Learning the Intel Threading Building Blocks Open Source 2.1 Library

An introduction

Arpan Sen

November 23, 2011

Discover a powerful alternative to POSIX and Windows-based threads—Intel Threading Building Blocks, a C++ based framework design specifically for parallel programming.

Parallel programming is the future, but how do you get to high-performance parallel programming that makes effective use of multicore CPUs? Using thread libraries like POSIX threads is certainly an option, but the POSIX framework for threads was originally introduced with the c language in mind. It's also way too low-level an approach—for example, you don't have access to any concurrent containers, nor are there any concurrent algorithms to use. On that note, Intel has introduced Intel® Threading Building Blocks (Intel TBB), a c++-based framework for parallel programming that comes with a host of interesting features and a higher level of abstraction than threads.

**Frequently used acronyms**

- **POSIX**: Portable Operating System Interface for UNIX

Downloading and installing Intel TBB requires nothing special: The extracted directory hierarchy is reminiscent of UNIX® systems with include, bin, lib, and doc folders. For the purposes of this article, I chose the tbb30_20110427oss stable release.

**Getting started with Intel TBB**

Intel TBB has a lot going for it. Here are a few points of interest to get you started:

- Instead of threads, you have a higher level of abstraction in tasks. Intel claims that on Linux® systems, starting and terminating a task is 18 times faster than starting and stopping a thread.
- Intel TBB comes with a task scheduler that can handle load balancing efficiently across multiple logical and physical cores. The default task scheduling policy in Intel TBB is different from the round-robin policies most thread schedulers have.
- Intel TBB offers off-the-shelf availability of thread-safe containers like `concurrent_vector` and `concurrent_queue`.
- Generic parallel algorithms, like `parallel_for` and `parallel_reduce`, are available.
• Lock-free (also known as mutex-free) concurrent programming support is available with the template class `atomic`. This support makes Intel TBB suited for high-performance applications, because Intel TBB handles locking and unlocking the mutex.
• It’s all C++! With no fancy extensions or macros, Intel TBB stays within the language, making heavy use of templates.

Intel TBB does have a fair number of prerequisites. Before getting started, you should have:

• C++ templates and some understanding of the Standard Template Library (STL).
• Knowledge of threads—either POSIX threads or Windows® threads.

Although not necessary, lambda functions in C++0x find fair bit of usage with Intel TBB.

This discussion of Intel TBB begins with creating and playing around with tasks and synchronization primitives (mutex) followed by using the concurrent containers and parallel algorithms. It ends with lock-free programming using the atomic template.

**Hello, World with Intel TBB tasks**

Intel TBB is based on the concept of tasks. You define your own tasks, which are derived from `tbb::task`, declared in `tbb/task.h`. Users are required to override the pure virtual method `task* task::execute( )` in their code. Here are some of the properties of every Intel TBB task:

• When the Intel TBB task scheduler chooses to run some task, the task’s `execute` method is called. That is the entry point.
• The `execute` method may return a `task*`, which tells the scheduler the next task to run. If it returns NULL, then the scheduler is free to choose the next task.
• `task::~task( )` is virtual, and whatever resources the user task has allocated must be released within this destructor.
• Tasks are allocated by a call to `task::allocate_root( )`.
• The main task runs the task to completion with a call to `task::spawn_root_and_wait(task)`.

Listing 1 below shows the first task and how it gets called:

**Listing 1. Creating the first Intel TBB task**

```cpp
#include "tbb/tbb.h"
#include <iostream>
using namespace tbb;
using namespace std;

class first_task : public task {
    public:
    task* execute( ) {
        cout << "Hello World!\n";
        return NULL;
    }
};

int main( ) {
    task_scheduler_init init(task_scheduler_init::automatic);
    first_task& f1 = *new(tbb::task::allocate_root()) first_task( );
    tbb::task::spawn_root_and_wait(f1);
}
```
To run Intel TBB programs, you must have the task scheduler appropriately initialized. The argument to the scheduler in Listing 1 is automatic, which lets the scheduler decide for itself on the number of threads. Of course, you can override this behavior if you want to control the maximum number of threads spawned. But in production code, unless you really know what you're doing, it's best to leave the job of determining the optimum number of threads to the scheduler.

