

+ 22

Refresher on Containers, Algorithms and Performance

VLADIMIR VISHNEVSKII



20
22

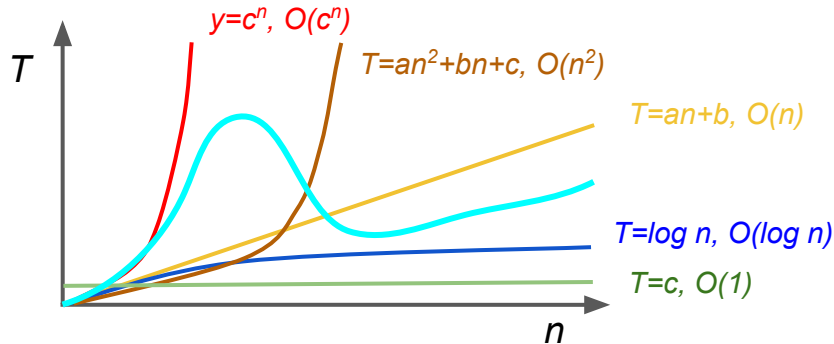


Motivation and agenda

- Performance of containers can be a crucial component of an application performance
- Performance is exciting topic as it is where multiple theoretical disciplines meet practice
- The purpose of the talk is to **revisit the basic factors defining efficiency of C++ containers and algorithms** and elaborate recommendations on effective usage considering individual characteristics of containers

Time complexity and big O notation

- Time complexity is estimated via the number of operations performed
- Big O notation describes scaling of an algorithm (growth of number of operations) by the means of known functions:



- Faster growing functions will overtake slower ones, but this can happen beyond the size of a real-life data set
- As the constants and smaller terms are ignored for big O notation, real life performance of algorithms can not be compared based on big O class

Time complexity and big O notation

- Can be used to reason about the relative performance of algorithms if the numbers of the same operations are compared. For instance, linear is faster:

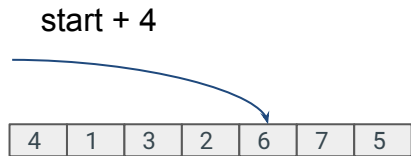
```
for (0..N)
    std::swap(...)
```

- Compared to quadratic:

```
for (0..N)
    for (0..N)
        std::swap(...)
```

Big O classes of typical operations

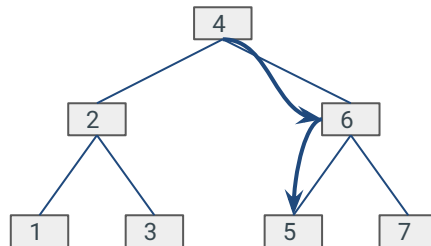
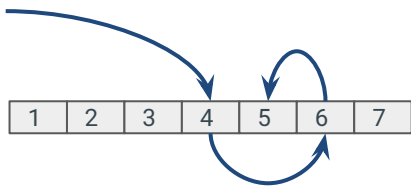
- Random access to the item in continuous storage is $O(1)$



- Traversal or linear search for continuous or linked container is $O(n)$

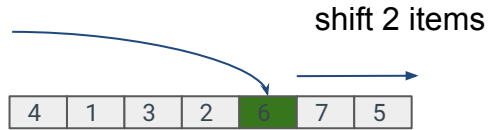


- Binary search in continuous sorted container or in Binary Search Tree is $O(\lg n)$

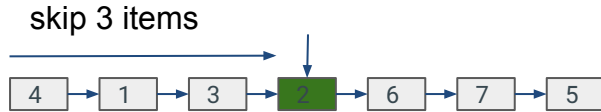


Big O classes of typical operations

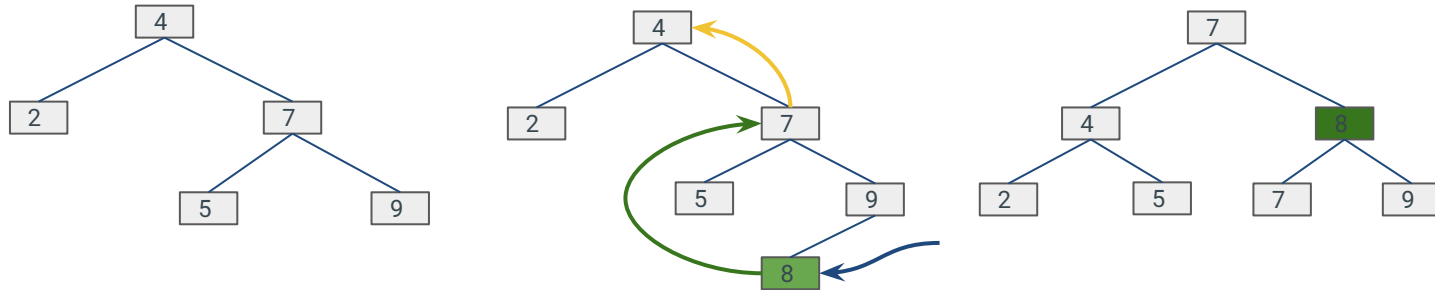
- Insertion into continuous storage is $O(n \text{ of item after insert position})$



- Insertion into linked list is $O(\text{position}) + O(1)$



- Insertion into balanced Binary Search Tree is typically $O(\log \text{ size})$ in worst case



Experiments. Basic setup

- OS: Ubuntu 20.04
- Compiler: gcc 10.3.0, libstdc++
- Optimization flags: -O3
- CPU: 8 cores x86_64 2112.01 MHz
- Google benchmark library is used (<https://github.com/google/benchmark>)

The benchmarking results are dependent on a setup. The trends and relative results will be analyzed in the presentation

Iteration through `std::vector` and `std::list`

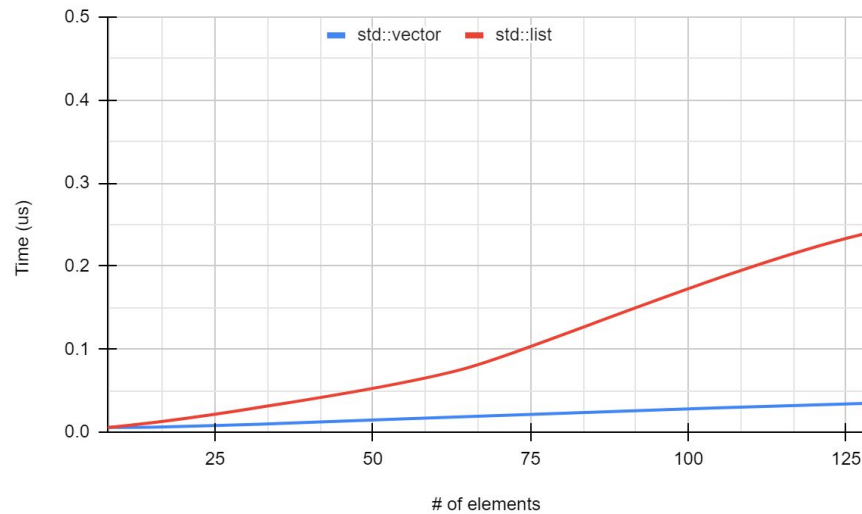
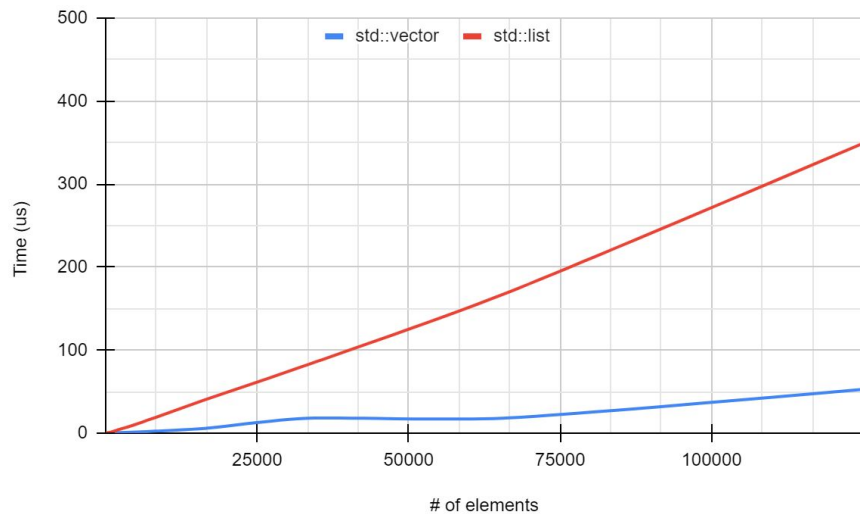
- `std::accumulate` is performed on `std::vector` and `std::list` holding `n` items of type `uint32_t`

```
auto const result = std::accumulate(container.begin(), container.end(), 0u);
```

- Each item is accessed once so the complexity is linear $O(n)$

Iteration through `std::vector` and `std::list`

- Linear graphs depict linear growth of execution time for both containers
- Starting from small number of items `std::vector` significantly outperforms `std::list`
- What defines the advantage of `std::vector`?

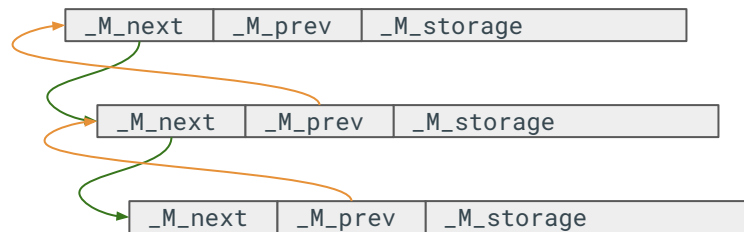


Memory access. Organization of `std::list`

- List items are allocated in arbitrary memory locations:

```
struct _List_node_base
{
    _List_node_base* _M_next;
    _List_node_base* _M_prev;
    //...
};

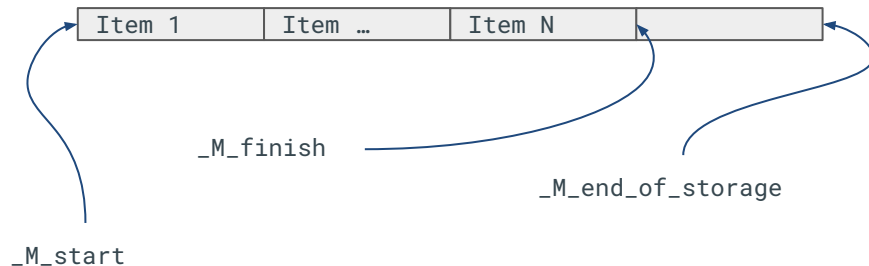
template<typename _Tp>
struct _List_node : public _detail::_List_node_base
{
    #if __cplusplus >= 201103L
        __gnu_cxx::__aligned_membuf<_Tp> _M_storage;
        //...
    };
```



Memory access. Organization of `std::vector`

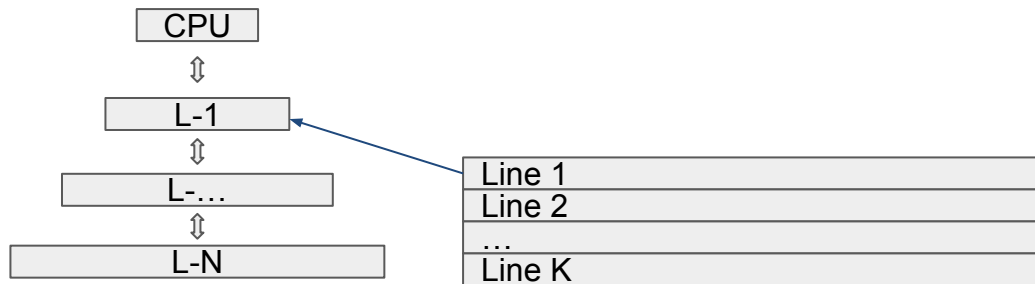
- Items in `std::vector` arranged in a continuous storage:

```
struct _Vector_impl_data
{
    pointer _M_start;
    pointer _M_finish;
    pointer _M_end_of_storage;
    //...
};
```



Memory access. Caches

- Relevant for the platforms with cache memory (for example X86_64)
- Faster and smaller memory to compensate expensive memory accesses
- Beneficial for memory accesses with **temporal and spatial locality**: the data that was recently accessed or data located near recently accessed will be accessed soon
- In case of miss (no data in cache) access to a level is significantly longer
- Cache is limited in size and is organized in aligned blocks (cache lines)



Run on (8 X 2112.01 MHz CPU s)

CPU Caches:

L1 Data 32 KiB (x4)

L1 Instruction 32 KiB (x4)

