Document number: P1470R0
Date: 2018-01-20 (pre-Kona)
Reply-to: David Goldblatt <davidtgoldblatt@gmail.com>
Audience: SG1

P1470R0: Against a standard concurrent hashmap

Abstract

We should not standardize a concurrent hash map (or most other concurrent equivalents of single-threaded data structures). If we do anyways, the versions standardized should at least have highly parameterizable functionality. (I'll occasionally refer to P0652 and hashmaps for specificity, but most of the arguments apply generally.)

The high level argument is:

- The benefits to standardizing, relative to open-sourcing, are small.
- The API we pick will likely either omit functionality that's important in some key use cases, or impose performance costs that are unacceptable in some key use cases.
- Even if the API we pick allows a performant implementation, structural challenges facing implementers mean that it's unlikely that we'll get one in practice.

If the first and third bullet points aren't persuasive, I suggest we attempt to mitigate the second one; we should parameterize the functionality of the data structures we standardize (this is also an approach suggested in P0387, Memory Model Issues for Concurrent Data Structures). Users could indicate which portions of an API they need, allowing implementers to specialize the implementation to omit costly functionality where required.

Why standardize anything?

To quote the LEWG chair[1]:
"So, what should go in the standard library? Fundamentals. Things that can only be done with compiler support, or compile-time things that are so subtle that only the standard should be trusted to get it right. Vocabulary. Time, vector, string. Concurrency. Mutex, future, executors. The wide array of "sometimes useful data structure" and "random metaprogramming" could and should be shuffled off into a semi-standard repository of available but distinct utility libraries. Those may themselves have different design and stability philosophies, like Boost or Abseil -

[1] https://abseil.io/blog/20180227-what-should-go-stdlib

the principles that make the standard what it is are not necessarily universal even if they are good for the standard.”

Concurrent maps, queues, counters, etc. are much closer to “sometimes useful data structures” than they are to core, can’t-live-without-it functionality like mutexes and futures. To try to expand on this a little, I’ll unscientifically and non-exhaustively categorize various types of things that get (or could get) standardized.

Hooks into the core language or runtime

Things like initiailizer_list and the contents of type_traits allow access to compiler magic; it’s the interface to a syntactic feature that makes C++ easier to use somehow. Things like terminate() and current_exception() access magical parts of the runtime.

These should be included because their functionality is important to end-users, and wouldn’t be accessible to end-users otherwise.

Vocabulary

Some abstractions are so important and universal that libraries written by different organizations need to agree on their names and functionality. This category includes overload points (e.g. hashing, swap()), most single-threaded containers (e.g. vector, string, map, unordered_map), implementation concerns that can be pulled out of algorithms and data structures (e.g. executors, allocators), patterns with common and conceptually simple implementations (e.g. smart pointers, tuple, optional, variant), concepts (e.g. iterators, ranges, Lockable), and some parts of the chrono library (e.g. durations).

These should be included because they enable library composition. Lots of libraries need some notion of a dynamic array with amortized constant time appends. But if we don’t pick a single name for it, user code won’t be able to take the result of one such library and pass it to another.

Portability

Sometimes important functionality is exposed by a wide variety of different implementations in a conceptually similar way that results in API differences. Standardizing these abstractions lets users write code without worrying about platform specifics. Examples of this include low-level I/O facilities, concurrency and parallelism primitives (threads, atomics, mutexes, low-level primitives for blocking and unblocking), filesystem support, networking, and some other parts of the chrono library (e.g. clocks).

These should be included, since portable abstractions for core functionality allow code built on top of those abstractions to be portable too. This reduces platform switching costs for users, and minimizes pain points for open-source libraries.

Simple APIs for simple, common problems

Basic CS primitives fall into this bucket; sorting, binary searching, filtering, and much of the rest of the algorithm header fall into this bucket. Often times, the vocabulary concepts are designed to allow some set of abstractions to compose straightforwardly, and including some implementations of those abstractions provides a base to build on top of. Much of the range views, iterator adaptors, and things like std::array fall into this bucket.

These allow a nice “out of the box” experience for the language. It’s irritating and unfortunate if students or newcomers need to find and install a set of utility libraries just to do basic CS 101 tasks. Libraries users depend on would have to either reimplement these basics themselves, or introduce a new dependency on some common utility library.