Now that you have created your first task, let's have the first_task from Listing 1 spawn some child tasks. Listing 2 below introduces a few new concepts:

- Intel TBB provides a container called task_list that is meant to serve as a collection of tasks.
- Each parent task creates a child task using the allocate_child function call.
- Before a task spawns any child tasks, it must make a call to set_ref_count. Failure to do so results in undefined behavior. If the intent is to spawn the child tasks, and then wait for them to finish, count must be equal to the number of child tasks + 1; otherwise, count should equal number of child tasks. More on this shortly.
- The call to spawn_and_wait_for_all does what its name suggests: It spawns child tasks and waits until all is done.

Here's the code:

**Listing 2. Creating multiple child tasks**

```cpp
#include "tbb/tbb.h"
#include <iostream>
using namespace tbb;
using namespace std;

class first_task : public task {
public:
    task* execute( ) {
        cout << "Hello World!\n";
        task_list list1;
        list1.push_back( *new( allocate_child() ) first_task( ) );
        list1.push_back( *new( allocate_child() ) first_task( ) );
        set_ref_count(3); // 2 (1 per child task) + 1 (for the wait)
        spawn_and_wait_for_all(list1);
        return NULL;
    }
};

int main( )
{
    first_task& f1 = *new(tbb::task::allocate_root()) first_task( );
    tbb::task::spawn_root_and_wait(f1);
}
```

So, why does Intel TBB require explicit setting of set_ref_count? The documentation says it's primarily for performance reasons. You must always set the ref count for a task before spawning children. See Related topics for links to more detail.

You can also create task groups. The following code creates a task group that spawns two tasks and waits for them to finish. The run method of a task_group has the following signature:

```cpp
template<typename Func> void run( const Func& f )
```
The `run` method spawns a task that computes $f(\cdot)$ but does not block the calling task, so control returns immediately. To wait for the child tasks to finish, the calling task calls `wait` (see Listing 3 below).

### Listing 3. Creating a task_group

```cpp
#include "tbb/tbb.h"
#include <iostream>
using namespace tbb;
using namespace std;

class say_hello( ) {
  const char* message;
  public:
    say_hello(const char* str) : message(str) {  }
    void operator( ) ( ) const {
      cout << message << endl;
    }
};

int main( ) {
  task_group tg;
  tg.run(say_hello("child 1")); // spawn task and return
  tg.run(say_hello("child 2")); // spawn another task and return
  tg.wait( ); // wait for tasks to complete
}
```

Note the syntactic simplicity of `task_group`—no calls required for memory allocation and so on when dealing directly with tasks, and you don't need to do anything with the ref count. That's about it for tasks. Hundreds of things are possible with Intel TBB tasks. Be sure to dive into the Intel TBB documentation for more details. Let's move on to concurrent containers.

### Concurrent containers: vector

Now, let's focus on one of Intel TBB's concurrent containers: the `concurrent_vector`. This container is declared in the header `tbb/concurrent_vector.h`, and the basic interface resembles the STL vector:

```cpp
template<typename T, class A = cache_aligned_allocator<T> >
class concurrent_vector;
```

Multiple threads of control can safely be added to the vector without the need for any explicit locking. To paraphrase from the Intel TBB manual, `concurrent_vector` has the following properties:

- It provides random access to its elements; indexing begins from position 0.
- Safe concurrent increases in size are possible, and multiple threads can be added simultaneously.
- Adding new elements doesn't invalidate existing indices or iterators.

Concurrency comes at a price, though. Unlike STL, where adding new elements involves moving of data, with `concurrent_vector` data isn't moved. Instead, the container maintains a series of contiguous memory segments. Obviously, this increases container overhead.
For concurrent additions to the vector, three methods are available:

- **push_back**—Append an element at the end of the vector.
- **grow_by(N)**—Append $N$ consecutive elements of type $T$ to the `concurrent_vector` and return the iterator to the first appended element. Each element is initialized with $T()$.
- **grow_to_at_least(N)**—Grow a vector to size $N$ if the current size of the vector is less than $N$.

