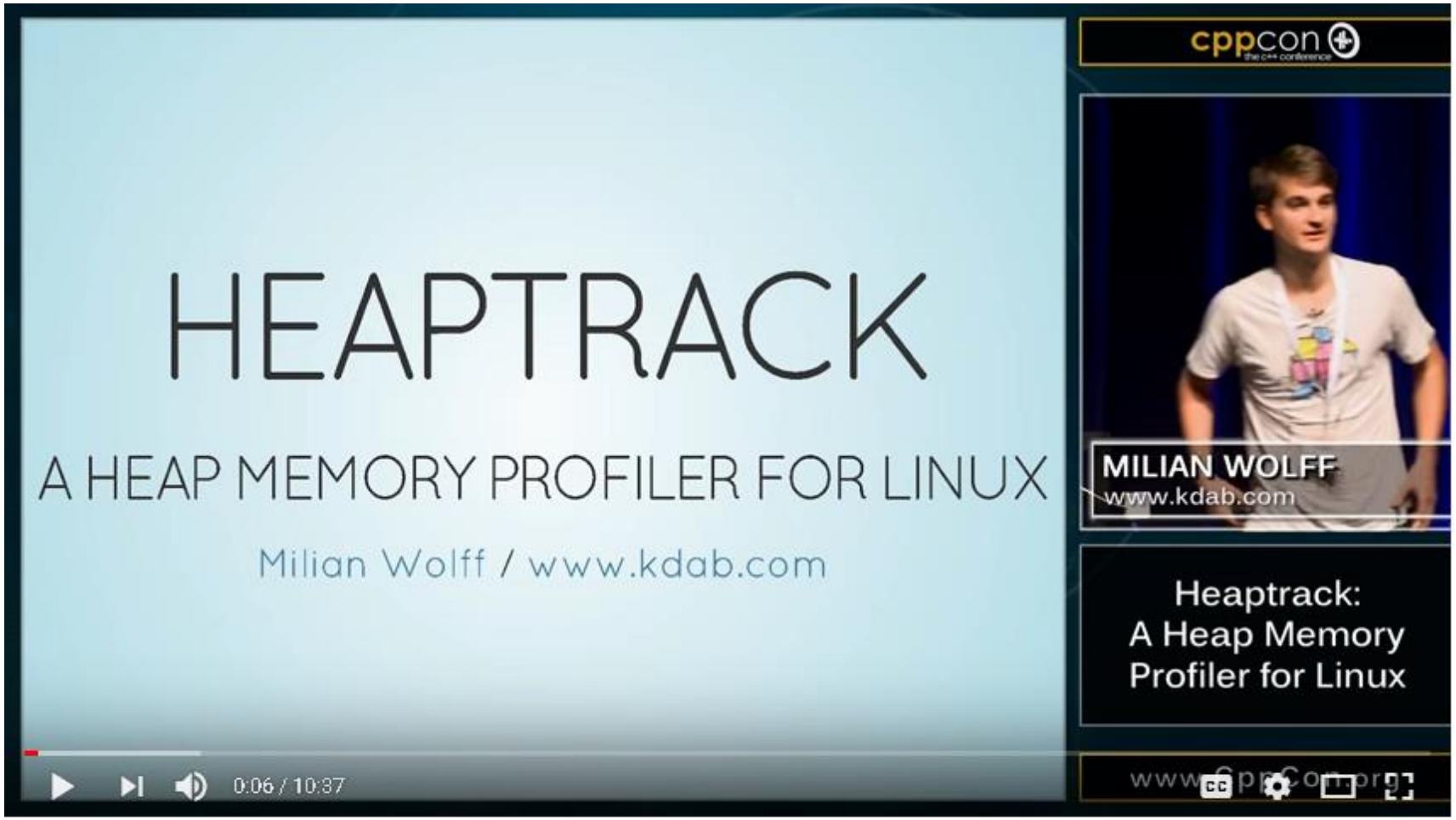
Reducing memory allocations
-
The case of C++ associative containers

Arnaud Desitter

ACCU Oxford
7 June 2018

Agenda

- Measuring memory allocations
- Memory allocations of associative containers
- How to improve them



CppCon 2015: Milian Wolff "Heaptrack: A Heap Memory Profiler for Linux"

Tip: heaptrack_gui is easy to build on a recent version of Linux (e.g. Ubuntu 18)

