Doc. No.: N3732
Date: 2013-08-30
Authors: Arch D. Robison, Anton Potapov, Anton Malakhov
Reply To: arch.robison@intel.com

Value-Oriented Concurrent Unordered Containers

Contents

1 Introduction **2**

2 Conceptual Interface **2**
2.1 find . . . 3
2.2 insert . . . 3
2.3 exchange . . . 4
2.4 erase . . . 4
2.5 reduce . . . 4
2.6 clear . . . 5
2.7 size . . . 5
2.8 for_each . . . 6
2.9 is_lock_free . . . 6

3 Discussion **7**
3.1 Requirements on User-Provided Objects . . . 7
3.2 Concurrent Erasure . . . 7
3.3 Emplace Less Useful . . . 7
3.4 No Hint Arguments . . . 8
3.5 Extension to other Unordered Associative Containers . . . 8
3.6 How to Return a Value? . . . 8
3.7 Exception Safety . . . 10
3.8 Redundant Construction . . . 11

4 Synopsis of Concrete Interface **11**

5 Comparison with the Reference-Based Approach **13**

6 Acknowledgments **13**

1 Introduction

An earlier paper N3425 proposed a design for concurrent unordered associative containers that closely adhered to STL conventions, notably by the way its operations returned iterators that pointed into the container. Doing so raised some issues with concurrent erasure, fears of easy misuse, and generally limited the possibilities for non-blocking implementations to split-ordered lists [1]. Feedback from SG1 was to consider not following STL so closely and consider a value-oriented approach, such as N3533, TBB, and PPL take for concurrent queues. This paper sketches such a design for concurrent maps with intent to elicit feedback.

Allowing concurrent operations on a hash table can improve throughput and reduce latency compared to serializing those operations. The existing `std::unordered_map` permits concurrent `find` operations, but not allow concurrent `find`, `insert` and `erase` operations. Frameworks such as Intel(R) Threading Building Blocks (TBB) and Microsoft's Parallel Patterns Library (PPL) offer concurrent hash tables, namely `concurrent_hash_map` (TBB) and `concurrent_unordered_map` (TBB and PPL). Some examples that can exploit concurrent lookup and modification are:

- Memoization tables for parallel dynamic Programming [2].
- Software caches, such as the directory cache in Linux [3].
- Shadow memory used by run-time program analyzers.

Typically, in the first example the table grows monotonically, and thus concurrent erasure is not required. The other two cases often require concurrent erasure.

2 Conceptual Interface

Concurrent operations on a `concurrent_unordered_map` should provide useful atomic actions. The suggested actions are:

find Lookup a key and return the value associated with it.

insert Insert a key-value pair if its key is not already present.

exchange Insert a key-value pair, replacing any existing value associated with the key.

erase Erase a key and its associated value.

reduce Insert a key-value pair, performing a reduction if the key is already present.

Each of the actions returns the previously associated value, wrapped in `optional<mapped_type>` (see N3527), which is "disengaged" if the key was previously absent from the table.

For example, the following snippet attempts to insert a new key-value pair, and reports if the key was already present.

```
if(optional<int> result = table.insert(make_pair(key,data))) {
    cout << key << " already associated with " << *result << endl;
}
```

Actions associated with a given key *synchronize with* each other. Whether stronger guarantees are worth the expense is debatable.

The following sections describe the conceptual interface in more detail. Though the format resembles the standard's descriptions, the wording should be considered informal.

2.1 find

`optional<mapped_type>` *map*::`find(const key_type& k) const;`

Returns: If key `k` is present, object engaged with copy of mapped value associated with the key; otherwise a disengaged object.

`size_type` *map*::`count(const key_type& x) const;`

Returns: 1 if key is present; 0 otherwise.

The name `count` is chosen for sake of consistency with existing associative container requirements and `std::map`.

2.2 insert

`template<class...Args>` *map*::`optional<mapped_type> emplace(Args&&...args);`
`optional<mapped_type>` *map*::`insert(const value_type& obj);`

`template<class P> optional<mapped_type>` *map*::`insert(P&& obj);`

Effects: Insert object if its key is not present. May temporarily construct a key-value pair that is *not* inserted if the key is present.

Returns: An object engaged with a copy of the mapped value previously associated with the given key, or a disengaged object if the key was not present.

The reason for permitting a temporary key-value pair to be constructed, even if not inserted, is to enable non-blocking implementations. These typically work by constructing the pair, attempting to insert it atomically into the container, and if the insertion fails (because the key is already present), destroying the copy. A concurrent `find` operation never sees the copy until (and if) it is actually inserted.