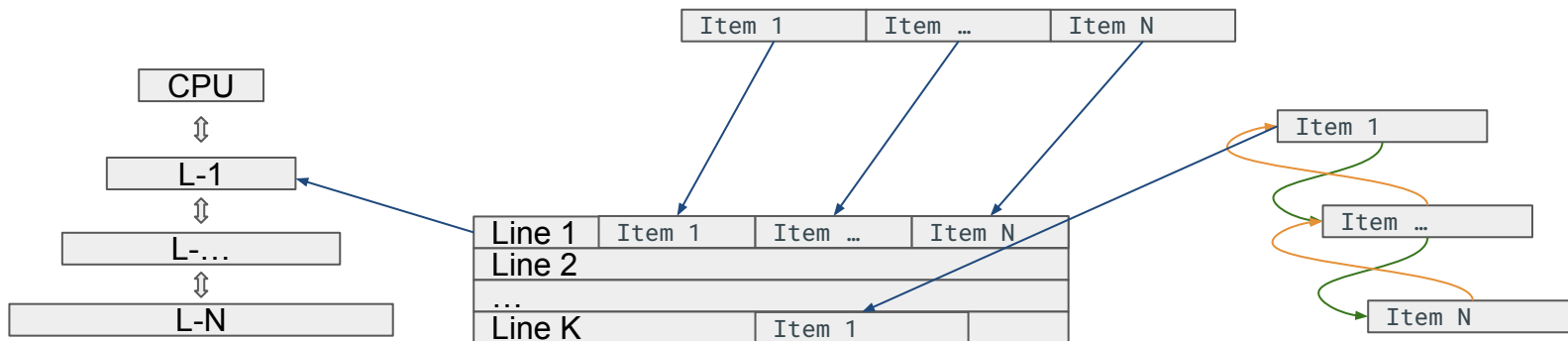
L2 Unified 256 KiB (x4)

L3 Unified 8192 KiB (x1)

Memory access. Caches

- Continuous access to continuous data is the best-case scenario for caches
- Nodes of `std::list` are allocated in arbitrary memory locations and thus have bad locality. Cachegrind emulation results (input size is 100000):

Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw	function
1,800,017	2	2	200,003	49,493	0	400,005	0	0	int std::accumulate(std::_List_const_iterato,...)
1,800,017	1	1	200,003	6,252	0	400,005	0	0	int std::accumulate(__normal_iterator, ...)



Code complexity. Iron Law of Performance

- Iron Law of Performance (by Douglas Clark):

CPU Time = # of instructions to be executed * cycles per instruction * cycle time

- Memory accessing instructions typically have higher latency
- More instructions and more memory accesses will increase execution time
(on the same platform)

Code complexity. Generated code

- Code generated for iteration
- `std::list` needs to access memory to get address of a next item, for `std::vector` only immediate offset is added to pointer

std::vector	std::list
<pre>.L23: add r12d, DWORD PTR [rax] add rax, 4 cmp rcx, rax jne .L23</pre>	<pre>.L8: add r12d, DWORD PTR [rax+16] mov rax, QWORD PTR [rax] cmp rax, rbp jne .L8</pre>

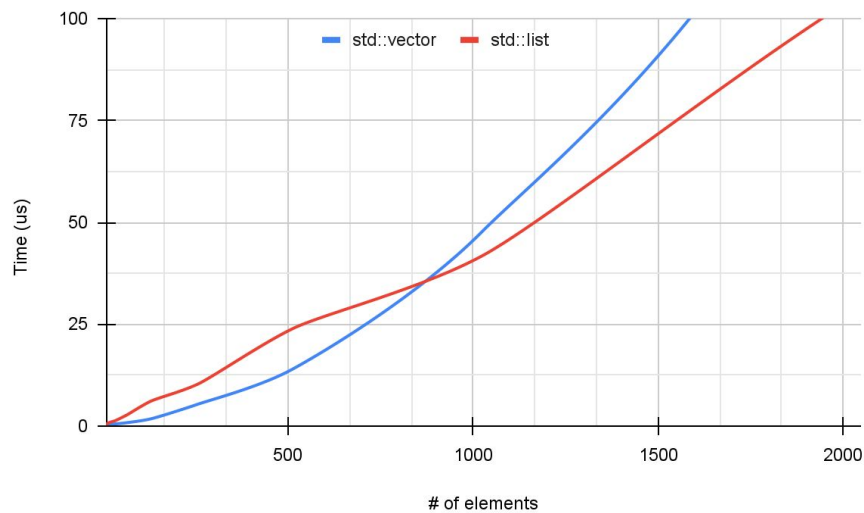
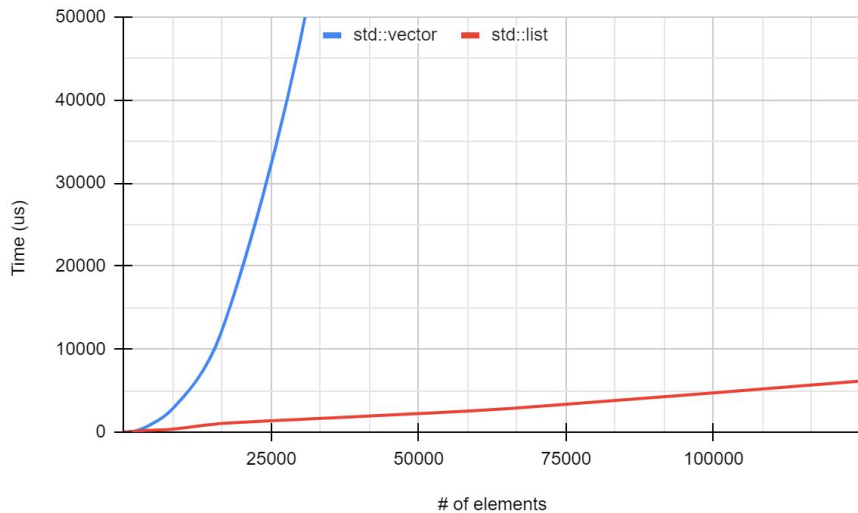
Insertion into front of `std::vector` and `std::list`

- Brute force reversion of the sequence
- Insertion of n items (`uint32_t`) is measured (not an individual insertion)
- Worst case for insertion into `std::vector` as the complexity is $O(n^2)$
- For `std::list` complexity is $O(n)$ as for n times of $O(1)$

```
std::list<uint32_t> container;  
for (auto const& it : data_to_insert) {  
    container.insert(container.begin(), it);  
}
```


Insertion into front of `std::vector` and `std::list`

- Up to ~800 items `std::vector` outperforms `std::list` despite higher complexity
- Allocation is performed for each node within `std::list` while in `std::vector` small blocks can still be effectively shifted



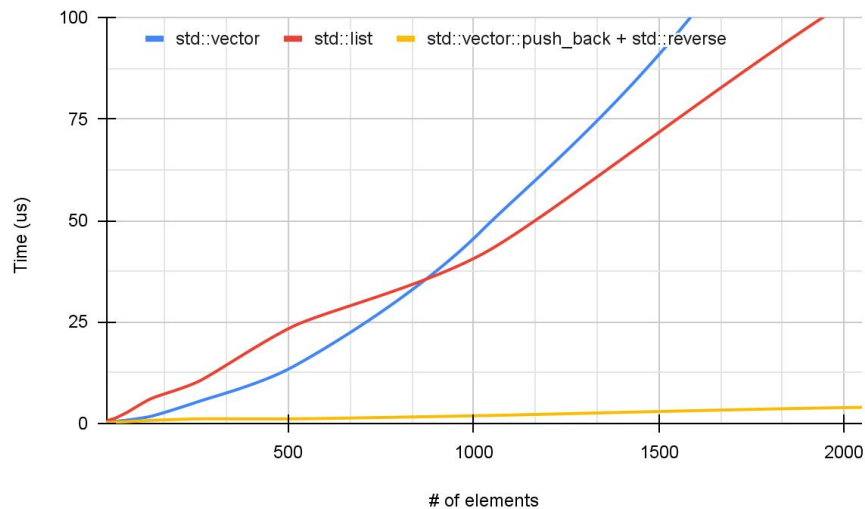
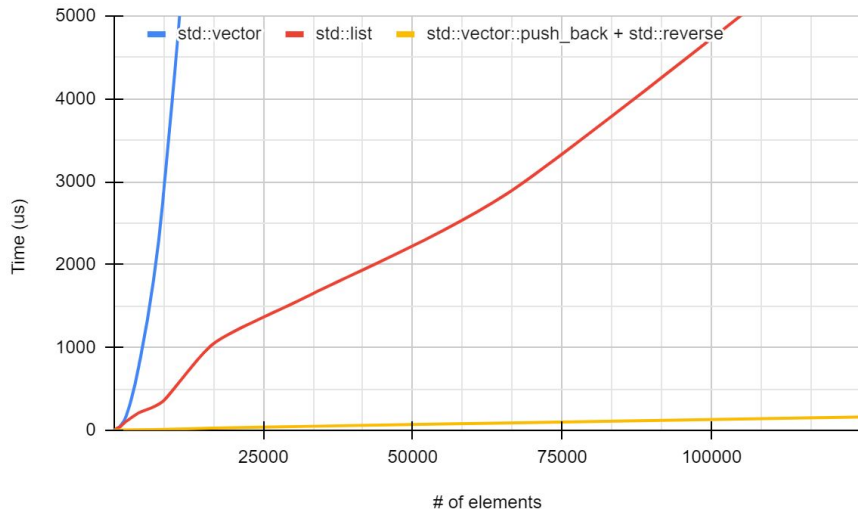
Insertion into front. “Post-processing”

- Insertion can be performed in two steps
- First, `std::vector::push_back` can be used to populate a vector
- Second, `std::reverse` can be applied

```
std::vector<uint32_t> container;  
for (auto const& it : data_to_insert) {  
    container.push_back(it);  
}  
std::reverse(container.begin(), container.end());
```

Insertion into front. “Post-processing”

- `std::vector::push_back` doesn't lead to copying/moving of existent items if no reallocation is required
- `std::reverse` performs $n/2$ swaps



Insertion into front. Parallel algorithm

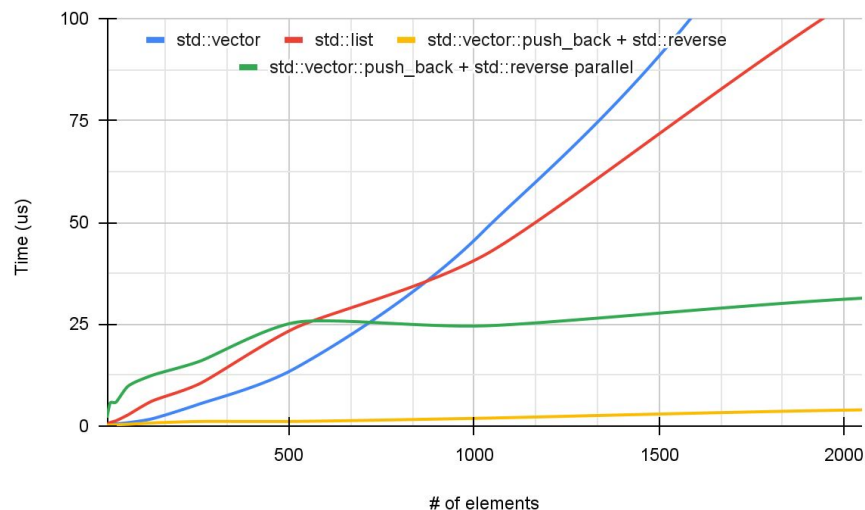
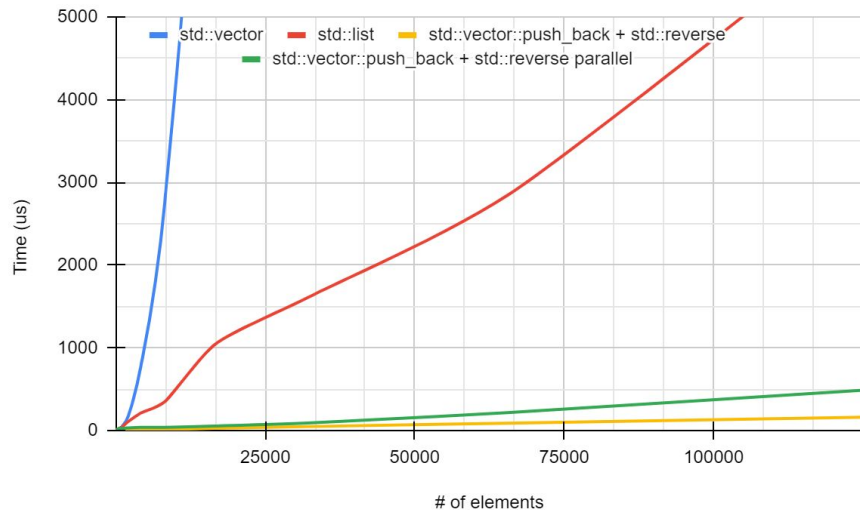
- C++17 standard introduces overloads supporting unsequenced execution for large subset of STL algorithms
- The overloads accept first parameter that specifies what execution policy should be applied
- `std::execution::par` requests parallel implementation

```
std::vector<uint32_t> container;  
for (auto const& it : data_to_insert) {  
    container.push_back(it);  
}  
std::reverse(std::execution::par, container.begin(), container.end());
```

