Document number: P1761R0
Date: 2019-06-16 (pre-Cologne)
Reply-to: David Goldblatt <davidtgoldblatt@gmail.com>
Audience: SG1

P1761R0 Concurrent map customization options (SG1 version)

Background

In the discussions of P0652 and P1470, SG1 took the following polls:

"We believe it is important to permit implementations that do not protect values from concurrent visits. (note: P0652 does protect values from concurrent visits)"
SF F N A SA
6 3 1 3 0

"We would like to see a policy-based design that allows customization of concurrent map (e.g., whether deletion is permitted)."
SF F N A SA
3 5 4 0 2

"We believe it is important that the concurrent map permits concurrent reads without contention."
Unanimous consent.

The room asked for:

- An SG1-focused paper on the design space of hashmap parameterization.
- An LEWG-focused paper, to determine to what extent type parameterization would ever make it out of LEWG. (There was concern that they would not be willing to advance any such parameterized data structures for library-design reasons).

This is the first of these papers.

A note on terminology: What counts as a read or write is to some extent context-dependent. We need to be clear on if we're talking about the structure of the map itself (i.e. inserting or removing an element, or finding the address of an element), or modifying an element that we've already looked up. A user with a map whose key set rarely changes may wish to modify the associated values often. Likewise, another user may have a frequently-changing key set, but never want to modify the associated values (say, because they are computed as a pure function of the associated key). I've tried to pick examples where it's clear whether terms like "read-mostly" or "write-mostly" refer to the structure of the map or its contents. By untying these notions from one another, concurrent map

implementations can substantially improve their performance in many real world scenarios; this fact is what drove the poll results above.

Example use cases

Having a few representative examples will concretize the problem domain, motivate the policies I'll suggest below, and, hopefully, let the reader come up with closer-to-home examples of their own. These will necessarily reflect my own biases, and tend towards the issues faced by highly-threaded, large-footprint, request-oriented server workloads. With each example, I'll try to point out a few features of a hypothetical concurrent hash-map that the use case does or doesn't care about. I'll address these again after going over the performance tradeoffs of various types of synchronization a concurrent map could provide; it may be useful to keep these examples in mind when reading the "feature set / performance" section.

Stats counting

A webserver wants to keep track of various event counts (how many requests timed out, how many used some uncommon code path, etc.). The key is a string (to allow counters determined at runtime; but regard it as an integer or pointer if you prefer) and the value is either an integer (for true event counts) or some more complex data structure (like a histogram). At some periodic interval (say, once a minute), a background thread iterates over all items in the map and sends them to a remote server for aggregation.

Some interesting things to note about this case:

- It's very heavily write-mostly, insofar as the values themselves are concerned
- It's very heavily read-mostly, insofar as the structure of the map goes (new keys are rarely added)
- Keys are never removed; the size of the map can only grow

Configuration information

A server needs to know a variety of data that can change at runtime. Some examples:

- Which feature flags are currently enabled? (Often times, dangerous code changes will be guarded by such a flag; first the feature is turned on for 1% of server processes, then 5%, then 10%, etc.).
- Should this server run high-overhead profiling functionality?
- What's the current value of the "spammy link" bloom filter?

The values of these settings change rarely; they may be updated via some diffing mechanism from a remote server. Here are some usage properties I expect this map to have:

- It's very heavily read-mostly. Periodically, though, it may go through phases during which it's heavily write-mostly.

- We expect a very strong hot-key effect, in which a small number of keys (say, those controlling currently turned on features, touched on every request) are read very frequently, and a large number are touched rarely.

Value cache

This one is straightforward: you have a computation-intensive pure function called from many threads and want to memoize the results. This ends up being a lot like the configuration information use case; heavily read-mostly, with hot keys. (Well, we hope it's true; otherwise caching won't be effective).

Proxy state machine

A proxy forwards bytes from clients to servers, providing services like rate limiting, encryption, etc. I/O threads access items in the map in order to route bytes appropriately and perform their logic. The logic within the state machine might be complex, keep track of lots of state, etc.; it's not easily amenable to parallel access. Removal may be common (say, when a client disconnects).

The bank account example

This is the naive deadlock example commonly used in "Intro to operating systems" classes. The map is from bank account numbers to information about those accounts. Balance queries (from a single account) and transfers (between accounts) are processed by accessing the map.

