UNIVERSITÀ DI PISA

TBB – Intel Thread Building Blocks

SPD course 2025-26

Massimo Coppola

12/03/2026

ISTITUTO DI SCIENZA E TECNOLOGIE DELL'INFORMAZIONE "A. FAEDO"

HPC lab

Thread Building Blocks : History

- A C++ template library that simplifies writing thread-parallel programs and debugging them
- Originated circa 2006 as a commercial product
  - First version was still very low-level, closer to a debugging tool, C++03 based
  - Strong emphasis was on how to performance-debug thread-parallel programs
- Several releases improved the abstraction level
  - first retooled toward general-purpose multicores, then forward-looking toward the massive multicore CPUs of 202X
  - Current TBB is a programming model & runtime
  - Subsumed within the OneAPI umbrella (TBB = oneTBB)
- Co-evolved with the C++ standard and STL
  - Many TBB original features are dropped as they enter the standards

Thread Building Blocks Release

- Latest release: oneTBB 2022.3.0 Oct 2025
- A C++ based pattern language for threads
  - Supports generic programming
  - Supports nested parallelism
- Double licensed - separate version for industrial users
  - Intel Simplified Software License
    - No commitment to support, no reverse engineering, no decompilation…
  - Open-source version
    - Stable versions (expected to be) aligned with commercial ones
    - Developer, source-only versions
  - Used to be GPL V2, then TBB 2017 moved to Apache 2.0
    - After V4.4 stable, update 6 released Sept. 2016, Intel changed released naming scheme
  - Documentation is provided online
    - Latest Specifications and reference docs
      https://oneapi-spec.uxlfoundation.org/specifications/oneapi/latest/elements/onetbb/source/nested-index
      https://uxlfoundation.github.io/oneTBB/main/reference/reference.html

Thread Building Blocks Compatibility

- Source code on github
  - https://github.com/uxlfoundation/oneTBB
  - https://github.com/oneapi-src/oneTBB
- **Multi-OS**
  - X86 Windows, Linux, MacOS 10.11+, direct support
  - Android Support (Apache 2 version)
  - More OS support from the community (e.g. FreeBSD 11, Windows/MacOS on ARM)
- Several **development environments**
  - OneAPI toolkits (formerly Intel Parallel Studio)
    - Merged into the OneAPI umbrella SDK
    - + other SW packages and tools from Intel
    - Most notably, Parallel STL
  - Microsoft Visual Studio
  - Works with GCC, Clang, MIcrosoft and Intel C compilers
    - Requires CMake and at least C++17 support)

What is oneTBB today

- A runtime and a template library for C++
- Eases writing thread programs by raising the abstraction level
  - OS-portable thread programs (Win, Linux, OS X)
  - HW independent programs, of course
  - Focus on task production/processing via threads, not on writing thread code
- C++ templates and classes for
  - Common forms of parallelism
  - Data structures used by these parallel "skeletons"
    - Heavy use of generics for expressiveness
  - Auxiliary data structures for parallelism management
    - e.g., **range** to define the set of values of a parameter
  - Use of Operators to specify each skeleton semantics
    - A form of encapsulation of sequential behaviour
- Parallel STL implementation
  - Intel open sourced its parallel STL implementation

