

Intel Thread Building Blocks, Part II

SPD course 2017-18

Massimo Coppola

26/03/2018

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

Splittable Concept

- A type is splittable if it has a so-called *split constructor* that allows splitting an instance in two parts
 - $X::X(X\& x, \text{ split})$
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 have a further split method that also 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...)
- 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*

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
```

Proportional split

- Class defining methods that allow control over the size of two split halves
- Passed as argument to methods performing a proportional split
 - `proportional_split(size_t _left = 1, size_t _right = 1)`
define a split object using the coefficients to compute the split ratio
 - `size_t left() const`
`size_t right() const`
return the size of the two halves
 - `operator split() const`
backward compatibility with simpler split (allows implicit conversion)

Range with proportional split

- Optional methods allowing proportional splits
 - `R::R(R& r, proportional_split proportion)`
optional constructor using a proportional split object to define the split ratio
 - static const bool `R::isSplittableInProportion`
true iff the range implementation has a constructor allowing the proportional split

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_each**
 - iteration via simple iterator, no partitioner choice
- **parallel_for**
 - iteration over a range, can choose partitioner
- **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_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 need 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) and a series of parameters (to the parallel pattern)
 - Args contain a user-defined 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)

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

- Loops over integral types, 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 (  
    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

- 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
    }
}
```

Temporary slides - to be revised

reduce

- Reduce has also two forms
 - “*Functional*” form, nice with lambda function definitions
 - “*Imperative*” form, minimizes data copying
 - *Please remember this is just TBB terminology*

```
template<typename Range, typename Value, typename  
        Func, typename Reduction>
```

```
Value parallel_reduce( const Range& range,  
                      const Value& identity, const Func& func,  
                      const Reduction& reduction,  
                      [, partitioner[, task_group_context& group]] );
```

```
template<typename Range, typename Body>
```

```
void parallel_reduce( const Range& range,  
                    const Body& body  
                    [, partitioner[, task_group_context& group]] );
```

“Functional” form

- Beside the function, several other objects have to be passed to the reduce
- Value Identity
 - left identity for the operator
- Value Func::operator()(const Range& range, const Value& x)
 - must accumulate a whole subrange of values starting from x (“sequential reduction”)
- Value Reduction::operator()(const Value& x, const Value& y);
 - Combines two values (“parallel” reduction)

Object-oriented form

- Computes the reduction on its Body object together with the associated Range
 - Data (reference) is held within the Body
 - The reduce can split() the body parameter, and will split() the range accordingly
 - Can also split only the range, and compute over a range that is smaller than the Body's data
 - This may allow saving some data copy operation when we exploit parallel slackness together with affinity
 - Results from each side will be combined
- Body object's state contains the reduced value
 - Final result is accumulated in initial Body object

Reduce

- Both the function-based form and the OO one can specify a custom partitioner
- Both forms can specify a task group that will be used for the execution

Reduce – deterministic variant

- `parallel_deterministic_reduce`
- Performs a deterministically chosen sets of splits, joins and computations
- Exploits the `simple_partitioner` → no partitioner argument allowed
- Computes the same regardless of the number of threads in execution
 - no adaptive work assignment is ever performed
 - grain size must be carefully chosen in order to achieve ideal parallelism
- Has both the functional form and the OO one

- Pipeline pattern
 - pipeline class not strongly typed
 - parallel_pipeline strongly typed interface
- Implements the pipeline pattern
 - A series of filter applied to a stream
 - You need to subclass the abstract filter class
 - Each filter can work in one of three modes
 - Parallel
 - Serial in order
 - Serial out of order

Pipeline class

- Pipeline is dynamically constructed
 - pipeline() create an empty pipeline
 - ~pipeline() destructor
 - void add_filter(filter& f) add a filter
 - clear() remove all filters
 - void run(size_t max_number_of_live_tokens
[, task_group_context& group])
- Run until the first filter returns NULL
- Actual parallelism depends on pipeline structure, and on parameter
 - max_number_of_live_tokens
- Pipelines can be reused, but NOT concurrently
- Stages can be added in between runs
- Can have all tasks belong in a specified optional group, by default a new group is created

- Abstract class implementing filters for pipelines
- Three modes, specified in the constructor
 - Parallel can process/produce any number of item in any order (e.g. nested parallelism)
 - Serial out of order filter processes items one at a time, and in no particular order
 - Serial in order filter processes items one at a time, in the received order
- Computation is specified by overriding the operator ()
 - virtual void* operator()(void * item)
 - Process one item and return result, via pointers
 - First stage signals with NULL the end of the stream
 - Result of last stage is ignored

Parallel_pipeline

- void parallel_pipeline(
 size_t max_number_of_live_tokens,
 const filter_t<void,void>& filter_chain
 [, task_group_context& group]);
- Strongly typed, can use lambdas
 - parallel_pipeline(max_number_of_live_tokens,
 make_filter<void,I1>(mode0,g0) &
 make_filter<I1,I2>(mode1,g1) & ...
 make_filter<In,void>(moden,gn));
- Employ the make_filter template to build filters on the spot from their operator() function
- Types are checked at compilation time
 - First stage must invoke fc.stop() and return a dummy value to terminate the stream

- Check that your compiler properly supports lambdas
- Installing TBB from sources and binary do not result in the same configuration
 - environment variables (paths, options) affect compilation
 - identify proper switches in compilation
- `tbbvars.sh` can set proper variables for you (but it is buggy in some TBB versions)
- Makefiles to compile examples tend to work reliably, but they rebuild their configuration each time