On its face, this sounds unrealistic; you couldn't really keep this information in memory, for a variety of account safety and regulatory compliance issues (maybe you're a bitcoin bank?). But variants of this can turn out less silly: suppose a process is tracking a binding from user sessions to the backend servers that respond to queries for those sessions (as a performance optimization; we're trying to make those bindings "sticky"). We have two maps, then; one from user -> session, and one from backend server -> {set of users}. During rebalancing operations (where a session is moved from an overloaded backend to less-loaded one), we may want to modify the mappings for the user and both backends all at once.

Feature set / performance tradeoffs

Here, we'll take a look at some of the performance consequences of extra synchronization guarantees.

The features

Synchronized visitation / map-provided locking

Here, the map provides safety not just for the structure of the map itself, but also for the access to its elements; write visitors get exclusive access, and read visitors get shared (among readers) access.

There are two reasonable ways to accomplish this:

1. Keep some per-bucket/per-shard/per-item atomic in the hashmap. Readers do a RMW operation on that object, and writers read it.
2. For each bucket / shard / map, keep a distributed (across threads, CPUs, NUMA nodes, etc.) set of atomics. Readers do a RMW (or just a release store, if they can guarantee exclusivity) on their atomic, and writers do a read on all the atomics.

(Transactional memory could theoretically be a third way, but the practical limitations are severe and non-portable here).

This is a costly guarantee to provide:

- Option 1 introduces reader-reader contention: On typical hardware, each reading thread has to acquire exclusive access to the cacheline containing the atomic. This serializes that phase of reader computation. Most maps have a “hot key” effect; a small fraction of keys are accessed much more commonly than most in the map. As a result, reader access is serialized by the coherence protocol.
- Option 2 forces one of a number of tradeoffs, none of them particularly desirable. Any decrease in reader-reader contention is met by a corresponding increase in the amount of work a writer has to perform (i.e. if you have a per-CPU atomic to minimize cache contention, the writer has to do O(numCPUs) reads of those counters per write). Getting reader->atomic sharding right is a tricky, research-level problem (e.g. mapping two threads running on CPUs on two different sockets to the same atomic is a pretty bad outcome on its own).

Aside from the cacheline contention, there’s the blocking incurred by excluding other accesses; a reader cannot proceed simultaneously with a writer, and vice versa; a long critical section can cause arbitrary-length stalls. This applies even in situations where this would be correct (say, the value is const, or otherwise thread-safe, or where the reader and writer access different fields). This is bad for CPU workloads, but worse for GPU ones.

Removal / replacement support (with synchronous destruction)

(I.e. when an element is removed from the map, the destructors of its key and value types run before the erase() call returns).

The lightest-weight way of accomplishing this is reference counting, or something very similar to it; readers or modifiers bump a counter when they look up an item, and decrement it once they're done with the item (if the map otherwise requires locks to support visitation, holding that lock suffices).

This has the same sorts of contention issues as visitation; it incurs reader overhead (even if that removal functionality is never used). It has a benefit, though: it avoids the blocking issues. On hardware with relaxed memory models, or workloads that can tolerate long lookup latencies, the cost of contention may be hidden somewhat.

It may seem odd that we're lumping together removal and replacement; they're not equivalent operations. However, the implementations end up looking pretty similar, and have similar tradeoffs.

Removal / replacement support (without synchronous destruction)

(I.e. when an element is removed from the map, the destructors of its key and value types run with the same sorts of guarantees as with hazard pointers or RCU).

This can be accomplished without introducing reader-reader contention, via the deferred reclamation techniques going into the concurrency TS (or any other latest<> backend). However, it does force either a heap-allocated, node-based implementation (which adds a pointer indirection down all paths), or tombstoning (which adds implementation complexity, and can introduce tricky pathologies if not done correctly).

Multiple element access

I.e. can one thread access two map entries at once? Or items in two different maps?

To support this, the implementation:

- Can't hold locks while the user accesses an item (to avoid deadlock)
- Must protect the lifetime of the key and value during their access (via a guard object or similar mechanism at an API level).

The amount of overhead this imposes on other operations depends heavily on the underlying implementation; if it supports visitation, we need an extra counter or set of locks (splitting object protection from its lifetime protection). On implementations that natively only provide lifetime protection, accessing multiple elements simultaneously usually falls out for free.

Exotic synchronization requirements

Say a reader thread's lookup of key "abc" fails, and a writer thread's insertion of key "abc" succeeds. Does the read failure synchronize with the write? What if the read succeeds, but sees

an older value? Which IRIW outcomes are possible if performed on map elements instead of atomics? Do we get a happens-before relationship there?

