AN INTRODUCTION TO THREADING IN C++ WITH THREADING BUILDING BLOCKS (TBB)

Mike Voss, Principal Engineer
Core and Visual Computing Group, Intel
We’ve already talked about threading with pthreads and the OpenMP* API

- POSIX threads (pthreads) lets us express threading but makes us do a lot of the hard work
- OpenMP is higher-level model and is widely used in C/C++ and Fortran
  - It takes care of many of the low-level error prone details for us
- OpenMP has weaknesses, especially for C++ developers...
  - It uses #pragmas and so doesn’t look like C++
  - It is not a *composable* parallelism model
Agenda

• What is composability and why is it important?
• An introduction to the Threading Building Blocks (TBB) library
  • What it is and what it contains
• TBB’s high-level execution interfaces
  • The generic parallel algorithms, the flow graph and Parallel STL
• Synchronization primitives and concurrent containers
• The TBB scalable memory allocator
There are different ways parallel software components can be combined with other parallel software components.
Nested composition

```c
int main() {
    #pragma omp parallel
    f();
}

void f() {
    #pragma omp parallel
    g();
}

void g() {
    #pragma omp parallel
    h();
}
```
Nested composition

Nested parallelism can lead to an exponential growth in the available parallelism, great! Or the number of threads, very bad!
Concurrent Composition

```c
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
    b[i] = f(a[i]);
}

#pragma omp parallel for
for (int i = 0; i < M; ++i) {
    d[i] = g(c[i]);
}
```

![Diagram showing parallel and concurrent execution]
Serial Composition

```c
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
    b[i] = f( a[i] );
}
```

```c
#pragma some_other_kind_of_parallel_for
for (int i = 0; i < N; ++i) {
    c[i] = f( b[i] );
}
```
Serial Composition

(a) ideal transition, same model

(b) transition with shutdown and startup, same model

(c) ideal transition, different models

(d) transition with shutdown and startup, different models
A composable threading model

- Executes efficiently when its constructs are composed with other constructs from the same threading model
  - nested, concurrent and serial
- Doesn’t negatively impact other threading models too much when composed with constructs in the other threading model
  - nested, concurrent and serial
  - it can’t control the other model, but it can be a “good citizen”
Agenda

• What is composability and why is it important?
• An introduction to the Threading Building Blocks (TBB) library
  • What it is and what it contains
  • TBB’s high-level execution interfaces
    • The generic parallel algorithms, the flow graph and Parallel STL
    • Synchronization primitives and concurrent containers
  • The TBB scalable memory allocator
Threaded Building Blocks (TBB)
Celebrated its 10 year anniversary in 2016

A widely used C++ template library for shared-memory parallel programming

**What**
Parallel algorithms and data structures
Threads and synchronization primitives
Scalable memory allocation and task scheduling

**Benefits**
Is a library-only solution that does not depend on special compiler support
Is both a commercial product and an open-source project
Supports C++, Windows*, Linux*, OS X*, Android* and other OSes
Commercial support for Intel® Atom™, Core™, Xeon® processors and for Intel® Xeon Phi™ coprocessors

http://threadingbuildingblocks.org
http://software.intel.com/intel-tbb
The Components in Threading Building Blocks (TBB)

TBB Parallel Execution Interfaces
- Generic Parallel Patterns
- Flow Graph
- Parallel STL (C++17 / C++2x Standardized Interfaces)

TBB Interfaces Independent of Execution Model
- Concurrent Containers
  - Hash Tables
  - Queues
  - Vectors
- Memory Allocation
  - Scalable Allocator
  - Cache Aligned Allocator
- Primitives and Utilities
  - Synchronization Primitives
  - Thread Local Storage
High-level execution interfaces map to parallelism in applications

Task Parallelism / Message Passing

fork-join

SIMD

fork-join

SIMD

Intel TBB helps to develop composable levels
High-level execution interfaces map to parallelism in applications

Task Parallelism / Message Passing

fork-join

fork-join

SIMD SIMD SIMD SIMD SIMD SIMD SIMD

Intel TBB helps to develop composable levels
High-level execution interfaces map to parallelism in applications

Task Parallelism / Message Passing

Generic Algorithms
Parallel STL (par policy)

Intel TBB helps to develop composable levels
High-level execution interfaces map to parallelism in applications

Task Parallelism / Message Passing

fork-join
fork-join

Parallel STL (unseq policy)

