Document number: P1762R0
Date: 2019-06-16 (pre-Cologne)
Reply-to: David Goldblatt <davidtgoldblatt@gmail.com>
Audience: LEWG

P1762R0 Concurrent map customization options (LEWG 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 second of these papers.

The issue

Different underlying concurrent hash map implementations support different workloads with different levels of efficiency. Supporting some features (say, removal of an element) comes with high overheads, even when those features are never used. Depending on the feature set required, fundamentally different designs are called for.

Possible questions, and responses

The parameterization we're suggesting here is more flexible than most data structures; I think only the executors proposals want a more flexible mechanism. Framing the explanation as a conversation will make clearer the flow of reasoning.

What does splitting up the feature set get you?

By getting a guarantee that certain types of operations are impossible, or that certain types of memory reordering are acceptable, the implementation can avoid costly protection mechanisms (you don't need to guard against lifetime issues if the map is never removed from) or memory fences.

Isn't this true of non-concurrent data structures too? std::unordered_map has an API that makes you pay for bucket functionality and iterator-stability (which force node-based implementations) even though they're rarely used

One difference is that the tradeoffs are more severe. The singled-threaded differences between node-based and open-addressing implementations are constant-factor ones. Some of the concurrent APIs under consideration result in changes to big-O complexity. Operations go from O(1) to O(numSimultaneousAccessors). For single-threaded cases, the differences from changing APIs might be in the hundreds of nanoseconds. For concurrent ones, they can range into the tens of thousands.

Another difference is the difficulty of working around the performance loss. In single-threaded use cases, it's usually easy and safe to drop in an API-compatible replacement. If you don't like the performance of std::unordered_map, you can switch your type to absl::node_hash_map. If it compiles, it probably works; you don't need to examine all surrounding code to make sure. (In some changes, there are pointer-stability issues, but those can be checked fairly easily with extra debug-mode checks.)
By contrast, in the concurrent setting, it's very difficult to tell whether or not replacing the standard concurrent map with one that provides looser memory ordering guarantees is correct, short of manually examining all the surrounding code. If you're lucky, you might catch it in a TSAN run. Even then, mapping the race back to the missing ordering provided by the map is nontrivial.

Lastly: maybe the decision for unordered_map was wrong too. At least twice (once in LEWGI, once in SG1), I've heard it expressed to general assent that the unordered_map interface was unfortunate because of the performance limitations it imposes.

But won't the vendor workload here be exponentially large? If we have 5 binary options, do we need 32 different implementations?

No. The "strength" of each option tends to be nested. A vendor can, if they wish, just implement the strongest possible semantics in a single, easy-to-write implementation. Those vendors that want to support higher-performance situations, though, can write a few different map implementations in a detail namespace. The exposed interface then just picks between them via template specialization. Having 2 or 3 implementations can give good performance across a wide portion of the design space, across a wide range of target architectures.

Why not have multiple types? Isn't that what we planned for concurrent queues?

Recall: rather than trying to unify both blocking and nonblocking queues into a single framework, LEWG and SG1 decided to only advance the blocking variant at first.

I think the key difference is the number of choices for customization points. Concurrent queues have just a few reasonable customization points at the API level; blocking and non-blocking, and bounded and non-bounded. Moreover, on popular queue implementations, supporting the blocking variants of operations can be accomplished without overhead down other pathways (e.g. the first step in a blocking push operation on a bounded queue is typically the same as a non-blocking try_push() on a bounded queue; supporting the blocking version doesn't substantially slow down the non-blocking one).
By contrast, the design space for hashmaps is more genuinely multidimensional.

- Lifetime: The only access to queue contents are via pushes and pops, which imply lifetimes straightforwardly. Maps, however, need answers to questions like "what happens if one thread tries to remove an item while another thread is reading it? When does its destructor run?".
- Memory ordering: Concurrent maps need answers to questions about ordering guarantees between operations on distinct elements in the map. Every queue operation affects the queue in a global sense.
- Scope creep: More people want more things from maps (some people want to piggyback on any internal locking the map might have; others can't tolerate that locking as a correctness issue). We ask more of maps, so they get a bigger API. Each part of that API provides the opportunity for different levels of strength.

Wait a minute; earlier, you said that vendors would only need 2 or 3 implementations to get good performance. But then, you said that there were too many options to capture the whole design space in just a few types. What gives?

One of the ways vendors can cut down on implementations is by pulling out shared functionality into separate internal classes. For example, one suggested option lets the user ask for mutual exclusion while they access on an element. This can be provided by a sharded lock table, keyed by the hash. That lock table implementation works for *any* underlying map implementation. We only want to include it, though, if the user requests it. So even with only 2 or 3 implementations, we still want the user to be able to pass options to those implementations.

A more fundamental issue is that the right way to split up the design space may in general be architecture-dependent. On a CPU implementation, a vendor may decide that the most important implementation concern is “workloads with lots of cache contention vs. workloads without lots of cache contention”, and so write one implementation that carves out the portion of the design space that allows readers to proceed without contending with other readers. On a GPU, the right split might be “workloads that can block” and “workloads that never block”. There’s no reason why a single correct split must exist.

Anything else?

The question of how to support template tuning parameters more effectively is something LEWG could stand to look at in a general sense. std::unordered_map (with no concurrency) has 3 defaulted template parameters at the end; changing its allocator requires writing out a type with 5 template arguments. In non-standardized code, it’s not uncommon to see even more in performance-sensitive code. A group of library people thinking about best practices for this sort of problem would be good, even if LEWG hates it for this particular use case.

Possible parameterization mechanisms

So far, we’ve dealt in abstracts. Here, I’ll suggest some possible concrete mechanisms that could let users customize library functionality. I have my own opinions on their relative merits, but I mostly regard the whole question as one for LEWG to decide.

Enum parameters

The std namespace defines some enums that can be OR’d together and passed as a template parameter. To get a concurrent map type that supports removal and synchronized element access, the user might write:

```
using MyMapType = std::concurrent_map<
    MyKey,
    MyVal,
    std::hash<MyKey>,
    std::equal_to<MyKey>,
```

```
    std::allocator<std::pair<const MyKey, MyVal>,
    std::allow_removal | std::synchronized_visitation>;
```

Non-type template parameter

Here, namespace std would define a constexpr inline object. That object would have member functions that return an object with that option set as desired (the way that some types have an “Options” struct passed to their constructor. The code above would then be:

```
using MyMapType = std::concurrent_map<
    MyKey,
    MyVal,
    std::concurrent_map_options
        .allow_removal()
        .synchronized_visitation()>;
```

P0443-style properties

```
using MyMapType = decltype(
    std::require(
        std::declval<std::concurrent_map<MyKey, MyVal>>(),
        std::allow_removal,
        std::synchronized_visitation));
```

Something else

I’ve seen other mechanisms for this functionality (e.g. Boost’s named parameter library, or the mechanism used in its multi-index container library), but these are fairly syntactically heavyweight; I’m not sure that they would be designed the same way if written with the benefit of recent C++ versions.