Source: youtube.com

C++ associative containers

	`std::map` `std::set`	`std::unordered_map` `std::unordered_set`
	Tree based	Hash based
Lookup complexity	O(log(n))	O(1) amortised, O(n) worst case
Functional requirement	A comparison function	A hash and an equality function
Source of dynamic allocations	Nodes	(Somewhat slimmer) nodes and buckets

std::set

```
#include <set>
void ex1() {
  std::set<std::size_t> set;
  constexpr std::size_t nmax = 1000;
  for (std::size_t i = 0; i != nmax; ++i)
    set.emplace(345 + i);
}
int main() { ex1(); }
```

```
>heaptrack ./a.out
heaptrack stats:
        allocations:                1000
```

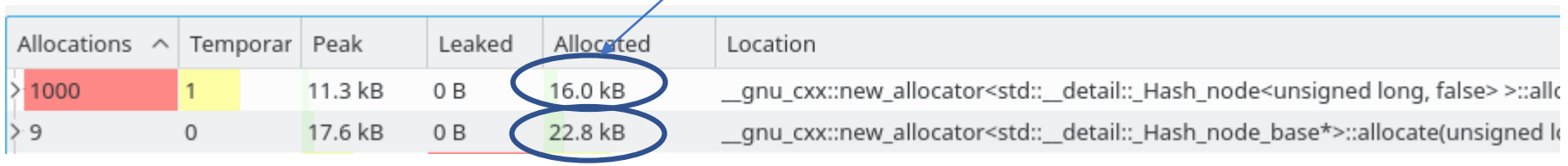
Allocations	Temporar	Peak	Leaked	Allocated	Location
1000	1	40.0 kB	0 B	40.0 kB	__gnu_cxx::new_allocator<std::_Rb_tree_node<unsigned long> >::allocate(unsigned

std::unordered_set (1)

```
#include <unordered_set>
void ex1() {
  std::unordered_set<std::size_t> set;
  constexpr std::size_t nmax = 1000;
  for (std::size_t i = 0; i != nmax; ++i)
    set.emplace(345 + i);
}
int main() { ex1(); }
```

```
>heaptrack ./a.out
heaptrack stats:
        allocations:                1009
```

Slimmer nodes



Buckets

std::unordered_set (2)