2.3 exchange

`optional<mapped_type>` *map*::`exchange(const value_type& obj);`
`template<class P> optional<mapped_type>` *map*::`exchange(P_type&& obj);`

Effects: Insert object if its key is not present, otherwise replace existing object with `obj`.
Returns: Same as for `insert`.

2.4 erase

`optional<mapped_type> erase(const Key& k);`

Effects: Erase object with given key from the container, if such is present.
Returns: Same as for `insert`.

2.5 reduce

`template<class F> optional<mapped_type>` *map*::`reduce(const value_type& obj, F f=std::plus<mapped_type>());`

Effects: Insert object if its key is not present; otherwise insert `f(`*old*,`obj.second`), where *old* is the mapped values of the existing item. Functor `f` may be invoked multiple times.

Returns: Same as for `insert`.

The intent, though not required, is that `f` is an associative and commutative functor, so that the net result of concurrent `reduce` operations does not depend on their particular evaluation order. For example, a histogram of word counts could be generated by using words for keys, and `int`s for values. Multiple threads could scan different parts of a text file and update the counts using the default `std::plus<int>()`. The final counts would not depend on the update order.

2.6 clear

`void` *map*::`clear() noexcept;`

Effects: Erases all objects from the map.

A point for discussion is whether `clear` should be safe to use concurrently with other methods. If so, an intuitive synchronization guarantee would be that if an insertion *happens before* a `clear` that *happens before* a `find`, the `find` will not see the inserted object. Certainly the container's destructor should not be safe to use concurrently with any other method.

2.7 size

Because `size()` is a global summary of a container's state, there is no way to implement it in constant time that is scalable. A non-constant-time approach is to use per-thread partial sizes, and have each thread update its piece to record its net change to the total size. The total size can then be computed in $O(P)$ time if P threads participated. However, guaranteeing even this level of support seems to yield more bane than boon for most applications, particularly since the reported size can be immediately invalidated by concurrent modifications. Hence the conceptual interface supports a weak form of `size()` that runs in $O(N)$ time, where N is the number of items in the table.

Furthermore, unless the table is quiescent, `size()` returns only an estimate that is not guaranteed to be linearizable with respect to other operations. Nonetheless, an estimate can be valuable for debugging, sanity checks, guiding resource usage. Furthermore, knowing the exact size in the quiescent case can be useful in programs that have phases where the map is known to be quiescent.

`size_type` *map*::`size() const noexcept;`
Returns: Estimate of the number of objects in the container. The estimate is exact if no modifications occur during the invocation.
Complexity: $O(\text{size}())$;

`bool` *map*::`empty() const noexcept;`
Returns: `size()!=0`.

2.8 for_each

Users of TBB's `concurrent_hash_map` sometimes request a way to walk the container while it is undergoing concurrent modification, and N3425 supported such a feature via a typical iterator interface. In a value-based approach, the equivalent would seem to be a method for copying the keys or objects into another container. A minimal solution is to offer a method that performs the walk. It could be extended to do parallel walks in the style of N3724.

`template<class Function> void` *map*::`for_each( Function f ) const;`
Effects: Evaluates `f`(x) for each object x in the container. This method is not necessarily linearizable with respect to modification operations.

An alternative would be to provide a `const_iterator`-like interface, though we fear that the iterator might have to carry much state information.

2.9 is_lock_free

Like the standard library atomics, the interface provides a way to query whether the implementation is lock-free. The answer presumes that any user-provided operations are lock-free.

`static bool` *map*::`is_lock_free() noexcept;`
Returns: `true` if operations on the container are always lock-free, assuming that user-provided operations are lock-free; `false` otherwise.

3 Discussion

3.1 Requirements on User-Provided Objects

Though the interface is "value-based", there is no avoiding reliance on user-defined functions that see references, for example, copy constructors. The expectation is that user-provided objects will provide the same level of thread safety as STL, that operations be safely invoked concurrently on logically const objects [4], that is not cause a race or a deadlock. For example, two concurrent operations on a `concurrent_unordered_value_map` might copy the same value concurrently, but would never assign to the same lvalue concurrently.

We would like to permit lock-based implementations, but avoid surprise deadlock in cases where user-defined code blocks. Thus implementations of the container may not hold internal *exclusive* locks while calling user-defined functions. It seems okay to allow implementations to hold internal *shared* locks while calling user-defined functions.

