A Standard `flat_map`

Document Number: P0429R0
Date: 2016-08-31
Reply to: Zach Laine whatwasthataddress@gmail.com
Audience: LWG/LEWG

1 Introduction

This paper outlines what a (mostly) API-compatible, non-node-based `map` might look like. Rather than presenting a final design, this paper is intended as a starting point for discussion and as a basis for future work. Specifically, there is no mention of `multimap`, `set`, or `multiset`. Those will be added in later papers.

2 Motivation and Scope

There has been a strong desire for a more space- and/or runtime-efficient representation for `map` among C++ users for some time now. This has motivated discussions among the members of SG14 resulting in a paper[1], numerous articles and talks, and an implementation in Boost, `boost::container::flat_map`[2]. Virtually everyone who makes games, embedded, or system software in C++ uses the Boost implementation or one that they rolled themselves.

Here are some numbers that show why. The graphs that follow show runtimes for different `map`-like associative containers. The containers used are Boost.FlatMap, `map`, `unordered_map`, and two thin wrappers over a sorted `vector`. The "custom pair" version of the sorted `vector` uses a simple `struct` instead of `pair` for its value type. All containers use either `<int, int>` or `<std::string, std::string>` for the value type.

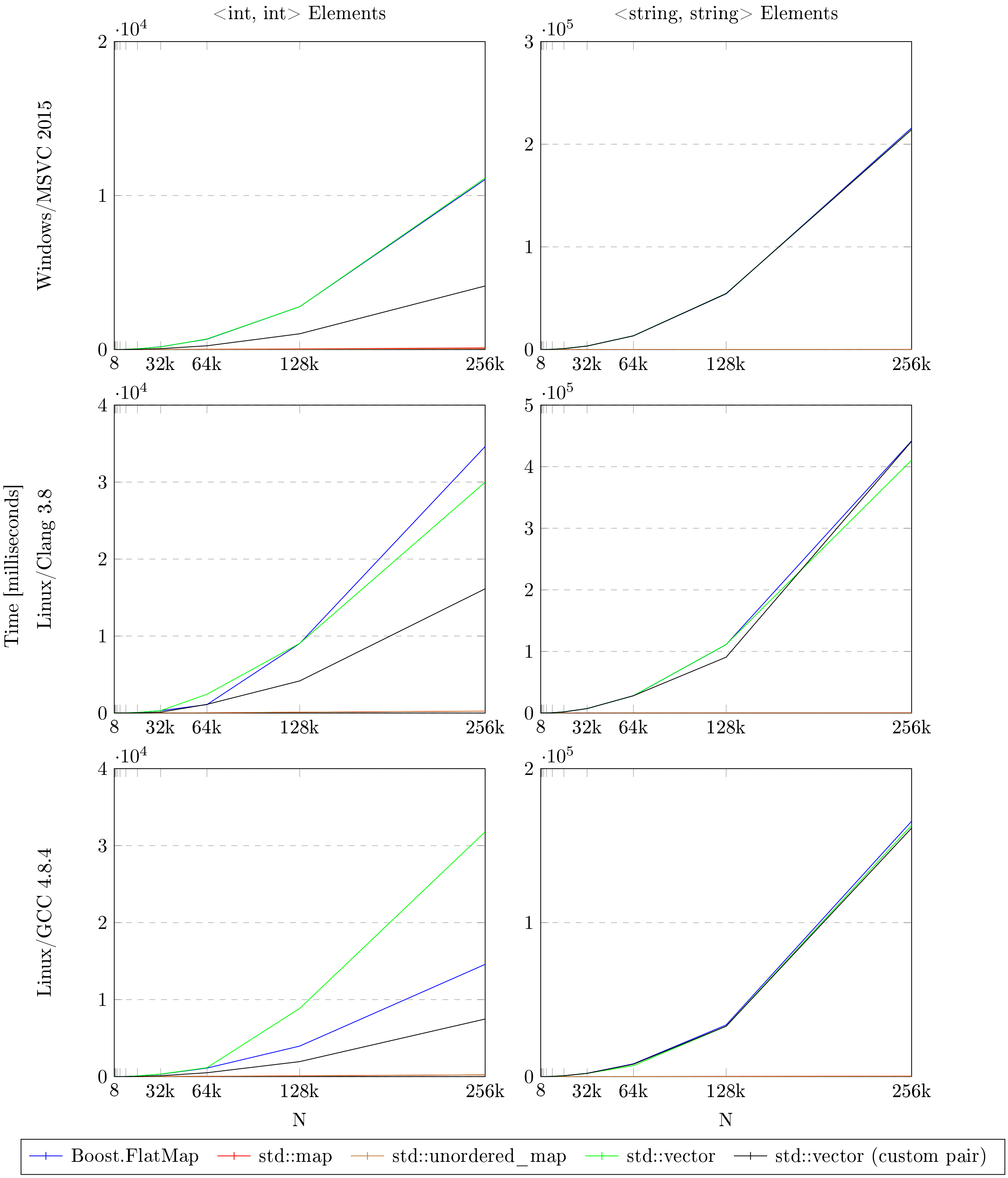
All data in the graphs below were produced on Windows with MSVC 2015, on Linux with Clang 3.8 and libc++, or on Linux with g++ 4.8.4 and libstdc++.

