Reducing memory allocations in a large C++ application

Arnaud Desitter

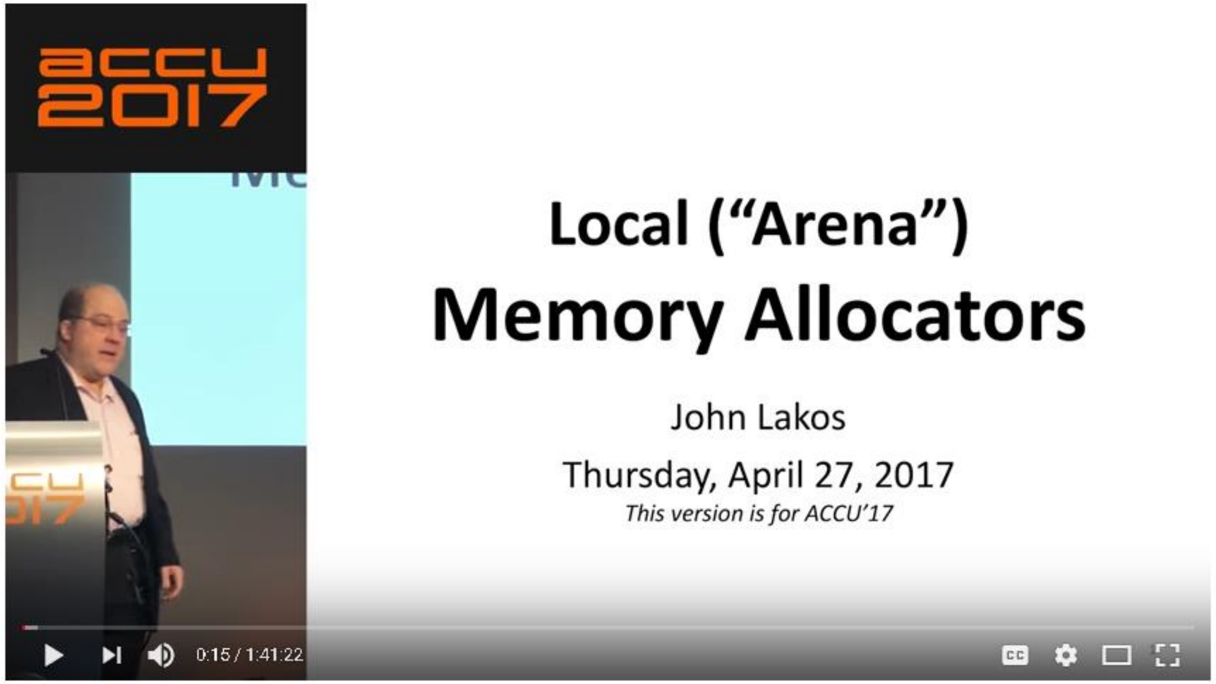
ACCU Oxford
18 July 2019

Roadmap

- Motivations
- Methodology
- Part I: a case study
- Part II: solutions to address excessive memory allocations
  - Vocabulary types
  - Patterns and anti-patterns
- Part III: C++17 pmr allocators (briefly)
- Conclusions

Motivations

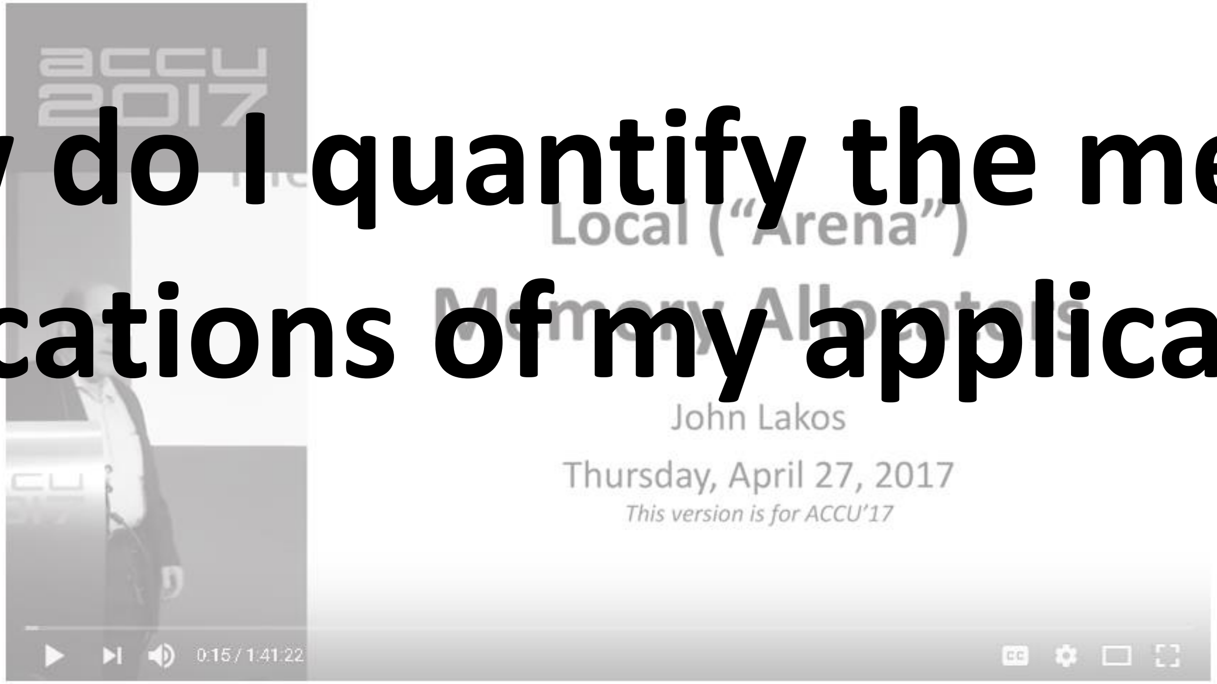
Custom allocators are a much discussed topic in the C++ industry.



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

Motivations

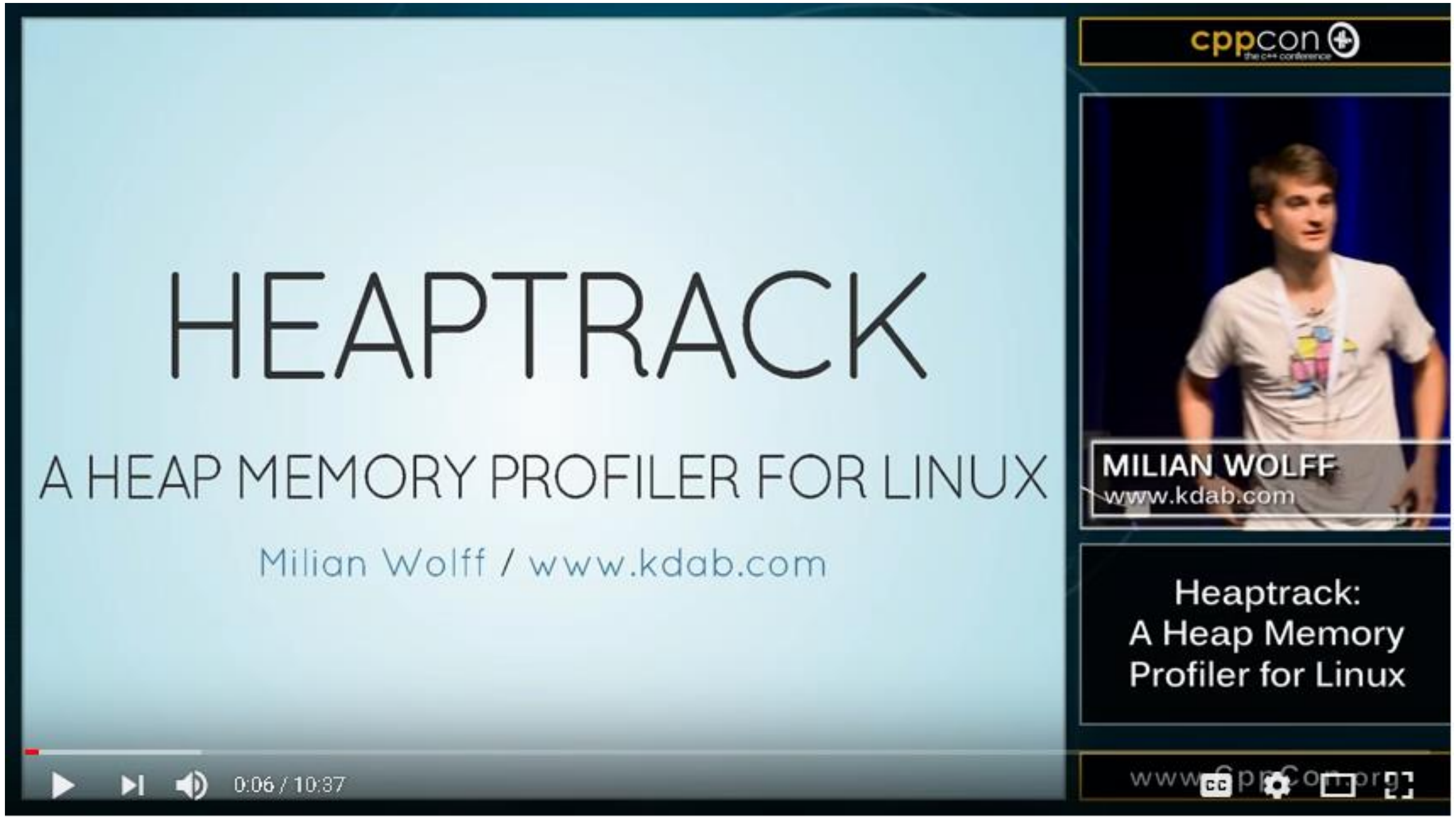
Custom allocators are a much discussed topic in the C++ industry.



