Parallel Programming course. TBB

Obolenskiy Arseniy, Nesterov Alexander

Nizhny Novgorod State University

April 2, 2025

Contents

OpenMP recap

- OpenMP is an open standard for shared-memory parallel programming
- It uses compiler directives (`#pragma omp`), runtime functions, and environment variables
- Simplifies parallel loops and regions for multi-threaded execution
- Serves as a baseline for comparing other parallel models

OpenMP vs TBB

- OpenMP:
  - Open standard (not a library) that defines a set of compiler directives, runtime library routines, and environment variables for parallel programming
  - Directive-based, integrated into the compiler (specific implementation is don on the compiler side)
  - Implements thread-level parallelism
- TBB (Threading Building Blocks):
  - C++ template library
  - Uses task-based parallelism with work-stealing scheduling
  - Provides higher-level constructs and generic parallel patterns
- TBB enables more flexible and fine-grained parallelism compared to OpenMP

Pros and Cons of TBB

Pros

- Task-based parallelism with dynamic work-stealing
- High-level constructs simplify parallel code
- Seamless integration with modern C++ and STL
- Scalable and efficient for fine-grained tasks
- Portable across different platforms

Cons

- Steeper learning curve compared to directive-based models
- Less intuitive for simple loop parallelism
- Threading issues debugging sometimes can be challenging due to dynamic scheduling

TBB history

- Originally developed by Intel to simplify parallel programming in C++
- Evolved into an open-source library and later integrated into Intel oneAPI
- Widely adopted in industry and academia for scalable task-based parallelism

TBB history timeline

- Early 2000s: Conceptual groundwork for task-based parallelism laid at Intel
- 2006: Initial development of TBB begins for internal projects
- 2007: First public release of Intel Threading Building Blocks
- 2010: Major updates introduce improved C++ integration and performance enhancements
- 2017: TBB is open-sourced, fostering community contributions
- 2019: Integration into Intel oneAPI, expanding its cross-platform reach

TBB fundamentals

- A C++ library for task-based parallelism
- Abstracts low-level thread management and uses a work-stealing scheduler
- Provides high-level constructs (e.g., `tbb::parallel_for`, `tbb::parallel_reduce`) that simplify parallel code implementation
- Promotes writing scalable code by focusing on tasks rather than threads

Work-stealing dynamic scheduler

A work-stealing dynamic scheduler in TBB is a type of scheduling algorithm designed to efficiently balance the workload across multiple threads in a parallel program
The idea: each thread maintains its own queue of tasks. When a thread finishes its own work and becomes idle, instead of waiting, it tries to "steal" tasks from other threads queues to stay productive. This helps to dynamically balance the workload and utilize CPU resources effectively

How TBB Implements Work Stealing

- Local Queues:
  - Each worker thread maintains its own local task queue (a double-ended queue)
  - New tasks are added to the bottom (LIFO) for cache-friendly, nested task execution
- Stealing:
  - When a thread's local queue is empty, it becomes a "thief"
  - The thief randomly selects a victim thread and steals a task from the top (FIFO) of its deque
- Minimizing Contention:
  - Most operations are thread-local, avoiding the need for synchronization
  - Stealing operations are synchronized but occur infrequently and in a randomized manner

Max thread number in TBB

- TBB uses a global task scheduler that automatically manages worker threads.
- You can limit the maximum number of threads using `tbb::global_control`.
- This is useful to match hardware capabilities or limit resource usage.
- Example: Limit TBB to use only a specific number of threads.

```
1  #include "tbb/global_control.h"
2  #include "tbb/parallel_for.h"
3
4  int main() {
5    // Limit TBB to use only 4 threads
6    tbb::global_control gc(tbb::global_control::max_allowed_parallelism, 4);
7
8    // Example parallel work
9    tbb::parallel_for(0, N, [&](int i){
10       // Compute something for index i
11   });
12
13   return 0;
14 }
15
```

TBB arenas: Isolated execution contexts

- TBB arenas let you create isolated execution contexts with custom concurrency levels.
- They allow you to control resource usage and isolate parallel work from the global scheduler.
- Use `tbb::task_arena` to define a pool of worker threads dedicated to a specific section of code.
- Execute tasks within an arena using the `execute()` method.

```
1  #include "tbb/task_arena.h"
2  #include "tbb/parallel_for.h"
3
4  int main() {
5    // Create a task arena with a maximum of 2 threads
6    tbb::task_arena arena(2);
7
8    arena.execute([&]{
9      tbb::parallel_for(0, N, [&](int i){
10       // Work executed within the arena
11       process(i);
12     });
13   });
14
15   return 0;
16 }
17
```