Each set of six graphs shows the performance of a single operation on all map-variants. The left column shows the `<int, int>` runs, and the right column shows the `<std::string, std::string>` ones. Each row shows one platform/compiler configuration.

These four sets of graphs cover the most commonly-used operations. The first set shows insertion of N elements with random keys; the second shows full iteration across all N elements; the third shows `map.find()` called once for each key used in the original insertions; and the fourth shows erasure of all N elements, by the keys used in the original insertions.

[1] See P0038R0, here.
[2] Part of Boost.Container, here.

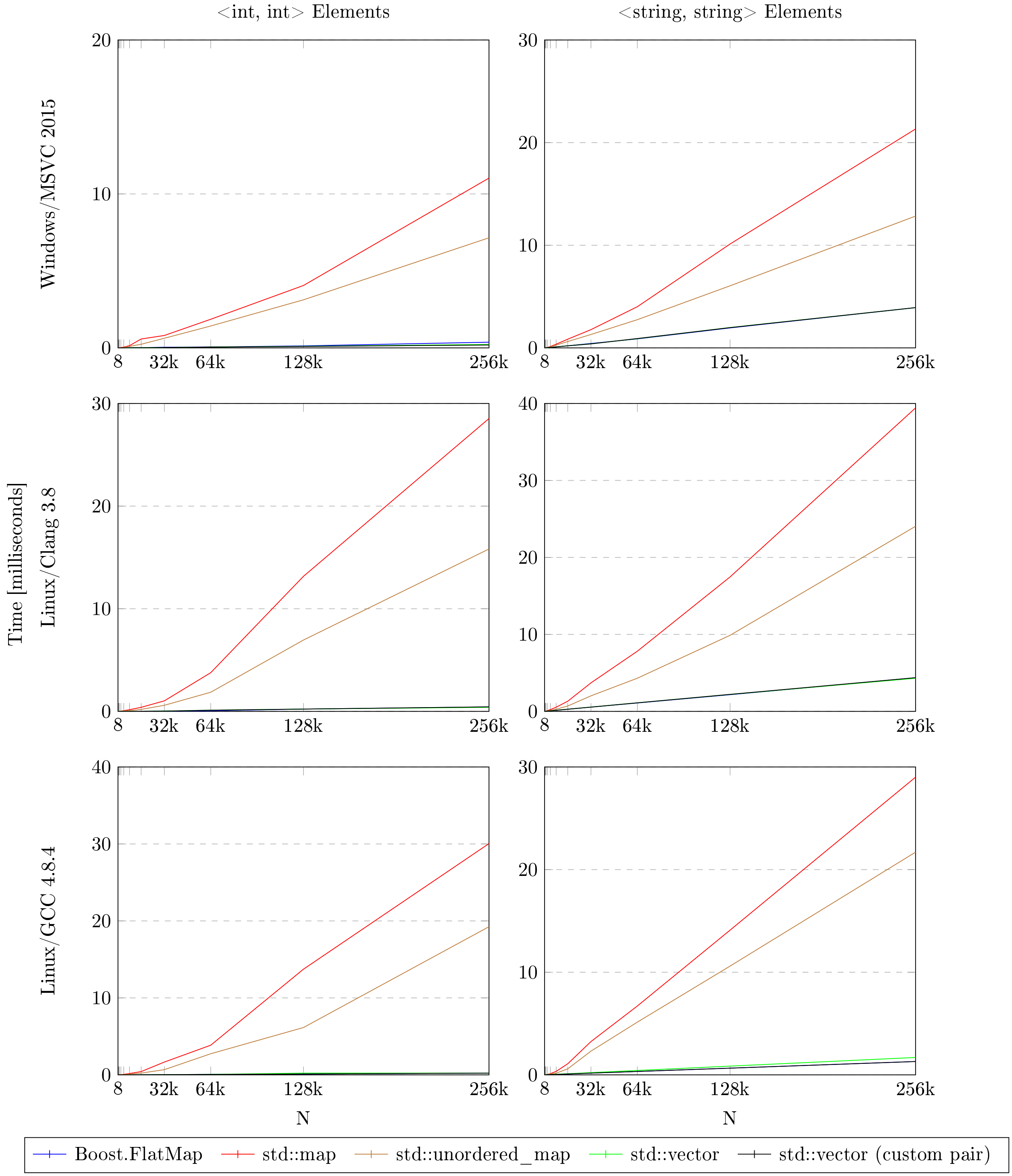
2.1 Insert



Unsurprisingly, insertion takes longer in contiguous-storage implementations. Boost.FlatMap and a sorted vector<pair<int, in have the steepest growth curves. The curve for sorted vector using a custom struct is dramatically flatter in its growth in the <int, int> runs. Note that the custom-pair vector does about the same as the vector of pair in the

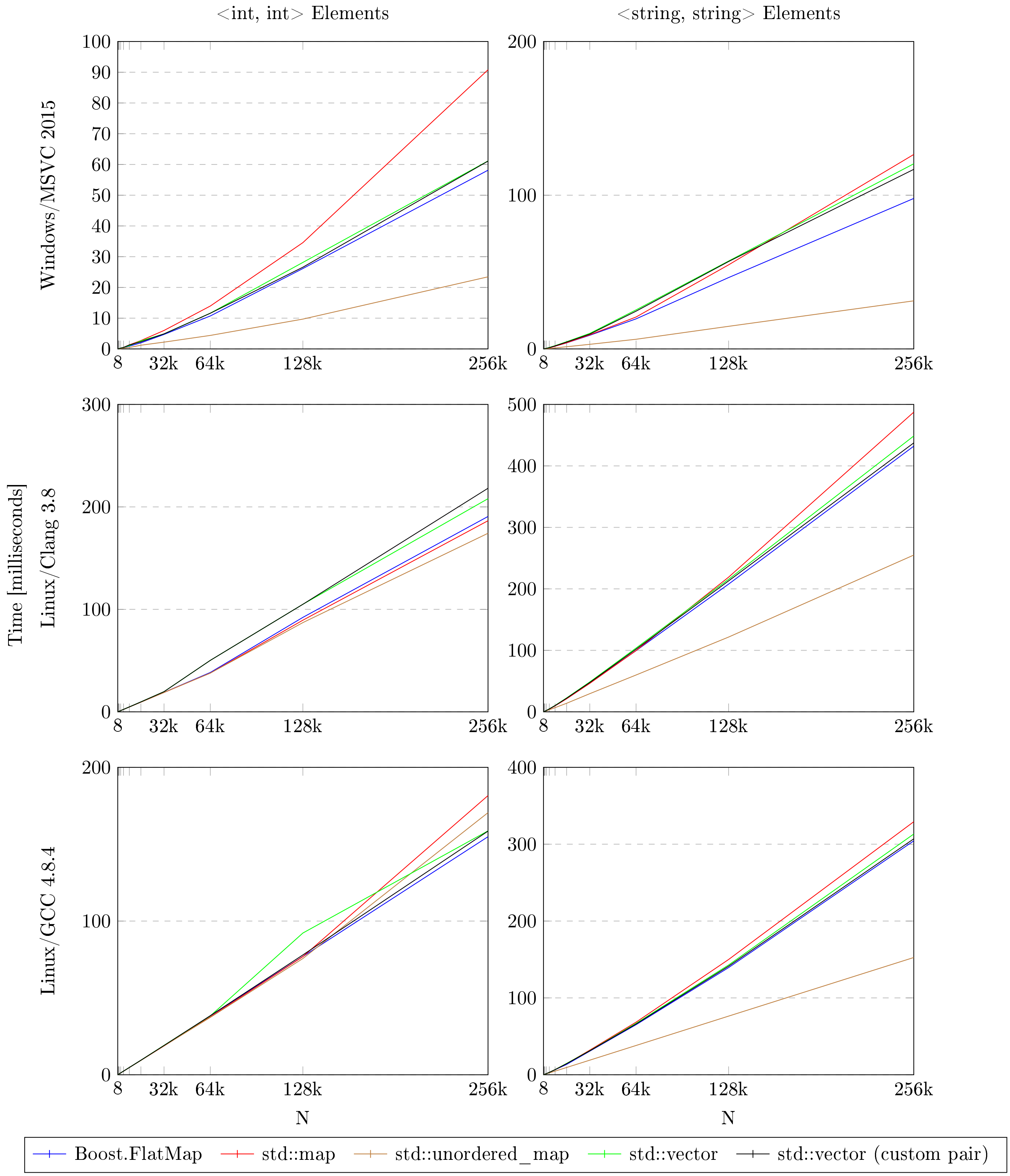
`<std::string, std::string>` runs.