- Most of the parallelism related details are Standard Library implementation specific

Insertion into front. Parallel algorithm

- The parallel version demonstrates decrease in performance

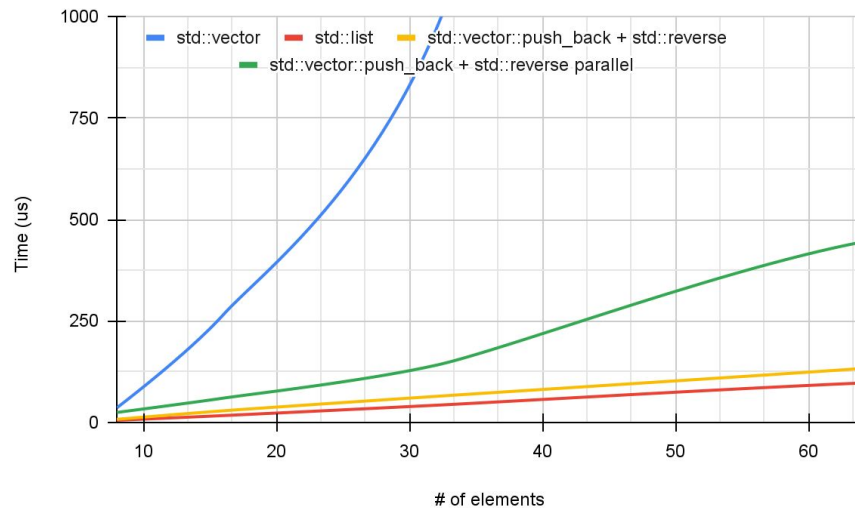
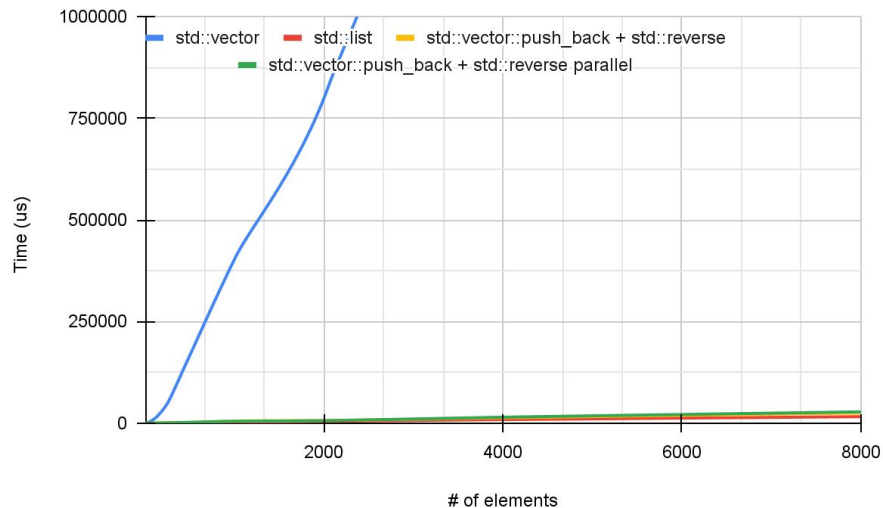


Insertion into front. Parallel algorithm

- Not all operations can be effectively parallelized
- Multi-core parallelism improves performance if parallel operations are CPU intensive and access to shared resources is minimized
- Overhead associated with parallelism orchestration can exceed benefits of parallel execution
- Memory access to adjacent locations mapped to the same cache line from multiple cores can lead to issues such as “false sharing”

Insertion into front. Large objects

- Non-movable objects of 512 bytes size
- As the copy/swap operations are expensive modification of `std::vector` has disadvantage
- `std::list` outperforms other options on all data set sizes



Insertion into front. Summary

- Operations with higher algorithmic complexity can outperform seemingly faster operations for particular data set size
- If use case allows, population of the container according to more performant pattern with subsequent transformation (`std::vector::push_back + std::reverse` in the example) can provide significant speedup
- Parallel algorithms should be applied with caution as they can increase execution time. They can't be considered as simple drop-in replacements and their applicability should be evaluated

Factors of performance. Summary

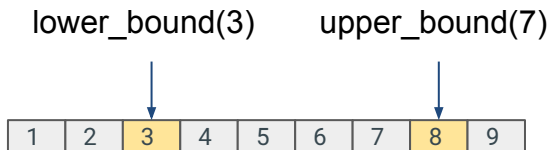
- Time complexity of algorithms for data organization and processing
 - Memory data access patterns (cash efficiency for systems where it is relevant)
 - Generated code complexity
 - Memory allocation patterns
-
- Nature of stored elements (cheap copy/movable, static footprint)
 - Potential for parallelization

Effective design should consider the individual container properties and usage scenarios to find proper application patterns

Sorted sequence

- Sorted sequence allows efficient queries of a data within a range
- `std::lower_bound` and `std::upper_bound` used to find bounds [from, to):

```
std::vector<uint32_t> sorted_vector{...};  
auto const it_from = std::lower_bound(sorted_vector.begin(), sorted_vector.end(), from);  
auto const it_to = std::upper_bound(sorted_vector.begin(), sorted_vector.end(), to);  
  
auto const result = std::accumulate(it_from, it_to, 0u);
```



- `std::set` has similar member functions
- Algorithmic complexity is $2 * O(\log n) + O(\text{number of items in range})$

Sorted sequence. Benchmark

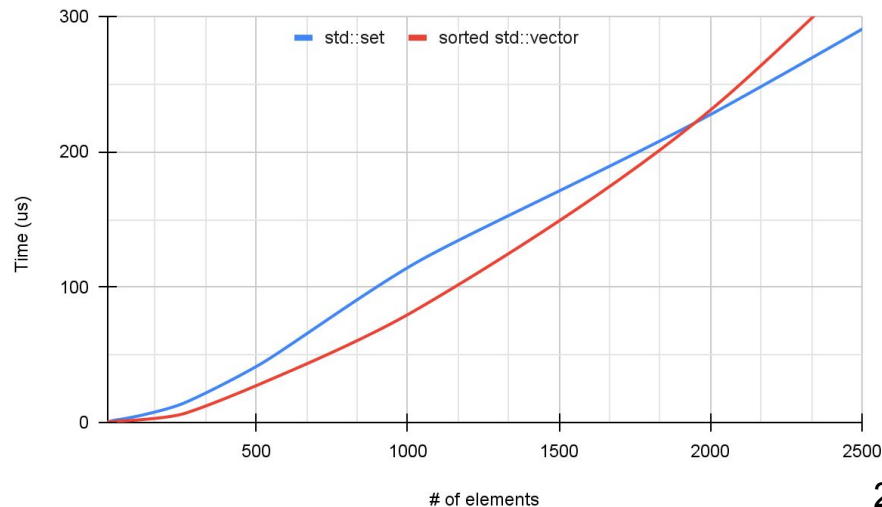
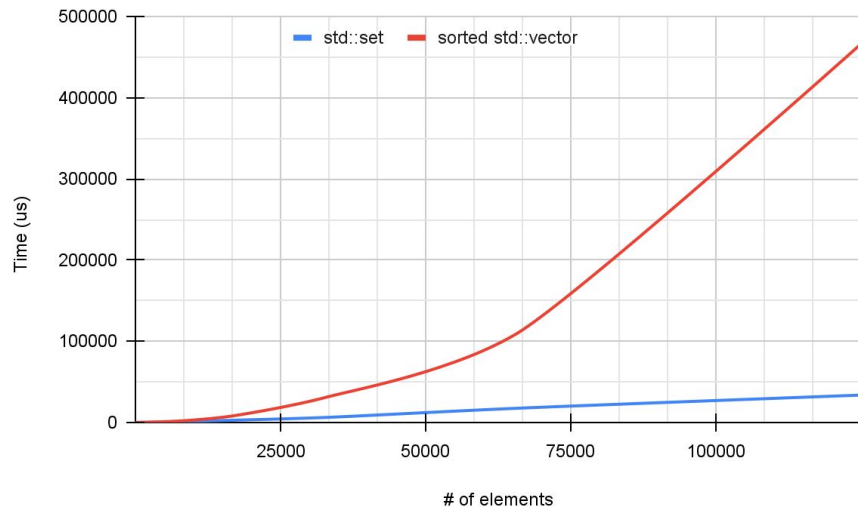
- All containers hold `uint32_t`
- First measurements are for insertion of `n` random items into containers maintaining sorted order (`std::set` and sorted `std::vector`)

```
std::set<uint32_t> container;
for (auto const& it : random_data) {
    container.insert(it);
}

std::vector<uint32_t> container; // will contain unique sorted values
for (auto const& it : random_data) {
    auto const position
        = std::lower_bound(container.begin(), container.end(), it);
    if (position == container.end() || *position != it)
        container.insert(position, it);
}
```

Sorted sequence. Insertion

- Similar to insertion into front benchmark. Node-based `std::set` outperforms on larger number of items `std::vector` due to overhead inflicted by shifting
- However, up to ~1900 items sorted `std::vector` still has advantage due to higher cost of node allocation in `std::set`



Sorted sequence. Improvements for std::vector

- Improving allocation with std::vector::reserve method

```
std::vector<uint32_t> container;  
container.reserve(random_data.size());
```

- Using unordered insertion + sorting (not applicable for any use case)

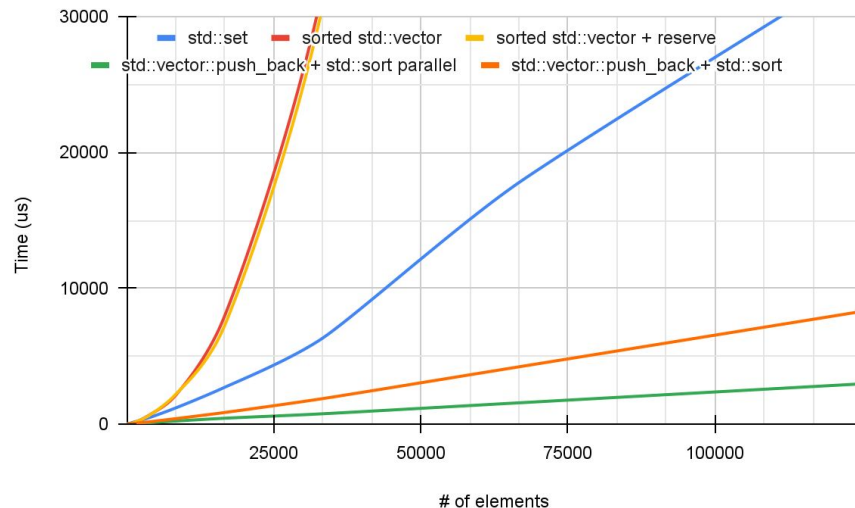
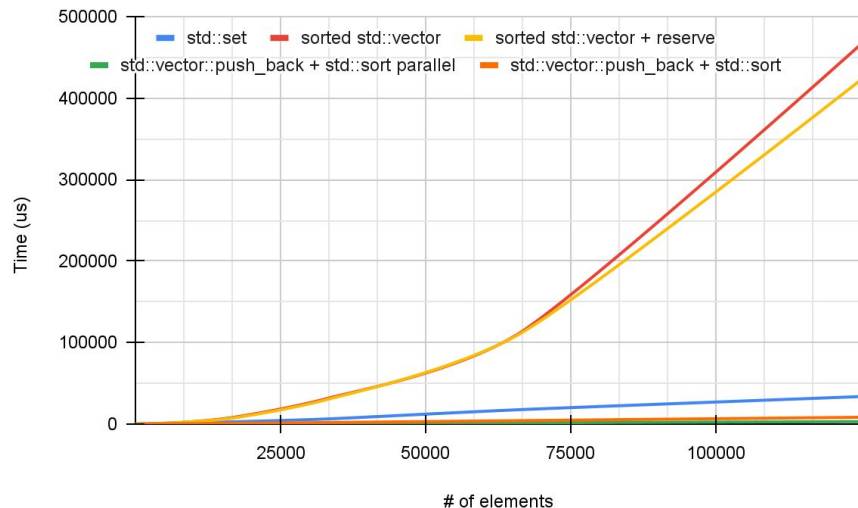
```
std::vector<uint32_t> container;  
for (auto const& it : random_data) {  
    container.push_back(it);  
}  
std::sort(container.begin(), container.end());
```