It's hard to provide those guarantees in a contention-free manner in the C++ memory model. Most hardware supports the SC semantics at the cost of a heavy fence on the read side (but, avoiding any contended accesses); the common exceptions are Itanium and (I'm less confident on this one) Nvidia.

It's hard to come up with a realistic example where this matters.

Comparing the features to other synchronization primitives

Various synchronization objects can be implemented in terms of a concurrent map, by making the key in the map the address of the synchronization object (just using the address as some unique identifier). It can be useful to think of the functionality in terms of these primitives; they are upper bounds on the performance of the map.

Reader/writer locks: Synchronized visitation

Simply run the critical section during the visitation.

atomic_shared_ptr<T>: Replacement with synchronous destruction

Make the value type shared_ptr<T>.

latest<T>: Replacement without synchronous destruction

Make the value type T. Access the T object using the lifetime protection provided by the map.

This breakdown roughly matches up with the analysis from the previous section. Reader/writer locks tend to scale poorly even for read-mostly workloads, and correspondingly we argued that a map providing synchronized visitation would likely do the same. atomic_shared_ptr<T> tends to be resilient to the latency spikes a long critical section might cause, but at the cost of high constant factors. latest<T> is fast, but at the cost of implementation complexity.

Use cases revisted

We can see a pattern: the more expensive the synchronization a map might provide is, the less commonly we need it. This analysis drove the SG1 polls at the beginning.

Stats counting

In the simple case of raw event counts, locking is extraneous; instead of using a lock to protect incrementing an integer, we should just make the integer atomic. For complex stats types (e.g. dynamically-bucketed histograms, time-series data), the externally-provided locking may be

helpful. Even there though: Facebook's experience has been that we often want some clever internally-synchronized data structure for our hot stats types (see e.g. folly's BufferedDigest classes, which compute quantile estimation and time series via CPU-local aggregation that is aggregated only periodically). Removal isn't a feature here; paying extra to provide it would be undesirable.

Configuration information

Here, the data in question is heavily read-mostly. The data is updated rarely enough that it's usually preferable to just replace the contents, rather than updating them in place. Reader-reader contention is unacceptable. Removals might be desired, but we don't need synchronous destruction of contents.

Value cache

This is another read-heavy case where the value type doesn't need any protection (indeed, they're const). Removals may be useful (say, to get an approximate bound on the size of the cache). Synchronous destruction gives earlier memory reclamation, but avoiding it allows higher performance.

Proxy state machine

This is a good use case for visitation; we don't expect much multithreaded access to individual values, and when we access a value we want our access to be truly exclusive. Minimizing the syntactic overhead of the concurrency is just what the user wants; the parallel nature of the map is not the user's primary concern, and paying extra costs to avoid having to deal with the difficulties of thread-level parallelism is acceptable to them.

The bank account example

We expect this to be read-mostly in the common cases, with occasional insertions and removals. The user must insert their own locking and deadlock avoidance into the value types, in the case where they want to perform atomic transactions across multiple accounts. This needs multiple-element access, but can be flexible in other respects.

The options

Here I'll list some possible parameterizations, and a rough sketch of their semantics. I've erred on the side of including too many; the idea being that it will be better to show too much of the possible design space than too little.

There are lots of mechanisms we could use to pass parameters to the map, but I regard the specific choice of one as an LEWG question, so I'll leave that discussion for the other paper.

You can think of them as enum values that get OR'd together, a configuration object, an executor-style policy interface, or whatever else your preferred mechanism may be.

Synchronized Visitation support

I think there are three reasonable levels of functionality one might want:

- Unsupported: The map provides no mutual exclusion on access to values
- Exclusive-only: The map provides exclusion on values equivalent to a mutex.
- Shared: The map provides exclusion on values equivalent to a reader/writer mutex.

The only reason one might want the exclusive-only mode over the shared mode is that exclusive mutexes can have lower constant-factor overheads.

Recalling the poll that attained unanimous consent: I think that any reasonable implementation that permits concurrent reads without contention cannot support synchronized visitation in any form.

Removal/Replacement support

There are four reasonable choices of strength of semantics

- Unsupported
- Supported, with asynchronous destruction of the removed element (i.e. via RCU, hazard pointers, etc.).
- Supported, with "locally synchronous" destruction of the removed element. That is to say, the completion of the destructor is sequenced before the return from erase().
- Supported, with "globally synchronous" destruction of the removed element. In this case, the completion of the destructor happens before any insertion of an item with the same key.