TBB Features

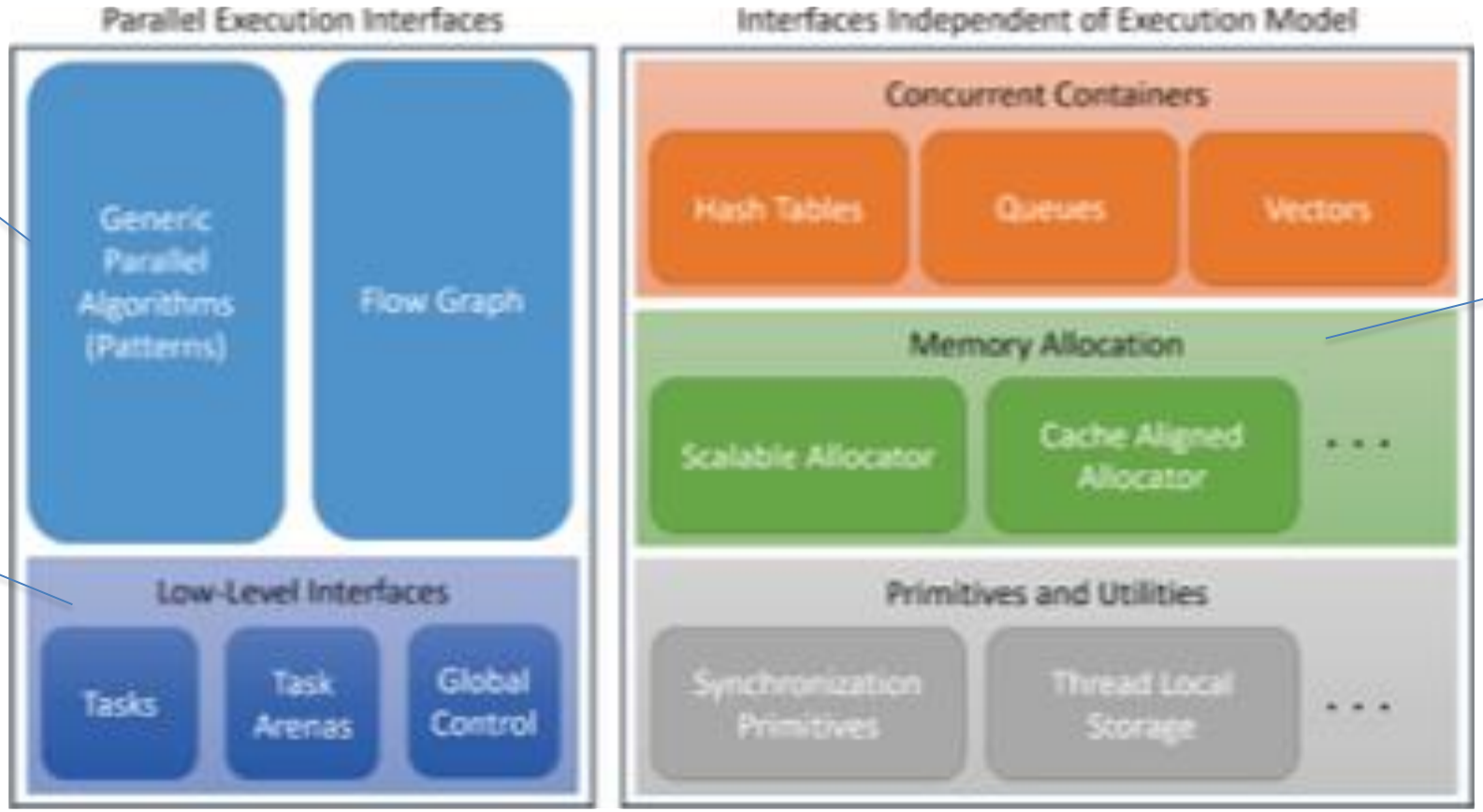
- Portable environment
  - Current 2022.3.0 release requires a C++17 standard compiler
  - Extensive use of templates
- No vectorization support (portability)
  - use vector support from C++17
  - check vectorization support in *Parallel STL*
- Full environment: compile time + runtime
- TBB supports patterns as well as other features
  - algorithms, containers, mutexes, tasks...
  - mix of high-level and low-level mechanisms
  - programmer must choose wisely

TBB Runtime support

- Runtime supports
  - memory allocation
  - synchronization
  - task management
- Provide operating system-independent basic primitives
- Two support libraries
    - may also be used independently
- Template library for
  - Task generation
  - Parallel patterns
  - Task scheduling to threads, work stealing
- A library for scalable memory allocation
  - thread allocation pools, cache alignment, interoperation with C++ memory allocation

