Can we make a faster linked list? Yes, basically, yes. Also, ignorance is bliss

- Matt Bentley, PacifiC++ 2017

plf::list

A more-contiguous drop-in replacement for std::list which is not quite as bad as std::list.

Open source, zlib permissive license, etcetera:

www.plflib.org/list.htm

"Because it was there."
– Sir Edmund Hillary

plf::list vs std::list

On average across types under GCC 7.1 x64 on Haswell:

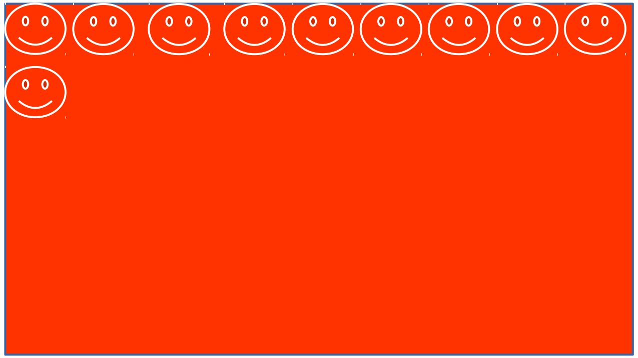
- 333% faster singular insertion
- 81% faster singular erasure
- 16% faster iteration
- 72% faster sorting
- 25% faster overall in ordered use-case benchmarking

All percentages vary substantially based on type size.

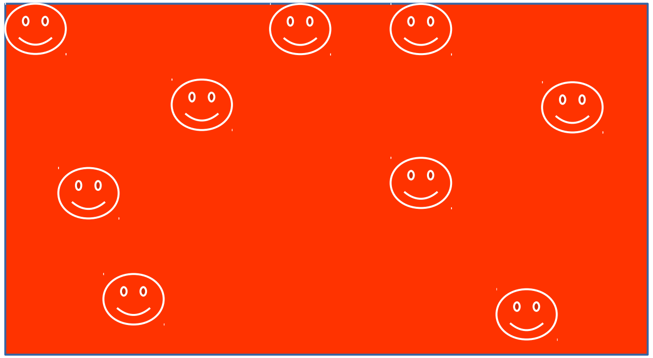
How?

- More contiguous storage (unrolling the list, node chunks)
- Smart erased-node re-use
- Alternative sort technique

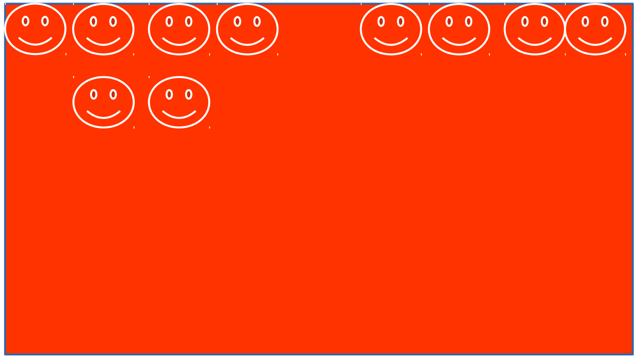
Vectors/arrays:



Most hashes, maps, lists etc:



Unrolled lists/deques/colony:



```
mandatory_history<linked_lists> need_to_know;

for(auto topic : need_to_know)
{
   hold_existential_dread_at_bay();
   listen(*topic);
}
```

“just use vector”

“Linked lists. Huh. Good god y’all, what are they good for (absolutely nothing)”
- Edwin Starr, musician (paraphrased)

Problems with linked lists?

- They cannot leverage SIMD (all containers except vector/array)
- They reduce the benefit of out-of-order execution
- They throw off hardware prefetching
- They reduce DRAM and TLB locality
- They are harder to send to GPUs

Source: *http://highscalability.com/blog/2013/5/22/strategy-stop-using-linked-lists.html*

And I would add to that:

- Following pointers to other pointers results in poor iteration performance, as the CPU can only cache a limited number of jump destinations.
- Sort performance is not great, except for large or non-trivially-copyable/movable types.
- Insertion, destruction slow due to individual allocations.
- Bidirectional iteration only, so no index access etc.

And yet...

People still use them!

(but ...why?)

What‘s actually Good?

- Fastest single erasure if done during iteration/via iterator
- Fastest non-back/front ordered insertion of std containers
- Fastest individual-insertion/sorting for large or non-trivially-copyable/movable types
- Iterators/pointers to non-erased elements stay valid regardless of operation (splice, erase, insert, merge etc)
- O(1) splicing together of lists
- Relatively easy to create concurrent versions

Where will this actually be faster?

Ordered situations with any of the following:

- A lot of non-back/front insertion and erasure
- A lot of splicing/merging
- Large types and a lot of sorting
- Objects that have significant copy/move costs
- Non-copyable/movable types

(plf::colony will be faster in unordered variants of the above)

What about intrusive lists?

- They combine the poor iteration speed of std::list with the non-front/back insert/erase speeds of a back-end container (usually vector/deque so, poor).
- Main focus: sharing lists of elements in a container owned by process A, with process B, in such a way that list nodes are automatically removed when elements are erased by A.
- If B can own the elements, plf::list will be faster due to better re-insertion locality & faster insert/erase.
- Side-note: if the back-end doesn't get erasures (or doesn't reallocate during erasure), a vector of pointers is faster, except when loads of non-back inserts to the vector occur.

How do we improve upon std::list?

Remove partial (range) splicing, which then allows:

- List unrolling, which in turn allows for:
- Memory re-use from erased elements

Also, sort hacking.

The pointer-array sort hack

- Enables any container with a bidirectional iterator to use std::sort or some other sort algorithm internally
- Temporary memory cost of N pointers
- On average across types, 72% faster sorting than std::list
- Still not as fast as vector + std::sort except for large structs
- Could use std::stable_sort if you want (slower)
- Also used by plf::colony

Pointer-array sort hack – Process:

1. Create temporary array of N pointers, fill it with pointers to each element
2. Sort the pointers by the value of the elements they point to via a dereferencing template functor and the supplied sort function
3. Process results according to the type of container involved
4. Delete temporary array

Stateful dereferencing template functor

```
template <class comparison_function>
struct sort_dereferencer
{
    comparison_function stored_instance;

    sort_dereferencer(const comparison_function &function_instance): stored_instance(function_instance) {}
    sort_dereferencer() {}

    bool operator() (const node_pointer_type first, const node_pointer_type second) const
    {
        return stored_instance(first->element, second->element);
    }
};
```

Pointer-array sort hack – list-specific:

In terms of plf::list, for step 1. we can iterate linearly over the node blocks rather than following the linked list sequence (faster).

In terms of linked lists in general, the process for step 3. follows:

(a) Process array sequentially, for every pointer at index i, dereference to node and set previous and next pointers to the addresses at i – 1 and i + 1 respectively.

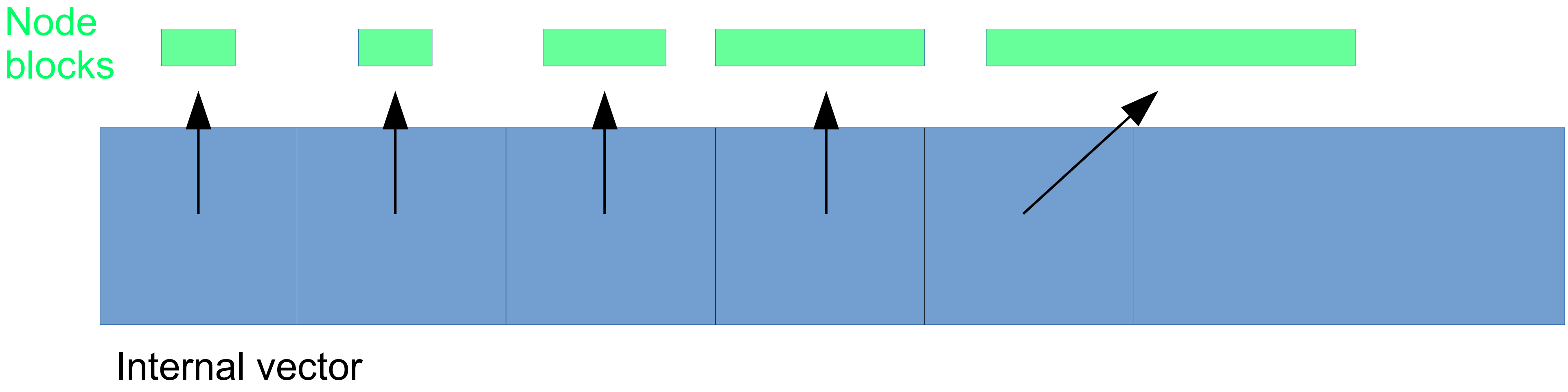
(b) Set the node pointed to by array index 0 as the front node, and the node pointed to by the last index in the array as the back node.

List unrolling

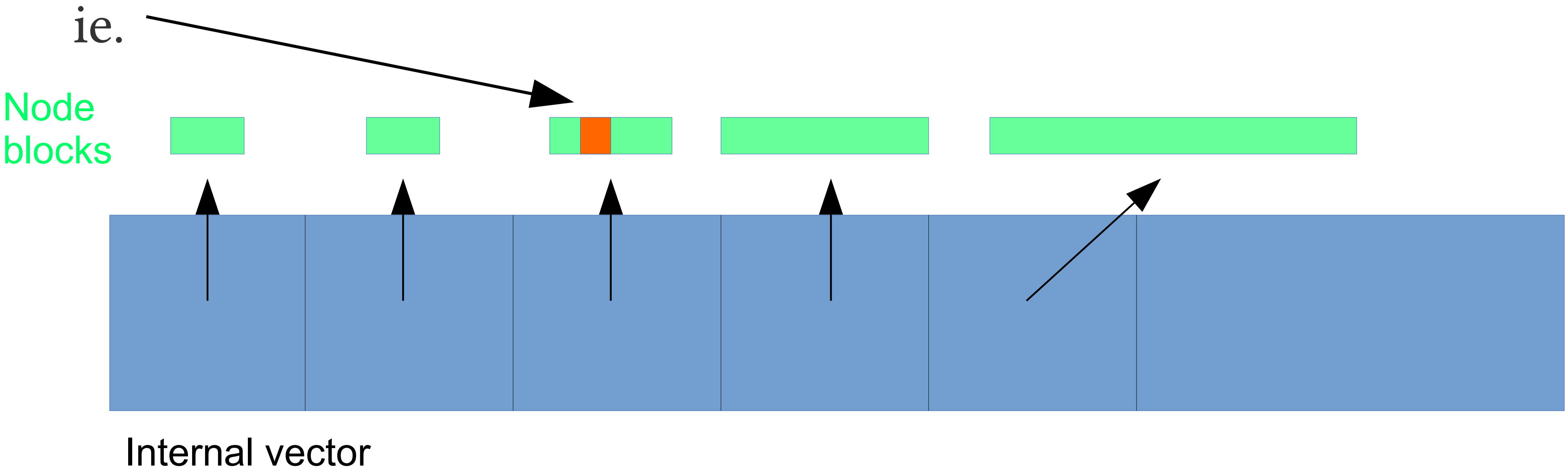
- Allocate chunks of nodes rather than individual nodes
- Increases memory locality and cache performance
- Reduces overall insertion/erasure/destruction cost