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

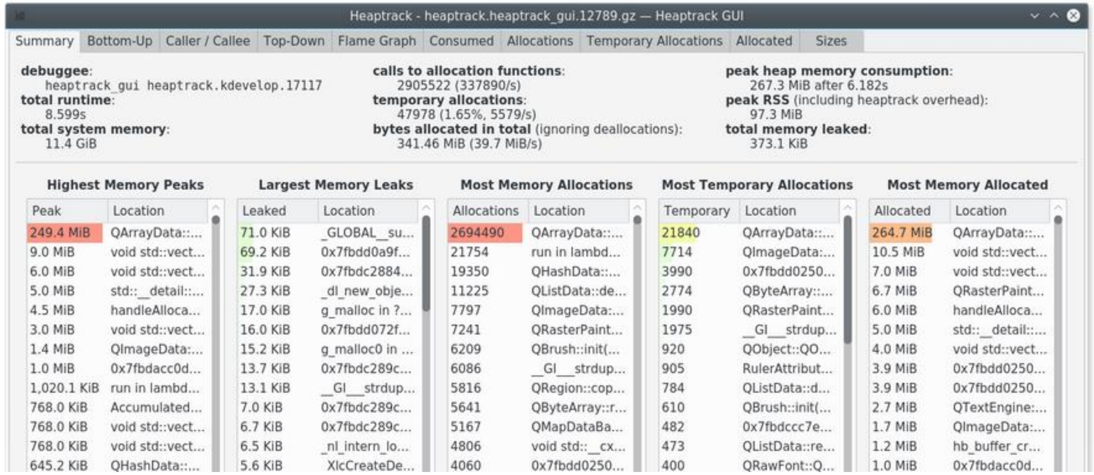
How do I quantify the memory allocations of my application ?

Motivations

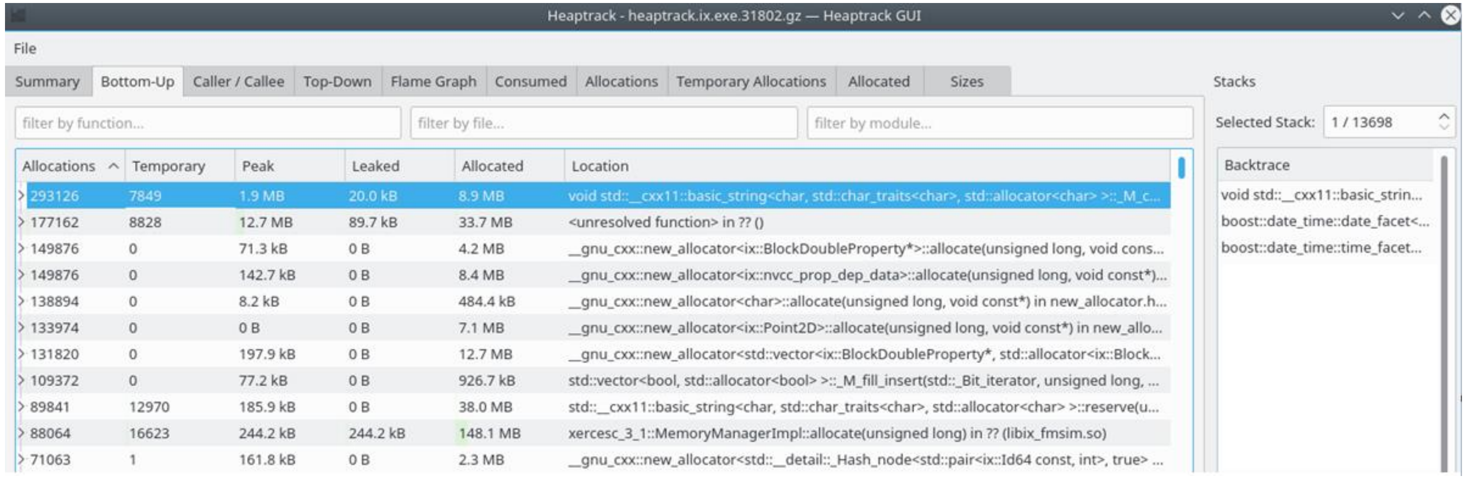


CppCon 2015: Milian Wolff "Heaptrack: A Heap Memory Profiler for Linux"

Heaptrack



Heaptrack - heaptrack.heaptrack_gui.12789.gz — Heaptrack GUI
Summary
Bottom-Up
Caller / Callee
Top-Down
Flame Graph
Consumed
Allocations
Temporary Allocations
Allocated
Sizes
debuggee:
heaptrack_gui heaptrack.kdevelop.17117
total runtime:
8.599s
total system memory:
11.4 GiB
calls to allocation functions:
2905522 (337890/s)
temporary allocations:
47978 (1.65%, 5579/s)
bytes allocated in total (ignoring deallocations):
341.46 MiB (39.7 MiB/s)
peak heap memory consumption:
267.3 MiB after 6.182s
peak RSS (including heaptrack overhead):
97.3 MiB
total memory leaked:
373.1 KiB
Highest Memory Peaks
Largest Memory Leaks
Most Memory Allocations
Most Temporary Allocations
Most Memory Allocated
Peak
Location
Leaked
Location
Allocations
Location
Temporary
Location
Allocated
Location
249.4 MiB QArrayData::...
9.0 MiB void std::vect...
6.0 MiB void std::vect...
5.0 MiB std::__detail::...
4.5 MiB handleAlloca...
3.0 MiB void std::vect...
1.4 MiB QImageData:...
1.0 MiB 0x7fbdacc0d...
1,020.1 KiB run in lambd...
768.0 KiB Accumulated...
768.0 KiB void std::vect...
768.0 KiB void std::vect...
645.2 KiB QHashData::...
71.0 KiB _GLOBAL__su...
69.2 KiB 0x7fbdd0a9f...
31.9 KiB 0x7fbdc2884...
27.3 KiB _dl_new_obje...
17.0 KiB g_malloc in ?...
16.0 KiB 0x7fbdd072f...
15.2 KiB g_malloc0 in ...
13.7 KiB 0x7fbdc289c...
13.1 KiB __GI__strdup...
7.0 KiB 0x7fbdc289c...
6.7 KiB 0x7fbdc289c...
6.5 KiB _nl_intern_lo...
5.6 KiB XlcCreateDe...
2694490 QArrayData::...
21754 run in lambd...
19350 QHashData::...
11225 QListData::de...
7797 QImageData:...
7241 QRasterPaint...
6209 QBrush::init(...
6086 __GI__strdup...
5816 QRegion::cop...
5641 QByteArray::r...
5167 QMapDataBa...
4806 void std::__cx...
4060 0x7fbdd0250...
21840 QArrayData::...
7714 QImageData:...
3990 0x7fbdd0250...
2774 QByteArray::...
1990 QRasterPaint...
1975 __GI__strdup...
920 QObject::QO...
905 RulerAttribut...
784 QListData::d...
610 QBrush::init(...
482 0x7fbdccc7e...
473 QListData::re...
400 QRawFont::Q...
264.7 MiB QArrayData::...
10.5 MiB void std::vect...
7.0 MiB void std::vect...
6.7 MiB QRasterPaint...
6.0 MiB handleAlloca...
5.0 MiB std::__detail::...
4.0 MiB void std::vect...
3.9 MiB 0x7fbdd0250...
3.9 MiB 0x7fbdd0250...
2.7 MiB QTextEngine:...
1.7 MiB QImageData:...
1.2 MiB hb_buffer_cr...
1.0 MiB 0x7fbdacc0d...