2.2 Iterate



For all variants but `map` and `unordered_map`, iteration is relatively similar, and much faster than `map`'s.

2.3 Find

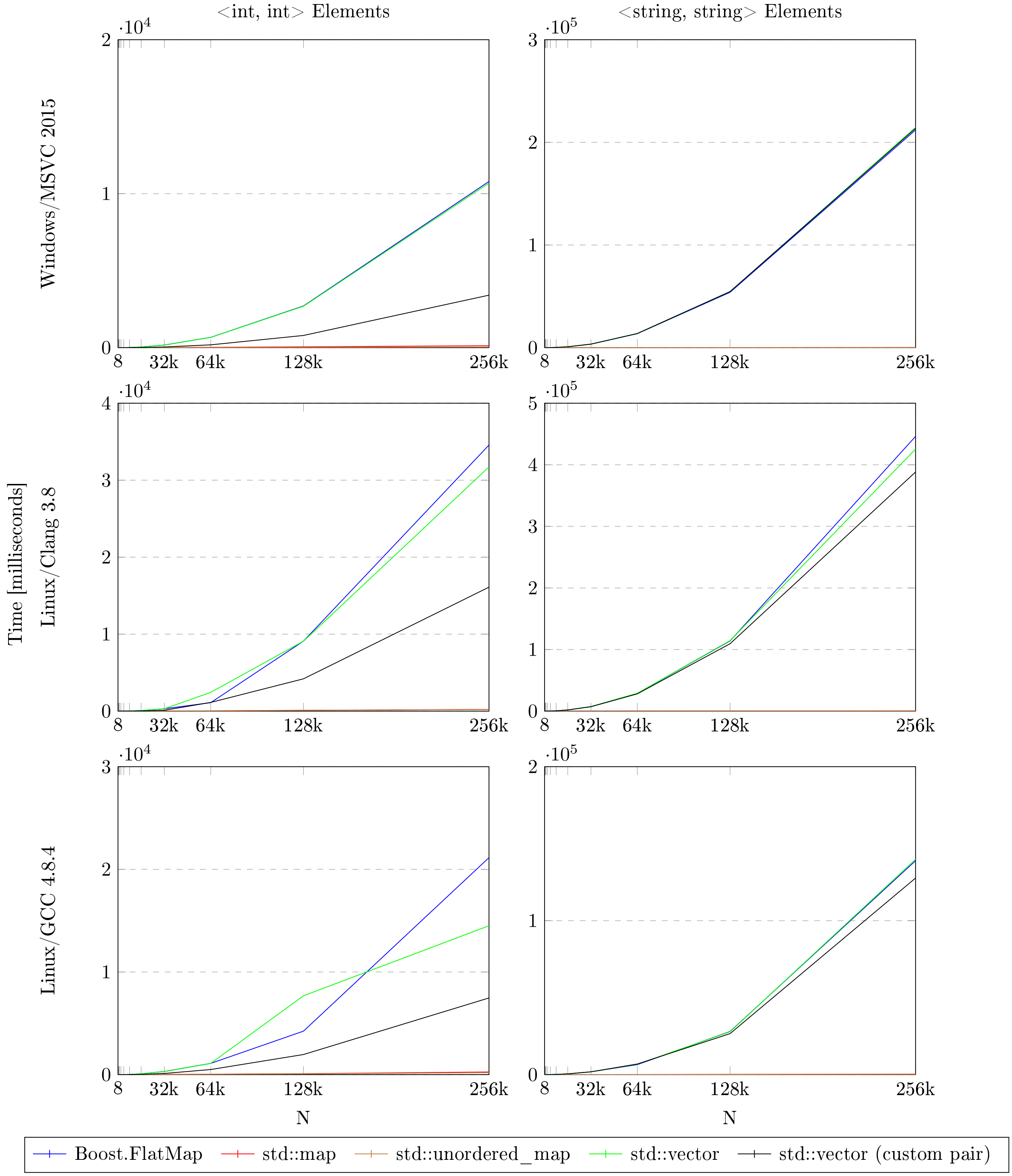


`find()` performance is where things get interesting. The different platforms produce strikingly different results.

In the MSVC runs, there is a large differentiation between Boost.FlatMap and `map`; in fact, Boost.FlatMap even beats both the sorted `vector` variants. Also, `unordered_map` is the clear winner, regardless of valkue type.

GCC and Clang on Linux produce nearly identical results. For `<int, int>` runs, all implementations are nearly identical. `unordered_map` is faster in the `<std::string, std::string>` runs, but all other variants are very close.

2.4 Erase



Erasure has a nearly identical performance profile to insertion.

2.5 Implications

Iteration is vastly cheaper for contiguous-storage variants. Any node-based associative container will always be slower than a flattened one for iteration. For use cases where there is a lot of iteration, this can be the deciding runtime performance consideration.

In all the graphs above, the reason the custom-`pair` sorted vector performs so much better than `vector<pair<int, int>>` seems to be that the custom-`pair` type has `nothrow` special functions. Implementing all the special functions and adding `nothrow(false)` to each makes the custom-`pair` version perform identically to the `pair<int, int>` version.

Boost.FlatMap differs significantly from a sorted `vector`. Clearly there are a lot of QOI choices that affect the runtime performance of a standard `flat_map`.

The fact that insertion and erasure operations produce such similar results implies that pre-`reserve()`ing space will probably not make much difference when using a flat map.

Use cases in which the runtime performance of a flat map would be no better than `map` or `unordered_map`, the user may still decide to use a flat implementation for the storage savings.

3 Proposed Design

3.1 Design Goals

Overall, `flat_map` is meant to be a drop-in replacement for `map`, jsut with better time- and space-efficiency. Functionally it is not meant to do anything other than what we do with `map` now.

The Boost.Container documentation gives a nice summary of the tradeoffs between node-based and flat associative containers (quoted here, mostly verbatim). Note that they are not purely positive:

- Faster lookup than standard associative containers.
- Much faster iteration than standard associative containers.
- Random-access iterators instead of bidirectional iterators.
- Less memory consumption for each element.
- Improved cache performance (data is stored in contiguous memory).
- Non-stable iterators (iterators are invalidated when inserting and erasing elements).
- Non-copyable and non-movable values types can't be stored.
- Weaker exception safety than standard associative containers (copy/move constructors can throw when shifting values in erasures and insertions).
- Slower insertion and erasure than standard associative containers (specially for non-movable types).