I think that any reasonable implementation that permits concurrent reads without contention can only provide one of the top two levels of support.

For some reason, no map I'm aware of implements the locally-synchronous guarantee (except as part of the globally synchronous guarantee); this despite the fact that:

- I can't think of any non-synthetic examples that require the extra strength of the global guarantee
- The local guarantee has some obvious tail-latency benefits for complex map types (you can move destruction out of the critical section).

Therefore, when describing existing implementations later on, I'll just use "synchronous destruction" to refer to the global guarantee.

With replacement, there's a question as to whether or not transient states become visible. If not, and the key type is not trivially copyable, then insertions and replacements require extra synchronization,

even if replacement is never actually used (lookups can remain lock-free and without contention). Making replacement support optional is another mechanism to support this.

Multiple-element access support

This requires the ability to “pin” an item, protecting it from being destroyed (or moved-from by an internal map operation). Pinning must not block pinning from other threads, for deadlock-avoidance reasons; it can’t be just a simple lock.

Ordering guarantees

Most litmus tests for atomics can straightforwardly be turned into an equally confusing ones for maps. I don’t think we need to replicate the whole memory model into the map specification, though. I see three useful levels of guarantee:

- Baseline: The insertion (resp. removal) of an element strongly happens before successful (resp. failed) lookups of that element.
- Per-item ordering: The same as the baseline guarantees, but we also provide the guarantee that lookups returning an earlier value associated with a given key strongly happen before insertions of a later one.
- Sequential consistency: The hashmap is linearizable, and its linearization can be incorporated into the SC ordering.

This omits an arguably important level of support, weaker than the baseline given here: the one where insertion is only dependency-ordered before lookup. Even the baseline guarantee here requires read-side fences on some architectures; there are real world use cases that don’t need or want this overhead. Given the uncertainty around the future direction of memory_order_consume, though, expanding its scope seems unwise.

Existing maps and their feature sets

It may be useful to see the portions of the design space that different approaches have picked. Here I’ll describe some. (In many cases, these guarantees aren’t formalized or documented; in such cases I’m giving my best guess best on documentation and the implementation, where available).

The P0652R2 API

Synchronized visitation support: Shared
Removal/replacement destruction semantics: Supported, with synchronous destruction
Replacement support: Supported, with globally synchronous destruction.
Multiple-element access support: No
Ordering guarantees: Sequential consistency

Of note is that it does not support unsynchronized visitation.

Intel TBB's concurrent_hash_map

This is broadly the same in its guarantees as P0652. It's my understanding that this was the inspiration for P0652.

Intel TBB's concurrent_unordered_map

Synchronized visitation support: Unsupported
Removal/replacement destruction semantics: Unsupported
Replacement support: No
Multiple-element access support: Yes
Ordering guarantees: Baseline

Libcds Michael map and Feldman Map

Synchronized visitation support: Unsupported
Removal/replacement destruction semantics: Supported, with asynchronous destruction
Replacement support: Yes
Multiple-element access support: Yes (not nicely at the API level, but the implementation does)
Ordering guarantees: Baseline

Libcds Striped map and Cuckoo map

Synchronized visitation support: Exclusive-only
Removal/replacement destruction semantics: Supported, with synchronous destruction
Replacement support: Yes
Multiple-element access support: No
Ordering guarantees: Sequential consistency

folly::ConcurrentHashMap

Synchronized visitation support: Unsupported
Removal/replacement destruction semantics: Supported, with asynchronous destruction
Replacement support: Yes
Multiple-element access support: Yes
Ordering guarantees: Baseline

Kokko::UnorderedMap

Synchronized visitation support: Unsupported
Removal/replacement destruction semantics: Erase is done in batch (the host calls begin_erase(), then the device makes some number of erase() calls in parallel, then the host calls end_erase()); I think this is morally closest to "Supported, with asynchronous destruction"

Replacement support: Yes
Multiple-element access support: Yes
Ordering guarantees: Baseline

Note: This map allows for inserts to fail, and moves the “rehash-on-failure” semantics to an outer loop (for better batching). I don’t think this is fundamental, but would require an extra layer of indirection.

java.util.concurrent.ConcurrentHashMap

Synchronized visitation support: Unsupported
Removal/replacement destruction semantics: Supported, with asynchronous destruction (though, note that this is less meaningful for Java).
Replacement support: Yes
Multiple-element access support: Yes
Ordering guarantees: Baseline

ConcurrencyKit’s ck_ht_t

Synchronized visitation support: Unsupported (readers are lock-free and the writer is wait-free; but see below)
Removal/replacement destruction semantics: Supported, (this is C, so the destructor aspect makes less sense; the map does tombstoning with the option for periodic user-initiated GC).
Replacement support: Yes
Multiple-element access support: No
Ordering guarantees: Weaker than baseline, but stronger than the consume-based ordering described above (there’s a load-load barrier between the metadata check and the key check).