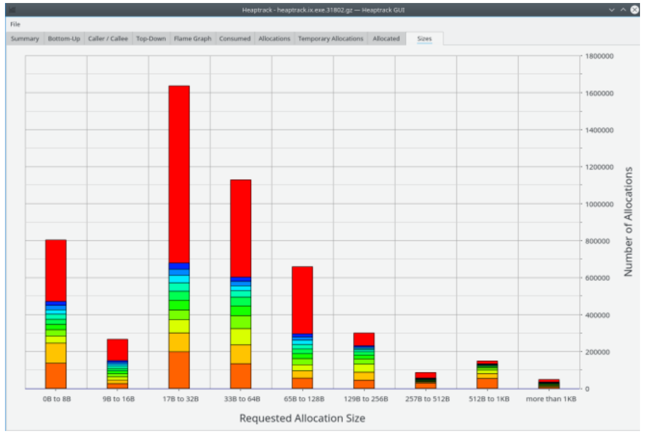


Heaptrack - heaptrack.ix.exe.31802.gz — Heaptrack GUI
File
Summary
Bottom-Up
Caller / Callee
Top-Down
Flame Graph
Consumed
Allocations
Temporary Allocations
Allocated
Sizes
Stacks
filter by function...
filter by file...
filter by module...
Selected Stack: 1 / 13698
Allocations
Temporary
Peak
Leaked
Allocated
Location
Backtrace
293126 7849 1.9 MB 20.0 kB 8.9 MB void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::_M_c...
177162 8828 12.7 MB 89.7 kB 33.7 MB <unresolved function> in ?? ()
149876 0 71.3 kB 0 B 4.2 MB __gnu_cxx::new_allocator<ix::BlockDoubleProperty*>::allocate(unsigned long, void cons...
149876 0 142.7 kB 0 B 8.4 MB __gnu_cxx::new_allocator<ix::nvcc_prop_dep_data>::allocate(unsigned long, void const*)...
138894 0 8.2 kB 0 B 484.4 kB __gnu_cxx::new_allocator<char>::allocate(unsigned long, void const*) in new_allocator.h...
133974 0 0 B 0 B 7.1 MB __gnu_cxx::new_allocator<ix::Point2D>::allocate(unsigned long, void const*) in new_allo...
131820 0 197.9 kB 0 B 12.7 MB __gnu_cxx::new_allocator<std::vector<ix::BlockDoubleProperty*, std::allocator<ix::Block...
109372 0 77.2 kB 0 B 926.7 kB std::vector<bool, std::allocator<bool> >::_M_fill_insert(std::_Bit_iterator, unsigned long, ...
89841 12970 185.9 kB 0 B 38.0 MB std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::reserve(u...
88064 16623 244.2 kB 244.2 kB 148.1 MB xercesc_3_1::MemoryManagerImpl::allocate(unsigned long) in ?? (libix_fmsim.so)
71063 1 161.8 kB 0 B 2.3 MB __gnu_cxx::new_allocator<std::__detail::_Hash_node<std::pair<ix::Id64 const, int>, true> ...
void std::__cxx11::basic_strin...
boost::date_time::date_facet<...
boost::date_time::time_facet...

Heaptrack

Flamecharts

Cumulated allocations



Sizes

Consumed

Tips

- Go to conferences !

  or watch them on YouTube.

- Do not be afraid to ask questions.

  at conferences or on the web.

- Try new tools.

  … and make improvements thanks to them.

Milian Wolff’s heaptrack

```
# Build it from source (search for “heaptrack build ubuntu”) or use pre-built AppImage
# See https://download.kde.org/stable/heaptrack/1.1.0/heaptrack-v1.1.0-x86_64.AppImage.mirrorlist
# wget …/download.kde.org/stable/heaptrack/1.1.0/heaptrack-v1.1.0-x86_64.AppImage
# chmod +x heaptrack-v1.1.0-x86_64.AppImage
```

```
# heaptrack operates on any executable. Stack traces available only with debugging symbols (compile with “-g”)
> ./heaptrack-v1.1.0-x86_64.AppImage /bin/ls
heaptrack output will be written to "heaptrack.ls.1877.zst"
starting application, this might take some time...
heaptrack.ls.1877.zst  heaptrack-v1.1.0-x86_64.AppImage
heaptrack stats:
        allocations:            44
        leaked allocations:     38
        temporary allocations:  2
Heaptrack finished! Now run the following to investigate the data:

  heaptrack --analyze "heaptrack.ls.1877.zst“
>./heaptrack-v1.1.0-x86_64.AppImage --analyze heaptrack.ls.1877.zst
```

(1) Execution with heaptrack data collection

(2) Data visualisation

Demo time

Profiling: a methodology

- Choose a case.
- Run it under a profiler.
- Spot a problem.
- Try to fix it.
- Profile again with the fix.
  - Discard fix if it does not work.
  - Submit if it does.
- Iterate until there are no more opportunities for change.
- Choose another case and iterate.