- Parallel sort as further improvement attempt

```
std::sort(std::execution::par, container.begin(), container.end());
```

Sorted sequence. Improvements for `std::vector`

- Preallocation (`reserve`) improves timing especially for large data set
- “Post-processing” (`std::sort` after `std::vector::push_back`) demonstrates significant speedup
- Parallel version of `std::sort` improves performance even further



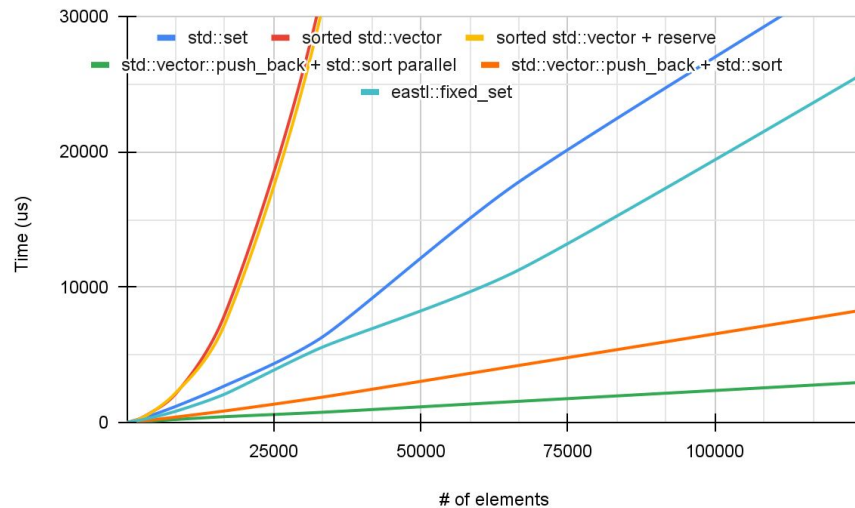
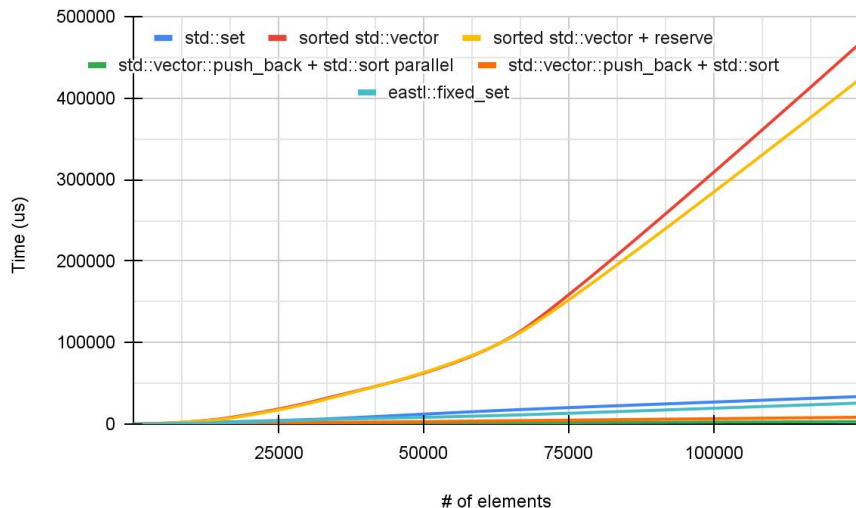
Sorted sequence. Improvements for `std::set`

- Allocation issues can be mitigated using custom allocation
- Alternative implementation with improved allocation can be used
- EASTL - Electronic Arts Standard Template Library
(<https://github.com/electronicarts/EASTL>)
- Contains fixed capacity containers and implementation of standard algorithms
- Implementation allocates new nodes in the continuous storage
- Can be used without heap allocation (suitable for Real Time/Embedded)

```
template <typename Key
    , size_t nodeCount
    , bool bEnableOverflow = true
    , typename Compare = eastl::less<Key>
    , typename OverflowAllocator = EASTLAllocatorType>
class fixed_set;
```

Sorted sequence. `eastl::fixed_set`

- Allocation pattern in `eastl::fixed_set` improves performance compared to `std::set`
- Best performance after `std::vector` with “post-processing”



Sorted sequence. `std::vector` alternative

- Sorted `std::vector` with binary search looks and behaves like flattened set
- `boost::container::flat_set` adapter from Boost container library (<https://github.com/boostorg/container>) provides set interface using random access container as a storage backend (`boost::container::vector`)

```
template <class Key, class Compare = std::less<Key>, class AllocatorOrContainer = new_allocator<Key> >
class flat_set : public dtl::flat_tree<Key, dtl::identity<Key>, Compare, AllocatorOrContainer> {
    std::pair<iterator, bool> insert(const value_type &x);
    bool contains(const key_type& x) const;
    iterator lower_bound(const key_type& x);
    iterator upper_bound(const key_type& x);
    iterator find(const key_type& x);
    //...
};
```

- `flat_multiset`, `flat_map`, `flat_multimap` are also available

Sorted sequence. `boost::flat_set`

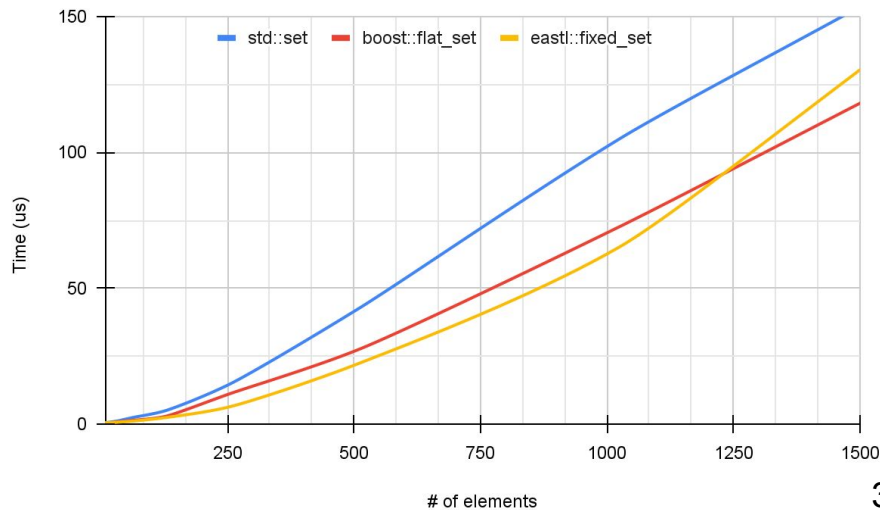
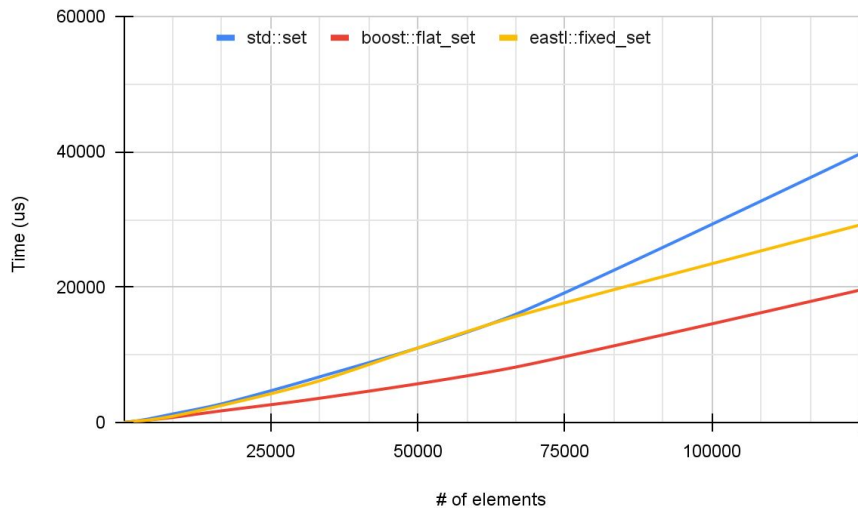
- For `boost::flat_set(*)` it is possible to extract internal and attach external sequence

```
boost::container::flat_set<uint32_t> container;  
auto sequence = container.extract_sequence();  
sequence.reserve(random_data.size());  
for (auto const& it : random_data) {  
    sequence.push_back(it);  
}  
  
std::sort(std::execution::par, sequence.begin(), sequence.end());  
container.adopt_sequence(  
    boost::container::ordered_unique_range, std::move(sequence));
```

- * Correct fully qualified name is `boost::container::flat_set`.
“container” part is omitted for shortness.

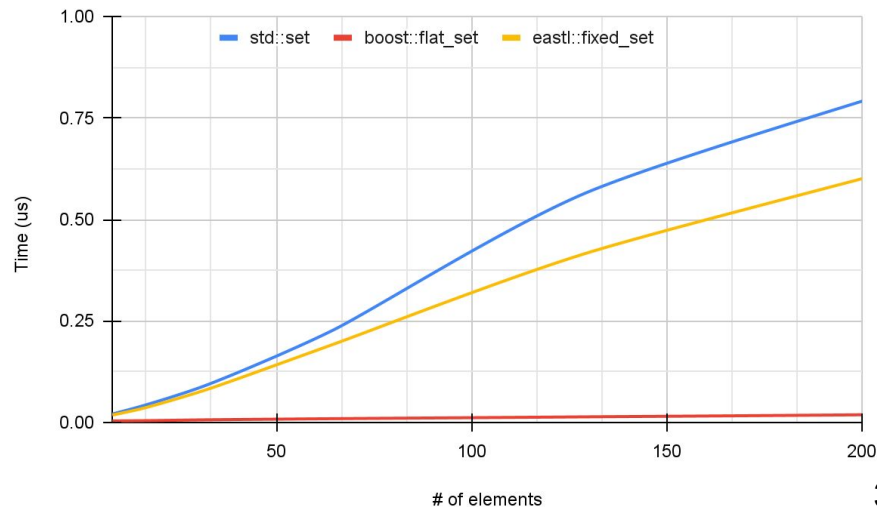
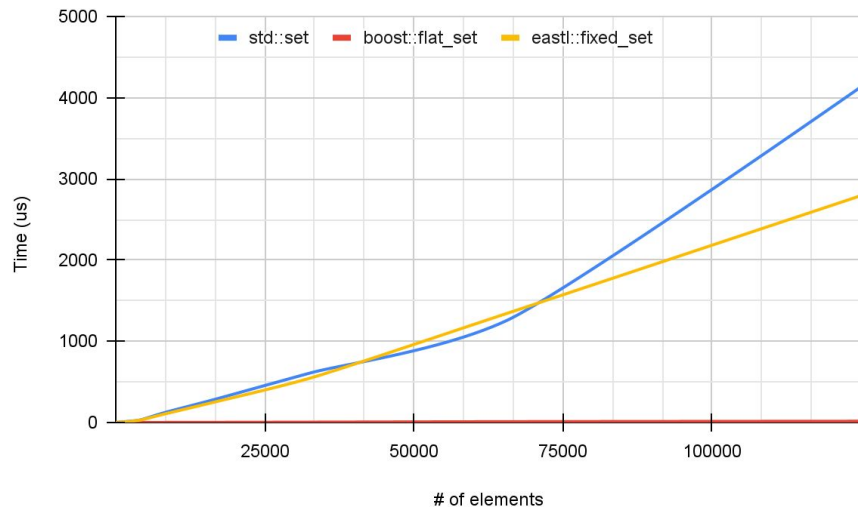
Sorted sequence. Search

- Due to random-access `boost::flat_set` performs faster on large data set
- On smaller data set `eastl::fixed_set` demonstrates an advantage