Unrolling: how plf::list does it

- Internal vector of groups (pointer to memory block + block metadata)
- Memory blocks start small, have a growth factor
- Maximum size relatively low to allow for better reinsertion



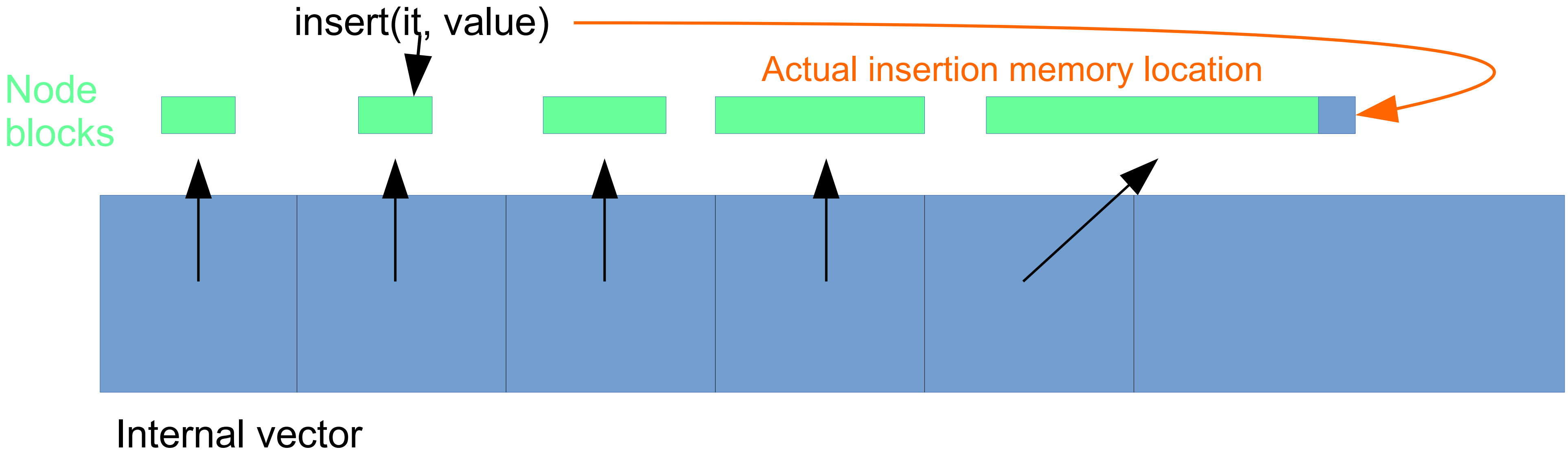
What to do with erased nodes?



Can‘t reallocate, or pointers to elements will invalidate, how to re-use? (empty blocks will get removed or sent to back)

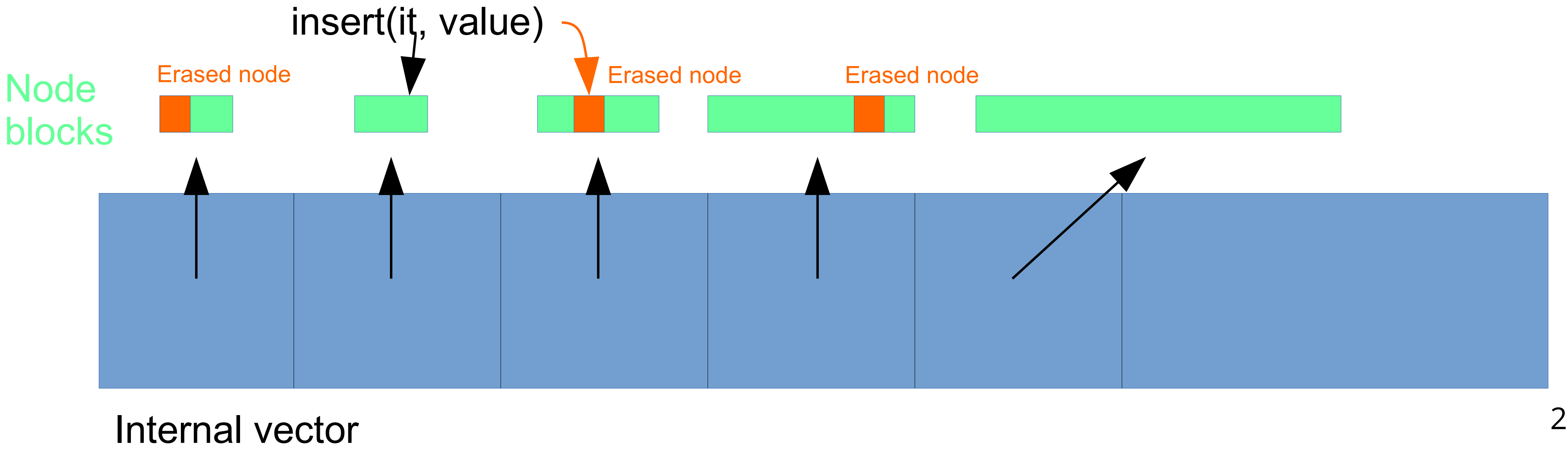
Inserting new elements

If no erased nodes are available, a new element is placed at back of last node block – regardless of insertion position and node order.



Inserting new elements

If erased nodes are available, we want to find the one nearest to the insertion point as defined by the insertion iterator parameter, in order to increase speed for iteration.



Re-use/search strategies

Ideal result: re-use location in memory as close to insertion location as possible (ie. location pointed to by `itr`in insert(itr, element);) to enable cache-friendlier iteration/processing.

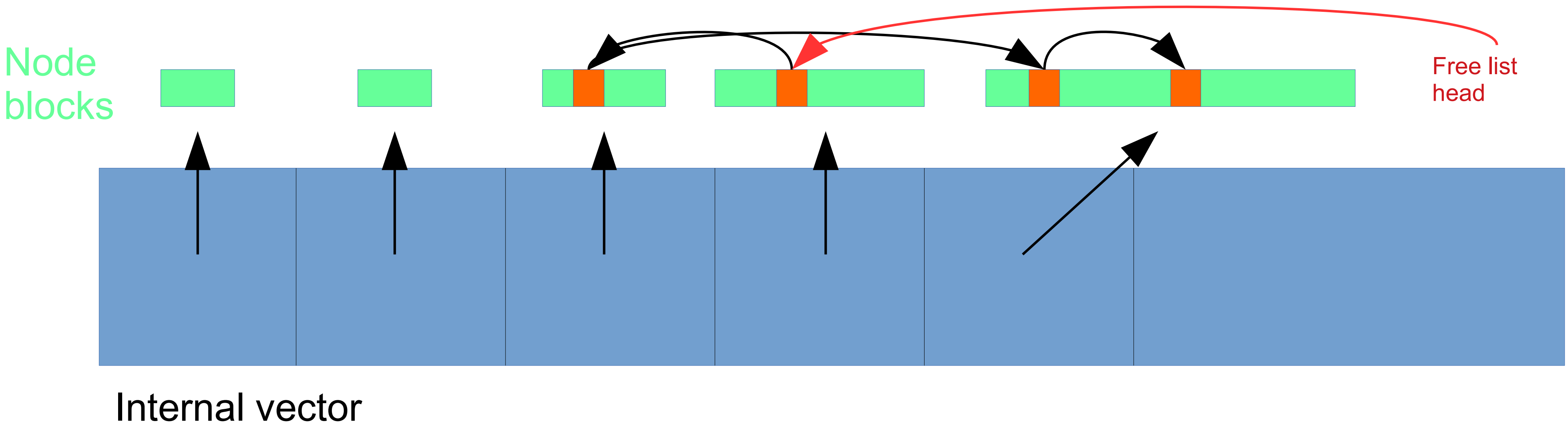
Three possible strategies to achieve this:

- Reversed jump-counting skipfield
- Global free-list
- Per-node-block free-list

Re-use strategy one: reverse skipfield

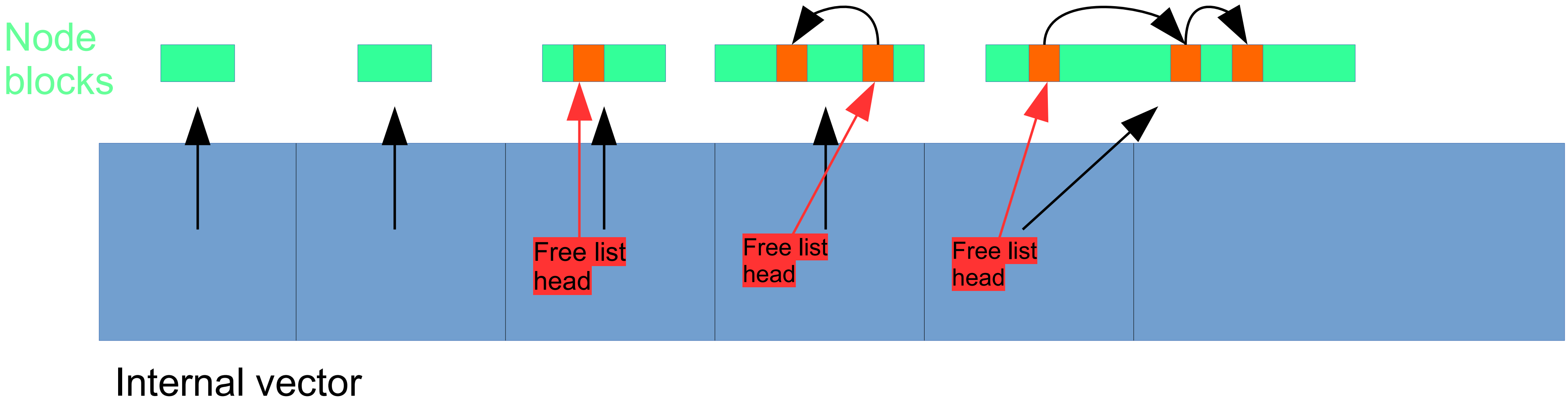
- Jump-counting skipfield: 0 0 1 0 4 2 3 4 0 0
- Non-zero = skipped
- Enables iterators to skip over runs of erased objects
- From any skipped node, can quickly find nearest non-skipped node
- Can use in reverse to find nearest erased list node from any given (non-erased) insertion point

Re-use strategy two: global free-list



- Use node pointers from erased nodes to form `free list`
- Very quick push and pop from free list
- Can process entire free list to find nearest location (but slow)

Re-use strategy three: per-block free-list



- One 'free list' head for each block
- Still very quick push and pop from free list
- Faster to find a re-use location close to insertion point

Strategy three sub-strategies

When there is no re-usable memory location in the same block as the insertion point, we can:

1) Scan all blocks from back or front block, or,
2) Scan left and right from insertion block, and/or,
3) Store a pointer to the last-used free list head.