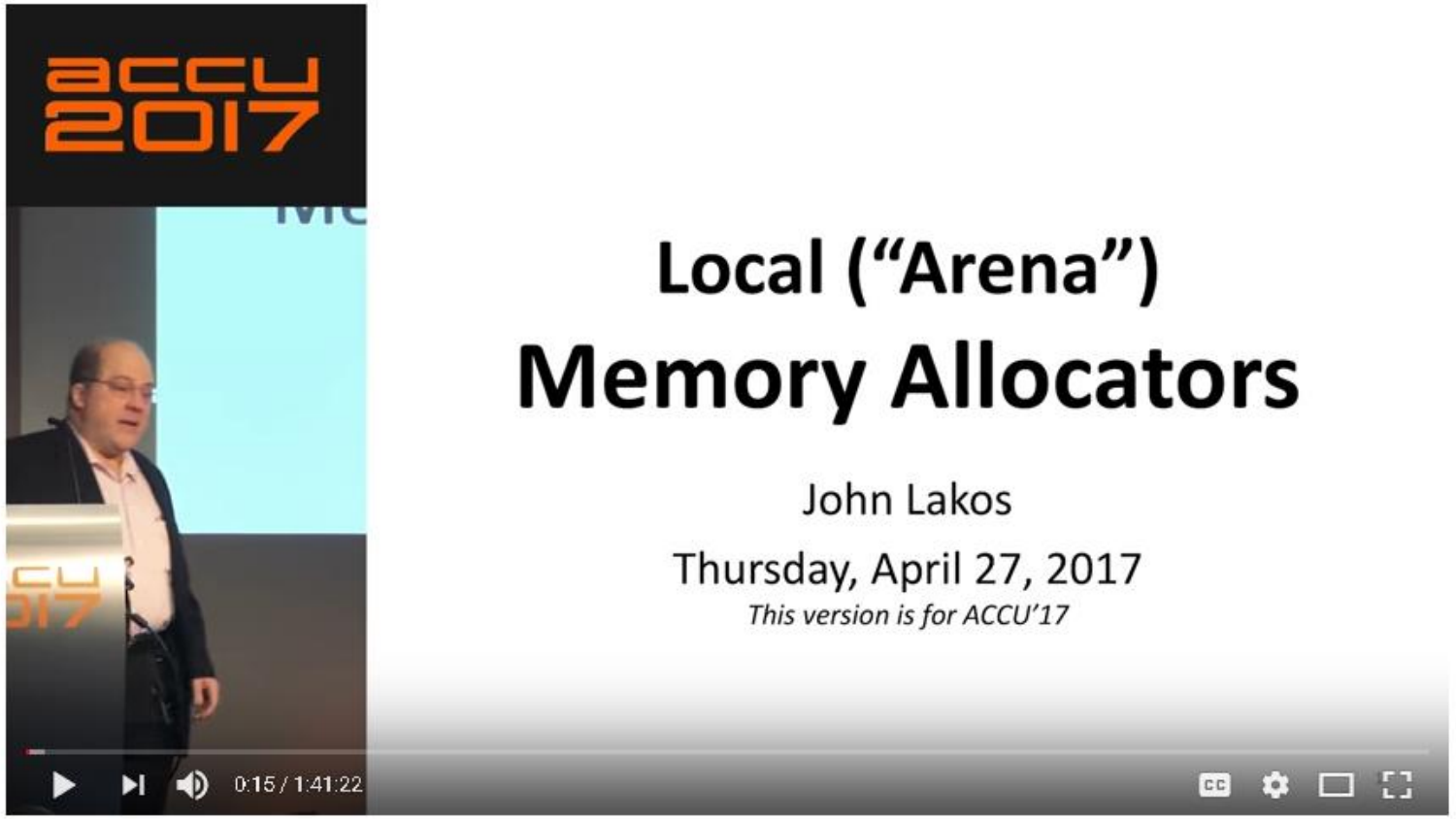
```
#include <unordered_set>
void ex1() {
  std::unordered_set<std::size_t> set;
  constexpr std::size_t nmax = 1000;
  set.reserve(nmax);
  for (std::size_t i = 0; i != nmax; ++i)
    set.emplace(345 + i);
}
int main() { ex1(); }
```

```
>heaptrack ./a.out
heaptrack stats:
        allocations:                1001
```

Allocations	Temporar	Peak	Leaked	Allocated	Location
1000	1	16.0 kB	0 B	16.0 kB	__gnu_cxx::new_allocator<std::__detail::_Hash_node<unsigned long, false> >::alloca
1	0	8.2 kB	0 B	8.2 kB	__gnu_cxx::new_allocator<std::__detail::_Hash_node_base*>::allocate(unsigned long

Associative containers will call the allocator at least “size” times.

