Bulk visitation in `boost::concurrent_flat_map`

Introduction

`boost::concurrent_flat_map` and its `boost::concurrent_flat_set` counterpart are Boost.Unordered's associative containers for high-performance concurrent scenarios. These containers dispense with iterators in favor of a *visitation*-based interface:

```
boost::concurrent_flat_map<int, int> m;
...
// find the element with key k and increment its associated value
m.visit(k, [](auto& x) {
  ++x.second;
});
```

This design choice was made because visitation is not affected by some inherent problems afflicting iterators in multithreaded environments.

Starting in Boost 1.84, code like the following:

```
std::array<int, N> keys;
...
for(const auto& key: keys) {
  m.visit(key, [](auto& x) { ++x.second; });
}
```

can be written more succintly via the so-called *bulk visitation* API:

```
m.visit(keys.begin(), keys.end(), [](auto& x) { ++x.second; });
```

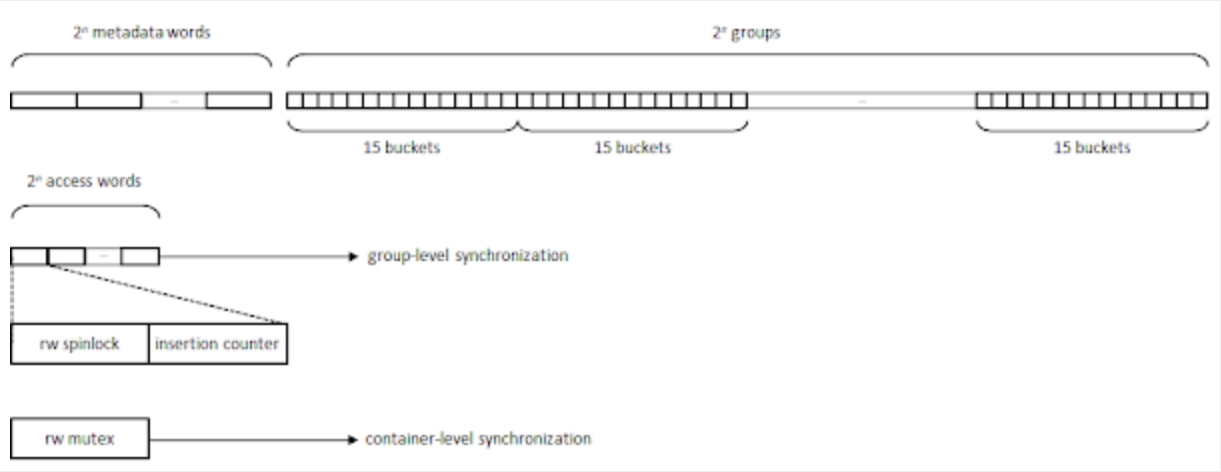
As it happens, bulk visitation is not provided merely for syntactic convenience: this operation is internally optimized so that it performs significantly faster than the original for-loop. We discuss here the key ideas behind bulk visitation internal design and analyze its performance.

Prior art

In their paper "DRAMHiT: A Hash Table Architected for the Speed of DRAM", Narayanan et al. explore some optimization techniques from the domain of distributed system as translated to concurrent hash tables running on modern multi-core

architectures with hierarchical caches. In particular, they note that cache misses can be avoided by batching requests to the hash table, prefetching the memory positions required by those requests and then completing the operations asynchronously when enough time has passed for the data to be effectively retrieved. Our bulk visitation implementation draws inspiration from this technique, although in our case visitation is fully synchronous and in-order, and it is the responsibility of the user to batch keys before calling the bulk overload of `boost::concurrent_flat_map::visit`.