You append a string to a `concurrent_vector` as follows:

```cpp
void append( concurrent_vector<char>& cv, const char* str1 ) {
    size_t count = strlen(str1)+1;
    std::copy( str1, str1+count, cv.grow_by(count) );
}
```

### Using parallel algorithms off the shelf with Intel TBB

One of the best things about Intel TBB is that it lets you parallelize portions of your source code automatically without having to get into the nuts and bolts of how to create and maintain threads. The most common parallel algorithm is `parallel_for`. Consider the following example:

```cpp
void serial_only (int* array, int size) {
    for (int count = 0; count < size; ++count)
        apply_transformation (array [count]);
}
```

Now, if the `apply_transformation` routine in the previous snippet isn't doing anything strange, such as applying some transformation only to individual array elements, then nothing stops you from distributing the load to multiple CPU cores. You need two classes from the Intel TBB library to get started: `blocked_range` (from tbb/blocked_range.h) and `parallel_for` (from tbb/parallel_for.h).

The `blocked_range` class is meant to create an object that provides `parallel_for` with the iteration range, so you need to create something like `blocked_range (0, size)` and pass it as an input to `parallel_for`. The second and final argument that `parallel_for` needs is a class with the requirements in Listing 4 (pasted from the parallel_for.h header).

### Listing 4. Requirements for the second argument to parallel_for

```cpp
/** \page parallel_for_body_req Requirements on parallel_for body
    Class \c Body implementing the concept of parallel_for body must define:
    - \code Body::Body( const Body& ); \endcode Copy constructor
    - \code Body::~Body(); \endcode Destructor
    - \code Body::operator()( Range& r ) const; \endcode Function call operator applying the body to range \c r.
**/
```

This code tells you that you need to create your own class with `operator ()`, with the `blocked_range` as the argument, and code the serial for loop you created earlier inside the method definition for `operator ()`. The copy constructor and destructor should be public, and you leave the compiler to provide the defaults for you. Listing 5 below shows the code.
Listing 5. Creating the second argument to parallel_for

```cpp
#include "tbb/blocked_range.h"
using namespace tbb;

class apply_transform{
    int* array;
    public:
        apply_transform (int* a): array(a) {}
        void operator()( const blocked_range& r ) const {
            for (int i=r.begin(); i!=r.end(); i++ ){
                apply_transformation(array[i]);
            }
        }
};
```

Now that you have successfully created the second object, you just invoke `parallel_for`, as shown below in **Listing 6**.

### Listing 6. Parallelizing the loop using parallel_for

```cpp
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"
using namespace tbb;

void do_parallel_the_tbb_way(int *array, int size) {
    parallel_for (blocked_range(0, size), apply_transform(array));
}
```

### Other parallel algorithms in Intel TBB

Intel TBB offers quite a few parallel algorithms, for example `parallel_reduce` (declared in `tbb/parallel_reduce.h`). Instead of applying a transformation on each individual array element, let’s say you want to sum up all the elements. Here’s the serial code:

```cpp
void serial_only (int* array, int size) {
    int sum = 0;
    for (int count = 0; count < size; ++count)
        sum += array [count];
    return sum;
}
```

Conceptually, running this code in a parallel context would mean that each thread of control should sum up certain portions of the array, and there must be a `join` method somewhere that adds up the partial summations. **Listing 7** below shows the Intel TBB code.

### Listing 7. Serial for loop to sum array elements

```cpp
#include "tbb/blocked_range.h"
#include "tbb/parallel_reduce.h"
using namespace tbb;

float sum_with_parallel_reduce(int*array, int size) {
    summation_helper helper (array);
    parallel_reduce (blocked_range<int> (0, size), 5, helper);
    return helper.sum;
}
```
When splitting the array into sub-arrays for each individual thread, you want to maintain some granularity (for example, each thread is responsible for summing \( N \) elements, where \( N \) is neither too big nor too small). That is the third argument of `blocked_range`. The Intel TBB requires that the `summation_helper` class fulfill two conditions: It must have a method named `join` to add partial sums and a constructor with special arguments (called the `splitting constructor`). Listing 8 provides the code:

**Listing 8. Creating the summation_helper class with the join method and splitting the constructor**

class summation_helper {
    int* partial_array;
public:
    int sum;
    void operator()( const blocked_range<int>& r ) {
        for( int count=r.begin(); count!=r.end(); ++count)
            sum += partial_array [count];
    }
    summation_helper (summation_helper & x, split):
        partial_array (x. partial_array), sum (0)
    {
    }
    summation_helper (int* array): partial_array (array), sum (0)
    {
    }
    void join( const summation_helper & temp ) {
        sum += temp.sum; // required method
    }
};

Here's what will happen. Intel TBB calls the splitting constructor (the second argument called `split` is a dummy argument that Intel TBB requires) and has the partial array filled up by some number of elements (the number is a function of granularity defined in `blocked_range`). When the summation is complete on the sub-array, the `join` method adds the partial result. Slightly complicated? At first glance maybe; just remember that you need three methods: `operator()` to add the array range, `join` to add the partial result, and the split constructor to initiate a new worker thread.