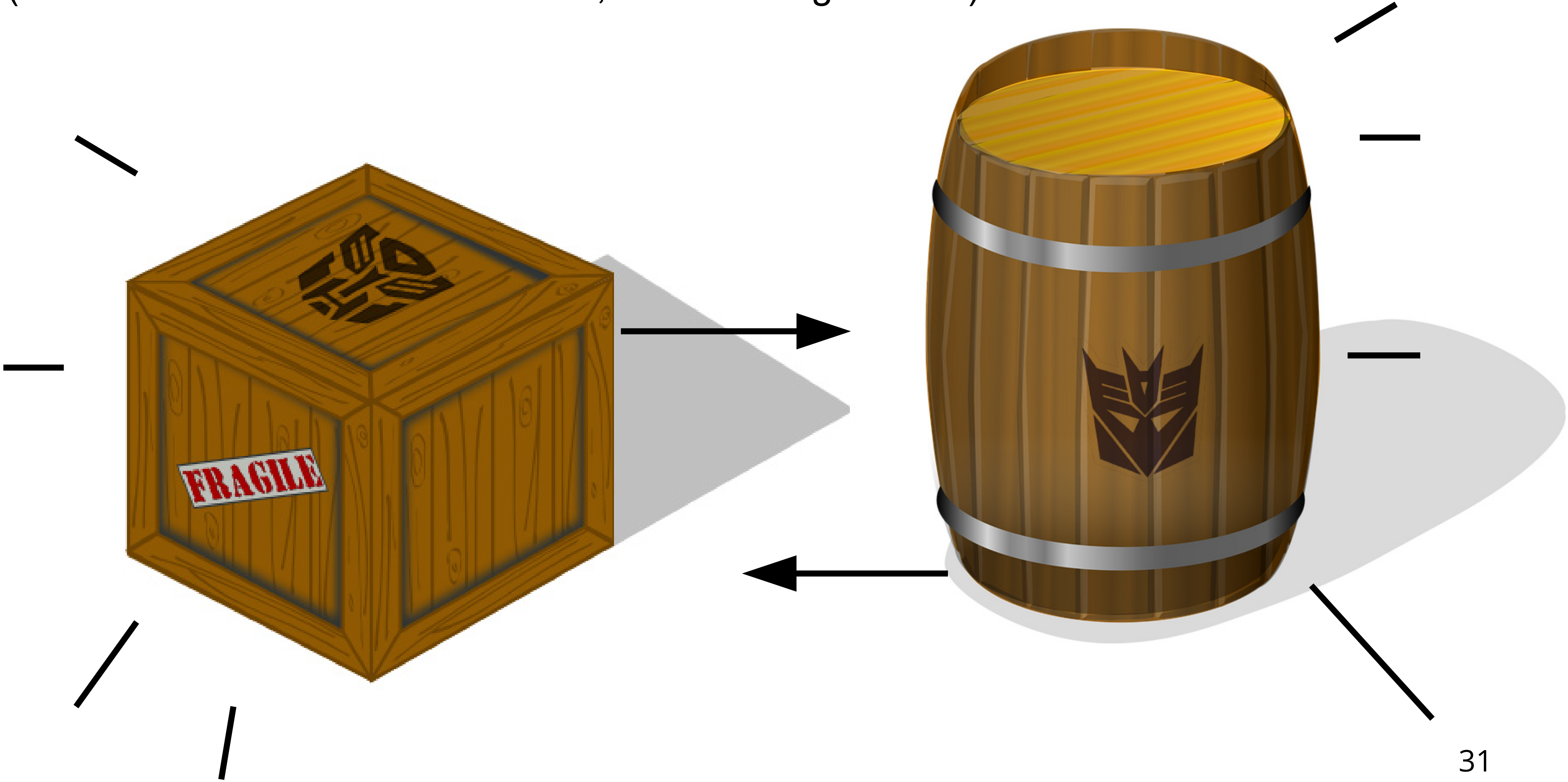
Combination of 2 and 3 turns out to have the best performance.

Node block size and additional notes

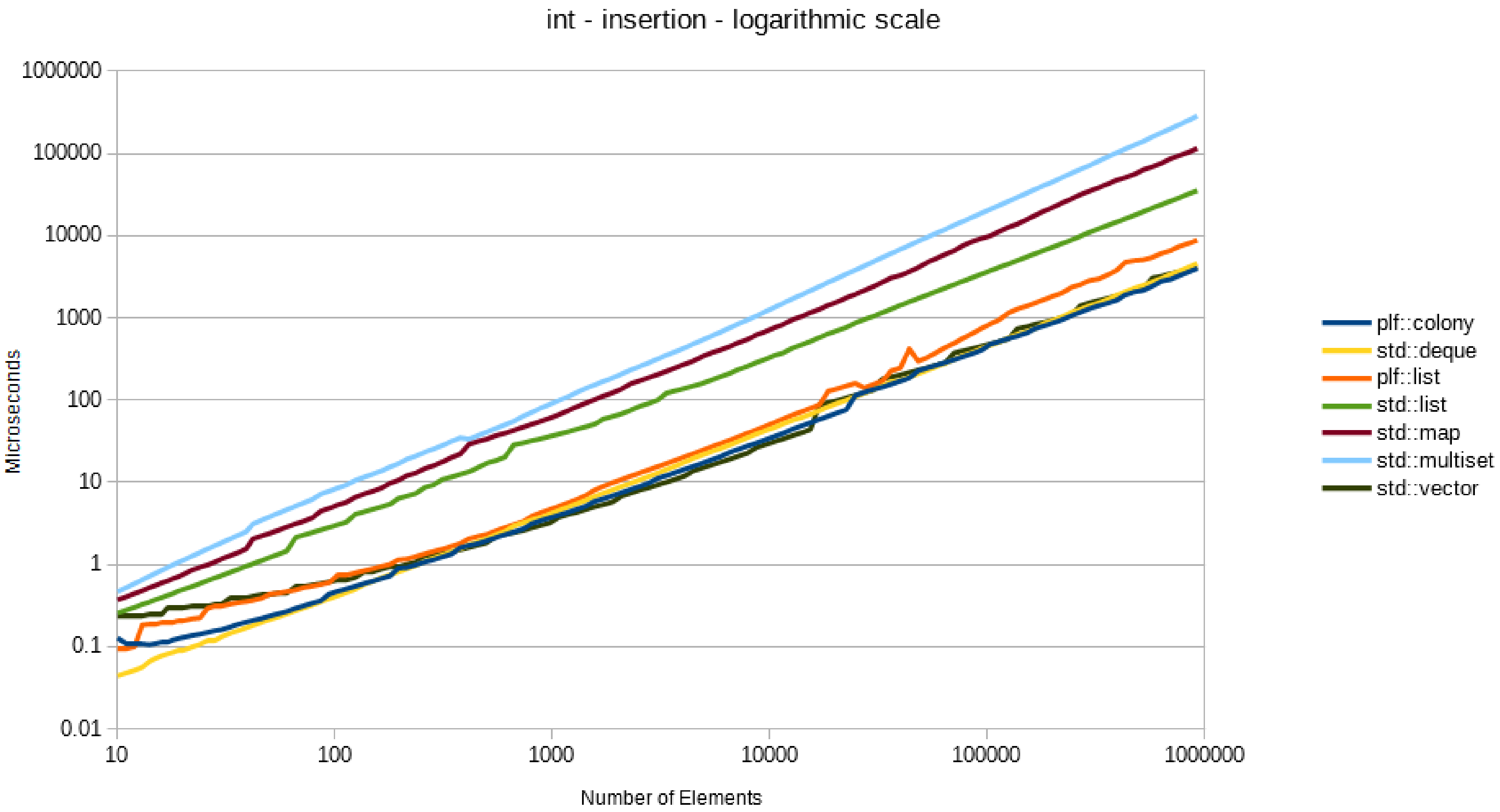
- Maximum group size: affects likelihood of any given free-list node being closer to insertion point. Can make 10-25% difference to overall performance.
- A month of benchmarking found ~2048 to be best max size overall, across types (with some outliers).
- Blocks of nodes mean we can iterate linearly instead of following list sequence for some tasks (sorting, destruction).
- Using 'next = previous' for free-list nodes indicates erased node during linear iteration ('next' is unused otherwise).

Container deathmatch!

