Intel Thread Building Blocks

SPD course 2017-18
Massimo Coppola
23/03/2018

Thread Building Blocks : History

- A library to simplify writing thread-parallel programs and debugging them
- Originated circa 2006 as a commercial product
  - First version was still very low-level
  - Little more than a debugging tool
  - Strong emphasis was put on how to performance debug thread-parallel programs
- Several releases improved the abstraction level
  - Current TBB is a programming model & runtime

Thread Building Blocks Release

- V4.4 stable, update 6 released Sept. 2016
  - Intel changed released naming scheme
- Latest release:TBB 2018 U2 – Dec. 2017

- 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, decompilation…
  - Open source version
    - Stable versions (expected to be) aligned with commercial ones
    - Developer, source-only versions
  - Used to be GPL V2, TBB 2017 moved to Apache 2.0
  - Documentation is provided online
    - https://software.intel.com/en-us/tbb-reference-manual

Thread Building Blocks Compatibility

- Source code on github
  - https://github.com/01org/tbb/releases.
- **Multi-OS**
  - Windows, Linux, OS X 10.11+, direct support
  - Android Support (Apache 2 version)
  - More OS support in the open source (e.g. FreeBSD 11)
- Several **development environments**
  - Intel Parallel Studio 2018 Beta
    - + other SW packages and tools from Intel
    - Most notably, Parallel STL
  - Microsoft Visual Studio 2017
  - Works with GCC, Clang, Intel C compilers (requires C++11 support)

What is TBB 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

TBB Features

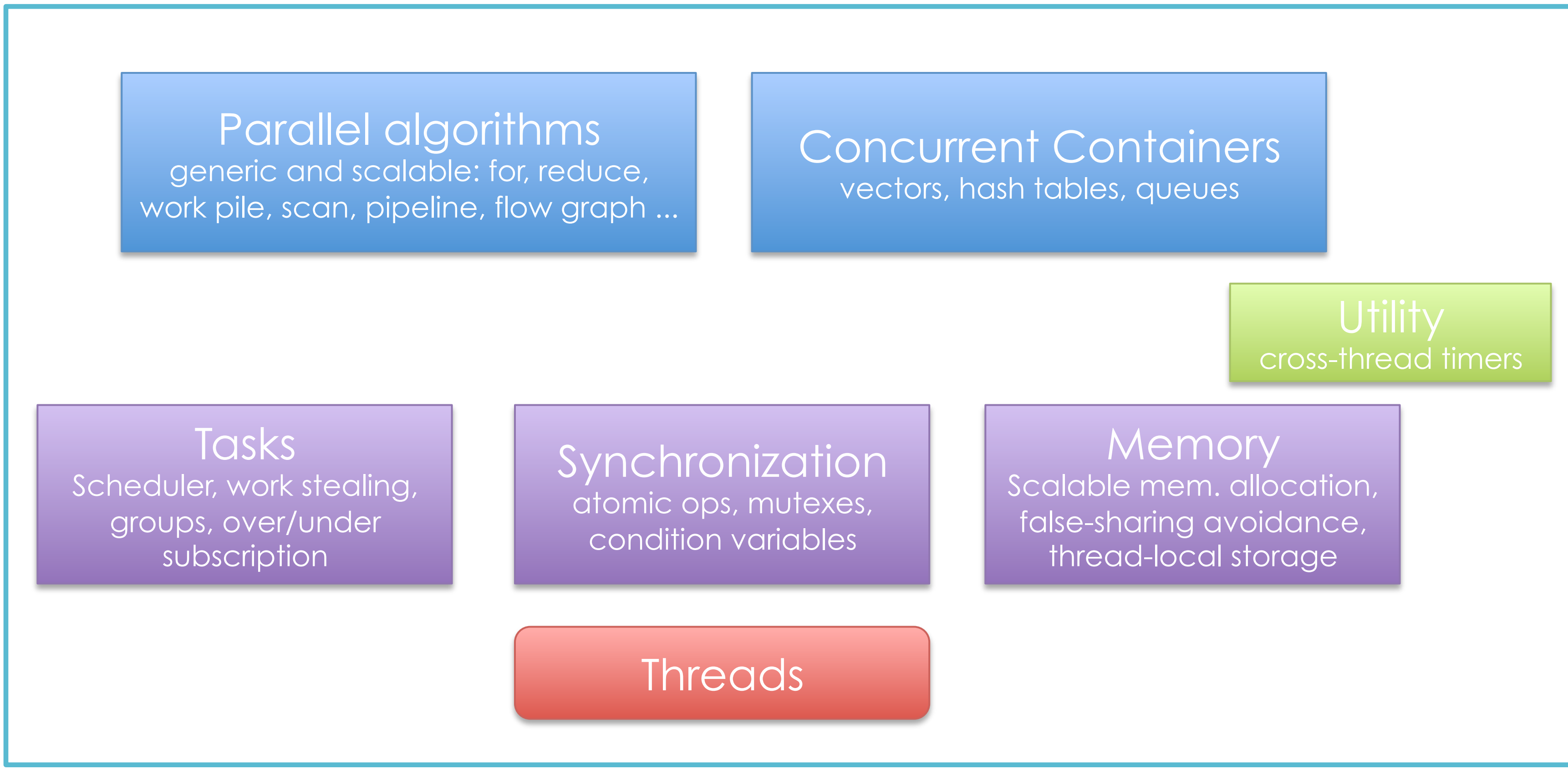
- Portable environment
  - Based on C++11 standard compilers
  - Extensive use of templates
- No vectorization support (portability)
  - use vector support from your specific compiler
  - Check vectorization support in ***Parallel STL***
- Full environment: compile time + runtime
- TBB supports patterns as well as other features
  - algorithms, containers, mutexes, tasks...
  - mix of high 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
  - The two can also be used independently
- One library for
  - Task generation
  - Parallel patterns
  - Task scheduling to threads,
- A specific library for scalable memory allocation