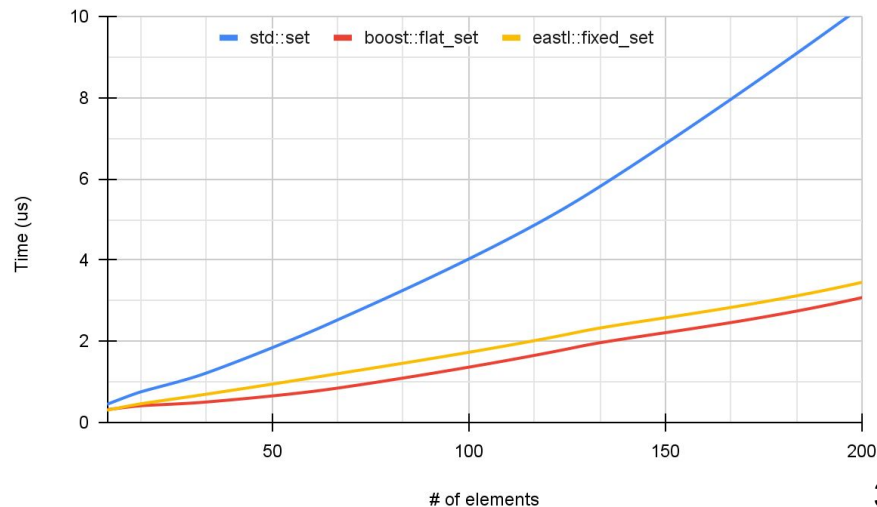
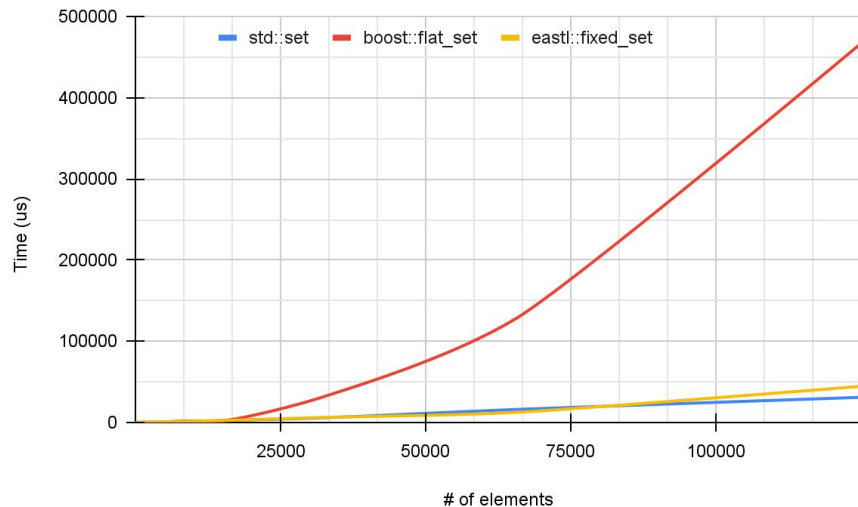
Sorted sequence. Traversal

- Continuous access to the items in `boost::flat_set` is the fastest option
- `eastl::fixed_set` shows better performance than `std::set` due to better data locality



Sorted sequence. Deletion

- Deletion in sequential containers has similar issues as insertion: shifts are performed for the items after the deleted one
- On smaller data set faster lookup for object to delete provides advantage for containers with continuous storage



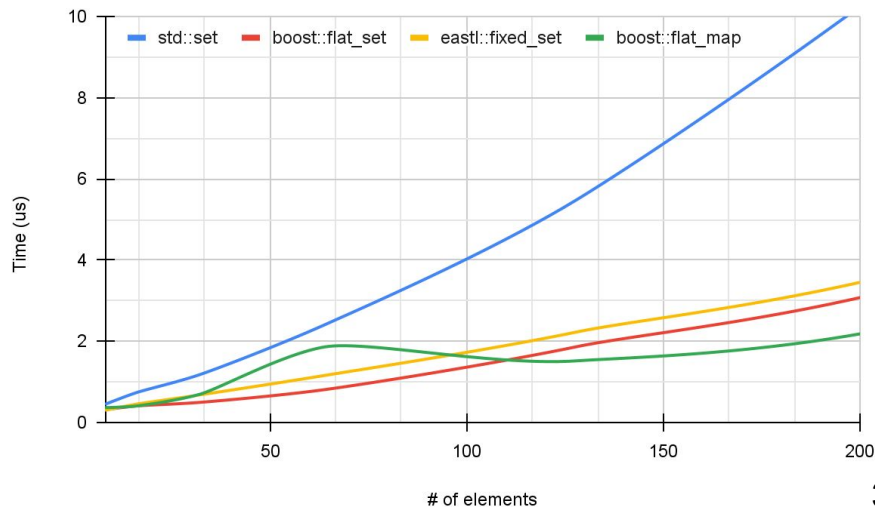
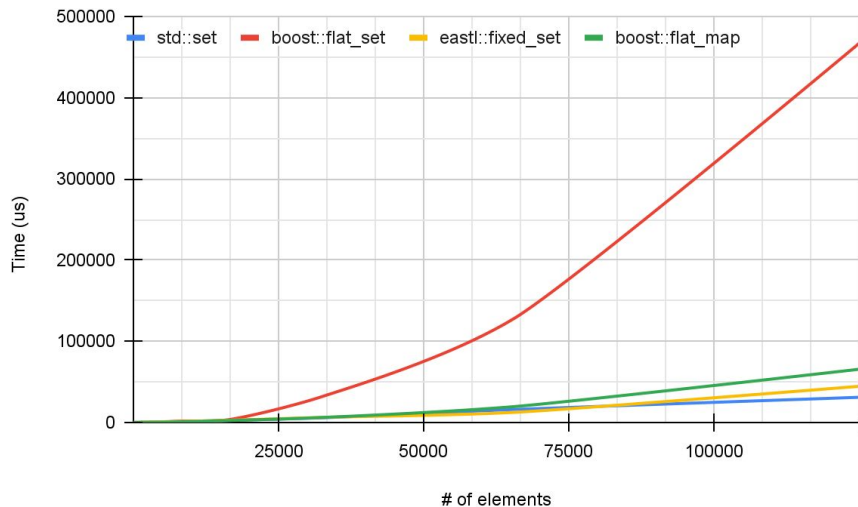
Sorted sequence. Deferred removal

- `boost::flat_map` can be used to store bool flag to indicate discarded items
- Erase/remove idiom can be applied if necessary

```
boost::container::flat_map<uint32_t, bool> container;  
//...  
auto erase_count{0u};  
for (auto const& it : data_to_delete) {  
    container[it] = true;  
    erase_count++;  
  
    if (erase_count == erase_threshold) {  
        erase_count = 0;  
        container.erase(  
            std::remove_if(container.begin(), container.end()  
                , [](auto const& it) { return it.second; } ), container.end());  
    }  
}
```

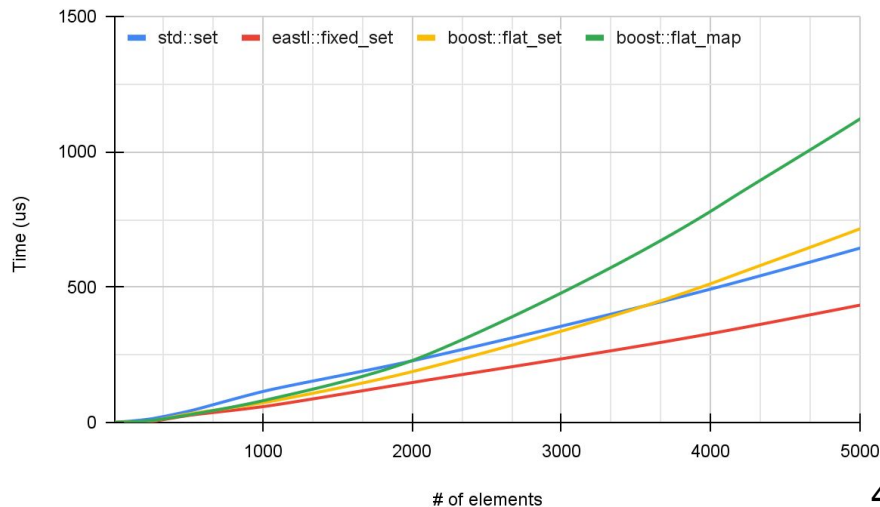
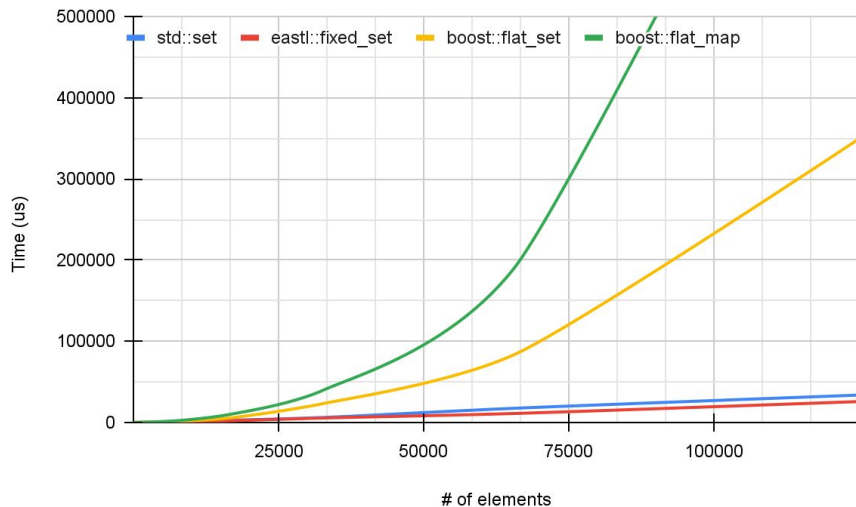
Sorted sequence. Deferred removal

- Optimized version significantly lowers deletion overhead as the number of shifts is reduced (all discarded items are removed in a single pass)
- Depicted version removes bulks of 100 items



Sorted sequence. Deletion optimization trade-off

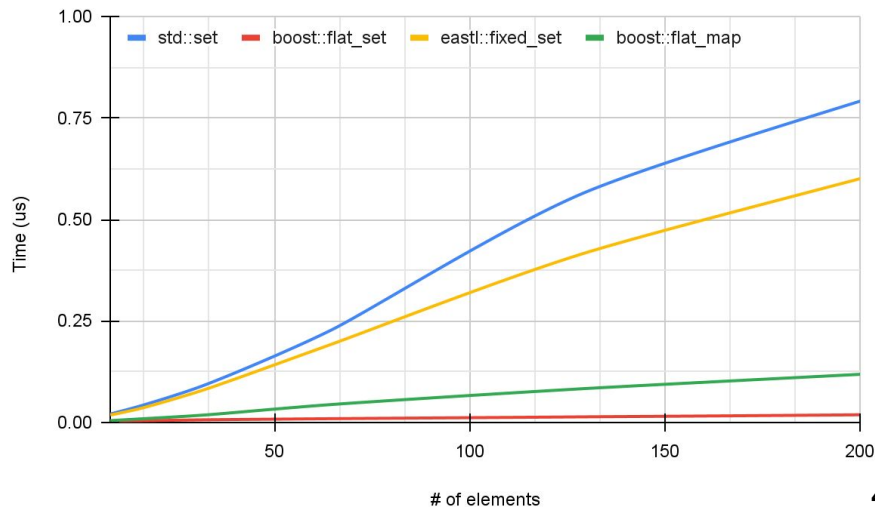
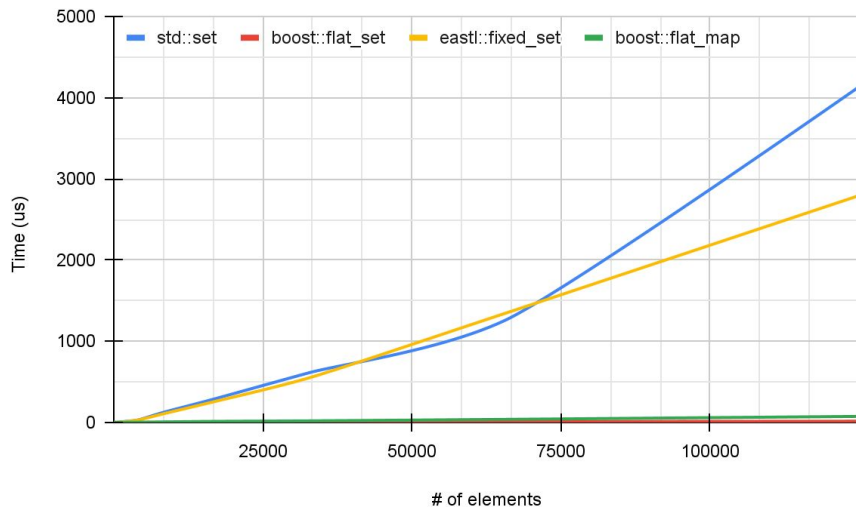
- Insertion into `boost::flat_set<uint32_t, bool>` has similar pattern as for `boost::flat_set`, but as the footprint of the objects increased the timing is worse
- Similar optimization with external `std::sort` for internal data is possible for `boost::flat_map` as well



