Intel Thread Building Blocks, Part II

SPD course 2014-15
Massimo Coppola
5/05/2015
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 $\rightarrow$ too many threads $\rightarrow$ 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
Summary

• A quick tour of TBB abstractions used to express parallelism
  – A few **C++ Concepts**, i.e. sets of template requirements that allow to combine C++ data container classes with parallel patterns
    • Splittable
    • Range
  – TBB Algorithms, i.e. the templates actually expressing thread (task) parallel computation
  – Data container classes that are specific to TBB
  – Lower-level mechanisms (thread storage, Mutexes) that allow the competent programmers to implement new abstractions and solve special cases
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 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
The Range concept

Class R implementing the concept of range must define:

```cpp
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

```cpp
#include <tbb/tbb.h>

namespace tbb {
  namespace {
    template <typename Value>
    class blocked_range {
    public:
      virtual ~blocked_range();
    private:
      Value range_start_;
      Value range_end_;  
    };
  } // namespace {

  namespace detail {
    template <typename RowValue, typename ColValue>
    class blocked_range2d {
    public:
      virtual ~blocked_range2d();
    private:
      RowValue y_range_start_;  
      RowValue y_range_end_;  
      ColValue x_range_start_;  
      ColValue x_range_end_;  
    };
  } // namespace detail

  namespace detail {
    template <typename PageValue, typename RowValue, typename ColValue>
    class blocked_range3d {
    public:
      virtual ~blocked_range3d();
    private:
      PageValue z_range_start_;  
      PageValue z_range_end_;  
      RowValue y_range_start_;  
      RowValue y_range_end_;  
      ColValue x_range_start_;  
      ColValue x_range_end_;  
    };
  } // namespace detail

} // namespace tbb
```
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::is_splittable_in_proportion
    true iff the range implementation has a constructor allowing the proportional split
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
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 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)
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

```c++
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

```cpp
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:
  ```cpp
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
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)

```cpp
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
    }
}
```
Intel Thread Building Blocks, Part III