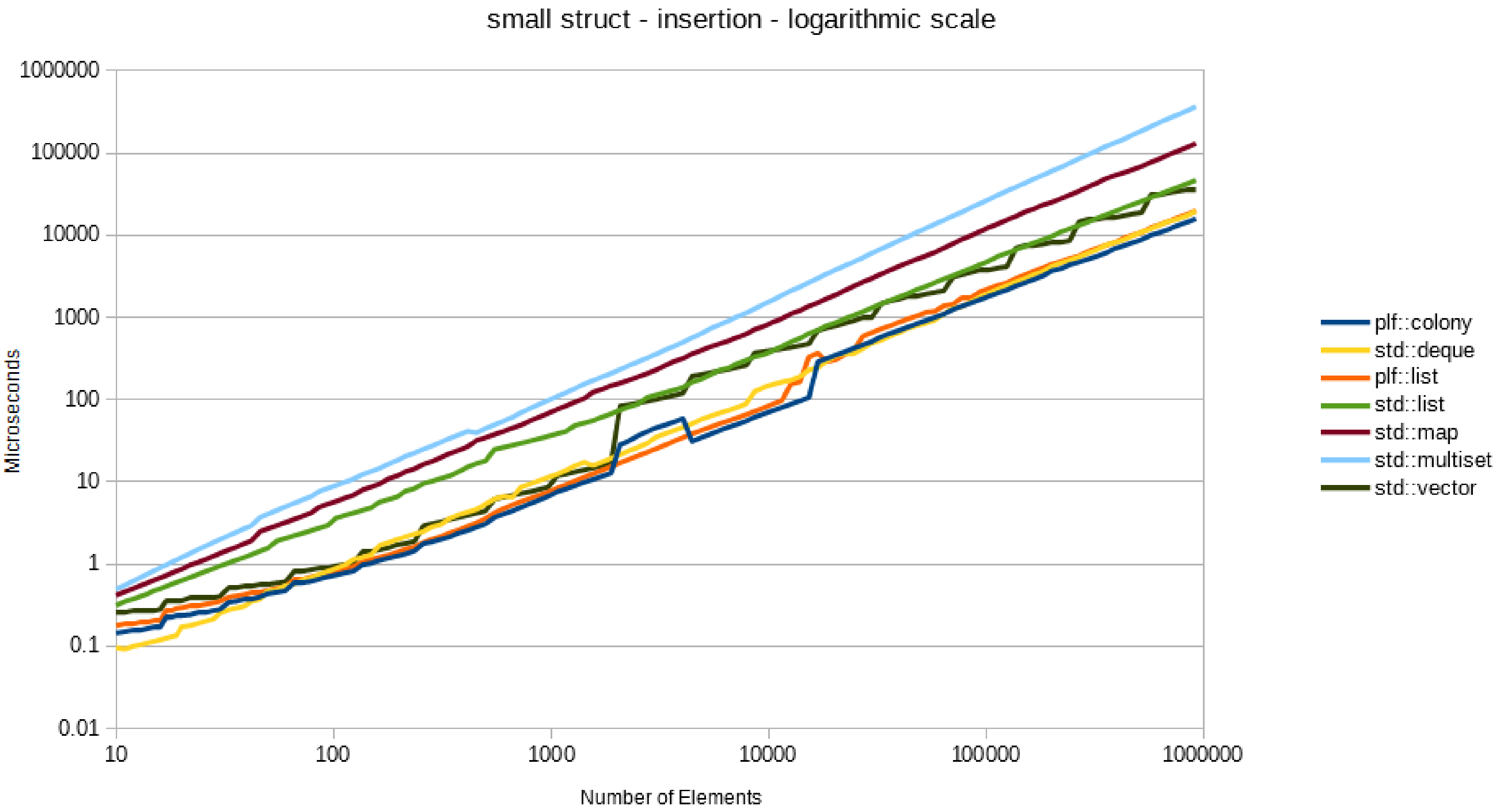
(GCC 7.1 x64 -O2 -march=haswell, both axis logarithmic)



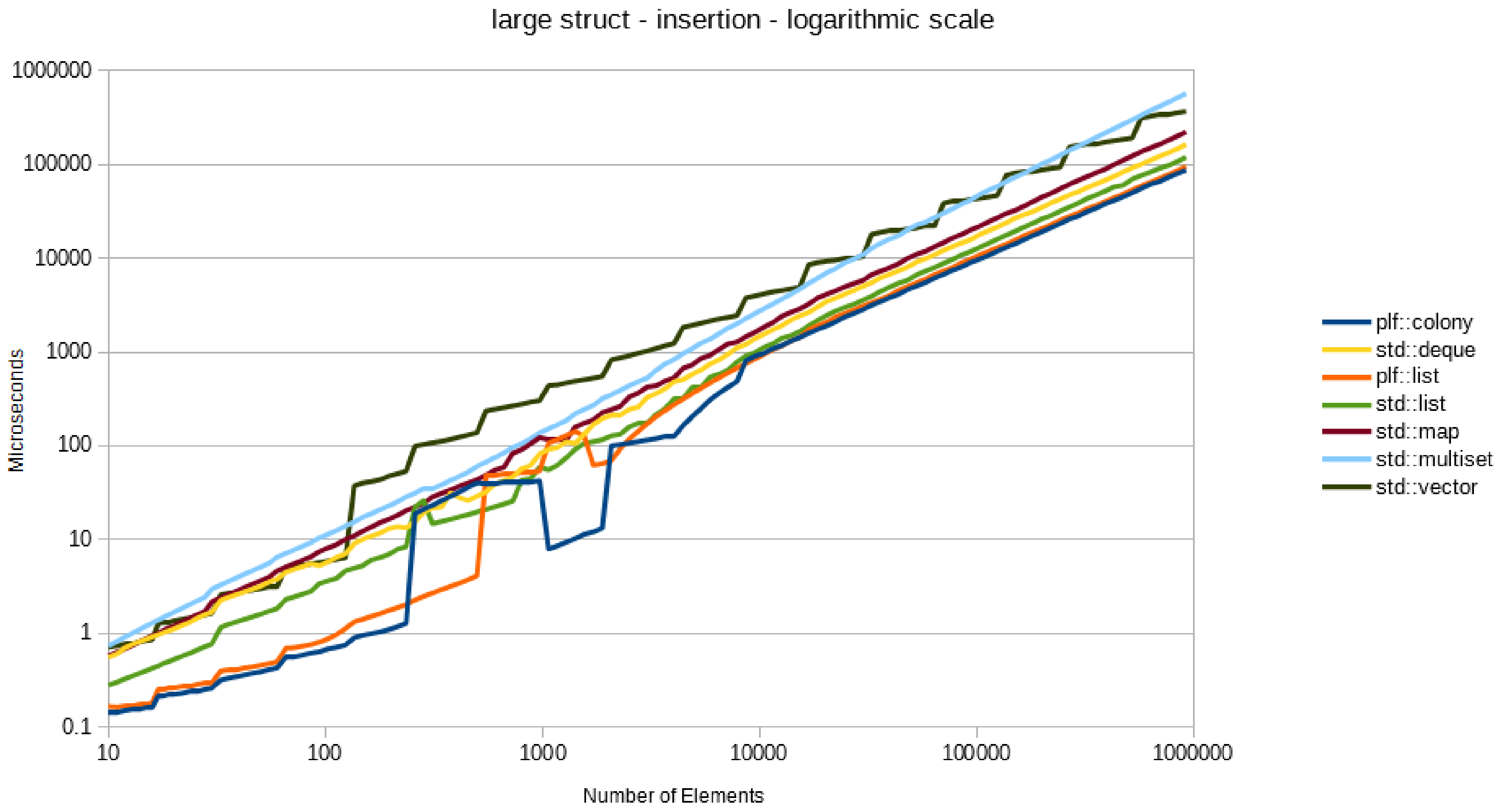
Insertion: int (2 bytes)



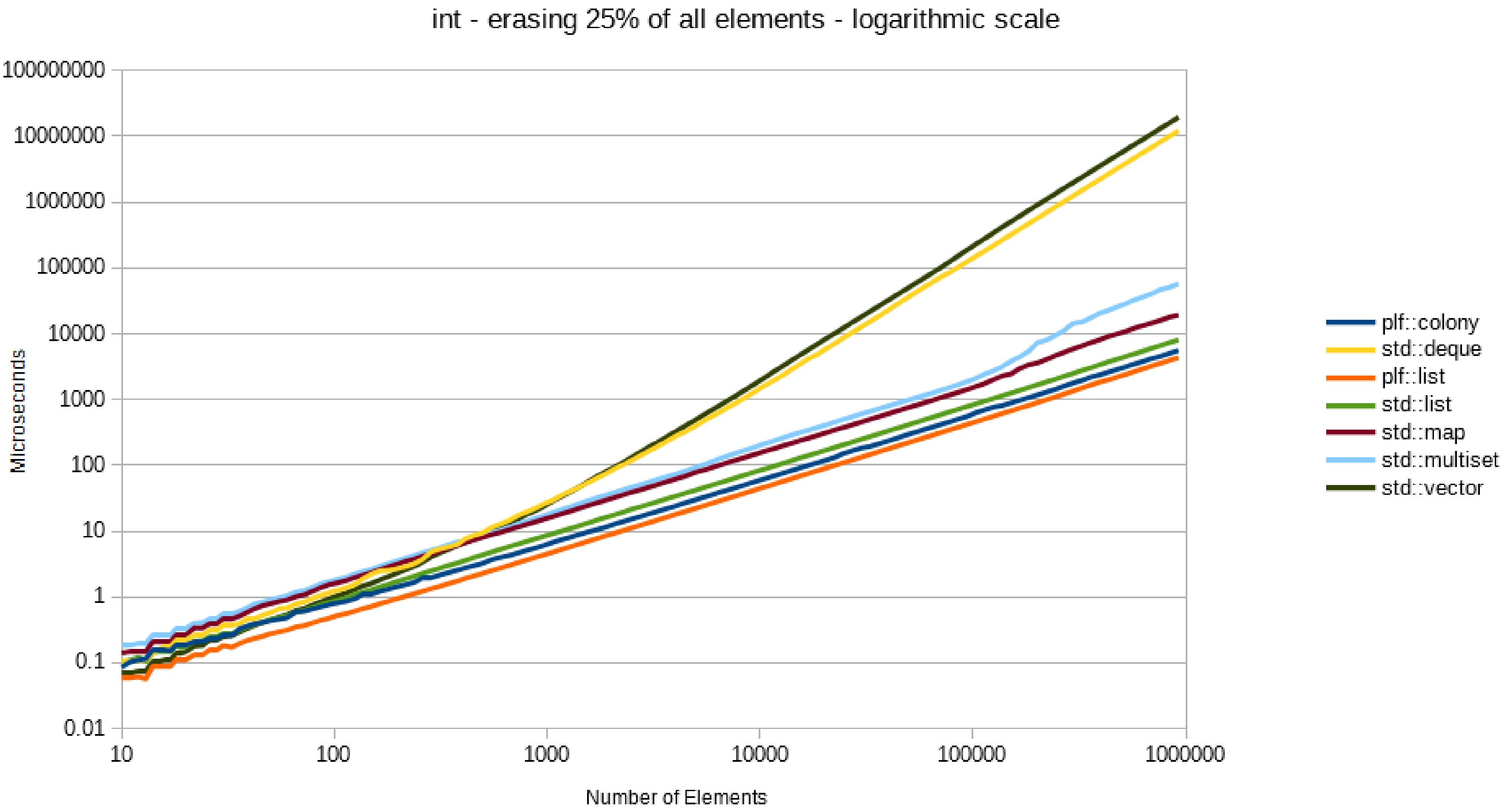
Insertion: small struct (20 bytes)