TBB core parallel execution constructs

- `tbb::parallel_for`: Distributes loop iterations among tasks
- `tbb::parallel_reduce`: Performs reductions over a range
- `tbb::parallel_scan`: Computes prefix sums in parallel
- `tbb::parallel_invoke` for tasks: Executes independent functions concurrently
- `tbb::task_group`: For more convenient parallel task management
- Concurrent containers and synchronization primitives

tbb::blocked_range

- Purpose: Defines a range of values to be processed in parallel
- Usage: Commonly used with `tbb::parallel_for` and `parallel_reduce`
- Range Specification: Represents a half-open interval $[begin, end)$ and supports an optional grain size
- Grain Size:
  - Specifies the minimum number of iterations in a subrange.
  - A small grain size increases task granularity, enhancing load balancing but may add overhead.
  - A large grain size reduces overhead but may cause imbalance if work per iteration varies.

```
1  #include "tbb/blocked_range.h"
2
3  // Example: process a range of indices [0, N)
4  tbb::blocked_range<size_t> range(0, N, grain_size);
5
6  tbb::parallel_for(range, [&](const tbb::blocked_range<size_t>& r) {
7    for (size_t i = r.begin(); i != r.end(); ++i) {
8      process(i);
9    }
10 });
11
```

Parallel loops (tbb::parallel_for)

- Splits a loop range into subranges executed in parallel
- Uses a blocked range to define the iteration space

```
1  #include "tbb/parallel_for.h"
2  #include "tbb/blocked_range.h"
3  #include <vector>
4
5  std::vector<int> data(N);
6
7  tbb::parallel_for(tbb::blocked_range<size_t>(0, N),
8    [&](const tbb::blocked_range<size_t>& r) {
9      for(size_t i = r.begin(); i != r.end(); ++i) {
10       data[i] = compute(i);
11     }
12   });
13
```

Or with range-based loop:

```
1  #include "tbb/parallel_for.h"
2  #include "tbb/blocked_range.h"
3  #include <vector>
4
5  std::vector<int> data(N);
6
7  tbb::parallel_for(tbb::blocked_range<size_t>(0, N),
8    [&](const tbb::blocked_range<size_t>& r) {
9      for(auto i : r) {
10       data[i] = compute(i);
11     }
12   });
13
```

Static scheduling with `static_partitioner`

- TBB uses dynamic work-stealing by default
- Use `tbb::static_partitioner` to enforce a static division of the iteration space
- This partitioner divides work evenly at the start, which can reduce overhead for regular, balanced workloads

```
1  #include "tbb/parallel_for.h"
2  #include "tbb/blocked_range.h"
3  #include "tbb/static_partitioner.h"
4  #include <vector>
5
6  std::vector<int> data(N);
7
8  tbb::parallel_for(
9    tbb::blocked_range<size_t>(0, N),
10   [&](const tbb::blocked_range<size_t>& r) {
11     for(auto i : r) {
12       data[i] = compute(i);
13     }
14   },
15   tbb::static_partitioner()  // Use static partitioning
16 );
17
```

Parallel reduction (tbb::parallel_reduce)

- Performs reduction (e.g., sum, max) across a range
- Divides the work and then combines partial results

```
1  #include "tbb/parallel_reduce.h"
2  #include "tbb/blocked_range.h"
3  #include <vector>
4  #include <functional>
5
6  std::vector<int> array(N);
7
8  int total = tbb::parallel_reduce(
9    tbb::blocked_range<size_t>(0, N),
10   0,
11   [&](const tbb::blocked_range<size_t>& r, int local_sum) -> int {
12     for(size_t i = r.begin(); i != r.end(); ++i)
13       local_sum += array[i];
14     return local_sum;
15   },
16   std::plus<int>());
17
```

Supported operations:

- Sum: `a + b`
- Product: `a * b`
- Minimum: `std::min(a, b)`
- Maximum: `std::max(a, b)`
- Logical AND / OR: `a && b` and `a || b`
- Custom operations: e.g., merging data structures or combining histograms

TBB tasks

- TBB tasks represent independent units of work
- They allow fine-grained parallelism and dynamic scheduling

```
1  #include "tbb/parallel_invoke.h"
2
3  tbb::parallel_invoke(
4      [](){ compute_task_A(); },
5      [](){ compute_task_B(); },
6      [](){ compute_task_C(); }
7  );
8
```