TBB “layers”

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



Threads and composability

- Composing 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
- Potential parallelism should be expressed
  - difficult or impossible to extract for the compiler
- Actual parallelism should be flexibly tuned
  - messy to define and optimize for the programmer, performance hardly portable
- 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
- A few **C++ Concepts**, i.e. sets of template requirements that allow to combine C++ data container classes with parallel patterns
  - Splittable
  - Range
- Lower-level mechanisms (thread storage, Mutexes) that allow the compentent programmers to implement new abstractions and solve special cases

Some supported abstractions

More patterns added with each version

- **parallel_for**
- **lambda expressions**
- **parallel_reduce**
- parallel_do
- pipeline
  - Extended to dags as supersets of pipelines
- **concurrency-safe containers**
- **mutex helper objects**
- atomic<t> template (atomic operations)

Parallel for (and partitioners)

- Express independent task computations
  - parallel_for (iteration space , function)
- Exploit a blocked_range template to express iteration space
  - 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 example

```
#include "tbb/tbb.h"
using namespace tbb;
class ApplyFoo {
    float *const my_a;
public:
    void operator()( const blocked_range<size_t>& r )
    const {
        float *a = my_a;
        for( size_t i=r.begin(); i!=r.end(); ++i )
            Foo(a[i]);
    }
    ApplyFoo( float a[] ) :
        my_a(a)
    {}
};


void ParallelApplyFoo( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0,n), ApplyFoo(a));
}
```

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
  - Task scheduler deals with pipelines and workflows

Partitioners and choosing grain size

- As always, small grain size → high overhead
  - Intel suggests 100.000 clock cycles as grain size
  - Also suggests experimental procedure to set
  - 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

Lambda expression

- Unnamed functions defined by the latest C++ 0x standard (ISO/IEC 14882:2011)
  – Released September 2011
- 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

Reusing concepts: Parallel reduce

- A brief introduction, we will back to it!
- Expresses the parallel reduction pattern
  - 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***

Container data Structures (I)

- 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 taken to avoid decreasing program performance
- concurrent_hash_map
  - Constant or update access to elements
  - Access to elements can block other threads

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()

Container data Structures (III)

- concurrent_queue
  - Simultaneous push/pop from concurrent threads
  - Ensure serialization and preserve object order
    - Bottleneck if improperly used
  - pop / push / try_push / size

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
- Check implementations in the docs:
  - mutex, recursive_mutex, spin_mutex, queueing_mutex, spin_rw_mutex, queueing_rw_mutex, null_mutex, null_rw_mutex
  - Specific reader/writer locks
  - Upgrade/downgrade operation to change r/w role

References

- Download docs and code from http://threadingbuildingblocks.org/
- Check the accompanying docs
  - Getting started – install and first compilation example ← *TRY IT*
  - Tutorial – tour of main functionalities with examples
  - Reference
- Quick summaries to lamba expressions in C++
  - `http://www.cprogramming.com/c++11/c++11-lambda-closures.html`
  - `http://www.nacad.ufrj.br/online/intel/Documentation/en_US/compiler_c/main_cls/cref_cls/common/cppref_lambda_lambdacapt.htm#cppref_lambda_lambdacapt`