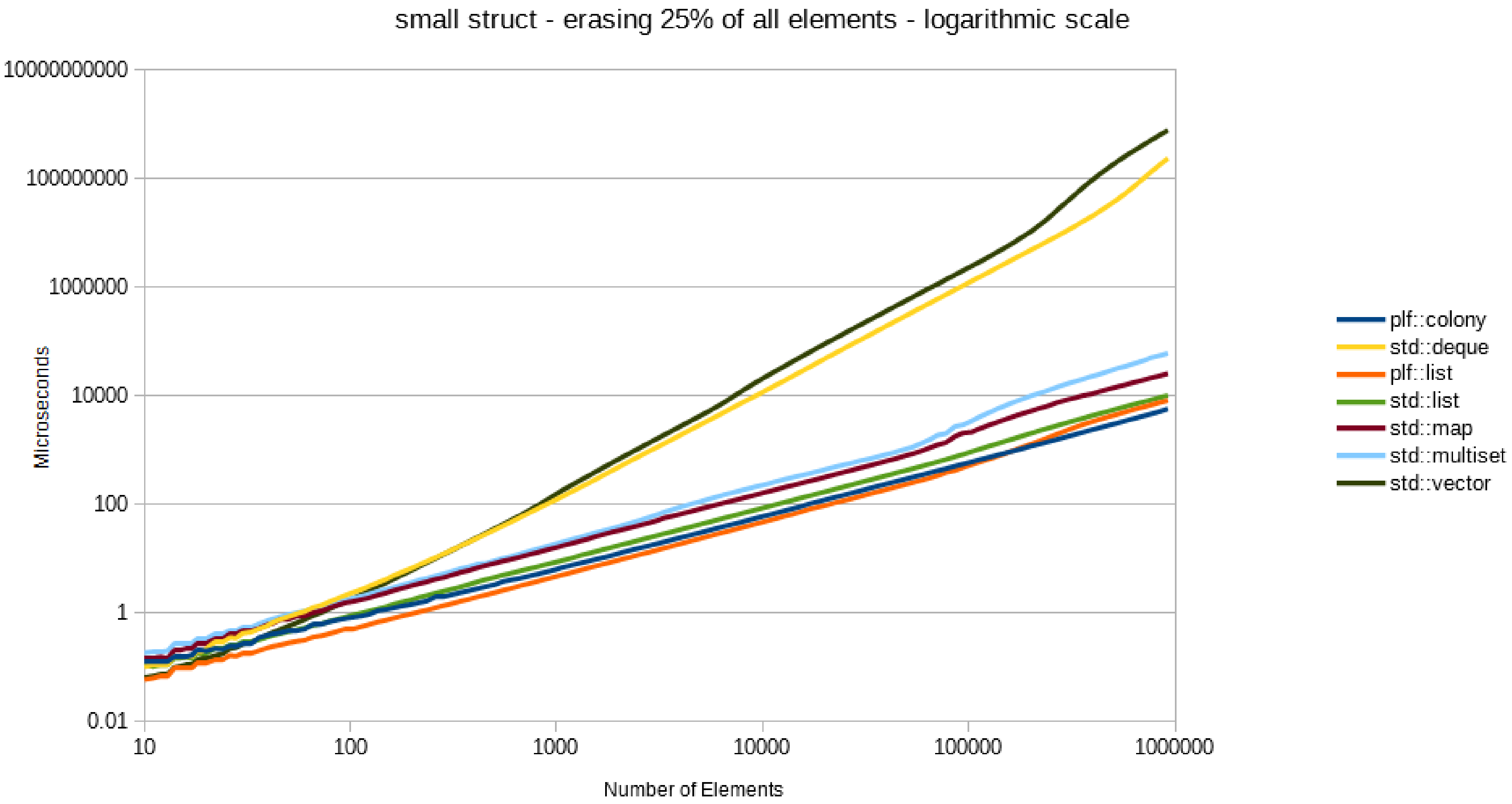
Insertion: large struct (270 bytes)



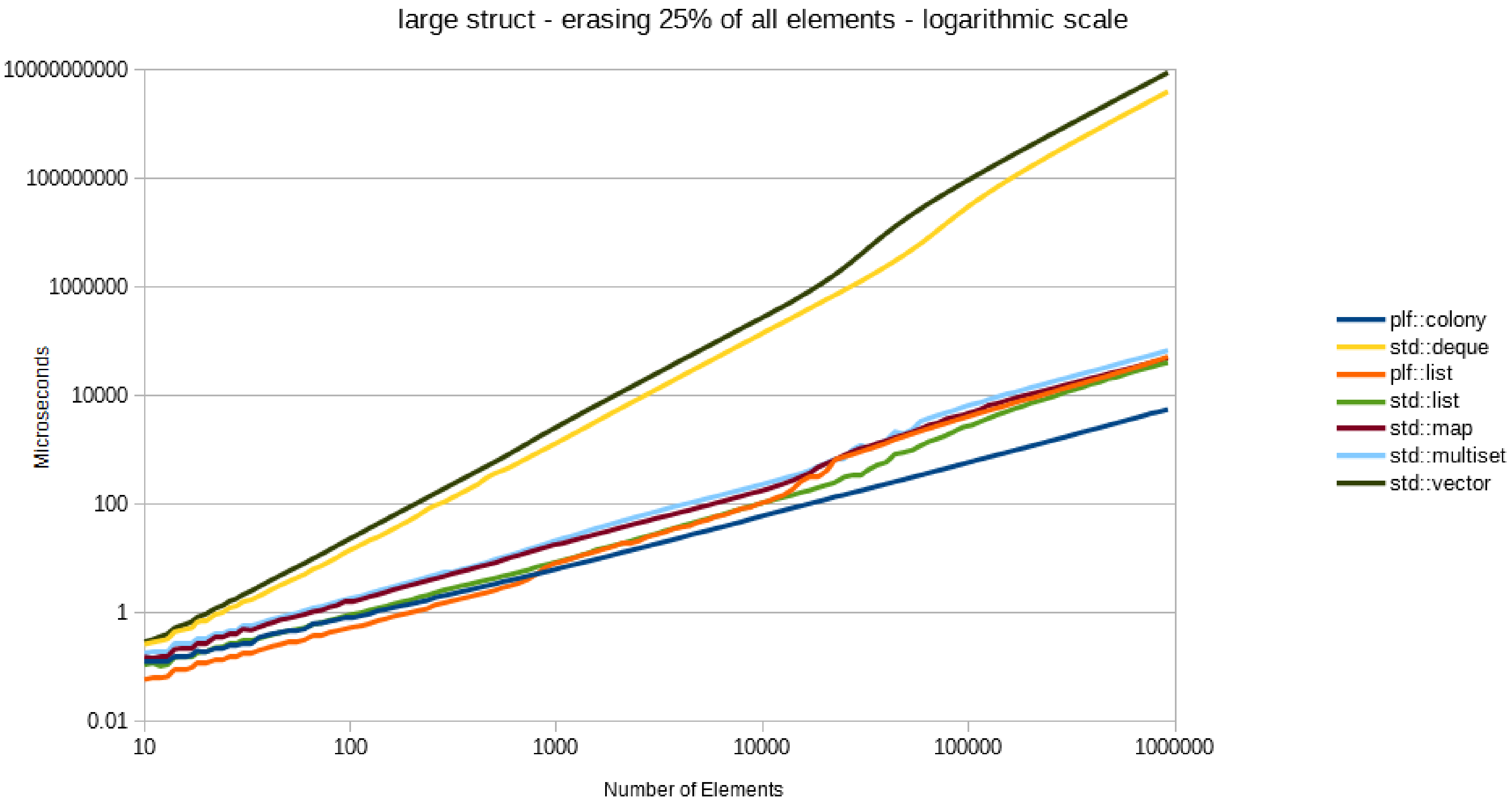
Erasure: int (2 bytes)



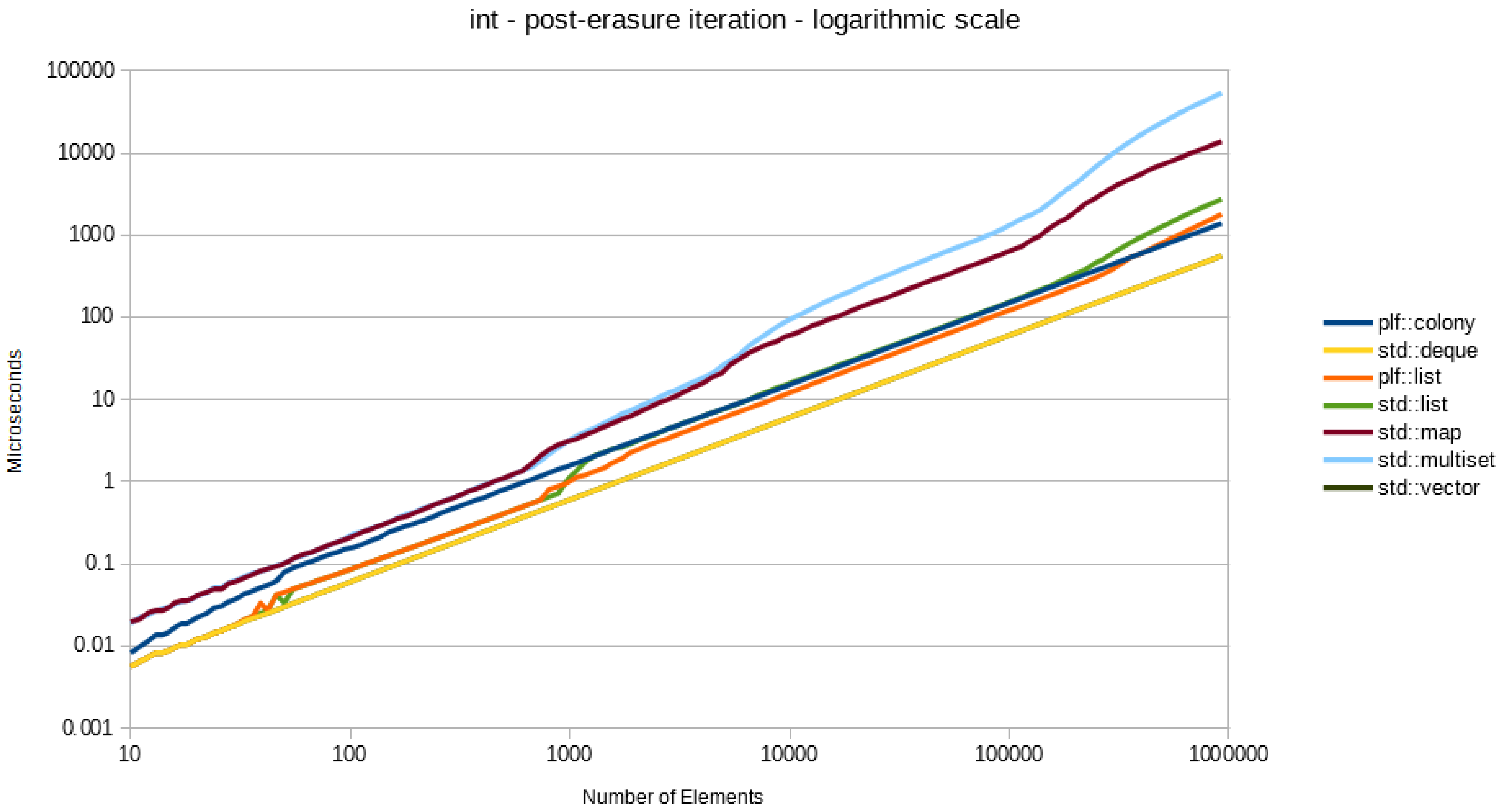
Erasure: small struct (20 bytes)