“Batteries included” APIs

These APIs could be large libraries in and of themselves, directed at a particularly knotty problem. Often times their complexity is caused by complexity that exists in the world, and must be designed with a depth of domain knowledge to solve. Sometimes they have to take an opinionated view of the problem, and cause friction for use cases not philosophically aligned with their design.

In this category I include things like the random number generators and distributions, calendar manipulation libraries, string formatting and regular expression libraries, and localization.

This is the riskiest category of things to standardize; there are hard-to-reason-about tradeoffs involved. If we get it wrong, will users make mistakes? How important is the problem? What are the user’s best alternatives? Do we have sufficient domain expertise? Indeed, many of the APIs commonly considered mistakes fall into this category. I’m thinking of locales, regexes, and iostreams manipulators, which many regard as poorly performing, hard to use, and, at least in regex’s case, buggy.

I claim that most concurrent data structures also fall into this category, and have unsatisfying answers to most of the tradeoff questions.

Concerns with concurrent data structures, generally

Here I'll argue that there are structural factors that limit the utility of standardized concurrent data structures. The argument is broadly: the only reason users would want to reach for such a data structure is higher performance (otherwise, a non-concurrent version wrapped by a mutex would suffice). But, in situations where users do care about performance, it’s unlikely the the

standardized structure will be what they want (it's necessarily too one-size-fits-all, and standard library vendors are not well-situated to produce the best implementation of a given API).

Some of these arguments apply to some extent to non-concurrent data structures too. There are a couple of differences, though:

- Non-concurrent data structures are more likely to be vocabulary types
- Concurrency multiplies existing tradeoffs in addition to adding new ones. At each layer of abstraction in the design of a single-threaded data structure, there are often many different ways to make that layer concurrent (for example; whether or not to heap-allocate nodes is a question for single-threaded hash maps. Concurrent hash maps also have that question, but also need to decide on the memory reclamation semantics for those nodes, whether or not their linkage is lock-free, and how to re-link on re-hash).

Practical performance requires iterating on design

Or, "no battle plan survives contact with the enemy". Synthetic data structure benchmarks are useful to a point, but any performance engineer has a long list of war stories where their code fell over because of some usage pattern they didn't anticipate. Here are some possibilities:

- How sharded should lock tables be?
- Are the operations "random"?
    - Are reads and writes evenly distributed in time? Or are there "bursts" of writes followed by long periods of read-mostly behavior.
    - For associative data structures, is the access pattern uniform? Or are some keys hotter than others?
    - If some keys are hotter than others, is the distribution for reads and writes similar? (I.e. are keys that are read most frequently also likely to be the keys that are written most frequently?)
- How much do we care about non-concurrent or small use cases? Sometimes, a container that *may* be concurrently accessed ends up being touched only by a single thread in most program executions. Likewise, although containers usually support arbitrarily many elements, many instances end up containing only a few.

(Variants of this some of these items will recur; here we're looking at these issues abstractly; next, we'll look at them in the context of hash tables; lastly, we'll see how they affect end users picking API parameters).

Producing an implementation that balances all these tradeoffs is hard to do ex nihilo. The path to a high-quality outcomes usually goes like the following:

- Some domain experts write an initial version that performs well on microbenchmarks.
- They deploy the implementation to a small number of hand-picked use cases; this is the first production use.

- They iteratively expand the uses of the implementation to more and more cases. As the data structure becomes more generally used, its implementation becomes less tuned to its initial uses, and the implementers get a better global view of use cases.
- They declare the data structure “ready”; begin encouraging it as the default tool to reach for in the relevant use cases.
- Periodically, use cases will pop up in which the chosen design exhibits pathologically bad behavior. The implementers deploy mitigations or redesign to remain resilient in the face of these pathologies.

It’s hard to iterate on design in standardized data structures

Once a data structure ships to end-users, changing it often needs an ABI break. Design iterations therefore can happen only infrequently and with high cost. Even when a change occurs, many users don’t update right away. There may be a long time in between when a change ships and when implementers receive feedback on it.

Iterating before shipping is not always feasible; delaying the release of a data structure means delaying conformance claims. These delays have a cost to end-users, who need to keep track of which features are available on which compiler versions. Moreover, implementation vendors don’t necessarily have access to a wide range of profiling data and canary tooling obtained from production use cases.

The TS process is a partial mitigation to some of these concerns, but not all of them; it perhaps allows an extra one or two iteration cycles.