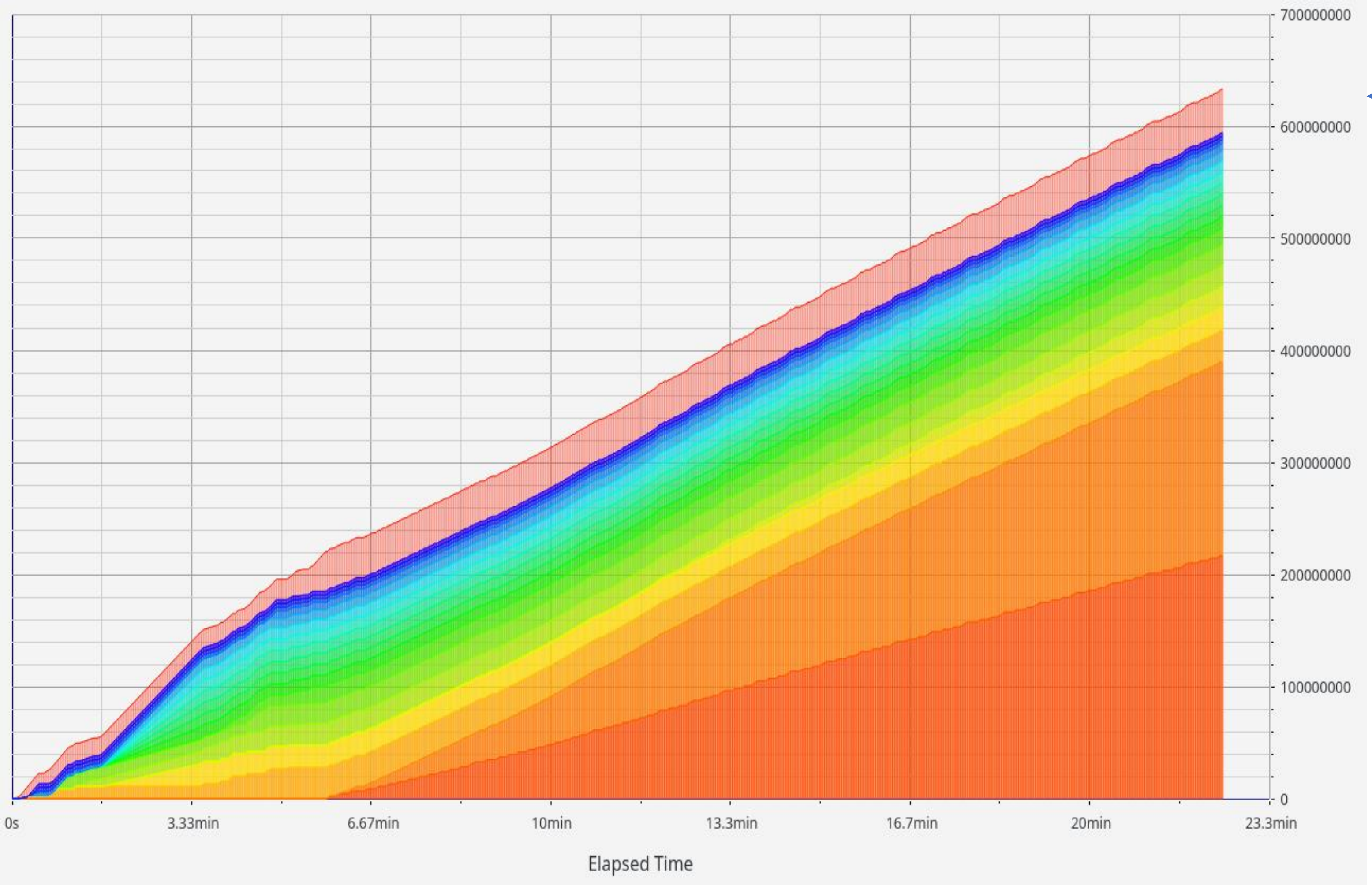
Part I: a case study

A numerical simulation

- Reasonably large synthetic problem
- … but shortened to a single day of simulation
- Dominated by floating point computation
- Heavily optimised over the years

- Run **without** any concurrency for the sake of this example
  - Considerably faster when many processes and cores are used
  - Allocations costs are amortised across processes.

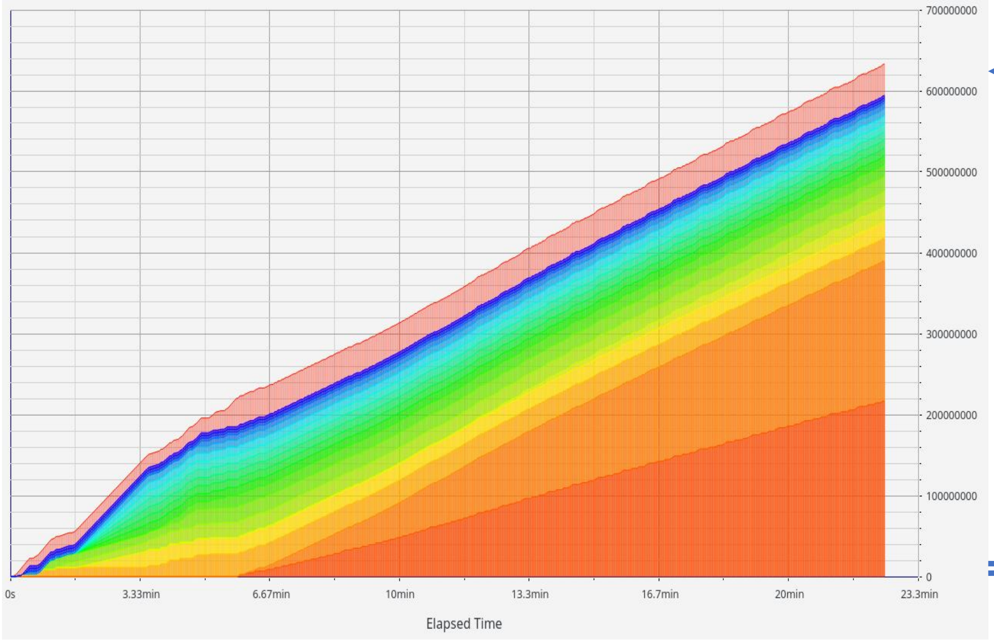
A case study



~630e6

Cumulated allocations

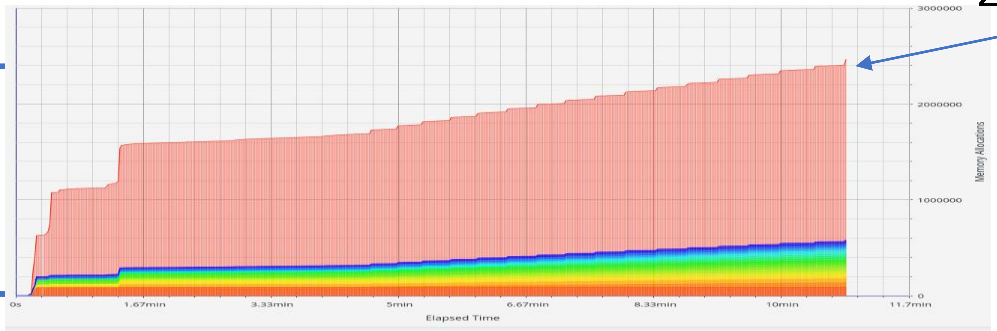
A case study



~630e6

Number of allocations reduced by x250

~2.5e6



Before

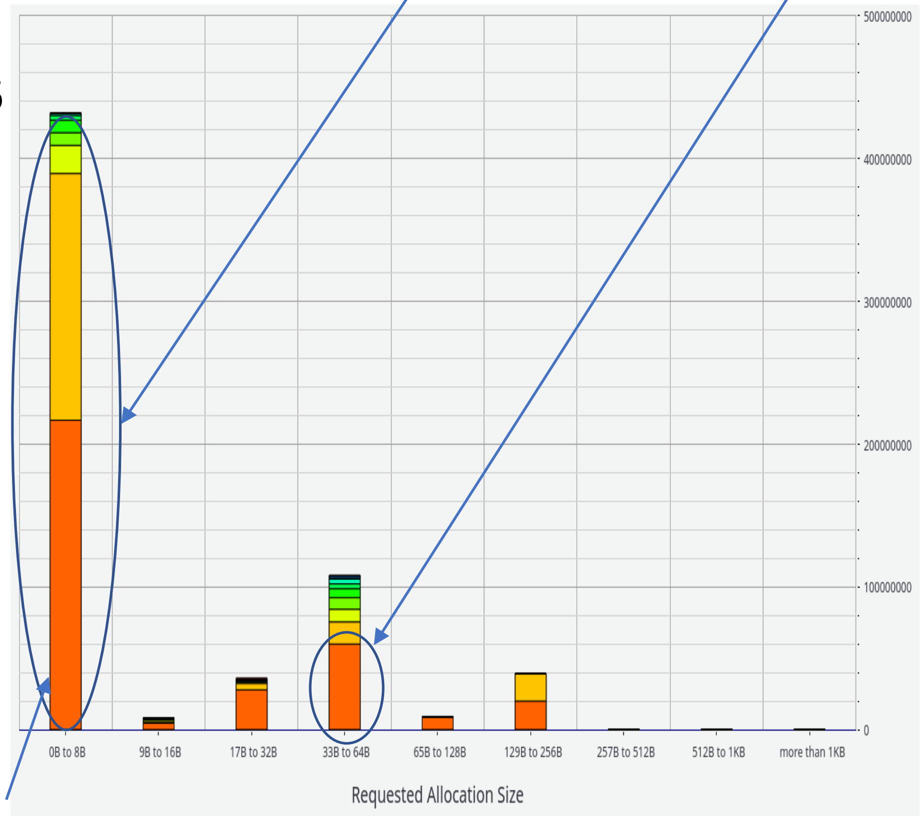
After

A case study

Most allocations were for 8 bytes or less.