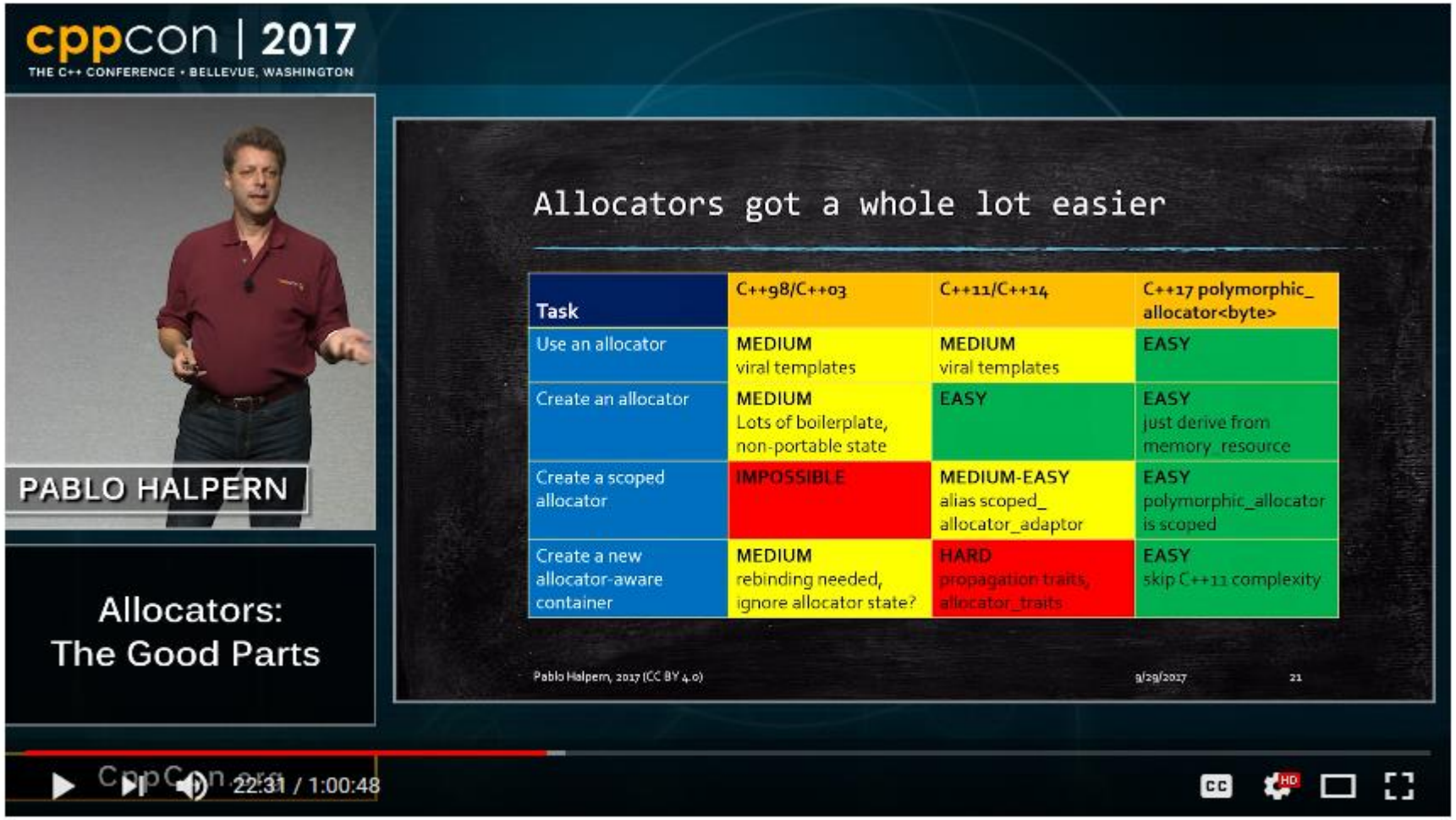
Local allocators to the rescue!



Local (arena) Memory Allocators - John Lakos [ACCU 2017]

Source: youtube.com

Extensive benchmarking: P01213R0, P0089R1



CppCon 2017: Pablo Halpern "Allocators: The Good Parts"

By the author of the C++17 allocators spec N3916.
The first 30 minutes are the most important.

Source: youtube.com

std::map

Defined in header <map>

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;                                                          (1)

namespace pmr {
    template <class Key, class T, class Compare = std::less<Key>>
    using map = std::map<Key, T, Compare,                            (2)  (since C++17)
                         std::pmr::polymorphic_allocator<std::pair<const Key,T>>>
}
```

“Allocator” is a customisation point.
By default, new and delete are used.

“pmr” stands for “polymorphic memory resource”.

Source: cppreference.com

What is the point of allocators and resources?

- Coalesce allocations of small blocks in pre-allocated larger blocks
  - Better locality
  - Less heap fragmentation

- Available memory resources in **C++17**:
  - `synchronized_pool_resource`: thread-safe pools of similar-sized memory pools
  - `unsynchronized_pool_resource`: non-thread-safe version
  - **`monotonic_buffer_resource`**: *very fast*, non-thread-safe allocation into buffers with do-nothing deallocation
    - Usually what we need for most use of associative containers if no erasure take place.

std::set

```
#include <set>
void ex1() {
  std::set<std::size_t> set;
  constexpr std::size_t nmax = 1000;
  for (std::size_t i = 0; i != nmax; ++i)
    set.emplace(345 + i);
}
int main() { ex1(); }
```

```
>heaptrack ./a.out
heaptrack stats:
        allocations:                1000
```

Allocations	Temporar	Peak	Leaked	Allocated	Location
1000	1	40.0 kB	0 B	40.0 kB	__gnu_cxx::new_allocator<std::_Rb_tree_node<unsigned long> >::allocate(unsigned

std::pmr::set