The overarching goal of this proposal is to define a `flat_map` for standardization that fits the above gross profile, while leaving maximum room for customization by users.

3.2 Design

3.2.1 `flat_map` Is Based On Boost.FlatMap

This proposal represents existing practice in widespread use – Boost.FlatMap has been available since 2011 (Boost 1.48).

3.2.2 `flat_map` Is Nearly API-Compatible With `map`

Most of `flat_map`'s interface is identical to `map`'s. Some of the differences are required (more on this later), but a couple of interface changes are optional:

- The overloads that take sorted containers or sequences.
- Making `flat_map` a container adapter.

Both of these interface changes were added to increase optimization opportunities.

3.2.3 `flat_map` Is a Container Adapter

`flat_map` is an adapter for an underlying storage type. This storage type is configurable via the template parameter `Container`. `Container` must be a *contiguous container* (§23.2.1/13). `vector` is a great candidate for this, but limiting `flat_map` only to use `vector` for its storage would be a mistake. Many other suitable replacements exist, each suited to a certain use. A user may have a small-buffer implementation of `vector`, like LLVM's `SmallVector`, or `boost::container::small_vector`. The user may also want to avoid allocations altogether, if the maximum number of elements N is known *a priori*. If so, `boost::container::static_vector` could be used. The user's specific performance requirements will dictate which of these is most appropriate.

There are certain optimization opportunities that are lost to the user of a non-adapter `flat_map`. For instance, if one does not care about the strong or weak exception guarantees in the code that uses `flat_map`, one can use a `Container` that blindly uses `move` all the time, even if exceptions may occur. This allows performance curves more like the `<int, int>` custom-pair sorted `vector` in the graphs above.

While this may not be a use case for a majority of users, there are numerous such niche use cases, and these niches are not well served by a fixed underlying storage implementation.

3.2.4 Interface Differences From `map`

- Members `capacity()`, `reserve()`, and `shrik_to_fit()` have been added, with the same semantics as the corresponding members of `vector`.
- Several new constructors have been added that take objects of the `Container` type. These members must only be used if the given container is already sorted.
- The `extract()` overloads from `map` are replaced with `Container extract()`, that moves out the entire storage of the `flat_map`. Similarly, the `insert()` members taking a node have been replaced with a member `void replace(Container&&)`, that moves in the entire storage.

  Many users have noted that M insertions of elements into a map of size N is O(M·log(N+M)), and when M is known it should be possible instead to append M times, and then re-sort, as one might with a sorted `vector`. This makes the insertion of multiple elements closer to O(N), depending on the implementation of `sort()`.

  Such users have often asked for an API in `boost::container::flat_map` that allows this pattern of use. Other flat-map implementations have undoubtedly added such an API. The extract/replace API instead allows the same optimization opportunities without violating the class invariants.
- Several new constructors and an `insert()` overload use a new tag type, `ordered_unique_sequence_tag`. These members must only be used if the given sequence is already sorted. This can allow much more efficient construction and insertion.

3.2.5 `flat_map` Requirements

Since the underlying container is contiguous and elements may be moved or copied during inserts and erases, the element type of `Container` must be `pair<Key, T>`, not `pair<const Key, T>`. Even so, the element type of `flat_map` should still be `pair<const Key, T>`, for drop-in compatibility with `map` (§23.2.4/5). This requires `flat_map` to have an iterator that adapts the underlying `Container` iterator.

Only the underlying container is allocator-aware. §23.2.4/7 regarding allocator awareness does not apply to `flat_map`.

Validity of iterators is not preserved when mutating the underlying container (i.e. §23.2.4/9 does not apply).

The exception safety guarantees for associative containers (§23.2.4.1) do not apply.

The rest of the requirements follow the ones in (§23.2.4 Associative containers), except §23.2.4/10 (which applies to members not in `flat_map`) and some portions of the table in §23.2.4/8; these table differences are outlined in "Member Semantics" below.

3.2.6 `Container` Requirements

Any contiguous container supporting operations `capacity()`, `reserve()`, and `shrik_to_fit()` can be used for the `Container` template parameter. `Container` must have a `value_type` of `pair<Key, T>`.