SPD course 2014-15
Massimo Coppola
05/05/2015
reduce

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

```cpp
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]] );
```

```cpp
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 form each side will the 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
filter

- 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
template<
typename InputIterator,
typename Body>
void parallel_do( InputIterator first,
InputIterator last, Body body
[, task_group_context& group] );

• Only has the object oriented syntax
• Applies a function object body to a specified interval
  – The body can add additional tasks dynamically
  – Replaces completely the deprecated parallel_while
  – Iterator is a standard C++ one
  – A purely serial input iterator is a bottleneck: use iterators over random-access data structures
Adding items in a do

\[ B::\text{operator}(\ T& \ \text{item}, \ \text{parallel}\_\text{do}\_\text{feeder}\<\text{T}>& \ \text{feeder} ) \ \text{const} \]

\[ B::\text{operator}() \ (\ T& \ \text{item} ) \ \text{const} \]

- The body class need to operate on the template T type
- It needs a copy constructor and a destroyer
- Two possible signatures for Body operator()
  - You can’t define both!
  - First signature, with extra parameter, allows each item to add more items dinamically in the do \( \rightarrow \) e.g. dynamically bound parallel do, divide & conquer
  - Second signature means the do task set is static
Intel Thread Building Blocks, Part IV

SPD course 2014-15
Massimo Coppola
05/05/2015
Containers

• container_range
  – extends the range to use a container class

• maps and sets:
  – concurrent_unordered_map
  – concurrent_unordered_set
  – concurrent_hash_map

• Queues:
  – concurrent_queue
  – concurrent_bounded_queue
  – concurrent_priority_queue

• concurrent_vector
container: Container Range

- extends the range class to allow using containers as ranges (e.g. providing iterators, reference methods)
  - Container ranges can be directly used in parallel_for, reduce and scan
- some containers have implementations which support container range
  - concurrent_hash_map
  - concurrent_vector
  - you can call parallel for, scan and reduce over (all or) part of such containers
Extending a container to a range

• Types
  – R::value_type Item type
  – R::reference Item reference type
  – R::const_reference Item const reference type
  – R::difference_type Type for difference of two iterators

• What you need to provide
  – R::iterator Iterator type for range
  – R::iterator R::begin() First item in range
  – R::iterator R::end() One past last item in range
  – R::size_type R::grainsize() const Grain size

• AND all Range methods: split(), is_divisible()…
concurrent map/set templates

• The key issue is allowing multiple threads efficient concurrent access to containers
  – keeping as much as possible close to STL usage
  – at the cost of limiting the semantics
  – A (possibly private) memory allocator is an optional parameter

• containers try to support concurrent insertion and traversal
  – semantics similar to STL, in some cases simplified
  – not all containers support full concurrency of insertion, traversal, deletion
  – typically, deletion is forbidden / not efficient
  – some methods are labeled as concurrently unsafe
    • E.g. erase
Types of maps

• We wish to reuse STL – based code as much as possible
  – However, STL maps are NOT concurrency aware
• Two main options to make them thread-nice
  – Preserve serial semantics, sacrifice performance
  – Aim for concurrent performance, sacrifice STL semantics
• Choose depending on the semantics you need
• concurrent_hash_map
  – Preserves serial semantics as much as possible
  – Operations are concurrent, but consistency is guaranteed
• concurrent_unordered_map, concurrent_unordered_multimap
  – Partially mimic STL corresponding semantics
  – drops concurrent performance hogging features
  – no strict serial consistency of operations
Concurrent_hash_map

• concurrent_hash_map
  – Preserves serial semantics as much as possible
  – Operations are concurrent, but subject to a global ordering to ensure consistency
  – Relies on extensive built-in locking for this purpose
  – Data structure access is less scalable, may become a bottleneck
  – Your tasks may be left idle on a lock until data access is not available
concurrent unordered (multi)map

- concurrent_unordered_map
- concurrent_unordered_multimap
  - associative containers, concurrent insertion and traversal
  - semantics similar to STL unordered_map/multimap but simplified
  - omits features strongly dependent on C++11
    - Rvalue references, initializer lists
  - some methods are prefixed by unsafe_ as they are concurrently unsafe
    - unsafe_erase, unsafe_bucket methods
  - inserting concurrently the same key may actually create a temporary pair which is destroyed soon after