3.2 Concurrent Erasure

The interface supports concurrent erasure. Since operations on the container return a `optional<mapped_type>` instead of an iterator pointing into the container, there is no fundamental problem with concurrently looking up and erasing an item. Erasure can remove the item from the map immediately, but defers destruction of it until any other threads currently copying or updating it have finished their operation.

A *Reclaimer* template parameter specifies the mechanism that the container's internal implementation should use to deal with access/destruction races. See the companion paper N3712 for a discussion of Reclaimers.

3.3 Emplace Less Useful

Emplace has value in serial containers because it avoids the need for copy-construction. However, in a value-based interface, copy-construction is often necessary anyway. An emplace interface is nonetheless included for sake of uniformity.

3.4 No Hint Arguments

In a concurrent environment, the value of a hint seems limited since it is likely to be invalidated between the time it is created and the time it is used. It could be useful for avoiding recomputation of a hash value, though that would require changing our return type of `optional<mapped_type>` to something that could hold the hash value too. Hence none of the proposed methods takes a hint argument.

3.5 Extension to other Unordered Associative Containers

It seems possible to have "set" and "multi" equivalents with a similar look and feel. Doing so could give the concurrent containers the kind of commonality that their serial equivalents have.

An interface for a `concurrent_unordered_value_set` could be similar to `concurrent_unordered_value_map`, with methods returning `bool` instead of `optional<mapped_type>`.

Likewise, an interface for a `concurrent_unordered_value_multiset` could return with `size_type` values instead of `optional<mapped_type>`, where the value would be the multiplicity of a key.

An interface for `concurrent_unordered_value_multimap` could return some kind of collection type instead of a `optional<mapped_type>`. The collection type could quack like an `optional<mapped_type>` that holds an array.

3.6 How to Return a Value?

This section presents a design choice for which feedback is sought. The issue is how to return values. The authors have considered five alternatives, each with strengths and weaknesses.

1. Return an `optional<T>`, as in the discussion so far.

2. Have two signatures, one that returns a `optional<T>` and one that returns void. They would be distinguished by a dummy parameter, similar to the way the throwing and non-throwing versions of `operator new` are distinguished. For example, the signatures for insertion might look like:

   ```
   void insert(const value_type& x, void_return_t);
   optional<mapped_type> insert(const value_type& x);
   ```

3. Assign the return value through an extra reference parameter. The signatures for `insert` would look like:

   ```
   bool insert(const value_type& x);
   bool insert(const value_type& x, mapped_type& old);
   ```

   where the `bool` indicates whether the insertion succeeded, and `old` (if supplied) is assigned the old value if the key was already present.

4. Return the result of applying a user-defined function to the value. The interface might look like:

   ```
   void insert(const value_type& x);
   template<class F> auto insert(const value_type& x, F f) ->
      decltype(f((const value_type*)()));
   ```

   If a matching key was already present, the second method would return `f(&`*old*`)`. Otherwise it would return `f((const value_type*)nullptr)`.

5. Return a count of how many old values were present, and apply a functor to those old values. The interface might look like:

   ```
   size_t insert(const value_type& x);
   template<class F> size_t insert(const value_type& x, F f);
   ```

   Functor `f` is applied to each value found, which is at most one for a `concurrent_unordered_map`, but possibly more for the "multi" variant.

Approaches 1 and 2, which return `optional<mapped_type>`, are syntactically nice and enable users to exploit shorthand such as `optional<T>::value_or`. But the need for a dummy parameter seems slightly awkward, and the interface requires that the values be copy-constructible.

Approach 3, with the reference parameter, offers another way to indicate whether the old value is of interest. On the other hand it imposes an additional requirement: assignability, and in general the standard library avoids returning values this way. Furthermore, extending it to the "multi" variant incurs another complication: the reference would need to be a container of some kind. If the container were generic, what would its signature requirements be?

Approach 4, with a functor, avoids the need for values to be copy-constructible. The functor could extract the minimum necessary information from it. The reason for passing an address, and not a reference, to `f` is to be able to use `nullptr` to indicate a "not-present" situation. The principle drawback of this approach is that it adds additional complexity to an otherwise simple signature. The complexity might be mitigated by providing a standard functor that creates an `optional<T>` from a `const T*`, so that the caller could write something like the following to insert an item into a `concurrent_unordered_map m` and get its previous value:

```
optional<mapped_type> v = m.insert( key, optional_from_pointer );
```

Extending the functor approach to a `concurrent_unordered_multimap` introduces the complication that the functor must operate on a sequence.