Map nodes

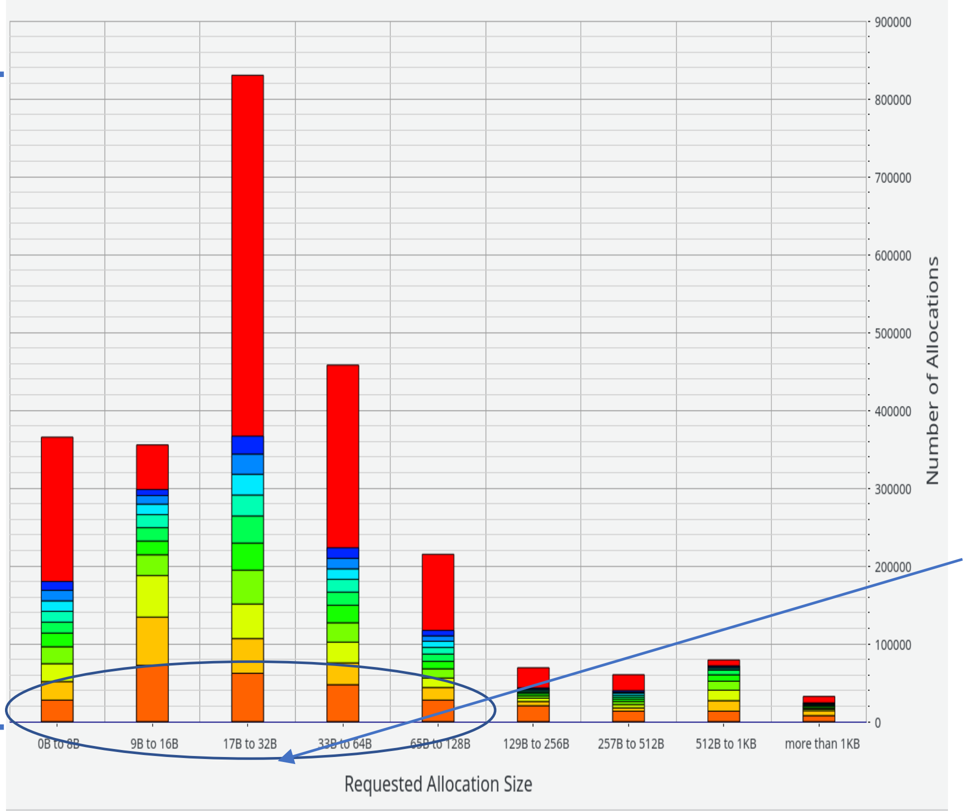
450e6



Large single source contributors

Before

0.8e6



Largest contributors

After

Cost of allocations are hard to predict

- Looking *carefully* with perf, about **5.5%** time was spent in malloc/free.
- We reduced it to **1.5%.**
- But we have a speed up well above 10%.

Reported time spent in allocator is an **under-estimate** of possible gain as excessive memory allocations may be a source of cache-misses.

Conclusions so far

- Heaptrack works really well.
  - Reasonable information with debug information provided by gcc and clang
  - Overhead is quite small once the number of allocations is under control.
  - Compatible with threads and MPI.

- It helped us to identify and then obtain significant speedup
  - Biggest gains come from removing allocations within loops.
  - Biased towards small size allocations.

Part II: solutions

- Do not copy if you can.
- Avoid allocations.
- Re-use allocated memory.
- Use contiguous containers.

Part II: solutions

- Do not copy if you can.
  - Avoid unused objects.
  - Use references.
  - Use views (gsl::span, std::string_view).
  - Use moves.
- Avoid allocation.
  - Use std::array, boost::container::small_vector.
  - Avoid pimpl when necessary. Use std::optional.
- Re-use allocated memory.
  - Use std::vector::reserve().
  - Make use of std::vector capacity.
- Use contiguous containers.
  - Avoid if possible std::map, std::set and std::list in critical code.

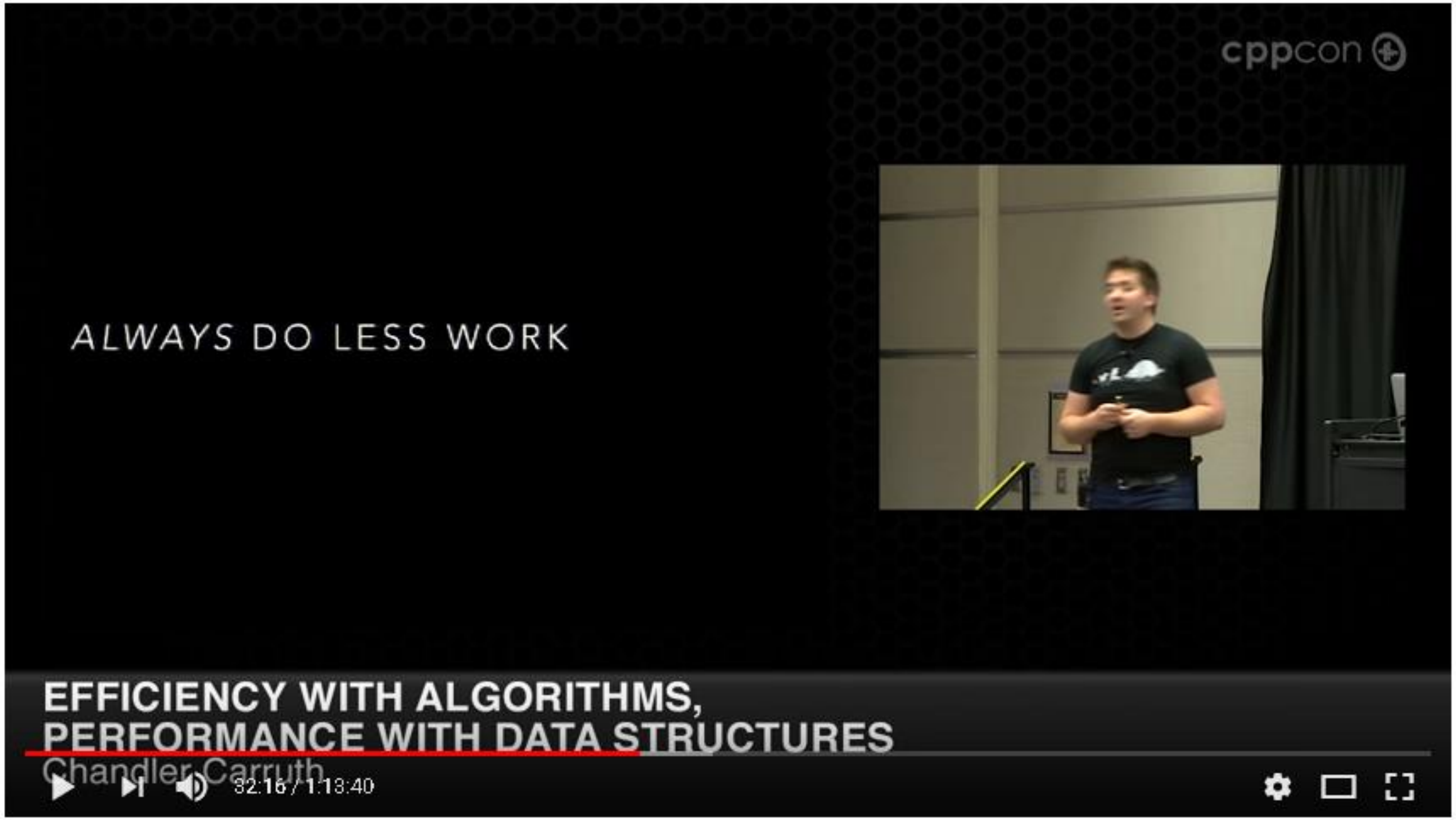
Part II: solutions

- Do not copy if you can.
  - **Avoid unused objects.**
  - Use references.
  - Use views (gsl::span, std::string_view).
  - Use moves.
- Avoid allocation.
  - Use std::array, boost::container::small_vector.
  - Avoid pimpl when necessary. Use std::optional.
- Re-use allocated memory.
  - Use std::vector::reserve().
  - Make use of std::vector capacity.
- Use contiguous containers.
  - Avoid if possible std::map, std::set and std::list in critical code.

