Intel Thread Building Blocks, Part II

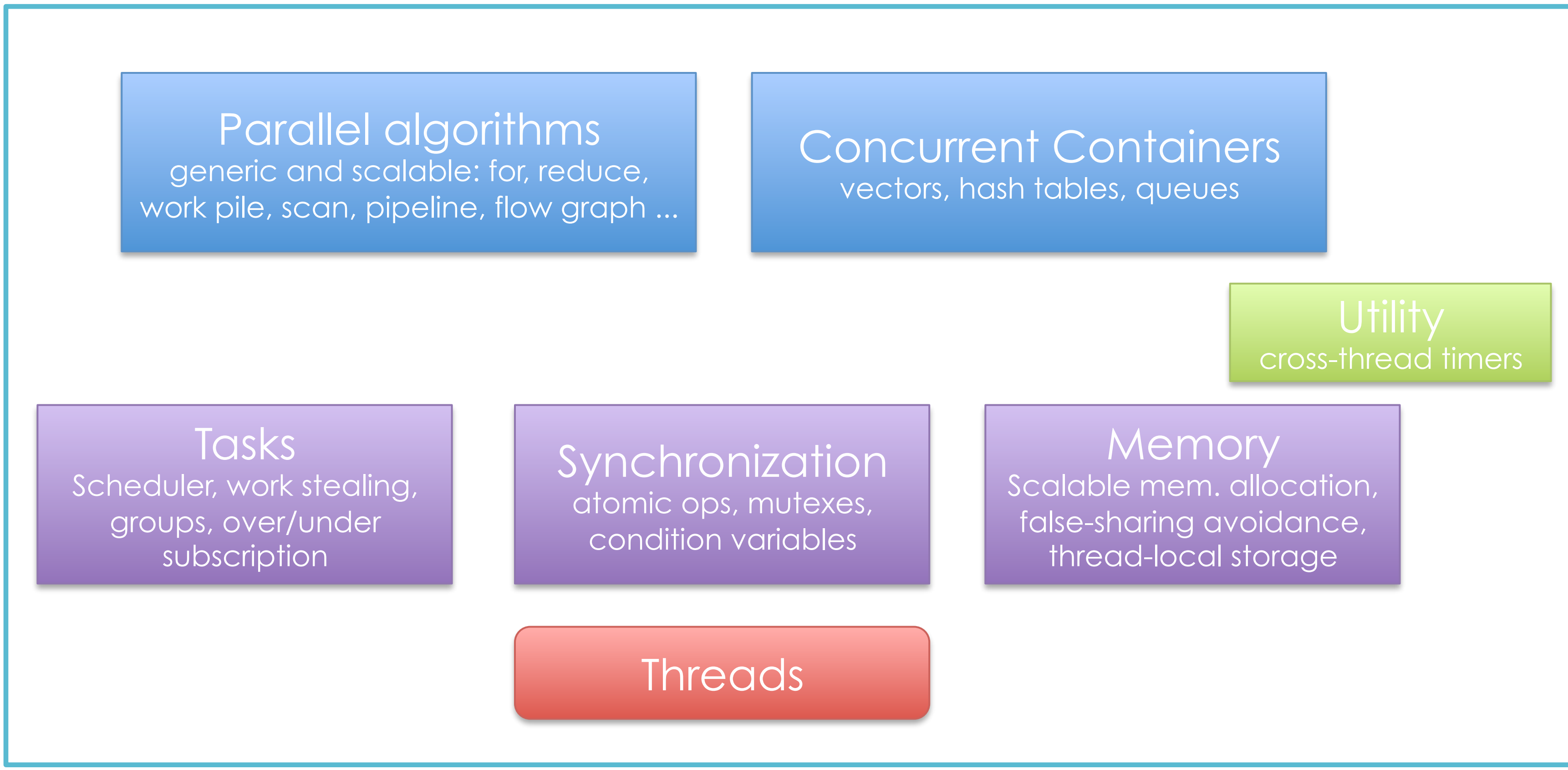
SPD course 2013-14
Massimo Coppola
25/03, 16/05/2014

TBB Recap

- Portable environment
  - Based on C++11 standard compilers
  - Extensive use of templates
- No vectorization support (portability)
  - use vector support from your specific compiler
- Full environment: compile time + runtime
- Runtime includes
  - memory allocation
  - synchronization
  - task management
- TBB supports patterns as well as other features
  - algorithms, containers, mutexes, tasks...
  - mix of high and low level mechanisms
  - programmer must choose wisely

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

TBB 4 Algorithms (1)

Over time, the distinction between parallel patterns and algorithms may become blurred
TBB calls all of them just "algorithms"

- **parallel_for**
  - iteration over a range, can choose partitioner
- **parallel_for_each**
  - iteration via simple iterator, no partitioner choice
- **parallel_do**
  - iteration over a set, may add items
- **parallel_reduce**
  - reduction over a range, can choose partitioner, has deterministic variant
- **parallel_scan**
  - parallel prefix over a range, can choose partitioner

TBB 4 Algorithms (2)

- **parallel_scan**
  - parallel prefix over a range, can choose partitioner
- *parallel_while* (deprecated, see parallel_do)
  - iteration over a stream, may add items
- **parallel_sort**
  - sort over a set (via a RandomAccessIterator and compare function)