Open-sourcing an implementation provides most of the benefits

Most of the arguments listed above (for standardizing something) don’t apply to concurrent data structures. They’re fairly obviously not hooks into the core language or a simple API for a common problem. There are occasionally portability arguments for standardizing some data structures (e.g. a lock-free hashmap can sometimes save a fence if it knows that the underlying platform is multicopy-atomic), but these are fairly rare. It has not been my experience that concurrent data structures are vocabulary types; they’re more often used to store data or order operations within an abstraction boundary, rather than share data across interfaces.[2]

A standardized concurrent data structure is then useful primarily in the “batteries included” sense. Reaching for a special-purpose concurrent data structure typically means that the programmer has some profiling information to indicate that a lock on top of a non-concurrent version is a bottleneck. Moreover, it’s rare that the concurrent version will be a drop-in

[2] The exception is building up message-passing frameworks on top of concurrent queues; but even there, the components on either end of the queue need to assume the whole framework rather than only the queue part of it.

replacement, both because of API changes and because there are usually surrounding bits of state that need to be synchronized differently after changing the map.

In such a case, where the user is motivated by performance, has identified a hotspot, and is willing to put in effort to fix it:

- A standardized, one-size-fits-all data structure is unlikely to be the best choice.
- The cost of exploring a variety of alternatives and pulling in third party libraries is acceptable in comparison.

To the extent that including it benefits the programmer, it does so because library distribution is difficult in the C++ ecosystem (compared to just writing “#include <concurrent_foo>”). It would be better to work on that problem directly than to use the standard library as a poor substitute for a package manager.

Standardizing imposes costs

Some of these costs are obvious; the standardization process requires time and money of its participants. Each new library imposes costs on implementations that want to claim full conformance, and make it harder to write new implementations. Depending on the specifics, the binary size of the runtime gets bigger (I suspect this will not be the case for most concurrent data structures, which are often header-only template implementations).

Those costs, though, are mostly not borne by end users. I’d also like to focus on one that is. Standardizing a data structure typically means making it one-size-fits-all. In some cases this is desirable, or even the point (e.g. for vocabulary types). But for data structures whose whole purpose is to increase performance, design decisions made to support parts of the API that end up being unused (for some particular instance of that data structure) incur costs with no benefits. The standardized version of an abstraction can then become an attractive nuisance. It’s hard to guide users towards data structures better tailored for their use cases when there’s one in the standard library that seems like it fits their goals.

Design considerations for concurrent hash maps

Note: P0652R1’s specification is still somewhat informal and does not yet document synchronization properties or state the guarantees on concurrent access. Those arguments specific to that proposal are relative to what I believe is the intent from the reference implementation and SG1 discussion (e.g. some lock is associated with a given key; const visitation requires holding it in shared mode, and non-const visitation requires holding it in exclusive mode).

Here (and subsequently), “visitation” could mean (as in P0652R1) an explicit API expecting a visitor object, but the same arguments also apply to any abstraction that’s morally similar (e.g.

obtaining some proxy object that guards an internal lock, explicit lock_for_key() and unlock_for_key() member functions, etc.). I mean visitation in the broader sense.

Performance issues with P0652

The access API for hashmaps in P0652 is fundamentally lock based; it's hard to imagine a reasonable implementation of those semantics that doesn't boil down to a conventional hash table design using sharded locks. These locks can introduce performance problems:

- Long-running visitation can block other accesses.
- Locking (even with reader/writer locks) introduces reader/reader contention. Readers end up serializing by transferring ownership of the cacheline containing the lock.

The second issue is the more important; the access patterns for "real" hash maps tend to be non-uniform; there are a large number of keys accessed rarely, and a small number of hot keys, which are accessed many times as often. Moreover, many hash maps go through phases of activity: they are write-mostly during infrequent update periods, but read-only the rest of the time; the locking is for a protection that's often unnecessary.

These issues are not fundamental to the problem domain; Facebook's open-source folly library defines a ConcurrentHashMap type that avoids reader-reader contention and prevents long accesses from blocking short ones. The broad strokes:

- The hash map is node based; buckets use a linked-list representation protected by hazard pointers.
- Writer-writer exclusion for structural changes to the map are provided by locks. They are held during equality comparisons in lookups.
- The map does not provide synchronization for its elements; in fact, the API encourages replacement over in-place modification, by giving out const handles to values wherever possible.