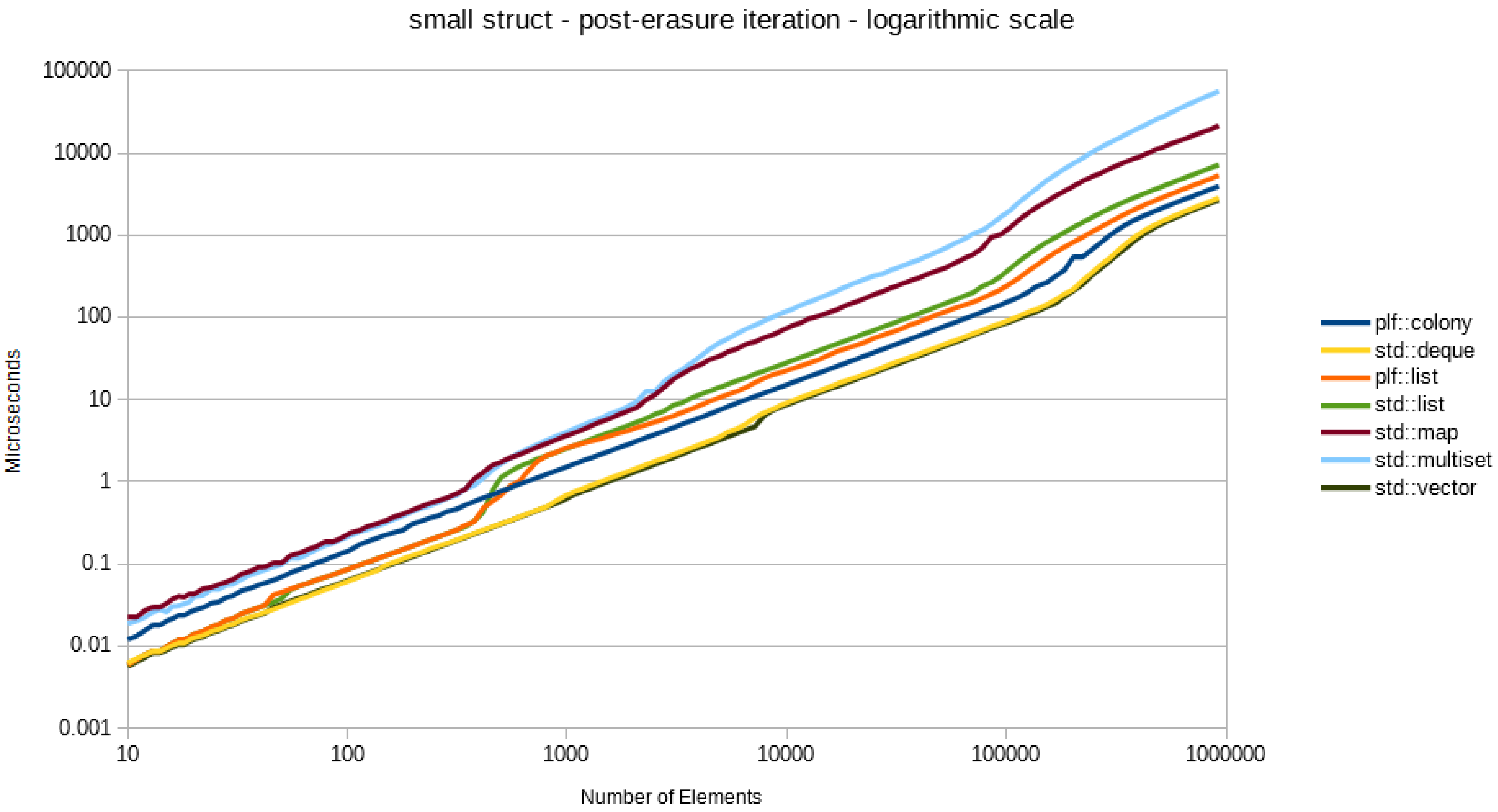
Erasure: large struct (270 bytes)



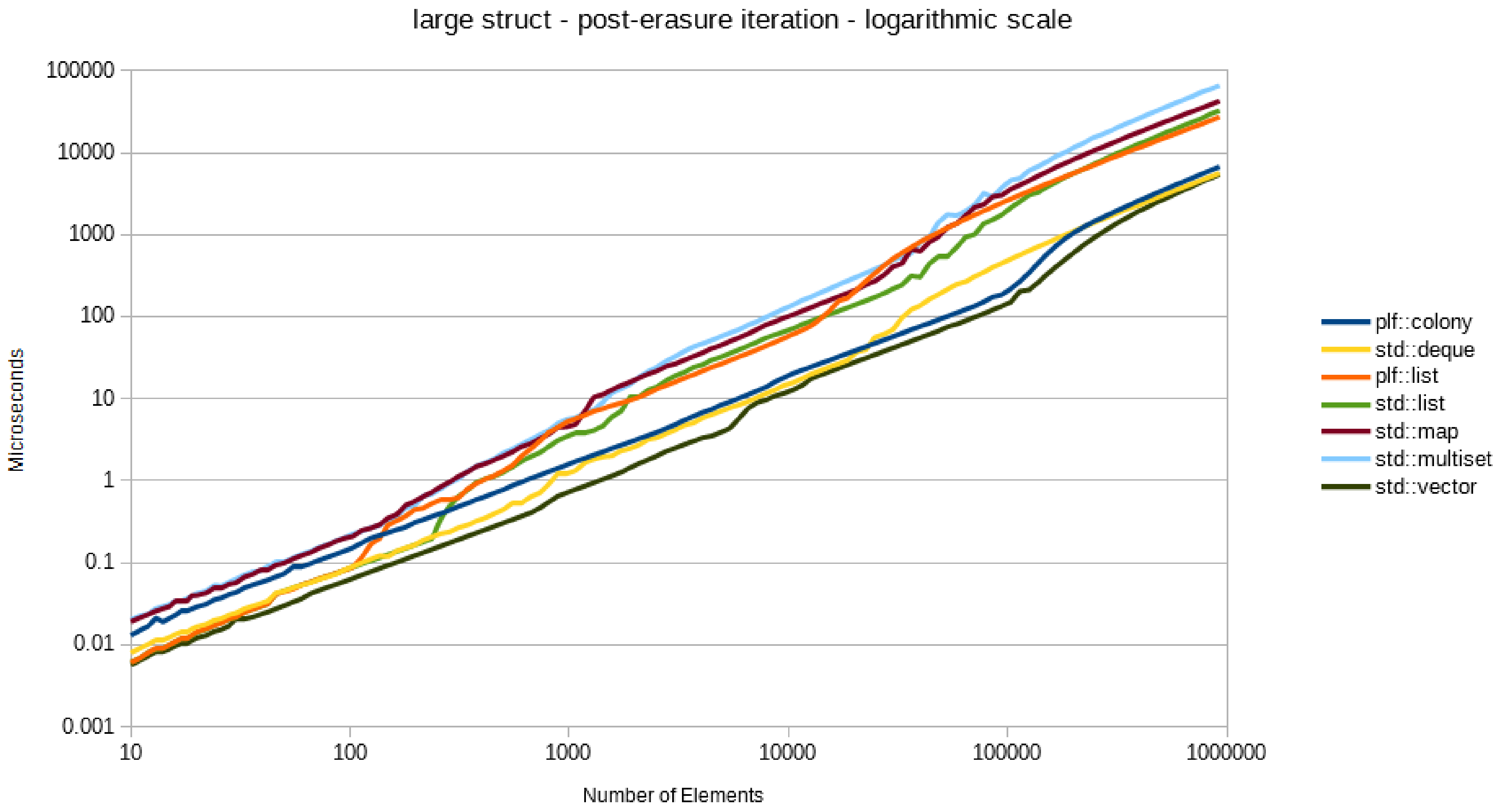
Iteration: int (2 bytes)



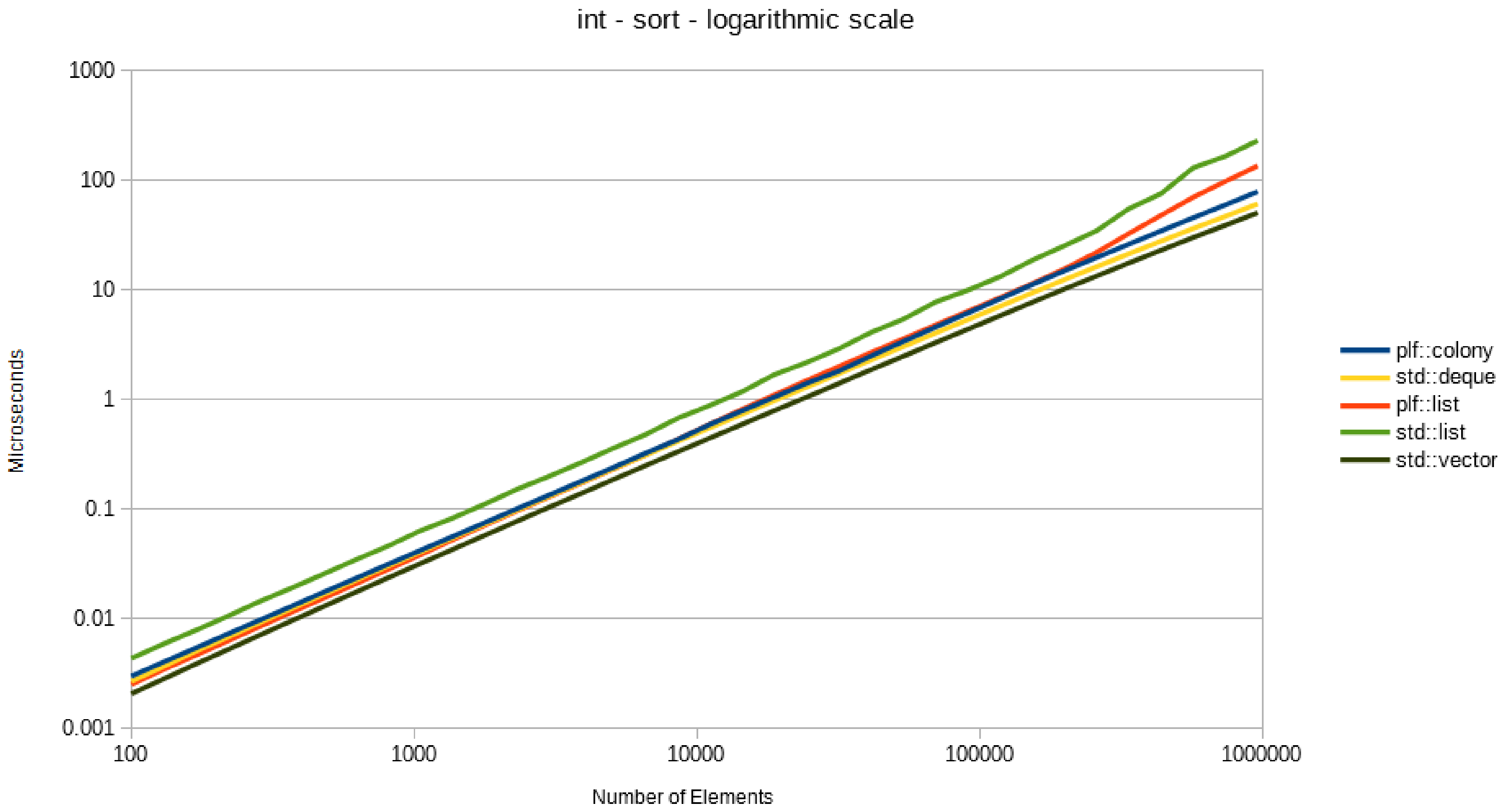
Iteration: small struct (20 bytes)