TBB “layers”

- All TBB architectural elements are present in the user API, **except** the actual threads



generic and scalable: for, reduce, work pile, scan, pipeline, flow graph

Scheduler, work stealing, groups, over/under subscription

Scalable mem. allocation, false-sharing avoidance, thread-local pools

Picture from “Today’s TBB”, 2nd ed., under CC BY/NC/ND

Threads and Pattern Composition

- Composing = nesting parallel patterns
  - a pipeline of farms of maps of farms …
  - a parallel for nested in a parallel loop within a pipeline
  - each construct can express more potential parallelism
  - deep nesting → too many threads → overhead
  - insufficient nesting → not enough threads → performance loss
- Potential parallelism should be expressed
  - difficult or impossible to extract for the compiler
- Actual expressed parallelism should be flexibly tuned
  - messy to define and optimize for the programmer, low performance portability
- TBB solution
  - Potential parallelism = tasks
  - Actual parallelism = threads
  - Mapping tasks over threads is largely automated and performed at run-time

Tasks vs threads

- Task is a unit of computation in TBB
  - can be executed in parallel with other tasks
  - the computation is carried on by a thread
  - task mapping onto threads is a choice of the runtime
    - the TBB user can provide hints on mapping
- Effects
  - Allow **Hierarchical Pattern Composability**
  - raise the level of abstraction
    - avoid dealing with different thread semantics
  - increase run-time portability across different architectures
    - adapt to different number of cores/threads per core

Basic TBB abstractions

- TBB Algorithms, i.e. the templates actually expressing thread (task) parallel computation
- Data container classes that are specific to TBB
  - some features are now provided in STL containers
- A few **C++ Concepts**, i.e. sets of template requirements that allow to combine C++ data container classes with parallel patterns
  - Range
  - Splittable
- Lower-level mechanisms (thread-local storage, Mutexes) that allow the compentent programmers to implement new abstractions and solve special cases while still producing generic, portable TBB code

Main TBB supported / used abstractions

- **parallel_for**
- **parallel_invoke**
- **parallel_for_each**
- **parallel_reduce**
  - parallel_deterministic_reduce
  - parallel_scan
- **parallel_pipeline**
- **parallel_sort**

Current TBB provides 8 "algorithms"

- **Flow Graph**
  - More general than pipeline, supports dags with various node policies and types of links
- **lambda expressions**
- **concurrency-safe containers**
- **mutex helper objects**
  - those not deprecated as superseded by standard C++ /STL features

parallel_invoke

- simple fork-join parallelism
- independent tasks, waits for completion

```
void example(myVector& v1, myVector& v2) {
    tbb::parallel_invoke(
        [&]() { myfunctionA(v1.begin(), v1.end()); },
        [&]() { myfunctionB(v2.begin(), v2.end()); }
    );
}
```

Parallel for (and partitioners)

- Express independent task computations
  - parallel_for (iteration space , function)
  - it is used to code parallel loops, but it is actually a map applied to a given set of items
  - it assumes there are **NO interaction** among the element computations = safe parallelism
- Exploit a blocked_range template to express iteration space and create tasks
  - The set is known at the beginning of the execution
  - Ranges can be recursively split by the library
  - 1D, 2D, 3D blocked ranges as of TBB 4.0
- Automatic dispatch to independent threads
  - Heuristics within the library, but it can be customized
    - Specify optional *partitioner* function to the parallel_for
    - Specify *grainsize* parameter in the range
  - **Partitioners** allow to customize the way ranges are split in order to obtain tasks amenable to concurrent computation
  - **Grainsize** is the standard parameter of partitioners

