Intel Thread Building Blocks, Part IV

SPD course 2013-14

Massimo Coppola

20/05/2014

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 succes
- 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

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.

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`