```
#include <memory_resource>
#include <set>
void ex1() {
  std::pmr::monotonic_buffer_resource res;
  std::pmr::set<std::size_t> map{ &res };
  constexpr std::size_t nmax = 1000;
  for (std::size_t i = 0; i != nmax; ++i)
    map.insert(345 + i);
}
int main() { ex1(); }
```

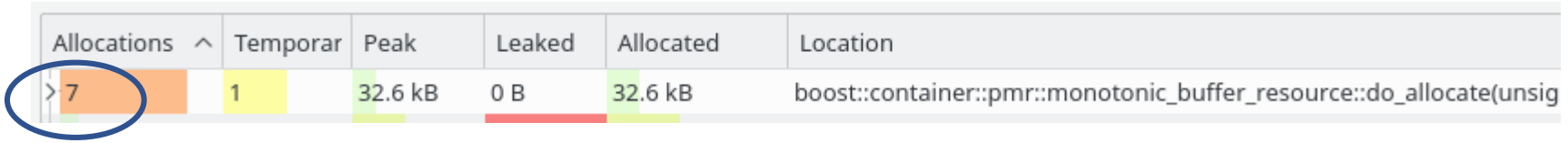
Unfortunately, no recent compilers implement the “pmr” facilities.

(with the notable exception of Visual Studio® 2017 update 6)

boost::container::pmr::set

```
#include <boost/container/pmr/monotonic_buffer_resource.hpp>
#include <boost/container/pmr/set.hpp>
void ex1() {
  namespace pmr = boost::container::pmr;
  pmr::monotonic_buffer_resource res;
  pmr::set<std::size_t> map{ &res };
  constexpr std::size_t nmax = 1000;
  for (std::size_t i = 0; i != nmax; ++i)
    map.insert(345 + i);
}
int main() { ex1(); }
```

```
>heaptrack ./a.out
heaptrack stats:
        allocations:              7
```



Alternatives

- Use an open source implementation
  - Boost (limited to boost containers)
  - https://github.com/mmcshane/pmr
  - https://github.com/phalpern/CppCon2017Code

- Roll your own:
  1. A subset of `std::pmr::monotonic_buffer_resource`,
  2. A simple, usable allocator optimized for node based containers,
  3. A series of type aliases

Home-brewed implementation (1)

```
// In C++17, when available, we can replace the code below with std::pmr::monotonic_buffer_resource.
namespace project { namespace pmr {
 class MonotonicBufferResource {
  public:
    MonotonicBufferResource();
    ~MonotonicBufferResource();

    void* allocate(std::size_t num_bytes, std::size_t alignment);
    void deallocate(void*)
    {} // Do not de-allocate individual blocks.
  private:
    struct PImpl; // see for instance https://github.com/mmcshane/pmr
    std::unique_ptr<PImpl> m_impl;
  };
}}
```

Home-brewed implementation (2)