Case study – problem #1 – Avoid unnecessary copies

Some leftovers of a debugging experiment: 210e6 allocations

```
std::vector<double> average_m(m.size(), 0.0);
for (int ic = 0; ic < m.size(); ic++)
  average_m[ic] = m[ic];
->
auto const& average_m = m;
```



CppCon 2014: Chandler Carruth "Efficiency with Algorithms, Performance with Data Structures"

Part II: solutions

- Do not copy if you can.
  - Avoid unused objects.
  - **Use references.**
  - Use views (gsl::span, std::string_view).
  - Use moves.
- Avoid allocation.
  - Use std::array, boost::container::small_vector
  - Avoid pimpl when necessary. Use std::optional.
- Re-use allocated memory.
  - Use std::vector::reserve().
  - Make use of std::vector capacity.
- Use contiguous containers.
  - Avoid if possible std::map, std::set and std::list in critical code.

Do not copy – “auto const& x = f()”

Returns “std::vector<int> const&”

```
for (…) { for (…) {
  std::vector<int> bs = the_sizes[…]->get_sizes();
->
for (…) { for (…) {
  // See GotW #88: A Candidate For the “Most Important const”
  auto const& bs = the_sizes[…]->get_sizes();
```

Do not copy arguments

```
// copy !
void add_res_from_buffer(std::vector<double> dbuffer);

->

void add_res_from_buffer(std::vector<double> const& dbuffer);
```

Beware of stateful functors

Copy!

```
class SortFunctor {
  std::vector<std::string> m_names; // ...
};
std::sort(begin(nodes), end(nodes), SortFunctor{names});
```

->

```
class SortFunctor {
  std::vector<std::string> m_names; // ...
  SortFunctor(SortFunctor const&) = delete;
  SortFunctor& operator=(SortFunctor const&) = delete;
};
const SortFunctor cmp{names}
std::sort(begin(nodes), end(nodes), cref(cmp));
```

Part II: solutions

- Do not copy if you can.
  - Avoid unused objects.
  - Use references.
  - **Use views (gsl::span, std::string_view).**
  - Use moves.
- Avoid allocation.
  - Use std::array, boost::container::small_vector.
  - Avoid pimpl when necessary. Use std::optional.
- Re-use allocated memory.
  - Use std::vector::reserve().
  - Make use of std::vector capacity.
- Use contiguous containers.
  - Avoid if possible std::map, std::set and std::list in critical code.

Vocabulary types - views

- Views are **pointers**.
  - They can dangle.
  - They are cheap to copy.
- C++17 std::string_view
  - A non-owning read-only view to a **contiguous** sequence of “char”.
  - May **not** have a trailing ‘\0’.
- GSL gsl::span<T> a.k.a. C++20 std::span<T>
  - A non-owning view to a **contiguous** sequence of T.
  - gsl::span<**T**>: read-write
  - gsl::span<**const T**>: read-only

Vocabulary types - views

- std::string_view
  - Avoid creating temporary std::string when passing a literal (“Hello world”).
  - Frequent solution to excessive allocations due to std::string.

- gsl::span
  - Usually used as function parameters
    - “std::vector<double> **&**” -> gsl::span<double> (if std::vector is not resized)
    - “std::vector<double> **const&**” -> gsl::span<**const** double>
  - Allow passing any contiguous sequences such as:
    - std::vector
    - std::array
    - boost::container::small_vector, boost::container::static_vector

Use std::string_view

```
int size = 0;
const std::unique_ptr<char[]> pchar = block->get_char(size);
std::string str(pchar.get(), size - 1); // Copy!
```

->

```
int size = 0;
const std::unique_ptr<char[]> pchar = block->get_char(size);
const std::string_view str{ pchar.get(), size - 1 };
```

string_view and std::map

```
class Map {
  std::map<std::string, int> m_map;
public:
  bool contain(std::string const& key)
  { return m_map.find(key) != m_map.end(); }
};
Map m; m.contain("a very long string");  // Temporary std::string created
```

->

```
class Map {
  std::map<std::string, int, std::less<>> m_map;  // C++14 transparent comparator
public:
  bool contain(std::string_view key) {
  { return m_map.find(key) != m_map.end(); }
};
Map m; m.contain("a very long string“sv);
```

string_view and std::map

Using std:string_view with std::unordered_map is possible thanks to the technique shown by **Marc Mutz, "StringViews, StringViews everywhere!" at "Meeting C++ 2017“.**
C++20 makes it easy: P0919.

```
class Map {
  std::map<std::string, int> m_map;
public:
  bool contain(std::string const& key)
  { return m_map.find(key) != m_map.end(); }
};
Map m; m.contain("a very long string");  // Temporary std::string created
->
class Map {
  std::map<std::string, int, std::less<>> m_map;  // C++14 transparent comparator
public:
  bool contain(std::string_view key) {
  { return m_map.find(key) != m_map.end(); }
};
Map m; m.contain("a very long string“sv);
```

Use gsl::span

```
 class TwoDTable { ...
  void lookup(double x, std::vector<double>& y) const;
};
std::vector<double> val(2);
m_table->lookup(x, val);
```

->

```
class TwoDTable { ...
    void lookup(double x, ix::span<double> y) const;
};
std::array<double,2> val = {};
m_table->lookup(x, val);
```

Part II: solutions

- Do not copy if you can.
  - Avoid unused objects.
  - Use references.
  - Use views (gsl::span, std::string_view).
  - **Use moves.**
- Avoid allocation.
  - Use std::array, boost::container::small_vector.
  - Avoid pimpl when necessary. Use std::optional.
- Re-use allocated memory.
  - Use std::vector::reserve().
  - Make use of std::vector capacity.
- Use contiguous containers.
  - Avoid if possible std::map, std::set and std::list in critical code.

Use move

```
 {
  std::vector<std::int64_t> vertex_ids_for_this_cell;
  ...
  cell_vertex_ids.push_back(vertex_ids_for_this_cell); // Copy!
}
```

->

```
{
  std::vector<std::int64_t> vertex_ids_for_this_cell;
  ...
  cell_vertex_ids.push_back(std::move(vertex_ids_for_this_cell));
}
```

Use move: rule of zero

```cpp
class X {
  ~X(); // Disable move operations generation
  std::string m_s;
};
X::~X() = default;
std::vector<X> v = ...;
std::reverse(begin(v), end(v)); // temporary copies are created
```

->

```cpp
class X {
  std::string m_s; // Rule of zero
};
static_assert(std::is_nothrow_move_constructible_v<X>);
std::vector<X> v = ...;
std::reverse(begin(v), end(v));  // Move operations are used.
```

Part II: solutions

- Do not copy if you can.
  - Avoid unused objects.
  - Use references.
  - Use views (gsl::span, std::string_view).
  - Use moves.
- Avoid allocation.
  - **Use std::array,** boost::container::small_vector.
  - Avoid pimpl when necessary. Use std::optional.
- Re-use allocated memory.
  - Use std::vector::reserve().
  - Make use of std::vector capacity.
- Use contiguous containers.
  - Avoid if possible std::map, std::set and std::list in critical code.

Use std::array

```
void calc(...) { ...
  std::vector<double> COF(15, 0.0);
  COF[0] = ...
  ...
  COF[14] = ...
->
void calc(...) {
  std::array<double, 15> COF = {
    ...
  };
```

Part II: solutions

- Do not copy if you can.
  - Avoid unused objects.
  - Use references.
  - Use views (gsl::span, std::string_view).
  - Use moves.
- Avoid allocation.
  - Use std::array**, boost::container::small_vector.**
  - Avoid pimpl when necessary. Use std::optional.
- Re-use allocated memory.
  - Use std::vector::reserve().
  - Make use of std::vector capacity.
- Use contiguous containers.
  - Avoid if possible std::map, std::set and std::list in critical code.

Vocabulary types: small_vector

- Popularised by LLVM.

  CppCon 2016: Chandler Carruth “High Performance Code 201: Hybrid Data Structures"

- Implemented as **boost::container::small_vector** (by Ion Gaztanaga)
- small_vector<T, 8>
  - Will handle any size
  - Will not allocate any memory as long as the size never exceeds 8.
  - Is clearly larger than vector<T>
  - Meets STL requirement even when T is “bool”.