- **pipeline** and **filter**
  - runs a pipeline of filter stages, tasks in = tasks out
- **parallel_invoke**
  - execute a group of tasks in parallel
- **thread_bound_filter**
  - a filter explicitly bound to a serving thread

Parallel For each

```
void    tbb::parallel_for_each (InputIterator first,
            InputIterator last, const Function &f)
```

- simple case, employs iterators
- drop-in replacement for std for_each with parallel execution
  - Easy-case parallelization of existing C++ code
- it was a special case of for in previous TBB
- Serially equivalent to:

```
for (auto i=first; i<last; ++i) f(i);
```

- There is also the variant specifying the context (task group) in which the tasks are run

Passing args to parallel patterns

- Beside the range of values we need to compute over, we nee to specify the inner code of C++ templates implementing parallel patterns
- Most patterns have two separate forms
  - Args are a function reference (computation to perform to perform) and a series of parameters (to the parallel pattern)
  - Args contain a user-define class "*Body*" to specify the pattern body,
    - *Body* is a concrete class instantiating a virtual class specified by TBB as a model for that pattern
    - TBB docs calls "requirements" the methods that the *Body* class provides and will be called by the pattern implementation
- Example: for_each uses the first method

Passing args to parallel patterns

- Advantages and disadvantages
- Using functions (TBB documentation calls it the "functional form"...)
  - Easier to use lambda functions
  - We are passing around function references
  - Static (compilation-time) type checking is in some cases limited as the template needs to be general enough
- Using Body classes (TBB calls it "imperative")
  - Slightly more lengthy code
  - Better static type-checking
  - Body classes can more easily contain data/ references – they can have state that simplifies some optimization (ex. see the parallel_reduce pattern)

Optional args to parallel patterns

- A partitioner
  - A user-chosen partitioner used to split the range to provide parallelism
  - see later on the properties of auto_partitioner, (default in any recent TBB) simple_partitioner, affinity_partitioner
- task_group_context
  - Allows the user to control in which task group the pattern is executed
  - By default a new, separate task group is created for each pattern

Parallel For

```
parallel_for (
  tbb::blocked_range<size_t> (begin, end,
  GRAIN_SIZE), tbb_parallel_task());
```

- Loops over integral tipes, positive step, no wrap-around
- one way of specifying it, where tbb_parallel_task is a *Body* user-defined class
- uses a class for parallel loop implementations.
  - The actual loop "chunks" are performed using the () operator of the class
  - the computing function (operator ()) will receive a range as parameter
  - data are passed via the class and the range
- The computing function can also be defined in-place via lambda expressions

Parallel For

```
parallel_for (
  tbb::blocked_range<size_t> (begin, end,
  GRAIN_SIZE), tbb_parallel_task(), partitioner);
```

- Extended version
- the partitioner is one of those specified by TBB (simple, auto, affinity)
- no real choice usually, just allocate a const partitioner and pass it to the parallel loops:

```
tbb::affinity_partitioner ap;
```

  – (unless you want to define your own partitioner)

Parallel_for, 1D alternate syntax

- template<typename Index, typename Func>
  Func parallel_for(Index first, Index_type last,
  const Func& f
  [, partitioner
  [, task_group_context& group]]);
- template<typename Index, typename Func>
  Func parallel_for(Index first, Index_type last,
  Index step, const Func& f
  [, partitioner
  [, task_group_context& group]]);
- Implicit 1D range definition, employs a function reference (e.g. lambda function) to specify the body

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…)

- The Range concept relies on five 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

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 subranges. One is returned via the parameter, the other one is the range itself, accordingly reduced

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
```

partitioners

- simple
  - generate tasks by dividing the range as much as possible (remember about the grain size!)
- auto
  - divide into large chunks, divide further if more tasks are required
- affinity
  - carries state inside, will assign the tasks according to range locality to better exploit caches

Combining the elements

- Apply a range template to your elementary data type
- Define a class computing the proper for-body over elements of a range
- Call the parallel_for passing at least the range and the function
- specify a partitioner and/or a grain size to tune task creation for load balancing

Example (with lambda)

```
void relax( double *a, double *b,
            size_t n, int iterations)
{
   tbb::affinity_partitioner ap;
   for (size_t t=0; t<iterations; ++t) {
      tbb::parallel_for(
         tbb::blocked_range<size_t>(1,n-1),
         [=]( tbb::blocked_range<size_t> r) {
            size_t e = r.end();
            for (size_t i=r.begin(), i<e; ++i)
               /*do work on a[i], b[i] */;
         },
         ap);
      std:swap(a,b); // always read from a, write to b
   }
}
```

Updated References

- Download docs and code from http://threadingbuildingblocks.org/
- Since TBB 4
  - many of the accompanying PDF (tutorial, reference) are no longer made available on the web site. Either
  - ask the teacher for TBB 3.0 copies
  - resort to books
- TBB Accompanying docs
  - download the full TBB source archive, it contains
    - an **example** directory with TBB examples and their description
    - a **doc** directory with full html reference docs
  - Getting started – install and compile examples ← *TRY IT*
- Quick summary to lamba expressions in C++
  - `http://www.nacad.ufrj.br/online/intel/Documentation/en_US/compiler_c/main_cls/cref_cls/common/cppref_lambda_desc.htm`