This map only supports single-producer multi-consumer workloads. To support writer-writer concurrency, the user would need to provide their own sharding and mutual exclusion (e.g. by having many sub-maps).

Some implementation sketches

A couple people in SG1 worried that we would be requiring vendors to write a new implementation for every setting combination; exponentially many in total. Here, we’ll see that need not be the case.

There are two reasons why:

- Most of the options can be ordered by strength, and implementing the stronger guarantees is a correct implementation of the weaker guarantees, too. (Implementing the stronger guarantees often turns out to be easier, too). Implementers can just provide the strongest implementation, and enable_if away any methods the options don’t ask for.

- Most options can be straightforwardly provided by using shared functionality that's generic across implementations. The core map implementation doesn't need to worry about how that functionality is provided. For example, synchronized visitation can be easily added as a mixin to any map: add a sharded lock table.

I'll sketch how a vendor might support all the listed options, first with a simple lock-based approach, and then gradually adding a second core implementation that grows in the functionality that it can provide via mixins.

The high-overhead implementation

Here, the implementer wants to minimize the amount of code they have to write; they're OK forcing locking and contention on all workloads to accomplish that goal.

This vendor can use a sharded lock table (with reader/writer locks), and a "pin" count per shard. To provide synchronized visitation, just acquire the corresponding lock. All the core hashmap operations are easy to provide (the whole point of locking is that it makes things easy!). To implement multiple-element access, bump the pin count and return a handle to the object (removers have to mark the item as logically removed, and wait for its pin count to drop to zero before executing the destructor).

If multiple-element-access support is not desired, simply omit pin counts from the lock table.

A two-underlying-maps implementation

Here, the implementer is willing to do a little more work in order to accommodate the desire for a higher-performance implementation.

They can add a second class to their detail namespace: a lock-free map (with all their atomic operations tagged memory_order_seq_cst) without any removal support. The public API chooses between each implementation in detail:: as follows:

- If the user requests removal or synchronized visitation, use the previous implementation
- Otherwise, use the lock-free one.

Adding parametric ordering

Here, the implementer is willing to do the two-maps approach, but also think about relaxed memory orders.

Instead of passing in memory_order_seq_cst on all their atomic operations, they define constants like the following:

```
constexpr static memory_order kLoookupMemoryOrder = (kBaselineOrdering ?
memory_order_acquire : memory_order_seq_cst);
constexpr static memory_order kInsertMemoryOrder = (kBaselineOrdering ?
memory_order_release : memory_order_seq_cst);
```

They replace the memory orders they pass with these constants.

A two-underlying-maps implementation, with parametric ordering and removal

Most lock-free maps can be made to support removal; the tricky part is the garbage collection. But, there are two deferred reclamation libraries targeting the same launch vehicle as the map proposal; that functionality comes for free.

To support removal, then, the map implements the algorithm, but hides the specific deferral mechanism behind something like the latest<T> API. If the user doesn't request removal, then the implementation is a no-op for all operations. If the user requests asynchronous removal, then the implementation uses hazard pointers or RCU as its backing mechanism. If the user requests synchronous removal, the implementation uses reference counting.

The complexity of the parametricity is "boxed up". The core parts of the map algorithm don't need to think about it at all, and the support for an option can be made small and self-contained.

Unanswered questions

Mostly I regard the big questions as API ones here, and arguably better suited for LEWG:

- What is the specific parameterization mechanism?
- What does the unsynchronized access API look like?
- etc.

There are also some more directly SG1-y ones:

- Should expensive functionality be opt-in or opt-out (i.e. should the default map be fast-but-slim or slow-but-featureful)?
- What does "same key" even mean in the presence of concurrency? The definitions of key equivalence used for std::unordered_map don't work here, since they fundamentally require the notion of a map with a set of keys that doesn't change within a call. You could phrase your guarantees in terms of the hash value itself (which, being trivially copyable and memcp-comparable, is simpler), but that feels wrong and rules out some types of perfectly reasonable implementation.

I have opinions on some of these, but none particularly polished.