Parallel_for minimal examples

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

tbb::parallel_for( 0, height,
   [&in_rows, &out_rows, width, tints](int i) {
      for ( int j = 0; j < width; ++j ) {
         const Pixel& p = in_rows[i][j];
         auto b = f_b(p.bgra[0],tints[0]);
         auto g = f_g(p.bgra[1],tints[1]);
         auto r = f_r(p.bgra[2],tints[2]);
         out_rows[i][j] = f_pixel(b,g,r);
      }
   }
);
```

Lambda expression

- Unnamed functions defined since the C++ 0x standard
- Use a stereotype for in-place defining an unnamed free function
  [variable_scope] type_def function_def;
  - some support for storing the definition
- Capture all variable references which are used inside, but defined outside the function
  - Variable scope spec can dictate capturing by reference, by value, or disallow use
  - In general, e.g. [] disallow [=] by value [&] ref.
  - For specific variable(s)
    [=,&z] all by value, with only z by reference

Parallelism extraction: Splittable Concept

- A type is splittable if it has a so-called *split constructor* that allows splitting an instance in two parts
  - X::X(X& x, split)
    Split X into X and newly constructed object
  - First argument is a reference to the original object
  - Second argument is a dummy placeholder
- Split concept is used to express
  - Range concepts, to allow recursive decomposition
  - Forking a body (a function object) to allow concurrent execution
    (see the reduce algorithm)
- The binary split is usually in almost equal halves
  - Range classes can also have a further split method that specifies the split proportion

TBB Range classes

- Range classes express intervals of parameter values and their decomposability
  - **recursively** splitting intervals to produce parallel work for many patterns (e.g. for, reduce, scan…)
  - a range is automatically constructed from the parallel algorithm parameters, when possible

- The Range concept relies on five mandatory and two optional methods
  - copy constructor
  - destructor
  - is_divisible() true if range is not too small
  - empty() true if range empty
  - split() split the range in two parts
  - *two more methods allow proportional split*

The Range concept

Class R implementing the concept of range must define:

```
R::R( const R& );
R::~R();
bool R::is_divisible() const;
bool R::empty() const;
R::R( R& r, split );
```

Split range R into two sub-ranges.

One is returned via the parameter,
one is the range itself (reduced as needed)

Blocked Range

- TBB 4 has implementations of the range concept as templates for 1D, 2D and 3D blocked ranges
  - 3 nested parallel for are functionally equivalent to a simple parallel for over a 3D range
  - the 2D and 3D range will likely exploit the caches better, due to the explicit 2D/3D tiling