3.2.7 Member Semantics

Members `capacity()`, `reserve()`, and `shrik_to_fit()` have the same semantics as the corresponding members of `vector`.

Each member taking a `Container` reference or taking a parameter of type `ordered_unique_sequence_tag` has the precondition that the given elements are already sorted by `Compare`, and that the elements are unique.

Each member taking an `Alloc` template parameter only participates in overload resolution if `uses_allocator_v<Container, Allo` is `true`.

Other member semantics are the same as for `map`.

3.2.8 `flat_map` Synopsis

```
namespace std {

struct ordered_unique_sequence_tag { };

template <class Key, class T, class Compare = default_order_t<Key>,
          class Container = vector<pair<Key, T>>>
class flat_map {
public:
    // types:
    using key_type               = Key;
    using mapped_type            = T;
    using value_type             = pair<const Key, T>;
    using key_compare            = Compare;
    using allocator_type         = typename Container::allocator_type;
    using pointer                = value_type*;
    using const_pointer          = const value_type*;
    using reference              = value_type&;
    using const_reference        = const value_type&;
    using size_type              = typename Container::size_type;
    using iterator               = implementation-defined;
    using const_iterator         = implementation-defined;
    using reverse_iterator       = implementation-defined;
    using const_reverse_iterator = implementation-defined;
    using container_type         = Container;

    class value_compare {
      friend class flat_map;
    protected:
      Compare comp;
      value_compare(Compare c) : comp(c) { }
    public:
      bool operator()(const value_type& x, const value_type& y) const {
        return comp(x.first, y.first);
      }
    };

    // construct/copy/destroy:
    explicit flat_map(const Container&);
    template <class Alloc>
```

```
  flat_map(const Container&, const Alloc&);
explicit flat_map(Container&& = Container());
template <class Alloc>
  flat_map(Container&&, const Alloc&);

explicit flat_map(const Compare& comp);
template <class Alloc>
  flat_map(const Compare& comp, const Alloc&);
template <class Alloc>
  explicit flat_map(const Alloc&);
template <class InputIterator>
  flat_map(InputIterator first, InputIterator last,
           const Compare& comp = Compare());
template <class InputIterator, class Alloc>
  flat_map(InputIterator first, InputIterator last,
           const Compare& comp, const Alloc&);
template <class InputIterator, class Alloc>
  flat_map(InputIterator first, InputIterator last, const Alloc& a)
    : flat_map(first, last, Compare(), a) { }

template <class InputIterator>
  flat_map(ordered_unique_sequence_tag, InputIterator first, InputIterator last,
           const Compare& comp = Compare());
template <class InputIterator, class Alloc>
  flat_map(ordered_unique_sequence_tag, InputIterator first, InputIterator last,
           const Compare& comp, const Alloc&);
template <class InputIterator, class Alloc>
  flat_map(ordered_unique_sequence_tag, InputIterator first, InputIterator last,
           const Alloc& a)
    : flat_map(first, last, Compare(), a) { }

template <class Alloc>
  flat_map(const flat_map&, const Alloc&);
template <class Alloc>
  flat_map(flat_map&&, const Alloc&);

flat_map(initializer_list<value_type>,
         const Compare& = Compare());
 template <class Alloc>
  flat_map(initializer_list<value_type>,
           const Compare&,
           const Alloc&);
template <class Alloc>
  flat_map(initializer_list<value_type> il, const Alloc& a)
    : flat_map(il, Compare(), a) { }
flat_map& operator=(initializer_list<value_type>);

// iterators:
iterator                  begin() noexcept;
const_iterator            begin() const noexcept;
iterator                  end() noexcept;
const_iterator            end() const noexcept;
reverse_iterator          rbegin() noexcept;
const_reverse_iterator    rbegin() const noexcept;
reverse_iterator          rend() noexcept;
const_reverse_iterator    rend() const noexcept;
const_iterator            cbegin() const noexcept;
const_iterator            cend() const noexcept;
const_reverse_iterator    crbegin() const noexcept;
const_reverse_iterator    crend() const noexcept;

// capacity:
```

```
bool      empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;
size_type capacity() const noexcept;
void reserve(size_type x);
void shrink_to_fit();

// element access:
T& operator[](const key_type& x);