Sorted sequence. Deletion optimization trade-off

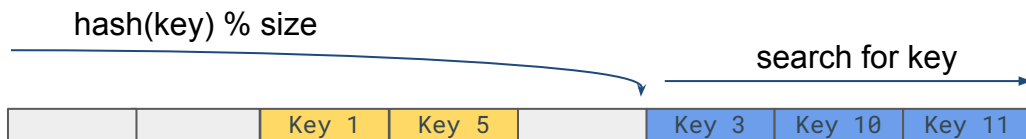
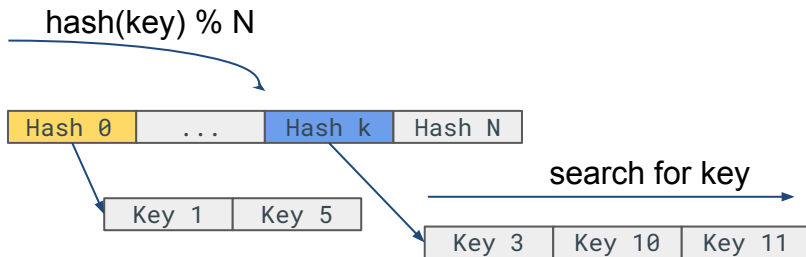
- Iteration is slower as additional logic is needed to check if the item is not discarded

```
auto const result = std::accumulate(container.begin(), container.end(), 0u,  
    [](auto const s, auto const& it) {  
        return s + (!it.second ? it.first : 0u);  
    });
```



Unordered containers

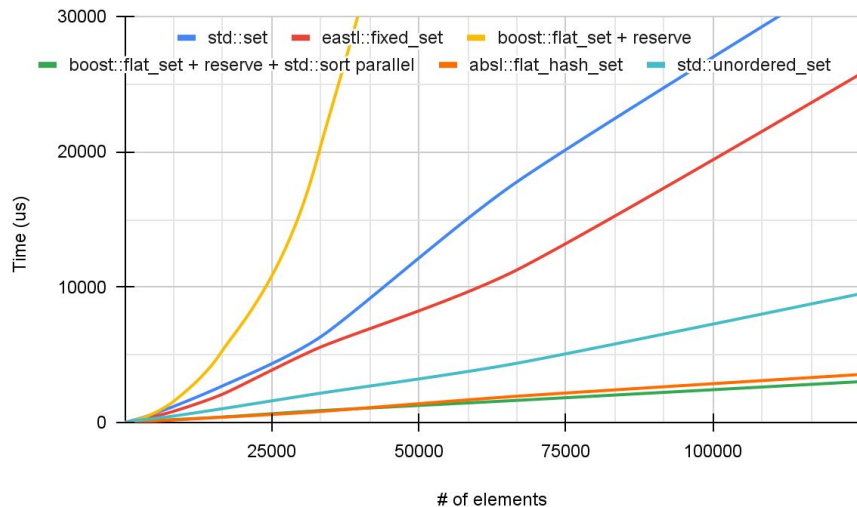
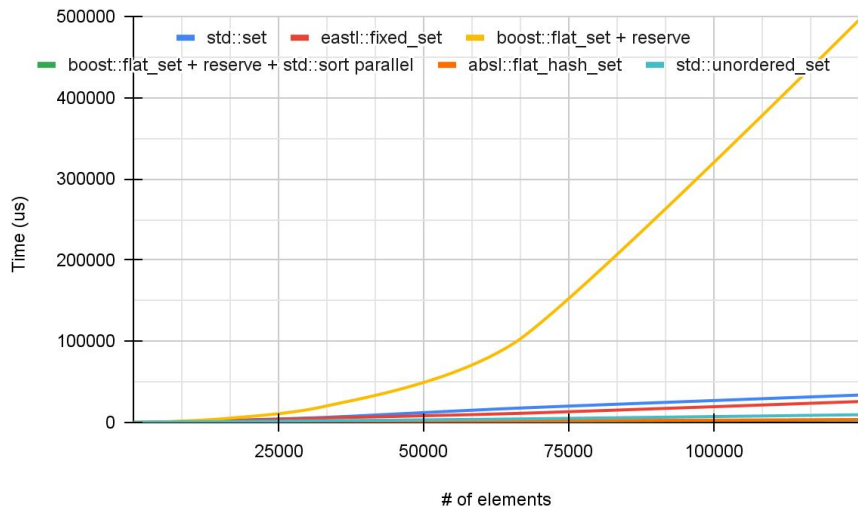
- If sorted order is not required unordered hash-based containers can be used
- Typically, $O(1)$ for insertion, deletion and search



- `std::unordered_set` and `std::unordered_map` - part of STL
- `absl::flat_hash_set`, `absl::flat_hash_map` - versions from abseil framework (<https://github.com/abseil/abseil-cpp>) using flat storage model

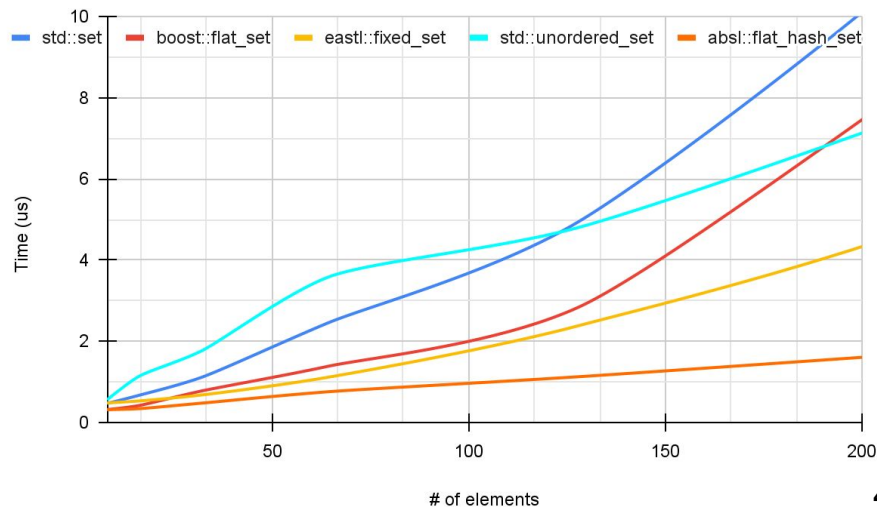
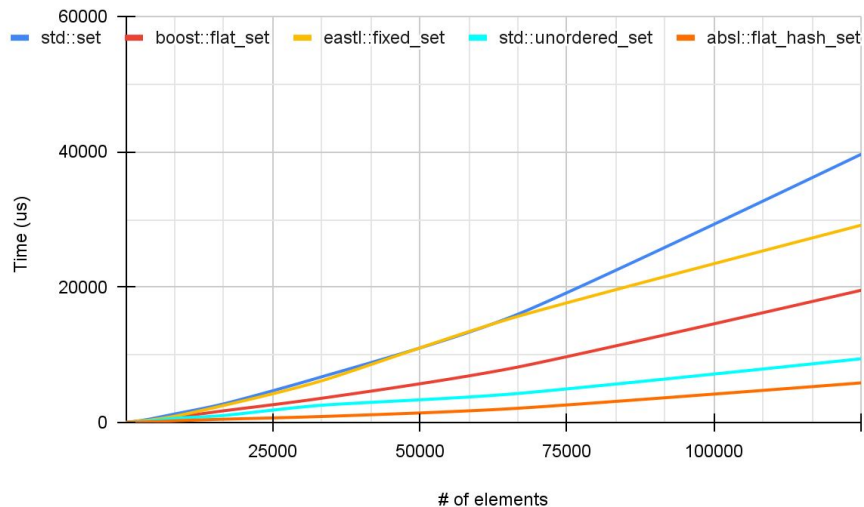
Unordered containers. Insertion

- `absl::flat_hash_set` outperforms `std::unordered_set`
- Additionally, `absl::flat_hash_set::reserve` is available to preallocate memory



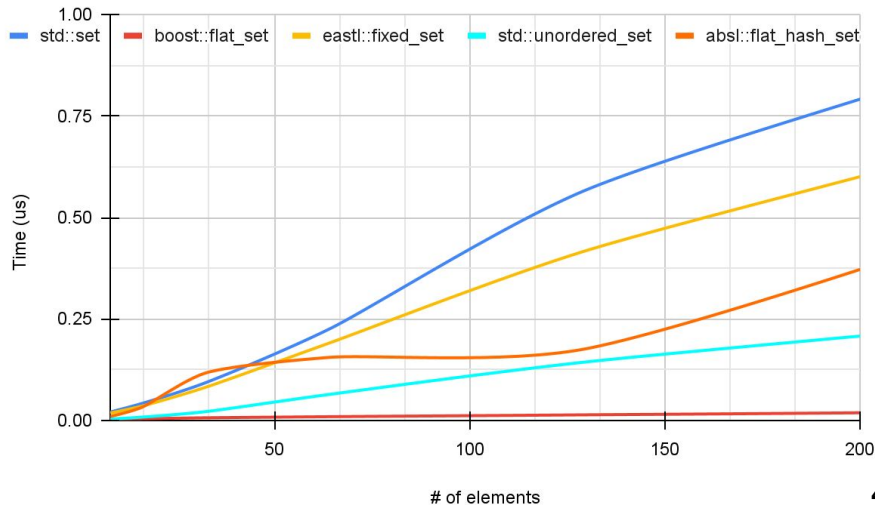
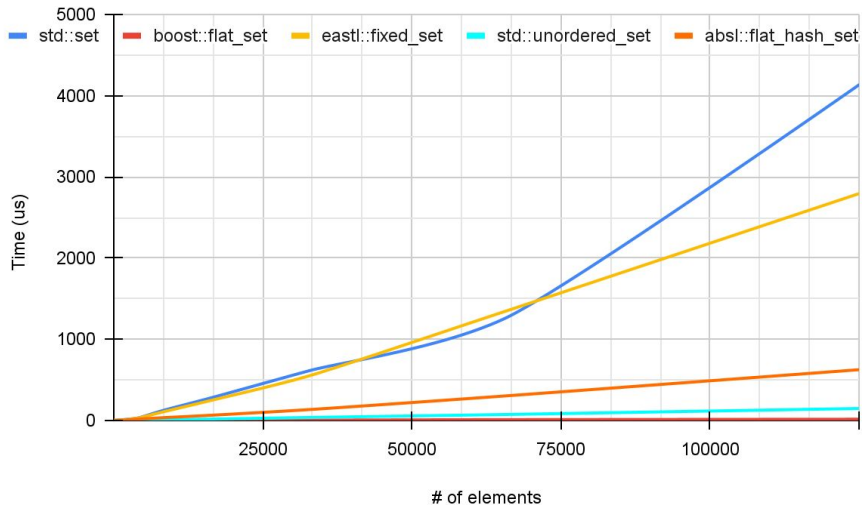
Unordered containers. Search

- Search performance of unordered containers is superior compared to other options
- With smaller data set size results vary so caution is required