```
tbb::blocked_range<  Value > Class
tbb::blocked_range2d<RowValue,   ColValue > Class
tbb::blocked_range3d<PageValue,  RowValue, ColValue > Class
```

Scheduling tasks to threads

- The **Partitioner** creates multiple tasks
  - by decomposing a range until we get enough parallelism OR we achieve the minimum task size
- Task **scheduler** dispatches tasks to threads
  - Automatically created by the library
  - Customizable by program to suit user needs
    - Define scheduler creation/destruction time
    - Number of created threads
    - Stack size for threads
  - Customizable per construct
    - via construct parameters
- Much more in the docs about the scheduler
  - The task scheduler also deals with all the other algorithms

Partitioners and choosing grain size

- As always, small grain size → high overhead
  - Intel used to suggest 100.000 clock cycles as grain size
  - They also suggest an experimental procedure to set it
  - You are expected to already know the issues, and take into account the number of cores and load balancing details in your algorithm
- Cache affinity can impact performance
  - *affinity partitioner* tries to exploit it when scheduling tasks to threads

Type	Use	Conditions
simple	Chunks given by grain size (Default until TBB 2.2)	g/2 < chunk size <g
auto	Automatic size (heuristics, default nowadays)	g/2 < chunks size
affinity	Automatic size (heuristics to exploit affinity)	g/2 <chunksize

Quick intro of the other algorithms (I)

- parallel_for_each
  - the iteration space is not known in advance, new elements may be added
  - compatible with iterators that do not allow splitting (less scalable)
  - full scalability if used with a splittable range
- tree based patterns
  (parallel_reduce, parallel_deterministic_reduce, parallel_scan)
- e.g. parallel_reduce
  - Basic form is analogous to the parallel for
    parallel_reduce (iteration_space, function)
  - Iteration space also defined as blocked_range
  - The function to apply has different C++ type template w.r.t to parallel loop
    - **Reduce operator does not have the same const-requirements as the one used in a for**
  - Also accepts an optional ***partitioner***

Quick intro of the other algorithms (II)

- parallel_pipeline
  - the usual pipeline, with stages (*filters*) transforming data on the stream
  - *serial* vs *parallel* filters : heavy stages can be replicated
  - parameters are the chain of filters and the maximum capacity of the pipeline
- parallel_sort
  - this is actually an algorithm
  - can sort sequences and containers, aiming at best parallelism
    - not stable and not deterministic

Container data Structures (I)

- C++ STL containers have limited support for concurrency
  - wrapping them in a mutex is not a scalable solution
- TBB containers are data structures
  - which are very often used in programs,
  - whose thread-safe implementation is not trivial
  - or it does not match standard semantics
  - Special care is taken to avoid decreasing program performance
  - built in locks and management policies are embedded in the container implementation,
  - they provide specific (non-standard) semantics that enables high scalability

Container data Structures (II)

- concurrent_vector
  - Random access by index, index of the first element is zero.
  - Growing the container does not invalidate existing iterators or indices.
    - **Multiple threads can grow the container and append new elements concurrently**
  - Destroying elements is not thread safe
  - Does not move its elements in memory when growing (and no insert() or erase())
    - Growing by too small a size increases memory fragmentation
  - Operations on the *whole* vector are not thread-safe; can move elements in memory (and reduce fragmentation)
    - notably reserve() and shrink_to_fit()
- meets requirements for Container and Reversible Container as specified in the ISO C++ standard
- It does **not** meet the Sequence requirements due to absence of methods insert() and erase()
- sports higher overhead than std:vector, but allows higher concurrency

Container data Structures (III)

- Associative Containers: they come in many flavors
  - ordered vs unordered (typically hash-based implementation)
  - map vs set
  - single values per key vs multiple values
- map, multimap, set, and multiset
  - both ordered and unordered
- concurrent_hash_map
  - predates both C++11 and STL
  - Constant or update access to elements
  - Access to elements (i.e. accessor fine-grain locks) can block other threads
- concurrent_queue
  - Simultaneous push/pop from concurrent threads
  - Ensure serialization and preserve object order
    - **Bottleneck if improperly used**
  - pop / push / try_push / size
  - regular, bounded and priority versions

Synchronization and Mutexes

- Classes to build *lock objects*
- The new lock object will generally
  - Wait according to specific semantics for locking
  - Lock the object
  - Release lock when destroyed
- Several characteristics of mutexes
  - Scalable
  - Fair
  - Recursive
  - Yield / Block
- A quick taxonomy:
  - mutex vs rw locks, recursive vs spin vs queueing locks, null locks
  - specific reader/writer locks, can upgrade/downgrade operation to change r/w role
  - some mutex implem.s still provided by TBB, some are now provided by C++ within **std::**

Synchronization and atomics

- Atomics are often less expensive and better preserve concurrency
  - if the operation you need is simple enough to be implemented with atomics
- Nowadays atomics are provided by C++
  - original TBB also defined atomic operations

References

- Michael J. Voss, James R. Reinders *"Today's TBB - C++ Parallel Programming with Threading Building Blocks"*, 2nd Edition
  - Preface – general concepts (mostly covered in previous courses)
  - Appendix A – TBB History, if you are curious
  - Chapter 2 – Algorithms (p. 29 – 35 + the introduction of each algorithm)
  - Chapter 3 – Data Structures for Concurrency (introduction)
  - Chapter 8 – Atomics (introduction)

Practice

- Download docs and example code from the provided links
- Check the accompanying docs
  - Getting started – install and first compilation example ← *TRY IT*