Intel TBB has several other useful algorithms, `parallel_sort` being one of the most useful. Refer to the Intel TBB reference manual (see Related topics) for details.

**Lock-free programming using Intel TBB**

One issue that frequently crops up during multithreaded programming is the number of CPU cycles wasted on the locking and unlocking of mutexes. If you're coming from the POSIX threads background, Intel TBB will surprise you with its `atomic` template. It's a far faster alternative to mutexes, and you could safely do away with the need for locking and unlocking code. Is `atomic` the panacea of all coding woes? No. It's severely restricted in its usage; nonetheless, it's quite effective if you want to create high-performance code. Here's how you declare an integer to be of `atomic` type:
#include "tbb/atomic.h"
using namespace tbb;

atomic<int> count;
atomic<float*> pointer_to_float;

Now, assume that the variable count from earlier is being accessed by multiple threads of control. Typically, you would want to guard count with a mutex during writes; but, that's no longer required with atomic<int>. Take a look at Listing 9.

**Listing 9. atomic fetch_and_add requires no locking**

```c
// writing with mutex, count is declared as int count;
{  
    // _ code  
    pthread_mutex_lock (&lock);
    count += 1000;
    pthread_mutex_unlock (&lock);
    // _ code continues  
}

// writing without mutex, count declared as atomic<int> count;
{  
    // _ code  
    count.fetch_and_add (1000); // no explicit locking/unlocking  
    // _ code continues  
}
```

Instead of +=, you use the fetch_and_add method of the atomic<T> class. And no, it does not use any mutexes internally as part of the fetch_and_add method. When fetch_and_add is executed, it has the effect of adding 1000 to count instantaneously—either all threads see the updated value of count at once, or all threads continue to see the old value. That's why count is declared as an atomic variable: Operations on count are atomic and cannot be interrupted by the vagaries of process or thread scheduling. No matter how threads are scheduled, there's no way count would have different values in different threads. For an in-depth discussion of lock-free programming, see Related topics.

The atomic<T> class comes with the following five fundamental operations:

```c
  y = x; // atomic read  
  x = b; // atomic write  
  x.fetch_and_store(y); // y = x and return the old value of x  
  x.fetch_and_add(y); // x += y and return the old value of x  
  x.compare_and_swap(y, z); // if (x == z) x = y; in either case, return old value of x
```

In addition, the operators +=, -=, ++, and -- are supported for convenience, but they are all implemented on top of fetch_and_add. As shown in tbb/atomic.h, here's how the operators are defined (Listing 10).

**Listing 10. Operators ++, --, +=, and -= defined using fetch_and_add**

```c
value_type operator+( D addend )  
    {  
        return fetch_and_add(addend)+addend;
    }
```
value_type operator-=(D addend) {
    // Additive inverse of addend computed using binary minus,
    // instead of unary minus, for sake of avoiding compiler warnings.
    return operator+=(D(0)-addend);
}

value_type operator++() {
    return fetch_and_add(1)+1;
}

value_type operator--() {
    return fetch_and_add(__TBB_MINUS_ONE(D))-1;
}

value_type operator++(int) {
    return fetch_and_add(1);
}

value_type operator--(int) {
    return fetch_and_add(__TBB_MINUS_ONE(D));
}

Note that the type T in atomic<T> can only be an integral type, enumeration type, or pointer type.

Conclusion

It is impossible to do justice to a library the scale of Intel TBB in a single article. Indeed, Intel's
web site has dozens of articles highlighting several aspects of Intel TBB. Instead, this article
attempted to provide insight into some of the compelling features that Intel TBB comes with—
tasks, concurrent containers, algorithms, and a way to create lock-free code. Hopefully, this
introduction ignites your interest and Intel TBB will gain yet another ardent user—much like the
author himself.
Related topics

- Learn more about Intel TBB atomic operations.
- Find details on lock-free programming.
- Intel TBB blogs provide good information on concurrent vector internals.
- Try out IBM software for free. Download a trial version, log into an online trial, work with a product in a sandbox environment, or access it through the cloud. Choose from over 100 IBM product trials.