SIMD SIMD SIMD SIMD SIMD SIMD SIMD

Intel TBB helps to develop composable levels
Agenda

• What is composability and why is it important?
• An introduction to the Threading Building Blocks (TBB) library
  • What it is and what it contains
• TBB’s high-level execution interfaces
  • The generic parallel algorithms, the flow graph and Parallel STL
  • Synchronization primitives and concurrent containers
• The TBB scalable memory allocator
But before we do that... a quick overview of C++ lambda expressions

- Lambda expressions are anonymous function objects
  
  \[
  \text{[ capture-list ] ( params ) -> ret \{ body \}}
  \]

- capture-list
  - a list of variables to capture from the enclosing scope
  - e.g. \([x,y]\) or to capture a reference then \([&x,y]\)

- params
  - The parameters of the function, just like for a named function

- ret is the return type

- body is the function body
Mandelbrot Example
Threading Building Blocks (TBB)

```cpp
int mandel(Complex c, int max_count) {
    int count = 0; Complex z = 0;
    for (int i = 0; i < max_count; i++) {
        if (abs(z) >= 2.0) break;
        z = z*z + c; count++;
    }
    return count;
}
```

Parallel algorithm

```cpp
parallel_for( 0, max_row,
[&](int i) {
    for (int j = 0; j < max_col; j++)
        p[i][j]=mandel(Complex(scale(i),scale(j)),depth);
};
```

Use C++ lambda functions to define function object in-line
TBB Generic Parallel Algorithms

**Loop parallelization**
- parallel_for
- parallel_reduce
- parallel_scan

**Parallel sorting**
- parallel_sort

**Parallel function invocation**
- parallel_invoke

**Streaming**
- parallel_do
- parallel_for_each
- pipeline / parallel_pipeline

The most common patterns used in parallel programming
TBB is a composable library because it uses tasks, a thread pool and a work-stealing task scheduler.
TBB is a composable library because it uses tasks, a thread pool and a work-stealing task scheduler.

Simplified work-stealing task dispatcher used by each worker thread:

- Pop from head (newest task)
- Steal from tail (oldest task)
A very nice distribution of a loop across 4 threads uses recursive splitting

\[
tbb::parallel_for(0, N, 1, \lambda (\text{int } i) \{ \\
\quad f(a[i]) ; \\
\}) ;
\]
A very nice distribution of a loop across 4 threads uses recursive splitting

```
tbb::parallel_for(0, N, 1, [a](int i) {
    f(a[i]);
});
```

But while thread 1 is doing this, along comes another thread that wants to help out...
A very nice distribution of a loop across 4 threads uses recursive splitting

```
tbb::parallel_for(0, N, 1, [a](int i) {
  f(a[i]);
});
```
A very nice distribution of a loop across 4 threads uses recursive splitting

```cpp
boost::parallel_for(0, N, 1, [a](int i) {
    f(a[i]);
});
```
A very nice distribution of a loop across 4 threads uses recursive splitting

\[
\text{tbb::parallel_for}(0, N, 1, [a](\text{int } i) \{ \\
\quad f(a[i]); \\
\}) ;
\]
(a) tasks as distributed by work-stealing across two threads

(b) the Task Dispatcher actions that acquire the tasks
TBB is a composable library because it uses tasks, a thread pool and a work-stealing task scheduler

- Nested parallelism just works
  - We create lots of small tasks but they execute on a limited number of threads in the thread pools – no explosion of threads

- Concurrent composition just works
  - Tasks are scheduled to the same threads – no problem

- Serial composition just works
  - The thread pool stays alive and as work becomes available, idle worker threads steal it
Agenda

- What is composability and why is it important?
- An introduction to the Threading Building Blocks (TBB) library
  - What it is and what it contains
- TBB's high-level execution interfaces
  - The generic parallel algorithms, the flow graph and Parallel STL
- Synchronization primitives and concurrent containers
- The TBB scalable memory allocator
Graph-based parallelism

```java
while ( img = get_image() ) {
    x = f1(img);
    y = f2(x);
    z = f3(x);
    f4(y, z);
}
```

Graphs express the operations and their input and output dependencies:

Given the operations and their input and output dependencies, a runtime scheduler:

Can exploit functional parallelism:

Can exploit pipeline parallelism:

Can exploit data parallelism:
Threading Building Blocks flow graph

Efficient implementation of dependency graph and data flow algorithms

Enables developers to exploit parallelism at higher levels

Nodes execute as TBB tasks

```cpp
graph g;
continue_node< continue_msg > h( g,
    []( const continue_msg & ) {
        cout << "Hello ";
    } );
continue_node< continue_msg > w( g,
    []( const continue_msg & ) {
        cout << "World\n";
    } );
make_edge( h, w );
h.try_put(continue_msg());
g.wait_for_all();
```
An example feature detection algorithm

