is its size. In order to avoid sharing of a cache line with any other data, which would likely result in a high degree of execution conflicts and related performance loss, an instance of speculative_spin_mutex occupies two cache lines.
Here is a code sample that uses an Intel TBB speculative lock:

```c++
#include <tbb/spin_mutex.h>
#include <set>

std::set<int> g_Set;

void thread_safe_add_to_set( int value ) {
    tbb::speculative_spin_mutex::scoped_lock lock(tsx mtx);
    g_Set.insert(value);
}
```

The class `speculative_spin_rwlock`, as you may guess from its name, implements a read-write spin lock with speculative execution. One may note that any speculative lock is by definition non-exclusive, allowing not only reads, but even non-conflicting writes being done simultaneously. So “speculative RW lock” may sound like a tautology. Remember, however, that a thread may acquire the lock “for real.” In this case, `speculative_spin_mutex` has to provide exclusive access for the thread, no matter whether it reads or modifies the data; therefore, it is not really a read-write lock. On the other hand, `speculative_spin_rwlock` will always allow multiple readers to proceed. Moreover, both real and speculative readers may run simultaneously. But this comes with a cost: internal data fields of the class need to be kept at separate cache lines. Together with false sharing avoidance, these lead to an instance of `speculative_spin_rwlock` occupying three cache lines.

Though `speculative_spin_rwlock` currently resides in `tbb/spin_rwlock.h` header file, and even uses `spin_rwlock` as part of the implementation, the two classes are not fully compatible. There are no `lock()` and `unlock()` methods in `speculative_spin_rwlock`. Rather it enforces a scoped locking pattern, that is, it has to be accessed via class `speculative_spin_rwlock::scoped_lock`. Since it is a preview feature, a TBB_PREVIEW_SPECULATIVE_SPIN_RW_MUTEX macro should be defined to a non-zero value prior to including the header file.

Unfortunately, the applicability and benefit of speculative locks are rather problem-specific. You should not think of these new classes as “better spin locks”—careful performance measurement is necessary to decide if speculative locks are the right instrument for your use cases.
User-Managed Task Arenas

Another significant piece of new functionality recently added to the Intel TBB library is user-managed task arenas.

In our terminology, an arena is a place for threads to share and steal tasks. Initially, the library supported just one global arena for an application. Later, we changed it to maintain separate arenas for each application thread, motivated by feedback that work submitted by different threads should be better isolated. Then, we received requests for concurrency control and work isolation to not be tied to application threads. To address this, we introduced user-managed task arenas. Currently, it is still a preview feature (and requires the TBB_PREVIEW_TASK_ARENA macro to be non-zero), but we are working toward making full support available later in 2014.

The API to user-managed arenas is provided by class `task_arena`. When creating a `task_arena`, a user may specify its desired concurrency and how much of this concurrency should be reserved for application threads.

```cpp
#define TBB_PREVIEW_TASK_ARENA 1
#include <tbb/task_arena.h>

tbb::task_arena my_arena(4, 1);
```

In the given example, an arena is created for four threads—where one place is always reserved for an application thread. It means that up to three worker threads managed by the Intel TBB library can join this arena and work on the tasks shared there. There is no limit on the number of application threads submitting jobs to the `task_arena`, but the overall concurrency will still be limited by four. All “extra” threads will be unable to join the arena and execute tasks there.

In order to submit some work to an arena, one should call its `execute()` or `enqueue()` methods:

```cpp
my_arena.enqueue( a_job_functor );
my_arena.execute( a_job_functor2 );
```

A job for any of these methods can be represented by a C++11 lambda expression or an instance of a functor class. The two methods differ in the way the job is submitted to the arena. `task_arena::enqueue()` is an asynchronous call for submitting a fire-and-forget job; the caller thread does not join the arena and returns immediately. In contrast, `task_arena::execute()` does not return until the submitted job is completed. If possible, the caller thread joins the arena and executes tasks there, otherwise it is blocked for the time necessary to complete its job.
While it is possible to submit a lot of serial job pieces into a task_arena, it is not an intended use case. Typically, one submits a job that creates sufficient parallelism, for example, with a parallel_for call:

```cpp
my_arena.execute( [&]{
    tbb::parallel_for(0,N,iteration_functor());
});
```

or with a flow graph:

```cpp
tbb::flow::graph g;
... // create the graph here
my_arena.enqueue( [&]{
    ... // start graph computations
});
... // do something else
my_arena.execute( [&]{
    g.wait_for_all();
}); // does not return until the flow graph finishes
```

More information on task arenas is available in the Intel TBB Reference Manual.

**Summary**

The Intel TBB C++ template library offers a rich set of components to efficiently exploit higher-level, task-based parallelism and implement portable future-proof applications to tap the power of multicore and many-core. The library lets developers focus on expressing the parallelism in their applications without focusing on the low-level details of managing that parallelism. In addition to efficient, high-performing implementations of the most commonly used high-level parallel algorithms and concurrent containers, the library provides efficient low-level building blocks such as a scalable memory allocator, locks, and atomic operations.

While the Intel TBB library is a mature technology, we continue to improve its performance and expand its capabilities. In the Intel TBB 4.0 release, we introduced the flow graph to allow developers to more easily express dependency and data flow graphs. In the Intel TBB 4.2 release, we responded to new architectural features by introducing new synchronization classes based on Intel TSX technology and responded to user requests for better control of concurrency and isolation by introducing user-managed task arenas.
You can find recent Intel TBB versions and information on the commercial and open source sites.


Intel TBB open source site: http://threadingbuildingblocks.org


References


Resource Guide for Intel® Xeon Phi™ Coprocessor Developers

BY TAYLOR KIDD »

This article identifies resources for software developers as they begin working with the Intel® Xeon Phi™ coprocessor, which is based on the Intel® Many Integrated Core (Intel® MIC) architecture. It is one of three such guides designed for people in one of the following roles:

- **Administrator** refers to a person responsible for administration of one or more servers equipped with the Intel Xeon Phi coprocessor (including clusters of such servers).
- **Developer** refers to a person programming for systems equipped with the Intel Xeon Phi coprocessor.
- **Investigator** refers to anyone else who needs to learn more about the Intel Xeon Phi coprocessor, particularly those deciding whether an organization should adopt the technology.

Each guide focuses on the resources most likely to be of primary interest to people in that role. For example, documentation on maintaining clusters is potentially of interest to an administrator, but far less likely to be valuable to a developer. Likewise, programming syntax and semantics are important to a developer, but typically not to an administrator. The content in each guide is tailored accordingly.


© 2014, Intel Corporation. All rights reserved. Intel, the Intel logo, Cilk, VTune, 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.

OpenCL and the OpenCL logo are trademarks of Apple, Inc. used by permission by Kronos.