Intel Thread Building Blocks

Expressing Parallelism Easily
Why are we here?

- Hardware is increasingly parallel and software must change to take advantage
- Coding with low level parallelism (e.g. pthreads) is time-consuming and not scalable
- Developers need to a way to express parallelism without adding undue complexity
More Parallelism Required
What is TBB?

Threading Building Blocks (TBB) is a C++ template library for writing software that take advantage of multi-core processors.

The library abstracts access to the multiple processors by allowing the operations to be treated as "tasks", which are allocated to individual cores dynamically by the library's run-time engine, and by automating efficient use of the CPU cache.
Specify *logical parallelism* instead of threads

Most threading packages require you to specify threads. Programming directly in terms of threads can be tedious and lead to inefficient programs, because threads are low-level, heavy constructs that are close to the hardware. Direct programming with threads forces you to efficiently map logical tasks onto threads. In contrast, the Intel® Threading Building Blocks run-time library automatically maps logical parallelism onto threads in a way that makes efficient use of processor resources.
Advantages of TBB - 2

Targets *threading for performance*.

Most general-purpose threading packages support many different kinds of threading, such as threading for asynchronous events in graphical user interfaces. As a result, general-purpose packages tend to be low-level tools that provide a foundation, not a solution. Instead, Intel® Threading Building Blocks focuses on the particular goal of parallelizing computationally intensive work, delivering higher-level, simpler solutions.
Advantages of TBB - 3

*Compatible with other threading packages.*

Because the library is not designed to address all threading problems, it can coexist seamlessly with other threading packages.
Advantages of TBB - 4

**Emphasizes scalable, data parallel programming.**

Breaking a program up into separate functional blocks, and assigning a separate thread to each block is a solution that typically does not scale well since typically the number of functional blocks is fixed. In contrast, Intel® Threading Building Blocks emphasizes *data-parallel* programming, enabling multiple threads to work on different parts of a collection. Data-parallel programming scales well to larger numbers of processors by dividing the collection into smaller pieces. With data-parallel programming, program performance increases as you add processors.
Relies on *generic programming*.

Traditional libraries specify interfaces in terms of specific types or base classes. Instead, Intel® Threading Building Blocks uses generic programming. The essence of generic programming is writing the best possible algorithms with the fewest constraints. The C++ Standard Template Library (STL) is a good example of generic programming in which the interfaces are specified by *requirements* on types. For example, C++ STL has a template function `sort` that sorts a sequence abstractly defined in terms of iterators on the sequence.

Specification in terms of requirements on types enables the template to sort many different representations of sequences, such as vectors and deques. Similarly, the Intel® Threading Building Blocks templates specify requirements on types, not particular types, and thus adapt to different data representations. Generic programming enables Intel® Threading Building Blocks to deliver high performance algorithms with broad applicability.
Advantages of TBB - 6

*Begin decomposing your problems in parallel manner.*

Software paradigms and programming models are rapidly changing to deal with the increased parallelism required by hardware. TBB may or may not be used specifically in 10 years. If your program is already designed to express parallelism, porting to the next parallel model is much easier.
TBB Overview

- **Generic Parallel Algorithms**
  Efficient scalable way to exploit the power of multi-core without having to start from scratch

- ** Concurrent Containers**
  Concurrent access, and a scalable alternative to containers that are externally locked for thread-safety

- **Flow Graph**
  A set of classes to express parallelism via a dependency graph or a data flow graph

- **Task Scheduler**
  Sophisticated engine with a variety of work scheduling techniques that empowers parallel algorithms & the flow graph

- **Thread Local Storage**
  Supports infinite number of thread local data

- **Synchronization Primitives**
  Atomic operations, several flavors of mutexes, condition variables

- **Memory Allocation**
  Scalable memory manager and false-sharing free allocators

- **Threads**
  OS API wrappers
Generic Parallel Algorithms

TBB provides the following:

- parallel_for
- parallel_reduce, parallel_scan
- parallel_do
- parallel_pipeline
- parallel_invoke, task_group
// Setup a TBB parallel for body

class ArraySummer {

    int *p_array_a, *p_array_b, *p_array_sum;

public:
    ArraySummer(int * p_a, int * p_b, int * p_sum) :
        p_array_a(p_a), p_array_b(p_b), p_array_sum(p_sum) { }

    // operator function contains the parallel computation
    void operator() ( const blocked_range<int>& r ) const {
        for ( int i = r.begin(); i != r.end(); i++ ) {
            p_array_sum[i] = p_array_a[i] + p_array_b[i];
        }
    }
};
Parallel For Loop Example - 2

p_A = new int[nElements];
p_B = new int[nElements];
p_SUM_TBB = new int[nElements];

parallel_for(blocked_range<int>(0, nElements, 100),
             ArraySummer( p_A, p_B, p_SUM_TBB ));
Alternative

#1 Handcode and optimize number of threads
#2 Use openMP
Concurrent Containers

A concurrent container allows multiple threads to concurrently access and update items in the container.

Containers offer a much higher level of concurrency, via one or both of the following methods:

- Fine-grained locking: Multiple threads operate on the container by locking only those portions they really need to lock. As long as different threads access different portions, they can proceed concurrently.
- Lock-free techniques: Different threads account and correct for the effects of other interfering threads.

