Programming for Shared Memory Parallel

- Pthreads
  - POSIX standard for Unix systems programming

- OpenMP
  - Language extensions (C, C++, Fortran)
  - Loop-level parallelism
  - Tasking
  - Requires OpenMP compiler

- Intel Threading Building Blocks (TBB)
  - Higher-level task-based programming
  - Parallel constructs
  - Generic class library
**Key Features of TBB**

- You can specify *tasks* instead of threads
  - TBB maps logical tasks onto threads
  - Full support for *nested parallelism*

- Targets threading for *scalable performance*

- Uses proven efficient parallel patterns

- Uses work-stealing to support the load balance of unknown execution time for tasks

- Open source and licensed versions available on:
  - Linux, Windows, and Mac OS X*
Advantages of TBB

- Do not have to be a threading expert
  - TBB works at a higher level than raw threads
  - Does not require language extensions or compilers

- TBB differs from other in the following ways:
  - Enables you to specify logical parallelism instead of threads
  -Targets threading for performance
  - Compatible with other threading packages
  - Emphasizes scalable, data parallel programming
  - Relies on generic programming (e.g., use of C++ STL)
Task-based Programming with Intel® TBB

- Tasks are light-weight entities at user-level
  - Much lighter than threads
  - Starting and terminating tasks is 18x faster
  - Threads have a system-level ID and own resources
  - Tasks are small routine with local data

- TBB parallel algorithms maps tasks onto threads
  - Done automatically
  - Task scheduler manages the thread pool
  - Work sharing / Work stealing
  - Scheduler is unfair
    - favors tasks that have been most recent in the cache

- Task-stealing scheduler
  - Allows management of core resources
  - Prevents oversubscription and undersubscription
## Overview

<table>
<thead>
<tr>
<th>Generic Parallel Algorithms</th>
<th>Concurrent Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficient scalable way to exploit the power of multi-core without having to start from scratch</td>
<td>Concurrent access, and a scalable alternative to containers that are externally locked for thread-safety</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flow Graph</th>
<th>Thread Local Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>A set of classes to express parallelism via a dependency graph or a data flow graph</td>
<td>Supports infinite number of thread local data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Scheduler</th>
<th>Synchronization Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sophisticated engine with a variety of work scheduling techniques that empowers parallel algorithms &amp; the flow graph</td>
<td>Atomic operations, several flavors of mutexes, condition variables</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Allocation</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalable memory manager and false-sharing free allocators</td>
<td>OS API wrappers</td>
</tr>
</tbody>
</table>
Intel® TBB Algorithms

- Task scheduler powers high level parallel patterns
  - Pre-packaged, tested, and tuned for scalability

**parallel_for, parallel_for_each**
- Load-balanced parallel execution of independent loop iterations

**parallel_reduce**
- Load-balanced parallel execution of independent loop iterations for reduction

**parallel_scan**
- Load-balanced computation of parallel prefix

**parallel_pipeline**
- Linear pipeline pattern using serial and parallel filters

**parallel_do**
- Load-balanced parallel execution of independent loop iterations
- Ability to add more work

**parallel_sort**
**parallel_invoke**
- Parallel execution of function objects or pointers to functions
**parallel_for Algorithm**

```cpp
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>

template<typename Range, typename Func>
Func parallel_for(const Range& range, const Func& f, [, task_group_context& group])
```

```cpp
#include "tbb/parallel_for.h"

template<typename Index, typename Func>
Func parallel_for(Index first, Index_type last [, Index step], const Func& f [, task_group_context& group]);
```

- `parallel_for` partitions original range into subranges, and deals out subranges to worker threads in a way that:
  - Balances load, uses cache efficiently, and scales
Range is Generic

- Library provides predefined ranges:
  - `blocked_range`, `blocked_range2d`, `blocked_range3d`

Diagram:
- `blocked_range2d`
- Split range...
- ... recursively...
- ... until `grainsize`.
- Tasks available to be scheduled to other threads (thieves)
Grain Size