T& operator[](key_type&& x);
T& at(const key_type& x);
const T& at(const key_type& x) const;

// modifiers:
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& x);
pair<iterator, bool> insert(value_type&& x);
template <class P> pair<iterator, bool> insert(P&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template <class P>
  iterator insert(const_iterator position, P&&);
template <class InputIterator>
  void insert(InputIterator first, InputIterator last);
template <class InputIterator>
  void insert(ordered_unique_sequence_tag, InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

Container extract();
void replace(Container&&);

template <class... Args>
  pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template <class... Args>
  pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template <class... Args>
  iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
template <class... Args>
  iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
template <class M>
  pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template <class M>
  pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template <class M>
  iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);
template <class M>
  iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);

void swap(flat_map& fm)
    noexcept(noexcept(declval<Container>().swap(declval<Container&>())));
void clear() noexcept;

template<class C2>
  void merge(flat_map<Key, T, C2, Container>& source);
template<class C2>
  void merge(flat_map<Key, T, C2, Container>&& source);
```

```
    template<class C2>
      void merge(flat_multimap<Key, T, C2, Container>& source);
    template<class C2>
      void merge(flat_multimap<Key, T, C2, Container>&& source);

    // observers:
    key_compare key_comp() const;
    value_compare value_comp() const;

    // map operations:
    iterator find(const key_type& x);
    const_iterator find(const key_type& x) const;
    template <class K> iterator find(const K& x);
    template <class K> const_iterator find(const K& x) const;
      size_type count(const key_type& x) const;
    template <class K> size_type count(const K& x) const;
    iterator lower_bound(const key_type& x);
    const_iterator lower_bound(const key_type& x) const;
    template <class K> iterator lower_bound(const K& x);
    template <class K> const_iterator lower_bound(const K& x) const;
    iterator upper_bound(const key_type& x);
    const_iterator upper_bound(const key_type& x) const;
    template <class K> iterator upper_bound(const K& x);
    template <class K> const_iterator upper_bound(const K& x) const;
    pair<iterator, iterator> equal_range(const key_type& x);
    pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
    template <class K>
      pair<iterator, iterator> equal_range(const K& x);
    template <class K>
      pair<const_iterator, const_iterator> equal_range(const K& x) const;
};

template <class Key, class T, class Compare, class Container>
  bool operator==(const flat_map<Key, T, Compare, Container>& x,
                  const flat_map<Key, T, Compare, Container>& y);
template <class Key, class T, class Compare, class Container>
  bool operator< (const flat_map<Key, T, Compare, Container>& x,
                  const flat_map<Key, T, Compare, Container>& y);
template <class Key, class T, class Compare, class Container>
  bool operator!=(const flat_map<Key, T, Compare, Container>& x,
                  const flat_map<Key, T, Compare, Container>& y);
template <class Key, class T, class Compare, class Container>
  bool operator> (const flat_map<Key, T, Compare, Container>& x,
                  const flat_map<Key, T, Compare, Container>& y);
template <class Key, class T, class Compare, class Container>
  bool operator>=(const flat_map<Key, T, Compare, Container>& x,
                  const flat_map<Key, T, Compare, Container>& y);
template <class Key, class T, class Compare, class Container>
  bool operator<=(const flat_map<Key, T, Compare, Container>& x,
                  const flat_map<Key, T, Compare, Container>& y);

// specialized algorithms:
template <class Key, class T, class Compare, class Container>
  void swap(flat_map<Key, T, Compare, Container>& x,
            flat_map<Key, T, Compare, Container>& y)
    noexcept(noexcept(x.swap(y)));

}
```

4 Acknowledgements

Thanks to Ion Gaztañaga for writing Boost.FlatMap.

Thanks to David Sankel, for reviews of early drafts of this paper.

Thanks to Bloomberg, for sharing performance data they used for internal decisionmaking on their use of flat maps.