CppCon 2016: Chandler Carruth "High Performance Code 201: Hybrid Data Structures"

boost::container::small_vector - overhead

```
// With gcc 64 bit:

 // sizeof(vector<int>)          = 24, sizeof(vector<double>)          = 24
 // sizeof(small_vector<int, 1>) = 32, sizeof(small_vector<double, 1>) = 32
 // sizeof(small_vector<int, 2>) = 32, sizeof(small_vector<double, 2>) = 40
 // sizeof(small_vector<int, 3>) = 32, sizeof(small_vector<double, 3>) = 48
 // sizeof(small_vector<int, 4>) = 40, sizeof(small_vector<double, 4>) = 56
 // sizeof(small_vector<int, 5>) = 40, sizeof(small_vector<double, 5>) = 64
 // sizeof(small_vector<int, 6>) = 48, sizeof(small_vector<double, 6>) = 72
 // sizeof(small_vector<int, 7>) = 48, sizeof(small_vector<double, 7>) = 80
 // sizeof(small_vector<int, 8>) = 56, sizeof(small_vector<double, 8>) = 88
 // ...
 // sizeof(small_vector<int,15>) = 80, sizeof(small_vector<double,15>) = 144
 // sizeof(small_vector<int,16>) = 88, sizeof(small_vector<double,16>) = 152
```

```
// With gcc 64 bit:

// sizeof(std::vector<bool>) = 40, sizeof(std::vector<char>) = 24
// N      sizeof(small_vector<bool, N>)
//  1:15  32 bytes
// 16:23  40 bytes
// 24:31  48 bytes
```

span + small_vector

- Step 1: replace **vector** by **span** in function parameters
  Any contiguous containers can now be used.

- Step 2: replace vector by **small_vector** where necessary and if beneficial

Case study – problem #2 – small_vector

170e6 allocations

```
std::vector<int> js(n);
```

->

```
boost::container::small_vector<int, 16> js(n);
```

Part II: solutions

- Do not copy if you can.
  - Avoid unused objects.
  - Use references.
  - Use views (gsl::span, std::string_view).
  - Use moves.
- Avoid allocation.
  - Use std::array, boost::container::small_vector.
  - **Avoid pimpl when necessary. Use std::optional.**
- Re-use allocated memory.
  - Use std::vector::reserve().
  - Make use of std::vector capacity.
- Use contiguous containers.
  - Avoid if possible std::map, std::set and std::list in critical code.

Avoiding memory allocations

- Avoid Pimpl when too expensive
  - Pimpl are about data hiding
  - Not worth it for internal code that is in the critical path

- Use `std::optional` for member data if delayed initialisation is necessary

Avoid Pimpl when necessary

```
class NodeContainerIterator {
  struct IIterator;
  std::shared_ptr<IIterator> m_pimpl;
->

class NodeContainerIterator {
  std::vector<Node*>::const_iterator m_iterator;
```

Use std::optional to delay initialisation

```
class ExecutorCommand {
  bool pre_execution(...);
  // “Context” not default initialisable. Initialised in pre_execution().
  std::unique_ptr<Context> m_ctx;
```

->

```
class ExecutorCommand {
  bool pre_execution(...);
  // Avoid memory allocation
  std::optional<Context> m_ctx;
```

Part II: solutions

- Do not copy if you can.
  - Avoid unused objects.
  - Use references.
  - Use views (gsl::span, std::string_view).
  - Use moves.
- Avoid allocation.
  - Use std::array, boost::container::small_vector
  - Avoid pimpl when necessary. Use std::optional.
- Re-use allocated memory.
  - Use std::vector::reserve().
  - Make use of std::vector capacity.
- Use contiguous containers.
  - Avoid if possible std::map, std::set and std::list in critical code.

Re-use vector capacity

```
for (…) {
  std::vector<int> bsz;
  for (…)
    bsz.push_back(…);
->
std::vector<int> bsz;  // hoisted to re-use capacity
for (…) {
  bsz.clear();         // clear it
  bsz.reserve(…);      // “reserve” safe after a clear()
  for (…)
    bsz.push_back(…);
```

The same technique can be used with std::string and std::stringstream.

Do **not** use reserve() in a loop

```
for (…) {
  auto block = received();
  if (block.empty())
    break;
  buffer.reserve(buffer.size() + block.size());
  for (auto const& elt : block)
    buffer.push_back(elt);
```

Performance killer

Create expensive objects only once

```
std::ostringstream oss;
oss.imbue(std::locale(oss.getloc(), new boost::posix_time::time_facet("%d-%b-%Y")));
```

->

```
using tf = boost::posix_time::time_facet;
static const auto owned_time_facet =
  std::make_unique<tf>
  ("%d-%b-%Y",
   tf::period_formatter_type{}, tf::special_values_formatter_type{}, tf::date_gen_formatter_type{},
   1); // Non zero ref count to retain ownership
std::ostringstream oss;
oss.imbue(std::locale(oss.getloc(), owned_time_facet.get()));
```

Part II: solutions

- Do not copy if you can.
  - Avoid unused objects.
  - Use references.
  - Use views (gsl::span, std::string_view).
  - Use moves.
- Avoid allocation.
  - Use std::array, boost::container::small_vector
  - Avoid pimpl when necessary. Use std::optional.
- Re-use allocated memory.
  - Use std::vector::reserve().
  - Make use of std::vector capacity.
- Use contiguous containers.
  - Avoid if possible std::map, std::set and std::list in critical code.

Avoid node-based containers

```
void process(std::vector<int> const& recvIndexSet)
{
  std::unordered_set<int> receivedNodes;
  for (int i : recvIndexSet) {
    // There is a chance of duplication in recvIndexSet.
    if (receivedNodes.insert(i).second) {
      …
```

Flagged as a source of allocations

Avoid node-based containers

```
std::sort(begin(recv_ind), end(recv_ind));
process(recv_ind);
```

Call site

```
void process(std::vector<int> const& recvIndexSet)
{
  std::set<int> receivedNodes;
  for (int i : recvIndexSet) {
    // There is a chance of duplication in recvIndexSet.
    if (receivedNodes.find(i) == receivedNodes.end()) {
      receivedNodes.insert(i);
      …
```

Avoid node-based containers

```
// Sort and remove duplicates
sort(begin(recv_ind),end(recv_ind));
recv_ind.erase(std::unique(begin(recv_ind), end(recv_ind)),
               end(recv_ind));
process(recv_ind);

void process(std::vector<int> const& recvIndexSet)
// Expect: recvIndexSet is sorted and contains no duplicates
{
  …
```

Part II: solutions

- Do not copy if you can.
  - Avoid unused objects.
  - Use references.
  - Use views (gsl::span, std::string_view).
  - Use moves.
- Avoid allocation.
  - Use std::array, boost::container::small_vector
  - Avoid pimpl when necessary. Use std::optional.
- Re-use allocated memory.
  - Use std::vector::reserve().
  - Make use of std::vector capacity.
- Use contiguous containers.
  - Avoid if possible std::map, std::set and std::list in critical code.

Part III: C++17 "pmr" allocators

- C++17 introduces the "polymorphic memory resources"
- Available in VS 2017 and GCC 9.
- Nicolai Josuttis "C++17 - The complete Guide", chapter 31

Howard Hinnant's stack_alloc is still handy.