- Part of `blocked_range<>`
  - Used by `parallel_for` and `parallel_reduce`
  - Not by underlying task scheduler

- Grain size exists to amortize overhead
  - Not balance load
  - Units of granularity are loop iterations

- Typically, only need to get it right within an order of magnitude

- OpenMP has similar parameter
An Example using parallel_for

- Independent iterations and fixed/known bounds
- Serial code

```cpp
const int N = 100000;

void change_array(float array, int M) {
  for (int i = 0; i < M; i++) {
    array[i] *= 2;
  }
}

int main() {
  float A[N];
  initialize_array(A);
  change_array(A, N);
  return 0;
}
```
An Example using `parallel_for` (2)

- **Using the `parallel_for` pattern**
  ```cpp
  #include <tbb/blocked_range.h>
  #include <tbb/parallel_for.h>
  using namespace tbb;
  
  void parallel_change_array(float *array, size_t M {
      parallel_for(blocked_range<size_t>(0, M, IdealGrainSize),
          [=](const blocked_range<size_t>& r) -> void {
              for(size_t i = r.begin(); i != r.end(); i++)
                  array[i] *= 2;
          });
  }
  ```
Task Scheduler

- A task scheduler is automatically created when TBB threads are required, and destructed after.

- You might want to control that in order to avoid overhead caused by numerous creations/destructions.

```cpp
#include <tbb/task_scheduler_init.h>
using namespace tbb;

int main (){
    task_scheduler_init init; //threads creation (with constructor)
    float A[N];
    initialize_array(A);
    parallel_change_array(A, N);
    return 0;
} //out of scope -> threads destruction
```

- You can set the maximum number of threads by passing it in argument to the constructor.
Generic Programming vs Lambda Functions

Generic programming:

```cpp
class ChangeArrayBody {
public:
    ChangeArrayBody(float *a): array(a) {} 
    void operator()( const blocked_range<size_t>& r ) const{
        for (size_t i = r.begin(); i != r.end(); i++){
            array[i] *= 2;
        }
    }
};

void parallel_change_array(float *array, size_t M {
    parallel_for(blocked_range<size_t>(0, M, IdealGrainSize
    =[](const blocked_range<size_t>& r) -> void {
        for(size_t i = r.begin(); i != r.end(); i++)
            array[i] *= 2;
    });
}
```

Lambda functions:

```cpp
void parallel_change_array(float *array, size_t M {
    parallel_for(blocked_range<size_t>(0, M, IdealGrainSize
    =[](const blocked_range<size_t>& r) -> void {
        for(size_t i = r.begin(); i != r.end(); i++)
            array[i] *= 2;
    });
}
```

blue = original code  
green = provided by TBB  
red = boilerplate for library
You can achieve the same performance with both, but some patterns might require generic programming.

Use Lambda way when it can be used.

Lambda functions are part of the C++11 standard.
Concurrent Containers

- TBB library provides highly concurrent containers
  - STL containers are not concurrency-friendly
  - Attempt to modify them concurrently can corrupt container
  - Standard practice is to wrap a lock around STL containers
    - turns container into serial bottleneck

- TBB library provides fine-grained locking or lockless implementations
  - Worse single-thread performance, but better scalability
  - Do not need the task scheduler
    - can be used with the library, OpenMP, or native threads
Concurrent Containers (2)

- concurrent_queue
- concurrent_vector
- concurrent_hash_map
- concurrent_unordered_map
How many threads are available?

- Do not ask!
  - Not even the scheduler knows how many threads really are available
  - There may be other processes running on the machine
  - Routine may be nested inside other parallel routines

- Focus on dividing your program into tasks of sufficient size
  - Task should be big enough to amortize scheduler overhead
  - Choose decompositions with good depth-first cache locality and potential breadth-first parallelism

- Let the scheduler do the mapping