In practice, the reduced synchronization costs from avoiding read-side locking more than make up for the worse cache behavior and heap activity (because of the linking). A similar effect shows up in the benchmarks in P0652: in the single-threaded case, the reference implementation is slower than an unordered_map protected by a lock. This is despite the fact that the reference implementation has a couple unfair advantages:

- unordered_map has significant implementation problems aside from the use of linked nodes; it is artificially slow when synchronization costs are discounted
- The synchronization for the unordered_map goes through platform APIs, but the reference implementation uses a custom lock type with an inlinable definition.
- The mutex guarding the unordered_map in the benchmark supports OS-level blocking (and therefore needs 2 RMWs per lock/unlock pair), while the reference implementation uses spinlocks (and therefore only needs 1).

The folly implementation has largely replaced P0652-style sharded maps within Facebook's codebase, at significant performance improvements. Users have not been complaining about the lack of external synchronization. Synchronization costs would rule out the reference implementation of P0652 (and, I believe, any likely implementation of the same API) for many of our use cases.

Performance and API tradeoffs are fundamental to the problem domain

No strong API will be a good fit for a truly broad range of performance-sensitive code. This is true even for the single-threaded case (folly's high performance hash tables have three possible data layout strategies; Abseil's have two; none are extraneous). When making things concurrent, the problems increase:

- How well-sharded do you need to be? If it is too finely sharded, low-contention uses see worse cache behavior. If too coarsely, there is unnecessary contention.
- Do readers lock? If so, you automatically hit many of the performance problems discussed above. If not, you probably lose some nice semantics.
- Are deletions allowed? Concurrent tombstoning is harding than the single-threaded.
- If reads don't block deletions or replacements, how do you handle memory reclamation?
- Is it possible to obtain a global, snapshotted view of the entirety of the hashmap's contents?

The answers to these questions drastically shape the best design for a concurrent hash map. I think that problem domains that require making a large number of context-dependent, opinionated decisions are not well-suited to standardization, especially if the primary justification is performance.

So shut down SG1?

This paper argues against a standardizing a class of useful concurrent code (and instead directing it towards open-source libraries, preferably as part of some sort of easy library distribution system). I think that this set is actually rather narrow. I'll give some examples as to why I don't object to many of the recent papers:

- Executors, retain_ptr, jthread, futures 2.0: Vocabulary types.
- RCU, Hazard Pointers, latest: These are less obvious vocabulary types. Memory protection for concurrently accessed data is a cross-cutting concern that different libraries need to be able to agree on the pattern for.
- Simd, memory model papers: Portability.

In general, APIs that can put most of their implementation behind an easy-to-update ABI are less likely to run into the implementation issues outlined above (because vendors can iterate at whatever pace they like without breaking code). A call to rcu_read_lock() can have the

implementation completely rewritten without the user noticing. Data structures need to have their internal layout publicly visible, and often need to have hot methods inlined for efficiency.

A middle ground

An intermediate step (that standardizes something, but at least allows for the possibility of high performance implementations) would be to make the exposed API highly parameterizable; this follows the policy approach suggested by P0387 (“Memory Model Issues for Concurrent Data Structures”).

In this approach, the map would take a template parameter indicating the range of functionality it needs to support. Questions that could be answered this way include:

- Does the map synchronize access to its elements? Or does it merely protect its own structure. (E.g. if thread 1 wants to access map[“foo”].x, and thread 2 wants to access map[“foo”].y, do they have to block waiting for each other?)
- Is deletion supported?
- Does a reader visitation of a key block removal of that key?
- Is iterator validity maintained after insertions or deletions?
- Is it possible to obtain a consistent view of the whole map at once?
- If the map uses deferred memory reclamation, does removal block waiting for readers to finish? Does destruction ensure all element destructors have executed?
- Can accessing multiple hashmap elements at once deadlock?

Users would then obtain the map type they desire by specifying the flags as appropriate:

```
using my_map_t = std::experimental::concurrent_hash_map<
  std::string, MyType, std::internally_synchronized | std::allow_deletion |
std::allow_snapshotting>;
```

This would not require the map to provide locking for its elements (i.e. they synchronize internally), but would allow element deletion and whole-map snapshots.

I don’t think that this solves the problems outlined above, and that structural barriers will prevent the resulting API from achieving its potential in performance-sensitive code. However, it at least allows for improvements at vendor ABI breaks. It would also let non-standardized libraries implement a similar API, and act as drop-in replacements where necessary.