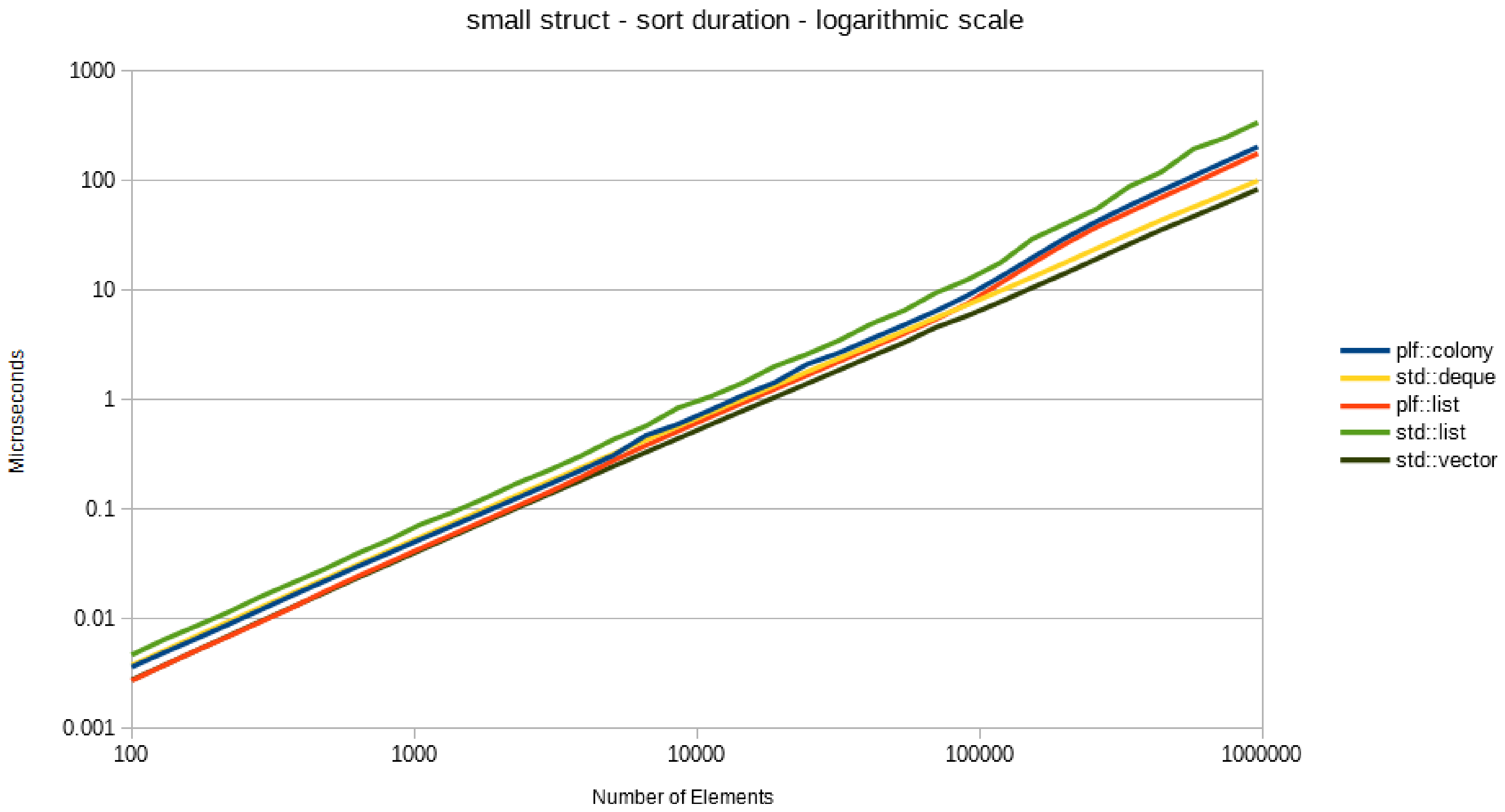
Iteration: large struct (270 bytes)



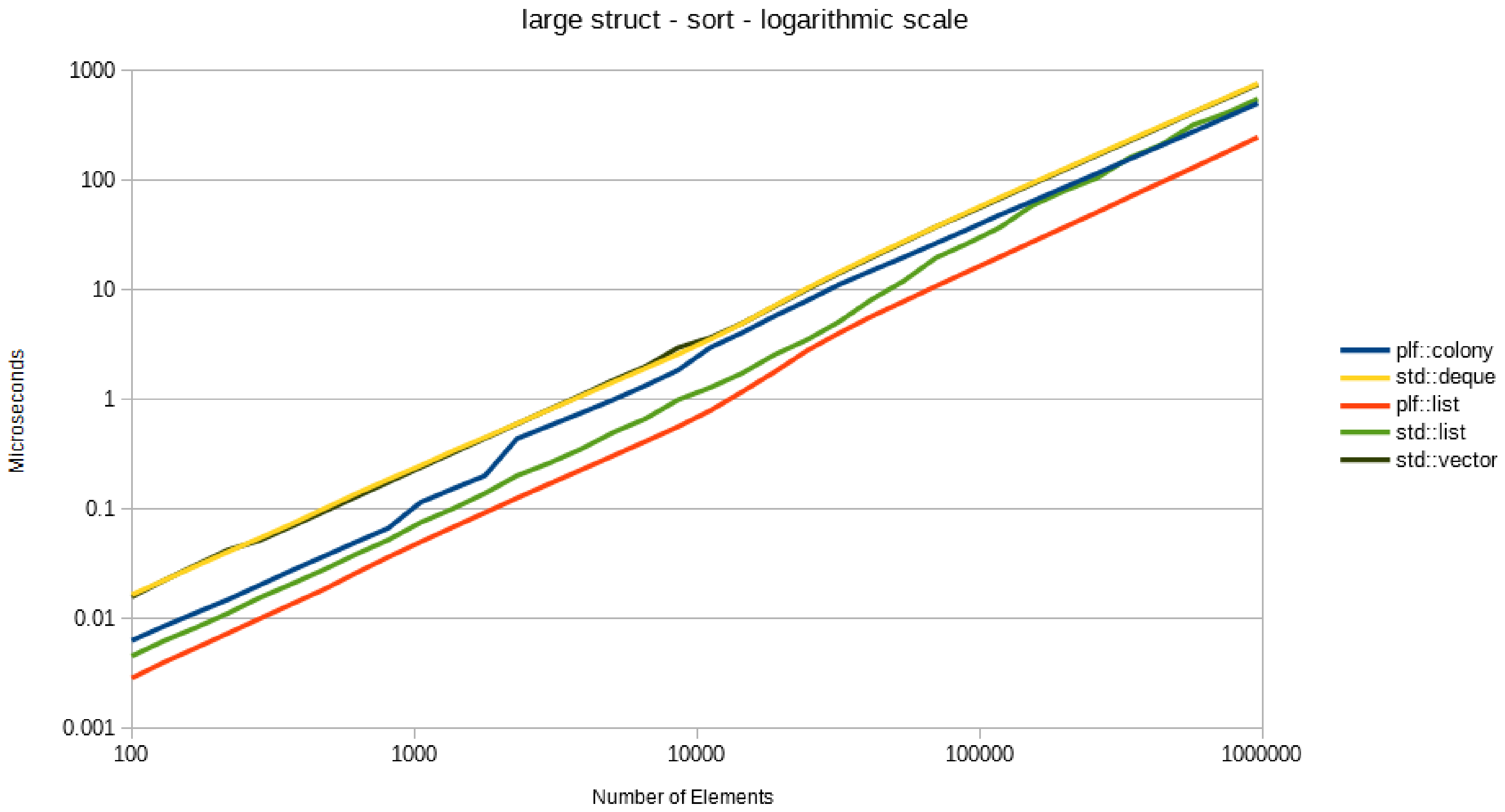
Sorting: int (2 bytes)