```
template<typename T>
class NodeAllocator {
  MonotonicBufferResource* m_res;
public:
  using value_type = T;   // Refer to https://howardhinnant.github.io/allocator_boilerplate.html
  value_type* allocate(std::size_t numObjects) {
    if (numObjects != 1)
      return static_cast<value_type*>(::operator new(sizeof(value_type)*numObjects));
    else
      return static_cast<value_type*>(m_res->allocate(sizeof(value_type), alignof(value_type)));
  }
  void deallocate(void* p, std::size_t numObjects) {
    if (numObjects != 1)
      ::operator delete(p);
    else
      m_res->deallocate(p);
  }
};
```

Home-brewed implementation (3)

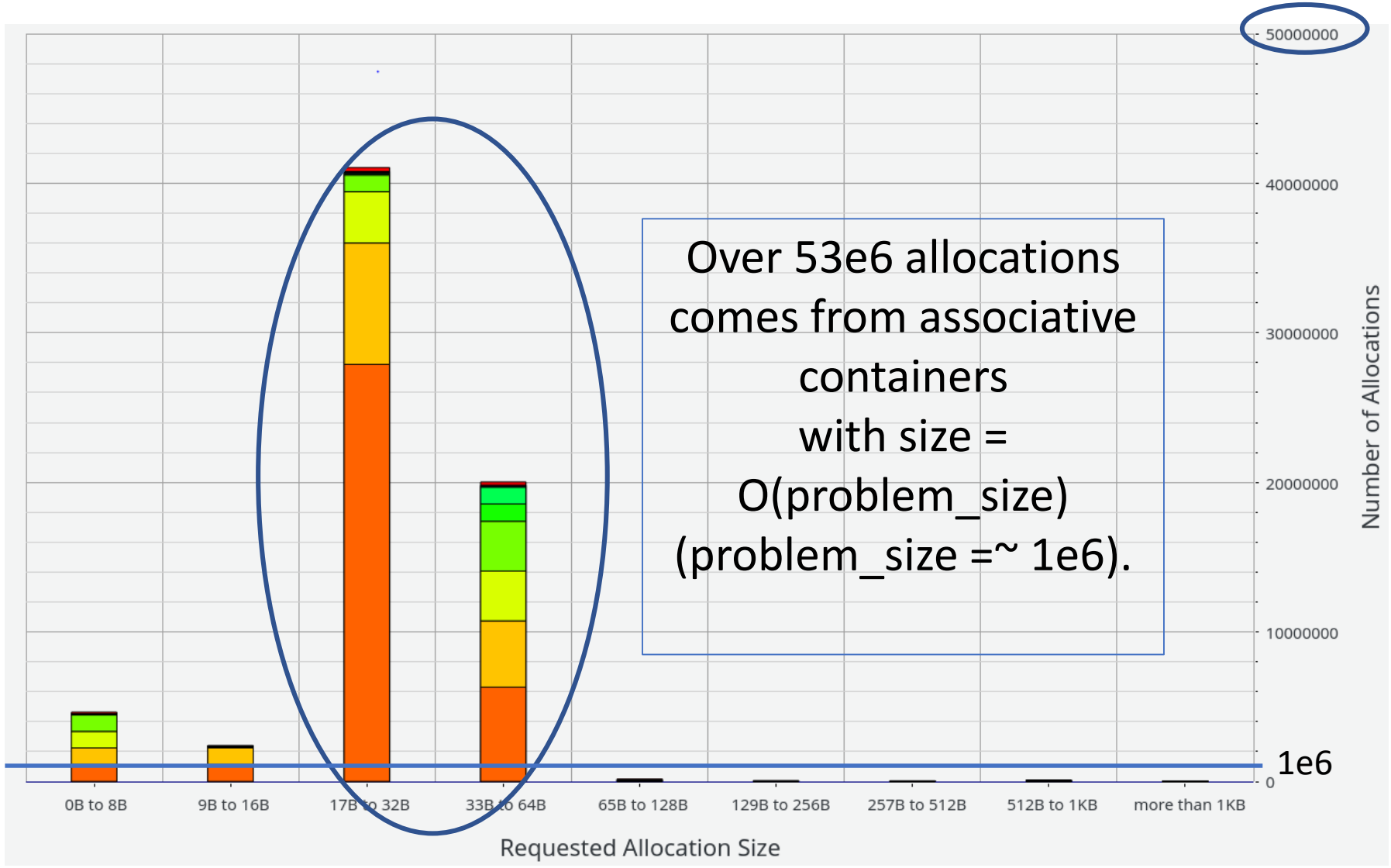
```
// In C++17, when available, we can replace the code below with std::pmr::set.
namespace project { namespace pmr {
  template <typename Key,
            typename Compare = std::less<Key>>
  using set = std::set<Key, Compare,
                       project::pmr::NodeAllocator<Key>>;