TBB provides hash map, vector, and queue.
Concurrent Queue Example

```cpp
concurrent_queue<int> queue;
for( int i=0; i<10; ++i )
    queue.push(i);

int popVal;
bool popped;
do {
    popped = s_queue.try_pop(popVal);
    cout << popVal << " ";
} while (popped);

See RunQueue.cpp
```
Alternative

#1 Wrap entire class in mutex = bottleneck
#2 Create your own fine-grained locks = error-prone and tedious
#include <tbb/scalable_allocator.h>
scalable_malloc( size );

Change

vector<int> to
vector<int, scalable_allocator<int> >
TBB provides two memory allocator templates that are similar to the STL template class `std::allocator`. These two templates, `scalable_allocator<T>` and `cache_aligned_allocator<T>`, address critical issues in parallel programming: *scalability* and *false sharing*.

**Scalability** - When using memory allocators originally designed for serial programs, threads must compete for access to a single shared pool in a way that allows only one thread to allocate at a time.

**False Sharing** - Problems arise when two threads access different words that share the same cache line. The problem is that a cache line is the unit of information interchange between processor caches. If one processor modifies a cache line and another processor reads (or writes) the same cache line, the cache line must be moved from one processor to the other, even if the two processors are dealing with different words within the line. False sharing can hurt performance because cache lines can take hundreds of clocks to move. Use the class `cache_aligned_allocator<T>` to always allocate on a cache line. Two objects allocated by `cache_aligned_allocator` are guaranteed to not have false sharing.
Alternative

#1 Handcode a thread specific allocate/deallocate system = time consuming
#2 Preallocate all memory = time consuming
# Synchronization Primitives

TBB provides synchronization primitives for *mutual exclusion* and to perform *atomic operations*. These are very similar to Boost interface.

An `atomic<T>` supports atomic read, write, fetch-and-add, fetch-and-store, and compare-and-swap. Type $T$ may be an integral type, enumeration type, or a pointer type. When $T$ is a pointer type, arithmetic operations are interpreted as pointer arithmetic.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>read the value of $x$</td>
</tr>
<tr>
<td>$x =$</td>
<td>write the value of $x$, and return it</td>
</tr>
<tr>
<td>$x$.fetch_and_store($y$)</td>
<td>do $y=x$ and return the old value of $x$</td>
</tr>
<tr>
<td>$x$.fetch_and_add($y$)</td>
<td>do $x+=y$ and return the old value of $x$</td>
</tr>
<tr>
<td>$x$.compare_and_swap($y, z$)</td>
<td>if $x$ equals $z$, then do $x=y$. In either case, return old value of $x$</td>
</tr>
</tbody>
</table>
Node* FreeList;
typedef spin_mutex FreeListMutexType;
FreeListMutexType FreeListMutex;

Node* AllocateNode() {
    Node* n;
    {
        FreeListMutexType::scoped_lock lock(FreeListMutex);
        n = FreeList;
        if( n )
            FreeList = n->next;
    }
    if( !n )
        n = new Node();
    return n;
}

void FreeNode( Node* n ) {
    FreeListMutexType::scoped_lock lock(FreeListMutex);
    n->next = FreeList;
    FreeList = n;
}
**Alternative**

#1 Assume atomic integers ( IF you're reading/writing 4-byte value AND it is DWORD-aligned in memory AND you're running on the I32 architecture, THEN reads and writes are atomic). = not portable

#2 Use Boost or C++ 11
TBB on ARC systems

TBB is installed in the module system. Load the Intel compiler and tbb module.

[mimarsh2@brlogin1 ~]$ module load intel mkl tbb
[mimarsh2@brlogin1 ~]$ cat test.cpp
#include "tbb/task_scheduler_init.h"

int main() {
    tbb::task_scheduler_init init();
    return 0;
}

[mimarsh2@brlogin1 ~]$ icpc -O2 test.cpp -ltbb -o test
TBB on MIC

You must be on a MIC node to access the MIC environment. Load the tbb-mic module.

[mimarsh2@br024 ~]$ module load intel mkl tbb-mic mic
[mimarsh2@br024 ~]$ module list
Currently Loaded Modules:
   1) intel/13.1  2) mkl/11  3) tbb-mic/4.1  4) mic/1.0
[mimarsh2@br024 ~]$ icpc -O3 test.cpp -tbb -lpthread -mmic -o test
[mimarsh2@br024 ~]$ ssh mic0
Warning: Permanently added 'mic0,10.11.110.24' (RSA) to the list of known hosts.
~ $ export LD_LIBRARY_PATH=/opt/apps/intel/13.1/tbb/lib/mic:$LD_LIBRARY_PATH
~ $ ./test
~ $
How much Parallelism on a MIC?

A good rule of thumb is that an Intel® TBB application should have approximately 10 tasks for every worker. This allows the work-stealing scheduler to redistribute work if one of the tasks is unexpectedly large. By default, a typical Xeon Phi coprocessor has 60 cores, and 4 hyperthreads/core:

\[ 60 \text{ cores} \times 4 \text{ hyperthreads/core} \times 10 \text{ tasks/thread} = 2400 \text{ tasks}. \]

Now factor in the parallelism provided by the vector units that can process 16 single precision floating point numbers simultaneously, and your loops should have a range of at least

\[ 2400 \text{ tasks} \times 16 \text{ lanes/instruction} = 38,400. \]

That’s a lot of parallelism.
Questions???
References

- https://www.threadingbuildingblocks.org/intel-tbb-tutorial