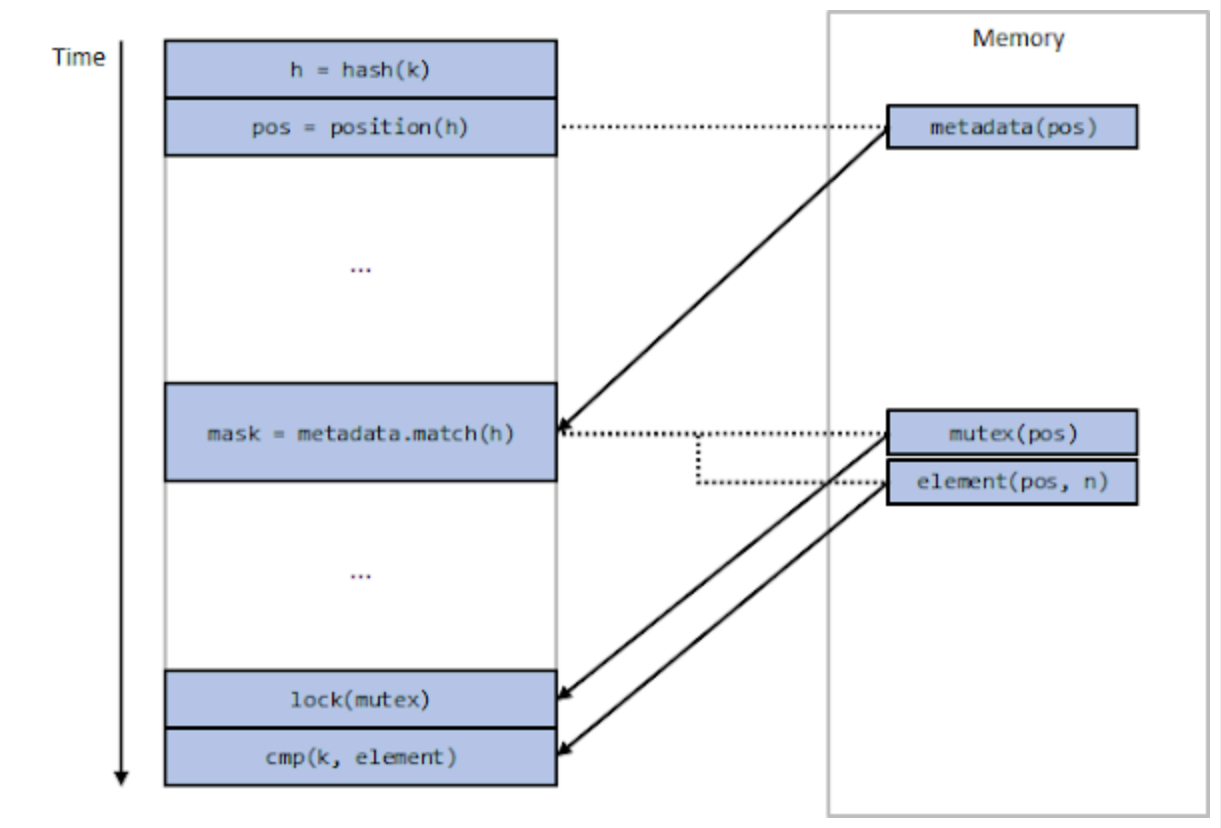
Bulk visitation design



As discussed in a previous article, `boost::concurrent_flat_map` uses an open-addressing data structure comprising:

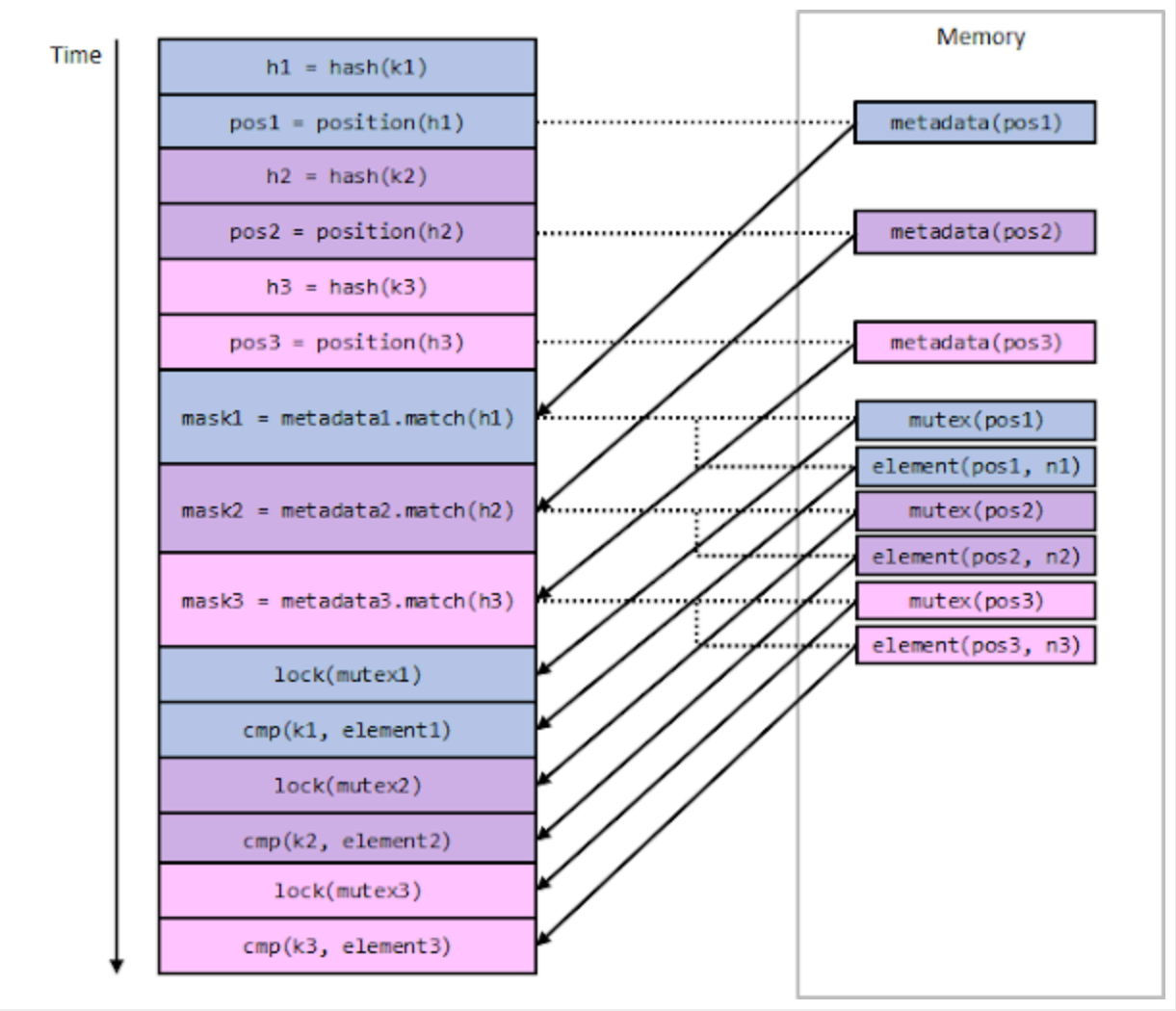
- A bucket array split into 2^n groups of $N = 15$ slots.
- A metadata array associating a 16-byte metadata word to each slot group, used for SIMD-based reduced-hash matching.
- An access array with a spinlock (and some additional information) for locked access to each group.

The happy path for successful visitation looks like this:



1. The hash value for the looked-up key and its mapped group position are calculated.
2. The metadata for the group is retrieved and matched against the hash value.
3. If the match is positive (which is the case for the *happy* path), the group is locked for access and the element indicated by the matching mask is retrieved and compared with the key. Again, in the happy path this comparison is positive (the element is found); in the unhappy path, more elements (within this group or beyond) need be checked.

(Note that happy *unsuccessful* visitation simply terminates at step 2, so we focus our analysis on successful visitation.) As the diagram shows, the CPU has to wait for memory retrieval between steps 1 and 2 and between steps 2 and 3 (in the latter case, retrievals of mutex and element are parallelized through manual prefetching). A key insight is that, under normal circumstances, these memory accesses will almost always be cache misses: successive visitation operations, unless for the very same key, won't have any cache locality. In bulk visitation, the stages of the algorithm are *pipelined* as follows (the diagram shows the case of three operations in the bulk batch):



The data required at step *N*+1 is prefetched at the end of step *N*. Now, if a sufficiently large number of operations are pipelined, we can effectively eliminate cache miss stalls: all memory addresses will be already cached by the time they are used.