TBB task groups

- The `task_group` interface simplifies tasks management
- It allows to launch and wait for a bunch of tasks

```
1  #include "tbb/task_group.h"
2
3  void compute() {
4    tbb::task_group tg;
5    tg.run([](){ compute_task_A(); });
6    tg.run([](){ compute_task_B(); });
7    tg.wait();  // Wait for both tasks to complete
8  }
9
```

Scan algorithm (tbb::parallel_scan)

- Useful for parallel prefix sum (scan) operations
- Supports both inclusive and exclusive scans

```
1  #include "tbb/parallel_scan.h"
2  #include "tbb/blocked_range.h"
3  #include <vector>
4  #include <numeric>
5
6  std::vector<int> data(N);
7  int initial = 0;
8
9  tbb::parallel_scan(
10   tbb::blocked_range<size_t>(0, N),
11   initial,
12   [&](const tbb::blocked_range<size_t>& r, int running_total, bool is_final_scan)
       -> int {
13     for(size_t i = r.begin(); i != r.end(); ++i) {
14       running_total += data[i];
15       if(is_final_scan)
16         data[i] = running_total;
17     }
18     return running_total;
19   },
20   std::plus<int>());
21
```

Synchronization

- TBB provides synchronization primitives such as:
  - `tbb::mutex` and `tbb::spin_mutex` for mutual exclusion
  - Atomic operations and concurrent containers for lock-free data access
- The runtime work-stealing scheduler minimizes contention by dynamically balancing tasks

Mutex (tbb::mutex)

- Provides mutual exclusion for protecting shared data
- Uses RAII (resource acquisition is initialization paradigm) with scoped_lock to automatically manage locking
- Lightweight and efficient for fine-grained synchronization

```
1  #include "tbb/mutex.h"
2  #include <vector>
3
4  tbb::mutex m;
5  std::vector<int> shared_data;
6
7  void update_data(int value) {
8    tbb::mutex::scoped_lock lock(m); // Lock acquired
9    shared_data.push_back(value);
10   // Lock released automatically when 'lock' goes out of scope
11 }
12
```

What about the barrier in TBB?

- Unlike OpenMP, TBB does not provide an explicit barrier construct
- Implicit Synchronization:
  - TBB parallel algorithms (e.g., `tbb::parallel_for`, `tbb::parallel_reduce`) return only after all tasks are complete
  - This behavior naturally synchronizes work without needing an explicit barrier
- Task Group Synchronization:
  - When using `tbb::task_group`, call `wait()` to ensure all spawned tasks have finished
- Design Philosophy: TBB's task-based model minimizes the need for explicit synchronization, improving scalability and reducing overhead

Brief advanced TBB features overview

- Pipelines and Flow Graphs:
  - `tbb::flow::graph`: Build complex workflows using nodes
  - Example: Create a pipeline that processes data through `function_node`, `buffer_node`, and `join_node` to orchestrate task dependencies
- Thread-safe data structures
  - `tbb::concurrent_vector`: Dynamic array with concurrent push-backs
  - `tbb::concurrent_hash_map`: High-performance hash table for concurrent access
  - `tbb::concurrent_queue`: Lock-free queue
  - `tbb::concurrent_unordered_map`: Unordered map optimized for parallel workloads
- Scalable Memory Allocation optimized for parallel environments:
  - `tbb::scalable_allocator`: Can be used with STL containers to reduce memory contention
  - Example: Using `tbb::scalable_allocator` with a `std::vector` for improved allocation performance in multi-threaded scenarios
- and others...

Performance: TBB vs OpenMP

- Performance factors:
  - TBB dynamic scheduling may introduce overhead for fine-grained tasks
  - OpenMP static scheduling can be more efficient for uniform workloads
- Scalability and load balancing:
  - TBB deals significantly better with unbalanced workloads with its work-stealing scheduler
  - OpenMP may perform better in highly regular, compute-intensive loops
- Optimization and tuning:
  - Both TBB and OpenMP are highly optimized
  - Real-world performance is case-dependent: benchmarking on target hardware and specific tasks is essential

Thank You!

References

- oneAPI Threading Building Blocks GitHub repository: `https://github.com/uxlfoundation/oneTBB`
- oneAPI Threading Building Blocks (oneTBB) documentation: `https://uxlfoundation.github.io/oneTBB/`
- CppCon 2015: Pablo Halpern "Work Stealing": `https://www.youtube.com/watch?v=iLHNF7SgVN4`
- Pushing the limits of work-stealing: `https://community.intel.com/legacyfs/online/drupal_files/managed/9d/48/ConfAnton-Pushing-the-limits-of-work-stealing-approved.pdf`