Can express **pipelining**, **task parallelism** and **data parallelism**
How flow graph nodes map to TBB tasks

```cpp
graph g;
function_node< int, int > n( g, unlimited, []( int v ) -> int {
    cout << v;
    spin_for( v );
    cout << v;
    return v;
});
function_node< int, int > m( g, serial, []( int v ) -> int {
    v *= v;
    cout << v;
    spin_for( v );
    cout << v;
    return v;
});
make_edge( n, m );
n.try_put( 1 );
n.try_put( 2 );
n.try_put( 3 );
g.wait_for_all();
```

One possible execution – stealing is random
Agenda

• What is composability and why is it important?
• An introduction to the Threading Building Blocks (TBB) library
  • What it is and what it contains
• TBB’s high-level execution interfaces
  • The generic parallel algorithms, the flow graph and Parallel STL
  • Synchronization primitives and concurrent containers
• The TBB scalable memory allocator
The C++ Standard Template Library

std::vector<float> float* transform

#include <algorithm>

void increment( float *in, float *out, int N ) {
    using namespace std;
    transform( in, in + N, out, []( float f ) {
        return f+1;
    });
}
Enter Parallel STL

- Extension of C++ Standard Template Library algorithms with the “execution policy” argument
- Support for parallel execution policies is approved for C++17
- Support for vectorization policies is being developed in Parallelism Technical Specification (TS) v2
The different execution policies for Parallel STL
Parallel STL Examples

// standard sequential sort
sort(v.begin(), v.end());

// explicitly sequential sort
sort(execution::seq, v.begin(), v.end());

// permitting parallel execution
sort(execution::par, v.begin(), v.end());

// permitting vectorization as well
sort(execution::par_unseq, v.begin(), v.end());

// Parallelism TS v2
// permitting vectorization only (no parallel execution)
sort(execution::unseq, v.begin(), v.end());
Parallel STL Examples

// standard sequential sort
sort(v.begin(), v.end());

// explicitly sequential sort
sort(execution::seq,v.begin(), v.end());

// permitting parallel execution
sort(execution::par, v.begin(), v.end());

// permitting vectorization as well
sort(execution::par_unseq, v.begin(), v.end());

// Parallelism TS v2
// permitting vectorization only (no parallel execution)
sort(execution::unseq, v.begin(), v.end());
Parallel STL includes many algorithms

- These are more specialized than the TBB generic algorithms
  - Like fill, find_if, etc...
- But contains some powerful functions
  - for_each, transform, reduce, transform_reduce, etc...
- Even so, they are less expressive than TBB since they work on sequences or containers
- But they have standardized C++ interfaces
Agenda

• What is composability and why is it important?
• An introduction to the Threading Building Blocks (TBB) library
  • What it is and what it contains
• TBB’s high-level execution interfaces
  • The generic parallel algorithms, the flow graph and Parallel STL
• Synchronization primitives and concurrent containers
• The TBB scalable memory allocator
TBB includes C++ versions of many of the synchronization primitives we’ve learned about

• atomic variables

• atomic<int> i;

• supports compare_and_swap, fetch_and_add, etc...

• Mutexes & locks

• spin_mutex, queuing_mutex, speculative_spin_mutex, etc...
But it also provides high-level thread friendly data structures

• maps, sets, queues and vectors

```cpp
extern std::queue q;
if (!q.empty()) {
    item = q.front();
    q.pop();
}
```

At this instant, another thread might pop the last element.

TBB provides a try_pop function instead.
But it also provides high-level thread friendly data structures

- maps, sets, queues and vectors

```c
extern concurrent_queue<T> MyQueue;
T item;
if( MyQueue.try_pop(item) ) {
    ...process item...
}
```

TBB provides a try_pop function instead.
Agenda

- What is composability and why is it important?
- An introduction to the Threading Building Blocks (TBB) library
  - What it is and what it contains
- TBB’s high-level execution interfaces
  - The generic parallel algorithms, the flow graph and Parallel STL
- Synchronization primitives and concurrent containers
- The TBB scalable memory allocator
TBB provides useful memory allocators

- `cache_aligned_allocator`
  - Helps to prevent false sharing by doing the right padding
- `scalable_allocator`
  - Some OSes use a single global heap for memory allocator, that is protected by a lock
  - If many threads starting allocating in parallel there is contention on the lock
  - The TBB scalable memory allocator uses per-thread heaps to avoid locking
To Learn More:
See Intel’s The Parallel Universe Magazine

http://threadingbuildingblocks.org

http://software.intel.com/intel-tbb
Legal Disclaimer & Optimization Notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Copyright © 2018, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
BACKUP