(https://howardhinnant.github.io/stack_alloc.html)

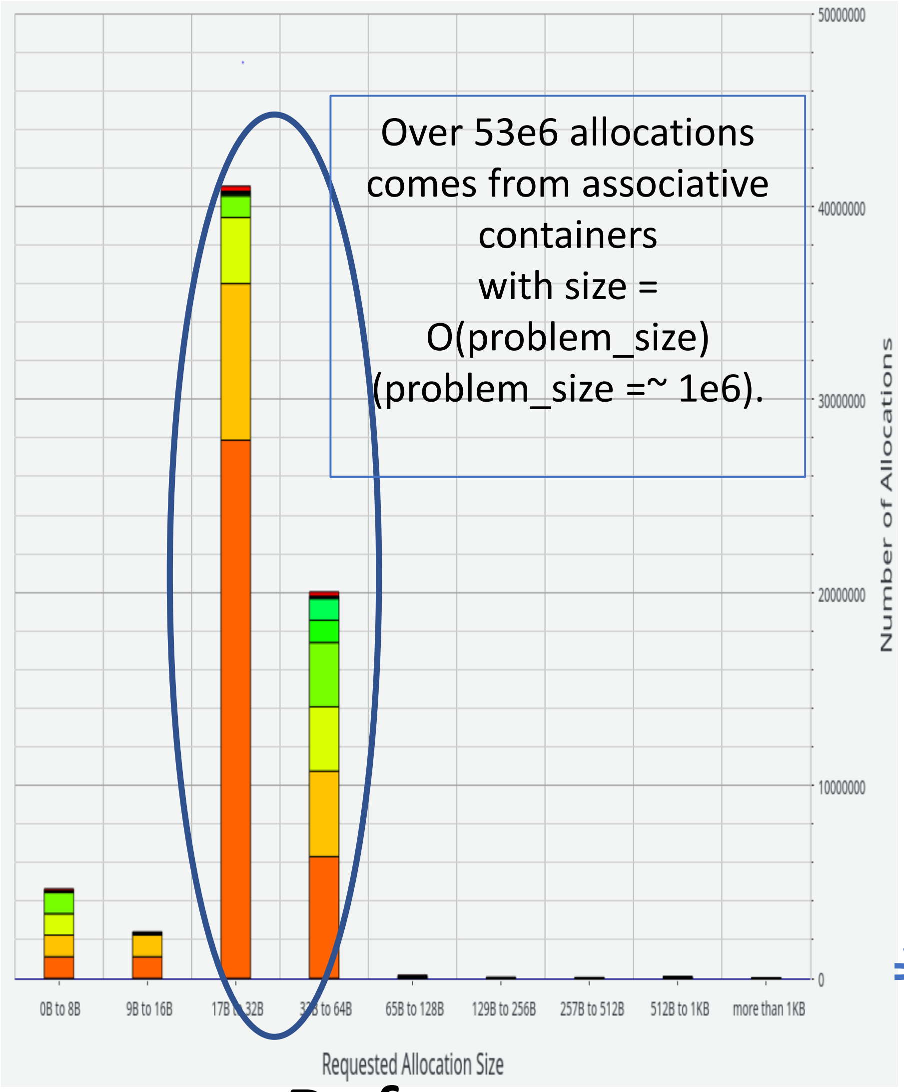
C++17 pmr allocators - vector

```
// Allocated and never resized
std::vector<double> f1(num_1);
std::vector<double> f2(num_1);
std::vector<double> f3(num_1*num_1);
std::vector<double> f4(num_2);
std::vector<double> f5(num_2*num_2);
->
std::array<char, 256> stack_buffer;
std::pmr::monotonic_buffer_resource mem_res{std::data(stack_buffer), std::size(stack_buffer)};
std::pmr::vector<double> f1(num_1, 0., &mem_res);
std::pmr::vector<double> f2(num_1, 0., &mem_res);
std::pmr::vector<double> f3(num_1*num_1, 0., &mem_res);
std::pmr::vector<double> f4(num_2, 0., &mem_res);
std::pmr::vector<double> f5(num_2*num_2, 0., &mem_res);
```

C++17 pmr allocators - map

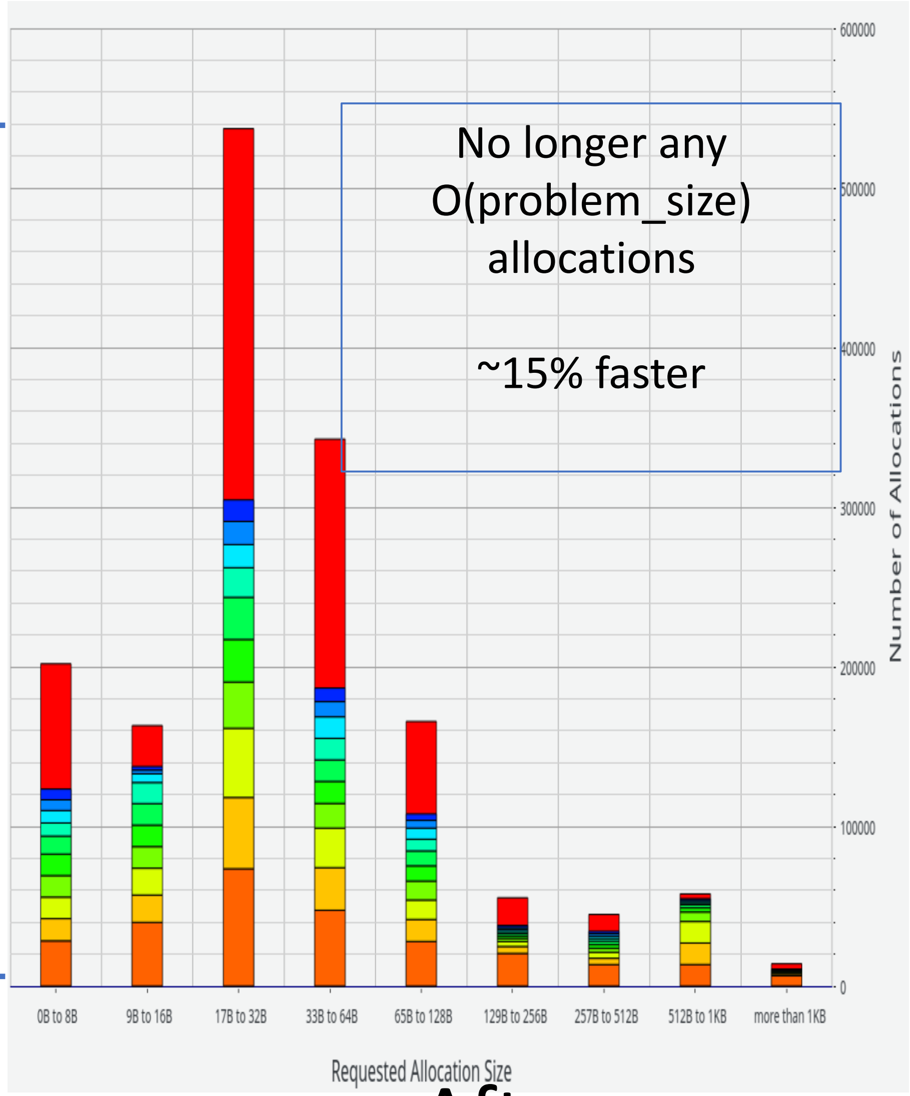
```
class X {
  // No individual erase operations
  std::unordered_map<std::int64_t, std::size_t> m_id_to_index;
public:
  X() {}
};
->

class X {
 std::pmr::monotonic_buffer_resource m_res;
 std::pmr::unordered_map<std::int64_t, std::size_t> m_id_to_index;
public:
  X() : m_id_to_index{&m_res} {}
};
```

~50e6



Before

~0.6e6



After

Anti-pattern: monotonic_buffer_resource and emplace/insert

```
class Cache {
  std::pmr::monotonic_buffer_resource m_res;
  std::pmr::unordered_map<std::size_t, int> m_cache;
public:
  X() : m_cache{&m_res} {}
  int get_index(std::size_t key) {
    // A temporary node is created resulting in slow memory growth.
    const auto r = m_cache.emplace(key,-1);
    if (!r.second)
      r.first->second = get_new_index(...);
    return r.second;
  }
};
```

Memory killer

Pattern: monotonic_buffer_resource and try_emplace

```
class Cache {
  std::pmr::monotonic_buffer_resource m_res;
  std::pmr::unordered_map<std::size_t, int> m_cache;
public:
  X() : m_cache{&m_res} {}
  int get_index(std::size_t key) {
    // C++17 try_emplace does not create any temporary node.
    const auto r = m_cache.try_emplace(key,-1);
    if (!r.second)
      r.first->second = get_new_index(...);
    return r.second;
  }
};
```

Conclusions

- Memory allocations can be surprisingly expensive.

- Heaptrack rocks.

- Do not copy unnecessarily.
- Learn about views (string_view, span) and small_vector.

Appendix

C++17 std::string_view

Supported by VS 2017, gcc 7.1 and later

For older compilers:

- boost's "boost/utility/string_view.hpp"
- https://github.com/tcbrindle/cpp17_headers
- std::string compatible std::hash support missing but easy to implement

gsl::span, C++20 std::span

- https://github.com/microsoft/GSL
- https://github.com/tcbrindle/span
- https://github.com/martinmoene/span-lite