  - the iterators defined are in the forward iterator category (only allow to go forward)
  - supports concurrent traversal (concurrent insertion does not invalidate the existing iterators)
Comparison of maps

• Choose depending on the semantics you need
  • concurrent_hash_map
    – Permits erasure, has built-in locking
  • concurrent_unordered_map
    – Allows concurrent traversal/insertion
    – No visible locking
      • minimal software lockout
      • no locks are retained that user code need to care about
    – Has [ ] and “at” accessors
  • concurrent_unordered_multimap
    – Same as previous, holds multiple identical keys
    – Find will return the first matching <key, Value>
      • But concurring threads may have added stuff before it in the meantime!
Map templates

- template<typename Key,
   typename Element,
   typename Hasher = tbb_hash<Key>,
   typename Equality = std::equal_to<Key>,
   typename Allocator =
   tbb::tbb_allocator<
   std::pair<const Key, Element>>
   class concurrent_unordered_map;

- template<typename Key,
   typename Element,
   typename Hasher = tbb_hash<Key>,
   typename Equality = std::equal_to<Key>,
   typename Allocator =
   tbb::tbb_allocator<
   std::pair<const Key, Element>>
   class concurrent_unordered_multimap;
Concurrent sets

- template<typename Key,
   typename Hasher = tbb_hash<Key>,
   typename Equality = std::equal_to<Key>,
   typename Allocator = tbb::tbb_allocator<Key>
   class concurrent_unordered_set;

- template<typename Key,
   typename Hasher = tbb_hash<Key>,
   typename Equality = std::equal_to<Key>,
   typename Allocator = tbb::tbb_allocator<Key>
   class concurrent_unordered_multiset;

- concurrent_unordered_set
  - set container supporting insertion and traversal
  - same limitations as map: C++0x, unsafe_erase and bucket methods
  - Forward iterators, not invalidated by concurrent insertion
  - For multiset, same find() behavior as with the maps
Concurrent queues

• STL queues, modified to allow concurrency
  – Unbounded capacity (memory bound!)
  – FIFO, allows multiple threads to push/pop concurrently with high scalability

• Differences with STL
  – No front and back access → concurrently unsafe
    • Iterators are provided only for debugging purposes!
    • unsafe_begin() unsafe_end() iterators pointing to begin/end of the queue
  – Size_type is an integral type
  – Unsafe_size() number of items in queue, not guaranteed to be accurate
  – try_pop(T & object)
    • replaces (merges) size() and front() calls
    • attempts a pop, returns true if an object is returned
Bounded_queue

- Adds the ability to specify a capacity
  - `set_capacity()` and `capacity()`
  - default capacity is practically unbounded
- push operation waits until it can complete without exceeding the capacity
  - `try_push` does not wait, returns true on success
- Adds a waiting `pop()` operation that waits until it can pop an item
  - `Try_pop` does not wait, returns true on success
- Changes the `size_type` to a signed type, as
  - `size()` operation returns the number of push operations minus the number of pop operations
  - Can be negative: if 3 pop operations are waiting on an empty queue, `size()` returns -3.
- `abort()` causes any waiting push or pop operation to abort and throw an exception
concurrent_priority_queue

• Concurrent push/pop priority queue
  – Unbounded capacity
  – Push is thread safe, try_pop is thread safe

• Differences to STL
  – Does not allow choosing a container; does allow to choose the memory allocator
  – top() access to highest priority elements is missing (as it is unsafe)
  – pop replaced by try_pop
  – size() is inaccurate on concurrent access
  – empty() may be inaccurate
  – Swap is not thread safe
Concurrent priority queue examples

- `concurrent_priority_queue(const allocator_type& a = allocator_type())`
  - Empty queue with given allocator

- `concurrent_priority_queue(size_type init_capacity, const allocator_type& a = allocator_type())`
  - Sets initial capacity

- Priority is provided by the template type `T`
Concurrent vector

- Random access by index
- Concurrent growth / append
- Growing does not invalidate indexes
- Some methods are NOT concurrent
  - Reserve, compact, swap
- Shrink_to_fit compacts the memory representation
  - Not done automatically to preserve concurrent access, invalidates indexes
- Implements the range concept
  - Can be used for parallel iteration
- Size() can be concurrently inaccurate (includes element in construction)
- Provides forward and reverse iterators
Intel Thread Building Blocks, Part V

SPD course 2014-15
Massimo Coppola
--/--/2015
thread local storage

- enumerable_thread_specific
- a container class providing local storage to any of the running threads

- outside of parallel contexts, the contents of all thread-local copies are accessible by iterator or using combine or combine_each methods
- thread-local copies are lazily created, with default, exemplar or function initialization