Approach 5 uses the functor in a slightly different way, so that it extends cleanly to `concurrent_unordered_multimap`. But it makes simple cases prolix. For example, to insert an item and get a copy of the old value would require writing something like:

```
optional<decltype(m)::mapped_type> old;
v = m.insert( key, [&](const decltype(m)::value_type& x){old=x;} );
```

The fundamental problem may be that any interface that supports the "multi" case is going to be inherently more complicated,

Assuming that moves are cheap, the return-value issue is mostly a problem for methods `emplace` and `insert`. Methods `exchange` and `erase` can simply move the old item into their return value. Method `count` can be used in place of `find` when the value is not of interest.

3.7 Exception Safety

If an operation throws an exception, the container remains in a consistent state. Ideally the state remains unchanged, but if the exception is thrown while copying the `optional<T>` return result, it may happen after the container was updated, and in a concurrent environment it is unsafe to undo the update since other threads may have already seen the effect of the update.

3.8 Redundant Construction

The required semantics are intended to allow both blocking and non-blocking implementations. In particular, an object may be constructed and destroyed an arbitrary number of times, even for a single insertion.

4 Synopsis of Concrete Interface

For sake of focusing discussion, the concrete interface here omits creature comforts such as an `insert` that takes a sequence.

```
namespace std {
    template< typename Key,
              typename T,
              typename Hash = hash<Key>,
              typename Pred = equal_to<Key>,
              typename Allocator = allocator<pair<const Key, T> >,
              typename Reclaimer = reclaimer
    >
    class concurrent_unordered_value_map{
    public:
        // types:
        typedef Key                      key_type;
        typedef std::pair<const Key, T> value_type;
        typedef T                        mapped_type;
        typedef Hash                     hasher;
        typedef Pred                     key_equal;
        typedef Allocator                allocator_type;
        typedef typename allocator_traits<Allocator>::pointer
            pointer;
        typedef typename allocator_traits<Allocator>::const_pointer
            const_pointer;
        typedef value_type&              reference;
        typedef const value_type&        const_reference;
        typedef implementation-defined size_type;
        typedef implementation-defined difference_type;

        // construct/copy/destroy:
        concurrent_unordered_value_map();
        concurrent_unordered_value_map(const
           concurrent_unordered_value_map&);
        concurrent_unordered_value_map& operator=(const
           concurrent_unordered_value_map&) = delete;
        ~concurrent_unordered_value_map();

        // capacity:
        bool empty() const noexcept;
        size_type size() const noexcept;
        size_type max_size() const noexcept;

        // modifiers:
        template<class...Args>
           optional<mapped_type> emplace(Args&&...args);

        optional<mapped_type> insert(const value_type& obj);
        template<class P> optional<mapped_type> insert(P&& obj);

        optional<mapped_type> exchange(const value_type& obj);
        template<class P> optional<mapped_type> exchange(P&& obj);

        optional<mapped_type> erase(const Key& k);

        template<class F>
        optional<mapped_type> reduce(const value_type& obj, F f=std
           ::plus<mapped_type>());

        void clear() noexcept;

        // observers:
        optional<mapped_type> find(const key_type& k) const;
        size_type count(const key_type& x) const;
        template<class Function> void for_each(Function f) const;
        static bool is_lock_free() noexcept;
    };
}
```

5 Comparison with the Reference-Based Approach

The grass often looks greener on the other side. Now that this paper has surveyed the other pasture, let's close with a comparison to the reference-based approach in N3425. The reference-based approach avoids three issues completely:

- The "how to return a value" issue discussed in Section 3.6.
- Needing to define a set of allowed atomic actions.
- The need for an ad hoc `for_each` method. The reference-based approach can use the usual `std::for_each` and other algorithm templates.

On the other hand, the reference-based approach allows accidental use of non-atomic actions on values.

As far as implementation, efficient implementation of the reference-based interface seems limited to split-order lists. The value-based interface would seem to allow many more approaches, such more traditional buckets, linear probing, or hopscotch hashing.

6 Acknowledgments

Artur Laksberg's comments on an earlier draft motivated some better explanations. Alexey Kukanov pointed out how extensions to multimaps and "How to return a value" were more tightly coupled than we first realized.

References

[1] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006.

[2] Alex Stivala, Peter J. Stuckey, Maria Garcia de la Banda, Manuel Hermenegildo, and Anthony Wirth. Lock-free parallel dynamic programming. *J. Parallel Distrib. Comput.*, 70(8):839–848, August 2010.

[3] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[4] Gotw #6a solution: Const-correctness, part 1. `http://herbsutter.com/2013/05/24/gotw-6a-const-correctness-part-1-3/`. Accessed 2013-08-23.