The operation `visit(first, last, f)` internally splits [`first`, `last`) into chunks of `bulk_visit_size` elements that are then processed as described above. This chunk size has to be sufficiently large to give time for memory to be actually cached at the point of usage. On the upper side, the chunk size is limited by the number of outstanding memory requests that the CPU can handle at a time: in Intel architectures, this is limited by the size of the *line fill buffer*, typically 10-12. We have empirically confirmed that bulk visitation maxes at around `bulk_visit_size` = 16, and stabilizes beyond that.

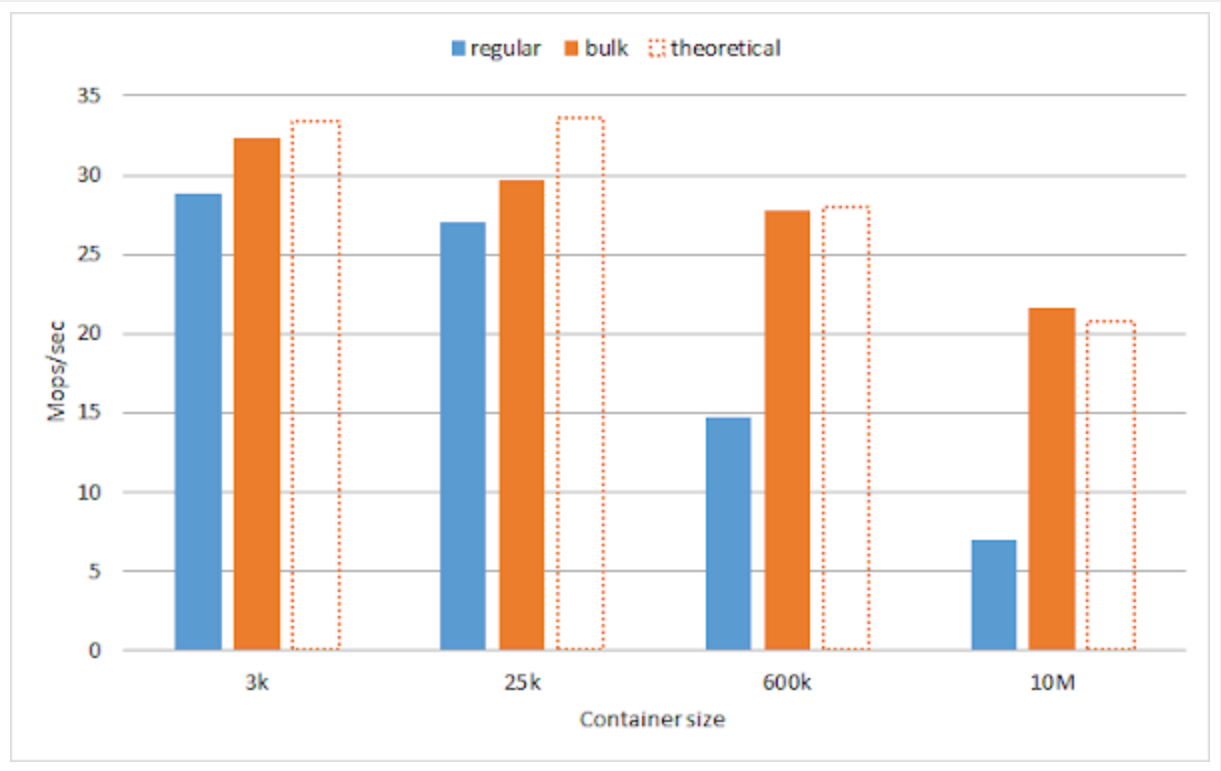
Performance analysis

For our study of bulk visitation performance, we have used a computer with a Skylake-based Intel Core i5-8265U CPU:

	Size/core	Latency [ns]
L1 data cache	32 KB	3.13
L2 cache	256 KB	6.88

	Size/core	Latency [ns]
L3 cache	6 MB	25.00
DDR4 RAM		77.25

We measure the throughput in Mops/sec of single-threaded lookup (50/50 successful/unsuccessful) for both regular and bulk visitation on a `boost::concurrent_flat_map<int, int>` with sizes N = 3k, 25k, 600k, and 10M: for the three first values, the container fits entirely into L1, L2 and L3, respectively. The test program has been compiled with `clang-cl` for Visual Studio 2022 in release mode.