- thread-local copies do not move (during lifetime, and excepting clear()) so the address of a copy is invariant.
  - the contained objects need not have operator=() defined if combine is not used.
  - enumerable_thread_specific containers may be copy-constructed or assigned.
  - thread-local copies can be managed by hash-table, or can be accessed via TLS storage for speed.
Synchronization mechanisms

- Low level mechanism to control low-level concurrent access to data structures
- Use with great care
  - Can cause software lockout
- Mutexes
  - Data structures that allow adding generic locking mechanisms to any data structures
- Atomic
  - Template that adds very simple, low overhead, hw-supported atomic behaviour to a few machine types available in the language
- PPL Compatibility
  - 2 constructs added for compatibility with Microsoft Parallel Pattern Library
- C++11 synchronizations
  - Supports a subset of the N3000 draft of the C++11 standard
  - Will change in future implementations of TBB
atomic objects

- template<typename T> atomic;
- Generate special machine instructions to ensure that operating on a variable in memory is performed atomically
- atomics within the C++11 standard (TBB goes beyond it)
- Integral type, enum type, pointer type
- Template supports atomic read, write, increment, decrement, fetch\&add, fetch\&store, compare\&swap operations
- Arithmetic
  - Pointer arithmetic is T is a pointer
  - not allowed if T is enum, bool or void*

* MCSN – M. Coppola – Strumenti di programmazione per sistemi paralleli e distribuiti
atomic objects

• Copy constructor is never atomic
  – It is compiler generated
  – Need to default construct, then assign
    
    atomic<T> y(x);  // Not atomic
    atomic<T> z; z=x;  // Atomic assignment
  – C++11 uses the constexpr mechanism for this

• atomic <T*> defines the dereferencing of data as
  – T* operator->() const;
Atomic methods

- `value_type fetch_and_add( value_type addend )`
  - Add atomically
- `value_type fetch_and_increment()`
- `value_type fetch_and_decrement()`
  - Increment/decrement atomically
- `value_type compare_and_swap( value_type new_value, value_type comparand )`
  - If the atomic has value “comparand” set it to “new_value”
- `value_type fetch_and_store( value_type new_value )`
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_rwlock_mutex, queueing_rwlock_mutex, null_mutex,
    null_rwlock_mutex
  – Specific reader/writer locks
  – Upgrade/downgrade operation to change r/w role
<table>
<thead>
<tr>
<th>Pseudo-Signature</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>M()</td>
<td>Construct unlocked mutex.</td>
</tr>
<tr>
<td>~M()</td>
<td>Destroy unlocked mutex.</td>
</tr>
<tr>
<td>typename M::scoped_lock</td>
<td>Corresponding scoped-lock type.</td>
</tr>
<tr>
<td>M::scoped_lock()</td>
<td>Construct lock without acquiring mutex.</td>
</tr>
<tr>
<td>M::scoped_lock(M&amp;)</td>
<td>Construct lock and acquire lock on mutex.</td>
</tr>
<tr>
<td>M::~scoped_lock()</td>
<td>Release lock (if acquired).</td>
</tr>
<tr>
<td>M::scoped_lock::acquire(M&amp;)</td>
<td>Acquire lock on mutex.</td>
</tr>
<tr>
<td>bool M::scoped_lock::try_acquire(M&amp;)</td>
<td>Try to acquire lock on mutex. Return true if lock acquired, false otherwise.</td>
</tr>
<tr>
<td>M::scoped_lock::release()</td>
<td>Release lock.</td>
</tr>
<tr>
<td>static const bool M::is_rwlock</td>
<td>True if mutex is reader-writer mutex; false otherwise.</td>
</tr>
<tr>
<td>static const bool M::is_recursive_mutex</td>
<td>True if mutex is recursive mutex; false otherwise.</td>
</tr>
<tr>
<td>static const bool M::is_fair_mutex</td>
<td>True if mutex is fair; false otherwise.</td>
</tr>
</tbody>
</table>
### Types of Mutexes

<table>
<thead>
<tr>
<th>Mutex Type</th>
<th>Scalable</th>
<th>Fair</th>
<th>Reentrant</th>
<th>Long Wait</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex</td>
<td>OS dependent</td>
<td>OS dependent</td>
<td>No</td>
<td>Blocks</td>
<td>&gt;=3 words</td>
</tr>
<tr>
<td>recursive_mutex</td>
<td>OS dependent</td>
<td>OS dependent</td>
<td>Yes</td>
<td>Blocks</td>
<td>&gt;=3 words</td>
</tr>
<tr>
<td>spin_mutex</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yields</td>
<td>1 byte</td>
</tr>
<tr>
<td>speculative_spin_mutex</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yields</td>
<td>2 cache lines</td>
</tr>
<tr>
<td>queuing_mutex</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yields</td>
<td>1 word</td>
</tr>
<tr>
<td>spin_rw_mutex</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yields</td>
<td>1 word</td>
</tr>
<tr>
<td>queuing_rw_mutex</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yields</td>
<td>1 word</td>
</tr>
<tr>
<td>null_mutex</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>empty</td>
</tr>
<tr>
<td>null_rw_mutex</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>empty</td>
</tr>
</tbody>
</table>