 template <typename Key,
            typename Hash = std::hash<Key>,
            typename Pred = std::equal_to<Key>>
  using unordered_set = std::unordered_set<Key, Hash, Pred,
                                           project::pmr::NodeAllocator<Key>>;
}}
```

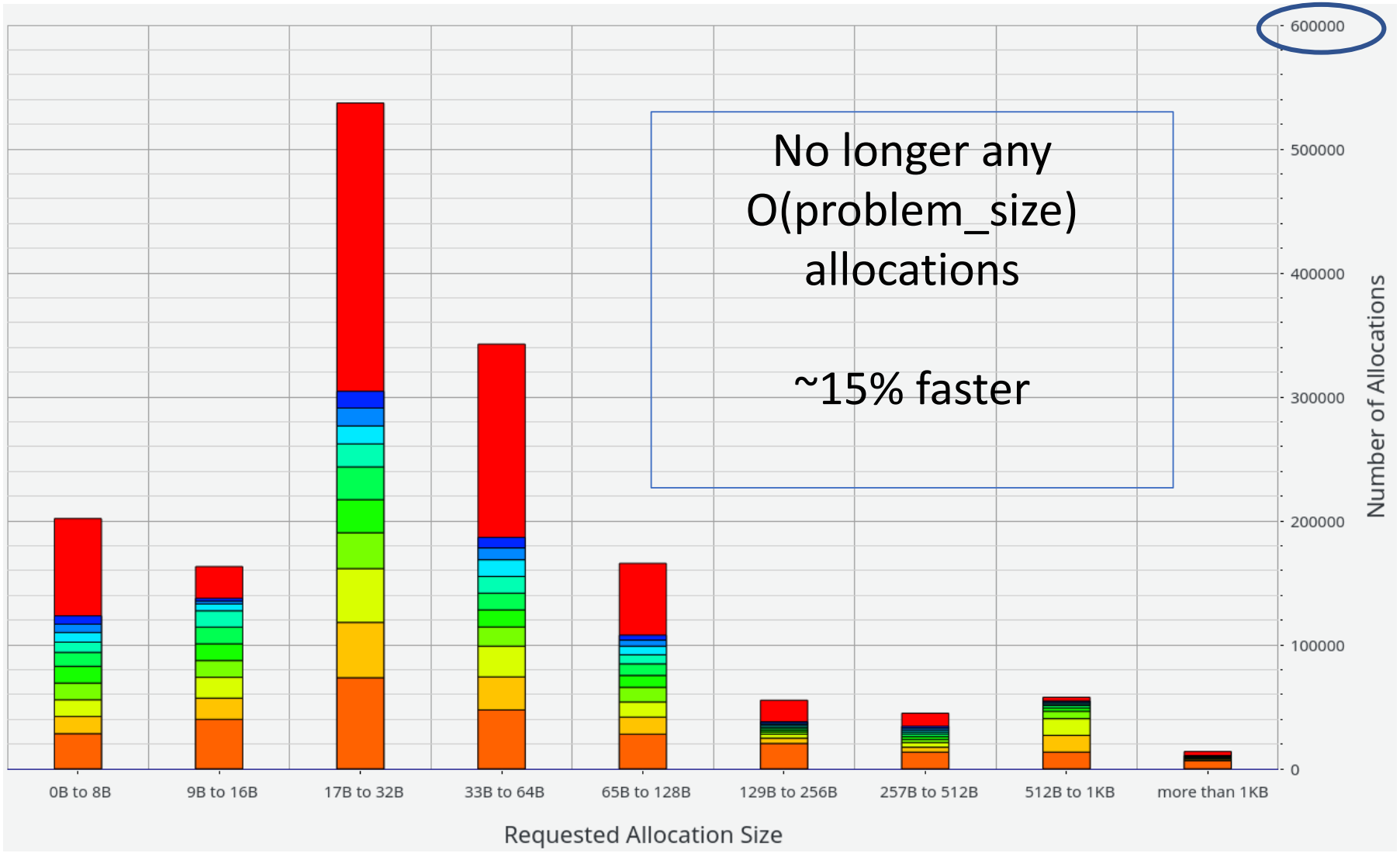
Usage

```
std::unordered_set<std::int64_t> vertex_ids;
```

->

```
project::pmr::MonotonicBufferResource monotonic_res;
project::pmr::unordered_set<std::int64_t> vertex_ids{&monotonic_res};
```


50000000
40000000
30000000
20000000
10000000
0
Number of Allocations
1e6
0B to 8B
9B to 16B
17B to 32B
33B to 64B
65B to 128B
129B to 256B
257B to 512B
512B to 1KB
more than 1KB
Requested Allocation Size
Over 53e6 allocations comes from associative containers with size = O(problem_size) (problem_size =~ 1e6).


No longer any
O(problem_size)
allocations
~15% faster
600000
500000
400000
300000
200000
100000
0
Number of Allocations
0B to 8B
9B to 16B
17B to 32B
33B to 64B
65B to 128B
129B to 256B
257B to 512B
512B to 1KB
more than 1KB
Requested Allocation Size

Conclusions

- Memory allocations can be surprisingly expensive.
- Heaptrack rocks.
- Use C++17 local memory allocators when appropriate.