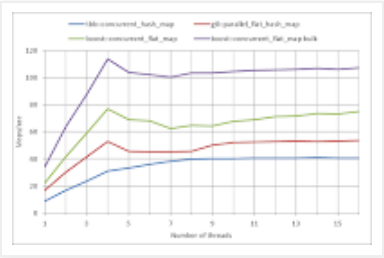
As expected, the relative performance of bulk vs. regular visitation grows as data is fetched from a slower cache (or RAM in the latter case). The theoretical throughput achievable by bulk visitation has been estimated from regular visitation by subtracting memory retrieval times as calculated with the following model:

- If the container fits in Ln (L4 = RAM), Ln–1 is entirely occupied by metadata and access objects (and some of this data spills over to Ln).
- Mutex and element retrieval times (which only apply to successful visitation) are dominated by the latter.

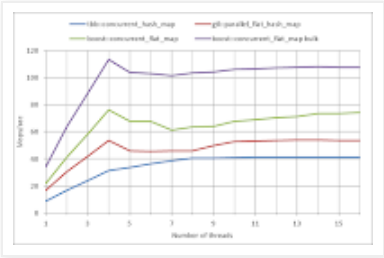
Actual and theoretical figures match quite well, which sugggests that the algorithmic overhead imposed by bulk visitation is negligible.

We have also run benchmarks under conditions more similar to real-life for `boost::concurrent_flat_map`, with and without bulk visitation, and other

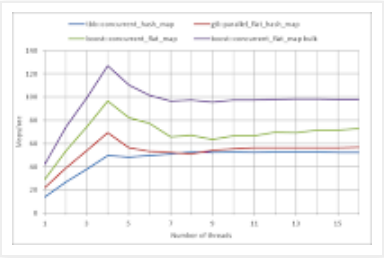
concurrent containers, using different compilers and architectures. As an example, these are the results for a workload of 50M insert/lookup mixed operations distributed across several concurrent threads for different data distributions with Clang 12 on an ARM64 computer:



5M updates, 45M lookups
skew=0.01



5M updates, 45M lookups
skew=0.5



5M updates, 45M lookups
skew=0.99

Again, bulk visitation increases performance noticeably. Please refer to the benchmark site for further information and results.

Conclusions and next steps

Bulk visitation is an addition to the interface of `boost::concurrent_flat_map` and `boost::concurrent_flat_set` that improves lookup performance by pipelining the internal visitation operations for chunked groups of keys. The tradeoff for this increased throughput is higher latency, as keys need to be batched by the user code before issuing the `visit` operation.

The insights we have gained with bulk visitation for concurrent containers can be leveraged for future Boost.Unordered features:

- In principle, insertion can also be made to operate in bulk mode, although the resulting pipelined algorithm is likely more complex than in the visitation case, and thus performance increases are expected to be lower.
- Bulk visitation (and insertion) is directly applicable to non-concurrent containers such as `boost::unordered_flat_map`: the main problem for this is one of interface design because we are not using visitation here as the default lookup API (classical iterator-based lookup is provided instead). Some possible options are:
    1. Use visitation as in the concurrent case.
    2. Use an iterator-based lookup API that outputs the resulting iterators to some user-provided buffer (probably modelled as an output "meta" iterator taking container iterators).

Bulk visitation will be officially shipping in Boost 1.84 (December 2023) but is already available by checking out the Boost.Unordered repo. If you are interested in this feature, please try it and report your local results and suggestions for improvement. Your feedback on our current and future work is much welcome.