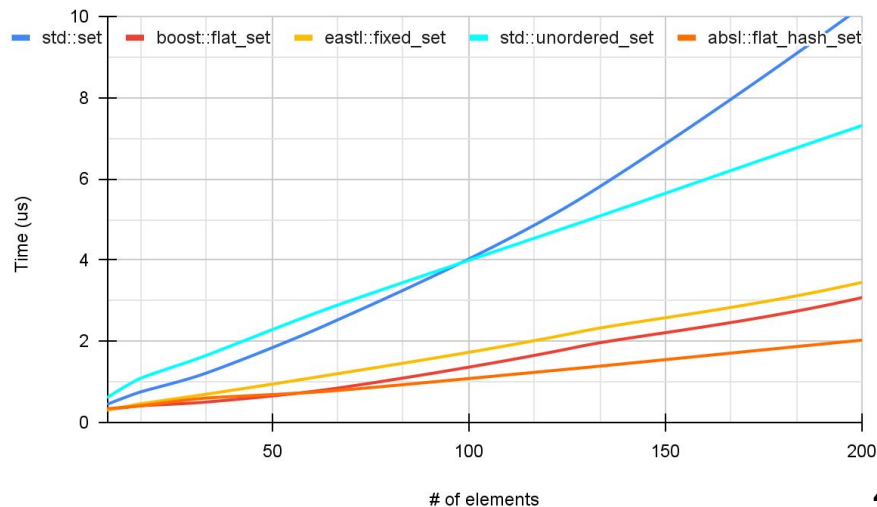
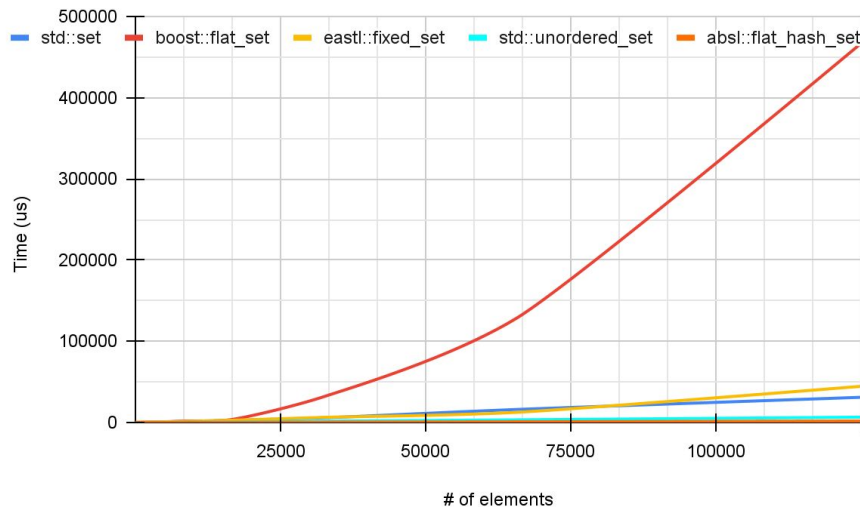
Unordered containers. Traversal

- `std::unordered_set` has relatively good traversal performance



Unordered containers. Deletion

- Like addition, deletion performance for unordered containers is superior in the test
- Similar to search benchmark caution is required for small data set sizes



Container combination. The case

- Given n records containing string label
- Labels are not unique and number of unique labels significantly smaller than n
- It is required to count records having labels within specified interval bounds (lexicographically compared)

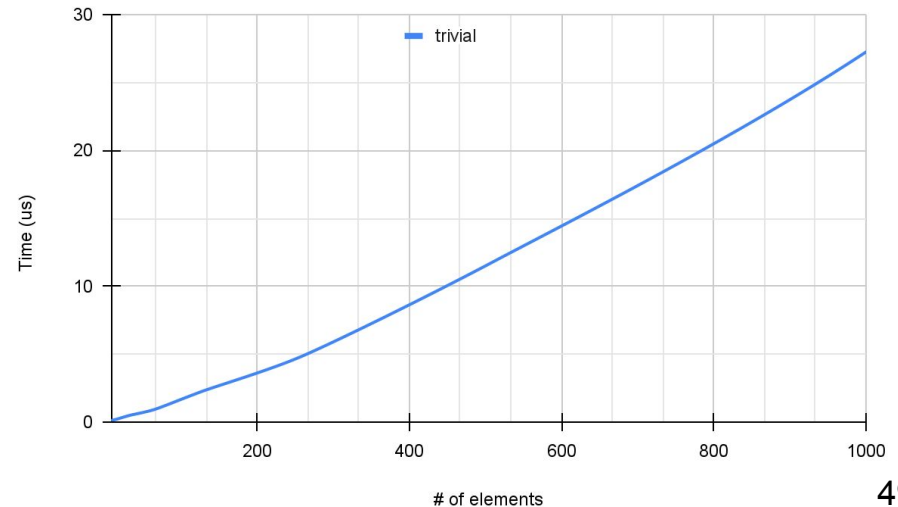
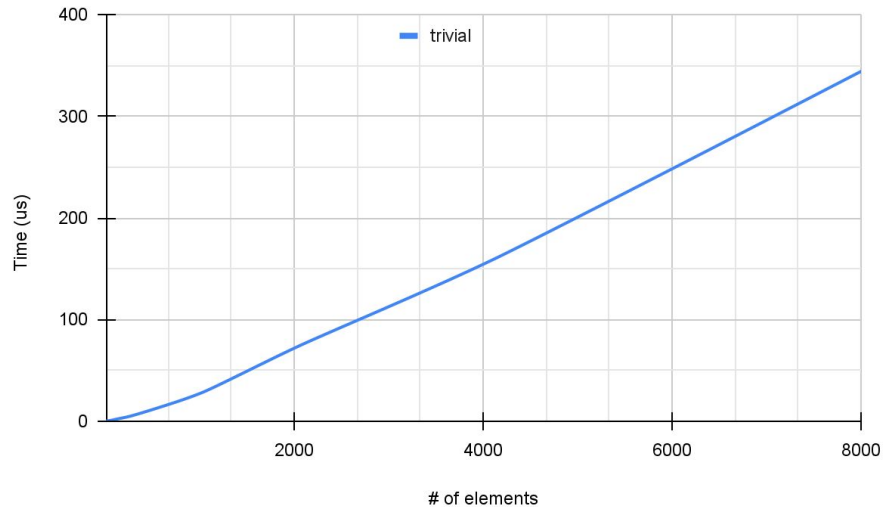
Container combination. Trivial approach

- Trivial `std::count_if` with string comparison

```
struct record {  
    std::string label;  
    size_t value;  
};  
  
std::vector<record> container{...};  
  
auto const result = std::count_if(  
    container.begin(), container.end(), [](auto const& it) {  
        return it.label >= range_from && it.label < range_to;  
    });
```


Container combination. Trivial approach

- Initial measurements as optimization starting point



Container combination. Avoiding indirection

- `eastl::fixed_string` to store the label “inline”
- Maximum length should be specified
- Increases the static footprint of the object so not always acceptable

```
using label_string = eastl::fixed_string<char, 40, false>;

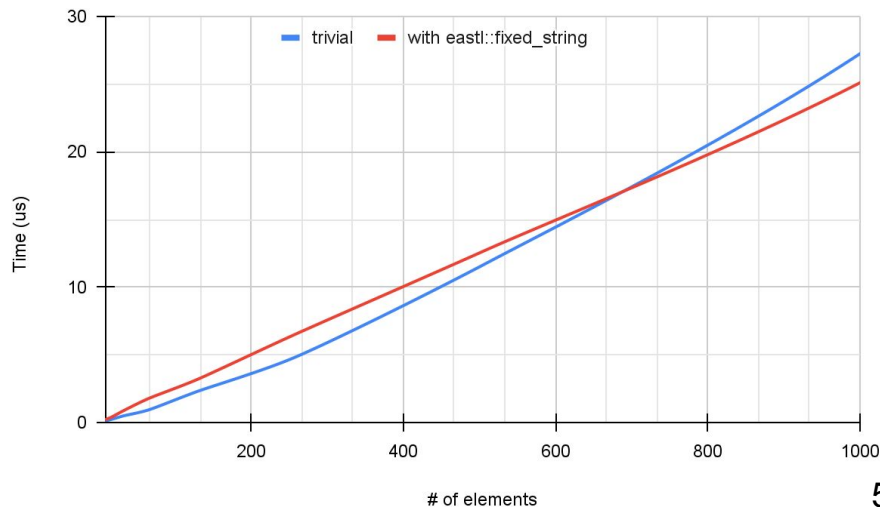
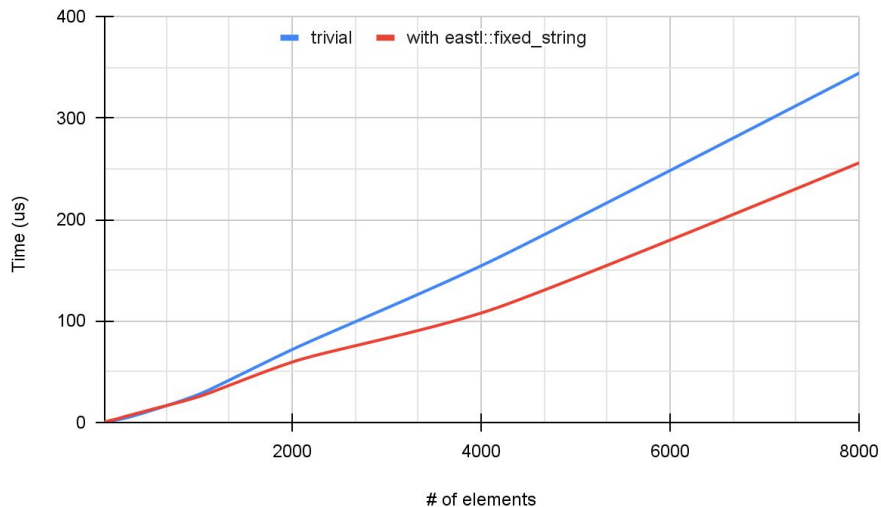
struct record_with_fixed_string {
    label_string label;
    size_t value;
};

std::vector<record_with_fixed_string> container{...};

auto const result = std::count_if(
    container.begin(), container.end(), [](auto const& it) {
        return it.label >= range_from && it.label < range_to;
    });
```

Container combination. Avoiding indirection

- Storing the string “inline” avoids indirection and provides better cache locality
- Effect of the improvement becomes visible for the large number of records



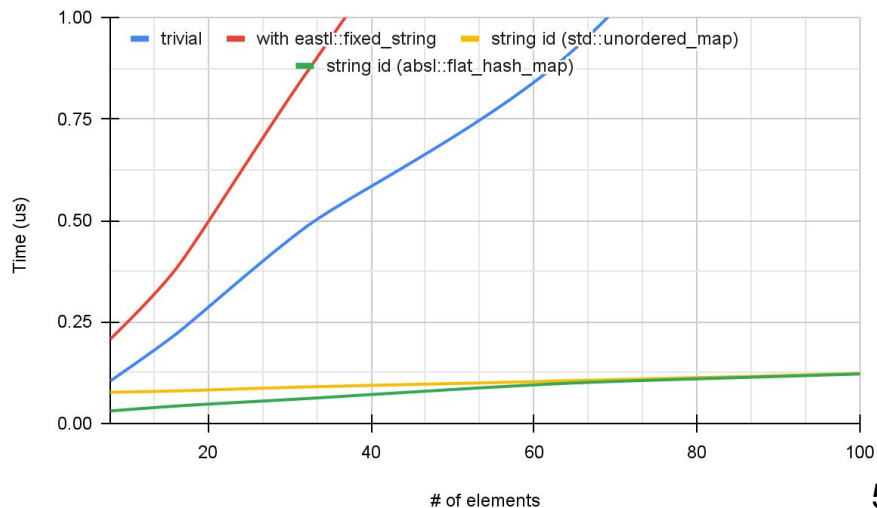
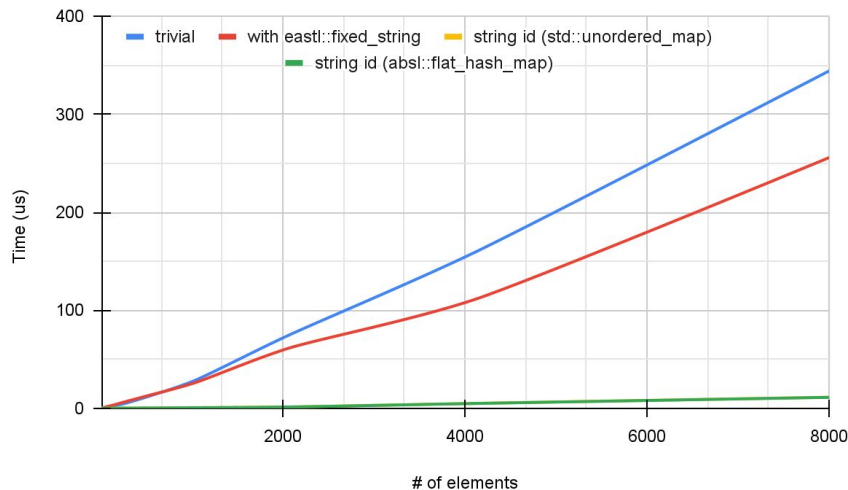
Container combination. Avoiding string comparison

- Assign integer identifier for each label according to it's sorted order position
- Store label id instead of string
- Requires precomputation (id for labels should be assigned)

```
struct record_with_label_id {  
    uint32_t label_id;  
    size_t value;  
};  
  
std::unordered_map<std::string, uint32_t> label_id_mapping;  
//absl::flat_hash_map<std::string, uint32_t> label_id_mapping;  
  
auto const id_from = label_id_mapping.at(range_from);  
auto const id_to = label_id_mapping.at(range_to);  
auto const result = std::count_if(container_label_id.begin(), container_label_id.end(),  
    [id_from, id_to](const auto& it) {  
        return it.label_id >= id_from && it.label_id < id_to;  
    });
```

Container combination. Avoiding string comparison

- Avoiding string comparison significantly reduces timing
- As search in `absl::flat_hash_map` is faster the difference with implementation using `std::unordered_map` is visible up to ~60 items (search contribution to execution time is substantial for small data set).



