Intel Thread Building Blocks, Part III

SPD course 2017-18
Massimo Coppola
26/03/2018

Parallel_do

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

- Parallel_do has two forms, both using the object-oriented syntax
- Applies a function object body to a specified interval
  - The body can add additional tasks dynamically
  - Iterator is a standard C++ one, however
    - a purely serial input iterator is a bottleneck
    - It is better to use iterators over random-access data structures
- Replaces the deprecated parallel_while

Parallel_do

```
Template<typename Container,
                typename Body>
void parallel_do( Container c, Body body
        [, task_group_context& group] );
```

- Shorter equivalent for processing a whole container
  - iterators are provided by the Container type
  - equivalent to passing std::begin() and std:: end() with the other syntax

Computing and adding items in a do

- The body class need to compute on the template T type e.g. operator()
  - Body also needs a copy constructor and a destroyer

```
B::operator()( T& item,
    parallel_do_feeder<T>& feeder ) const

B::operator()( T& item ) const
```

- Adding items depends on the signature of the Body operator() -- two possible signatures
  - First signature, with extra parameter, allows each item to call a feeder callback to add more items dinamically → e.g. allows dynamically bound parallel do, feedback, divide & conquer
  - Second signature means the do task set is static
  - You can't define both!

Containers

- TBB containers aim at increasing performance for heavy multithreading, while providing a useful level of abstraction
- Mimic STL interfaces and semantics whenever possible
- Change/drop features and introduce minimal locking to provide better performance (separate implementations)
  - Fine grain locking
  - Lock free techniques
- Lower multithread overhead still has a cost
  - may mean higher data management or space overhead

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

ISTITUTO DI SCIENZA E TECNOLOGIE DELL'INFORMAZIONE "A. FAEDO"

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 (can be negative!)
  - 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 with respect 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**
- Provides forward and reverse iterators
- Implements the range concept
  - Can be used for parallel iteration
- Some methods are NOT concurrent
  - Reserve, compact, swap
  - Shrink_to_fit compacts the memory representation
    - Never performed automatically in order to preserve concurrent access: it invalidates indexes
- Size() can be concurrently inaccurate (includes element in construction)

thread local storage

- **enumerable_thread_specific**

```
template <typename T, typename
                Allocator=cache_aligned_allocator<T>,
                ets_key_usage_type ETS_key_type=ets_no_key>

class enumerable_thread_specific;
```

- a container class providing local storage for a type T to any of the running threads
  - it has a combine() method, applying a binary, associative functor (T,T)->T to perform a reduction over the storage items
  - It has a combine_each() to apply a functor T->T to all items
- thread-local copies are lazily created, with default, exemplar or function initialization
- two implementations
  - the default one is not resource constrained, a bit of overhead
  - the more performant one consumes one TLS key each time a thread requires TLS storage; TLS keys are limited (64~128)

thread local storage

- outside of parallel contexts, the contents of all thread-local copies are accessible by iterator or by using combine / combine_each methods
- the address of a copy is invariant, as thread-local copies do not move
  - during their whole lifetime, but **with the exception of clear()**
  - clear() also frees the TLS key, if it was allocated
  - the contained objects need not have operator=() defined if combine methods are 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 generick locking mechanisms to any data structures
- Atomic
  - template that add 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 syncronizations
  - Supports a subset of the N3000 draft of the C++11 standard
  - Subject to changes with new 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 if T is a pointer
  - not allowed if T is enum, bool or void*

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)
  - Atomically add and fetch previous value
- value_type fetch_and_increment()
- value_type fetch_and_decrement()
  - Atomically Increment/decrement and fetch pr.val.
- 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)
  - Atomically fetch previous value and store new one