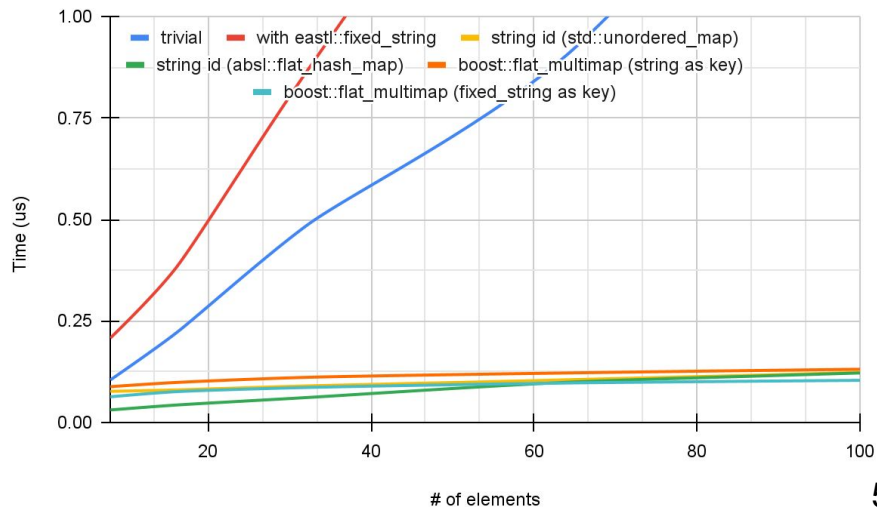
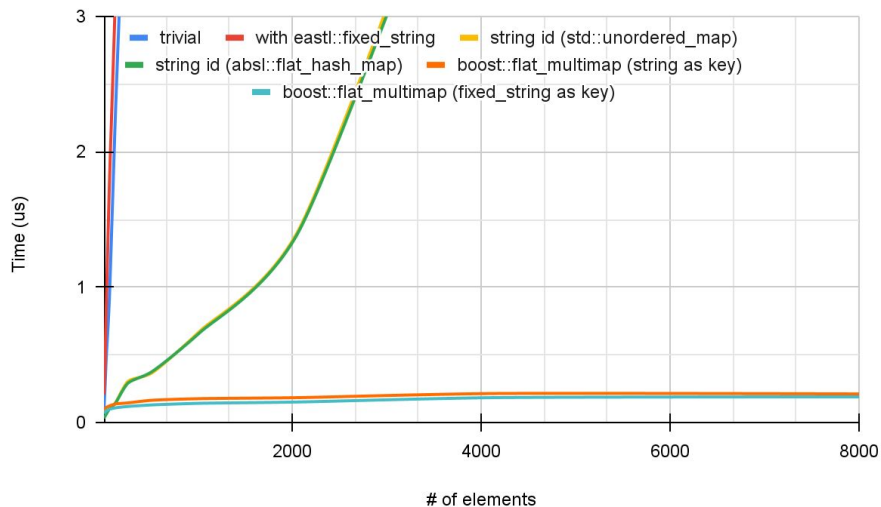
Container combination. Sorted sequence

- Use label as a key in sorted multimap
- The implementation has complexity $O(\log n)$ as binary search is used to find bounds and `std::distance` has time $O(1)$ for random access iterator
- Requires building sorted sequence (preparation can be expensive)

```
boost::container::flat_multimap<std::string, record> label_record_map;  
// using label_string = eastl::fixed_string<char, 40, false>;  
// boost::container::flat_multimap<label_string, record> label_record_map;  
  
auto const it_from = label_record_map.lower_bound(range_from);  
auto const it_to = label_record_map.upper_bound(range_to);  
auto const result = std::distance(it_from, it_to);
```

Container combination. Sorted sequence

- Offers significant improvement due to reduced number of operations
- Using `eastl::fixed_string` as a key provides extra speedup



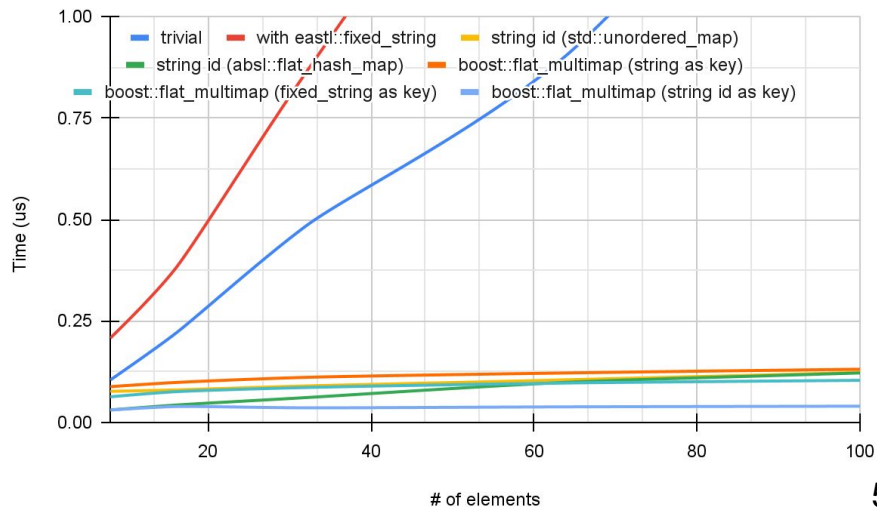
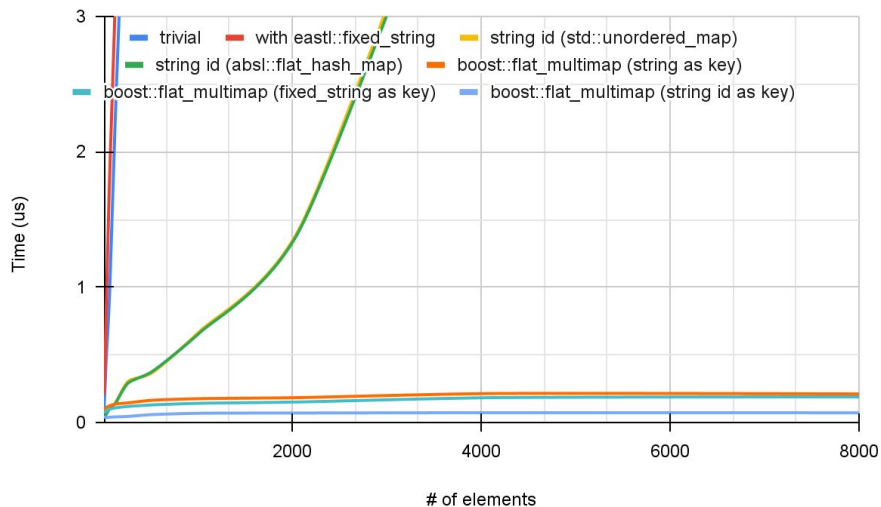
Container combination. Sorted sequence

- Further improvement is to use label id as a key for `boost::flat_multimap`

```
absl::flat_hash_map<std::string, uint32_t> label_id_mapping;  
boost::container::flat_multimap<uint32_t, record> label_id_record_map;  
  
auto const id_from = label_id_mapping.at(range_from);  
auto const id_to = label_id_mapping.at(range_to);  
auto const it_from = label_id_record_map.lower_bound(id_from);  
auto const it_to = label_id_record_map.upper_bound(id_to);  
auto const result = std::distance(it_from, it_to);
```


Container combination. Sorted sequence

- Using string id as a key in the `boost::flat_map` provides best performing implementation
- The structure is flattened, and no string comparison is executed



Container combination. Summary

- Minimization of memory indirection by flattening data structures can demonstrate substantial speedup
- Reduction of complex types into simpler ones by mapping reduces code complexity leading to reduction of execution time
- Using precomputed data structures tailored for specific access pattern allows to reduce algorithmic complexity or minimize number of required operations
- For insert/search/delete operations unordered hash-based containers can be preferable, but their performance can vary depending on data set size

Indirection

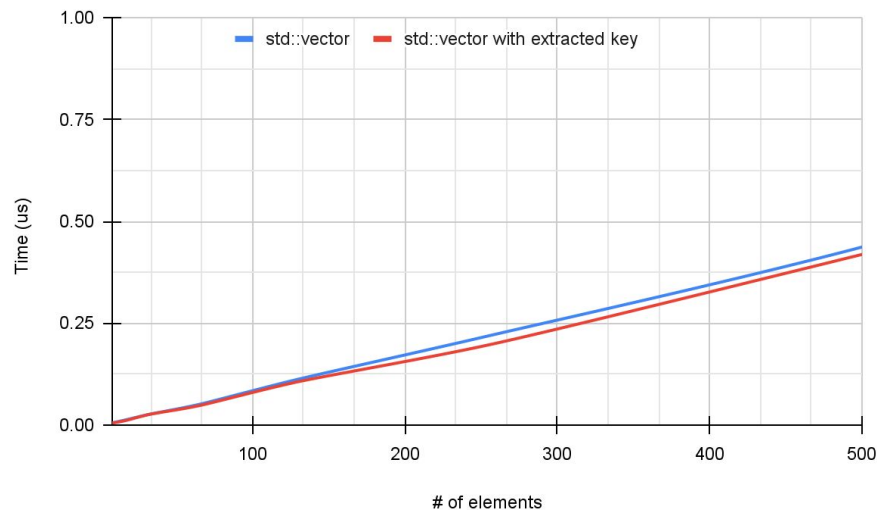
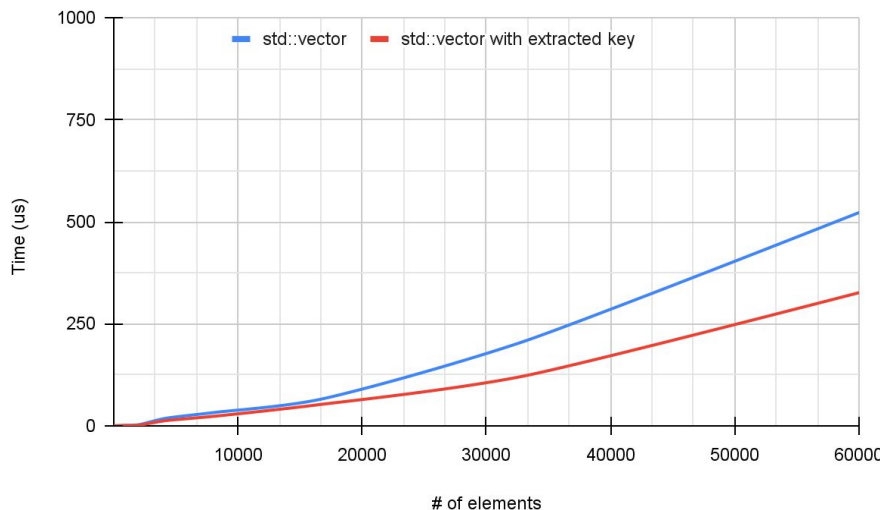
- Object is accessed indirectly. Key extracted and stored inline

```
using key_t = uint32_t;
struct record {
    key_t key;
    uint32_t value;
};
std::vector<std::unique_ptr<record>> vector;
auto const sum_v = std::accumulate(vector.begin(), vector.end(), 0u, [](auto const s, auto const& it) {
    return s + (it->key & 1 ? it->value : 0u);
});

using key_and_record = std::pair<key_t, std::unique_ptr<record>>;
std::vector<key_and_record> extracted_key_vector;
auto const sum_e = std::accumulate(extracted_key_vector.begin(), extracted_key_vector.end(), 0u,
    [](auto const s, auto const& it) {
        return s + (it.first & 1 ? it.second->value : 0u);
    });
```

Indirection

- Version with extracted key performs indirect access only if condition is met
- Minimization of indirect accesses provides performance increase



Summary

- Although some reasoning about performance can be done based on knowledge about existing containers, only benchmarking and profiling can validate the hypothesis about a performance for particular settings
- Apart from standard (STL) containers, third party alternatives can provide drop-in replacements often exhibiting better performance
- Combination of containers can complement functionality and mitigate downsides
- Separation of data preparation and data access can allow to pick best suitable patterns and containers
- C++17 parallel algorithms should be considered as they can provide speed up, but their contribution should be evaluated

Thank you!

Questions?