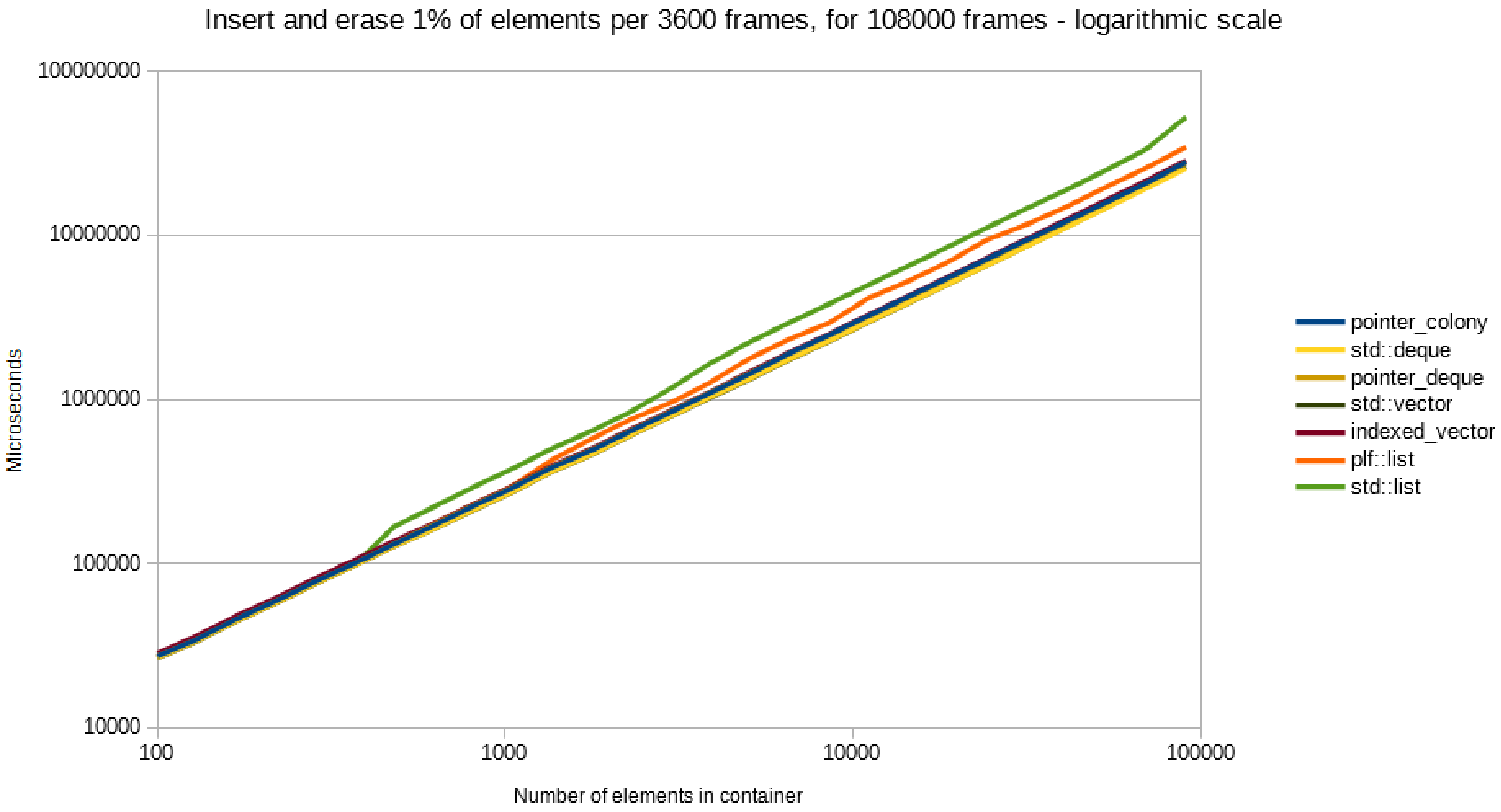
Sorting: small struct (20 bytes)



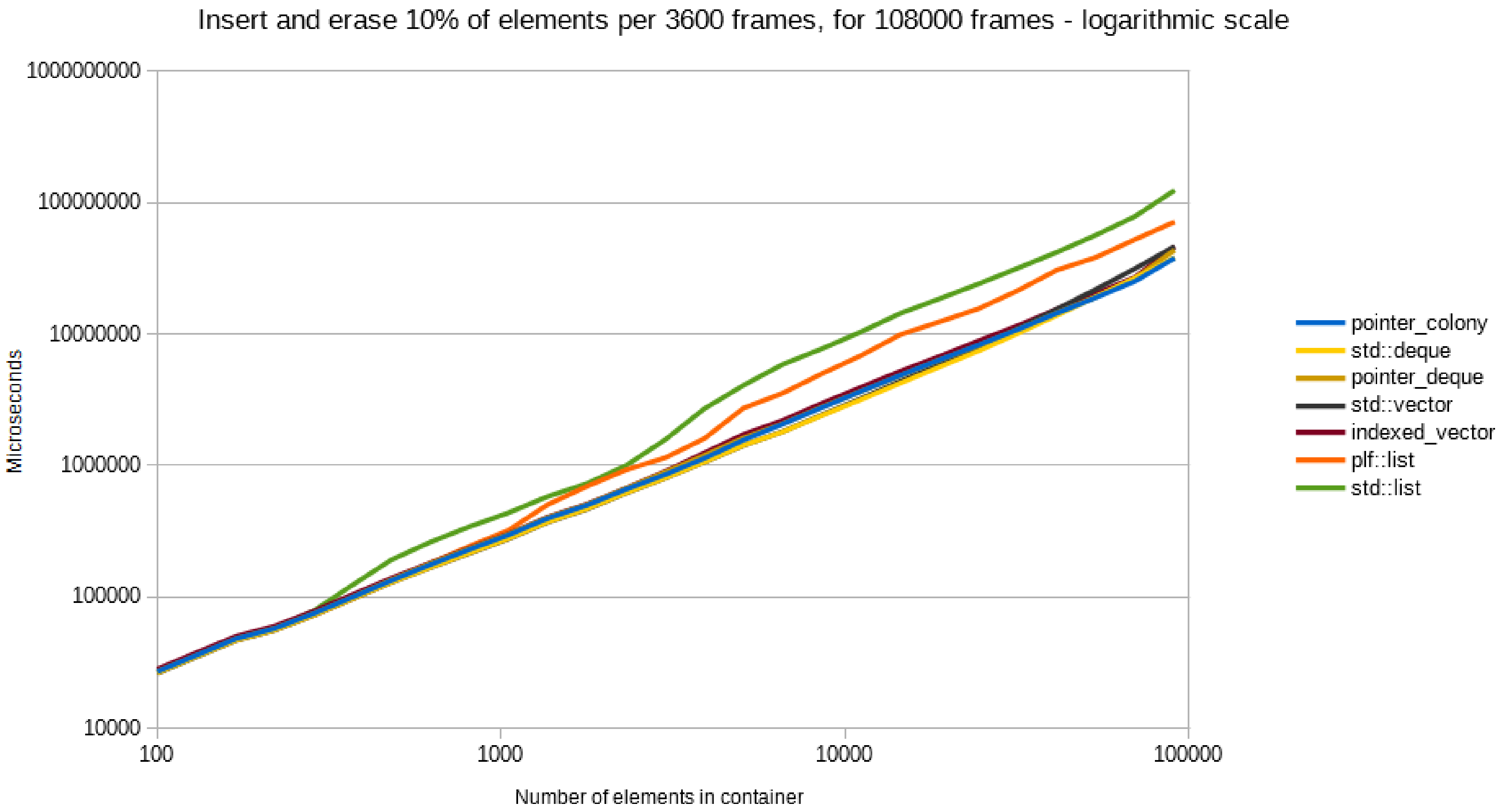
Sorting: large struct (270 bytes)



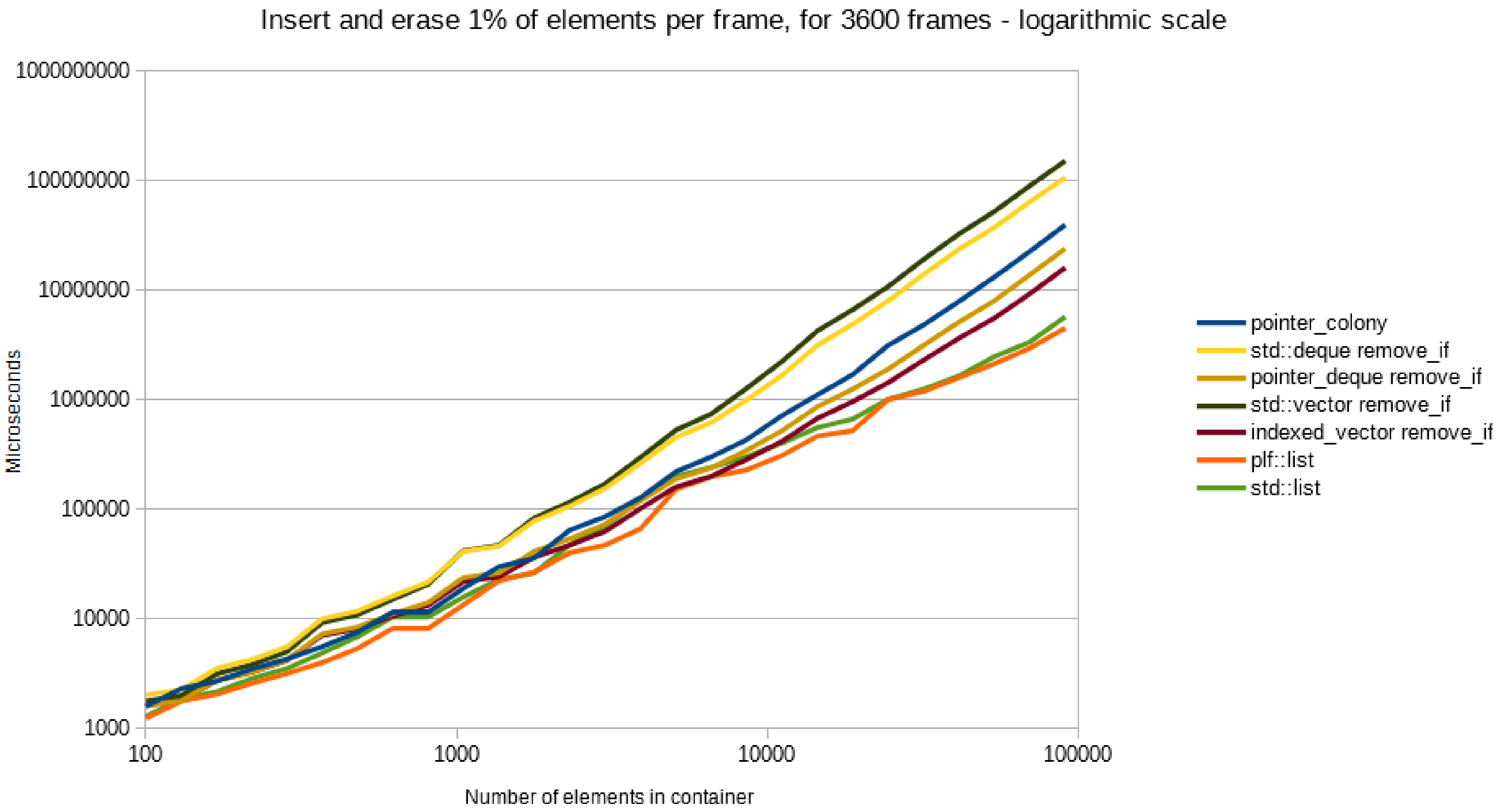
Ordered use-case: 1% per minute



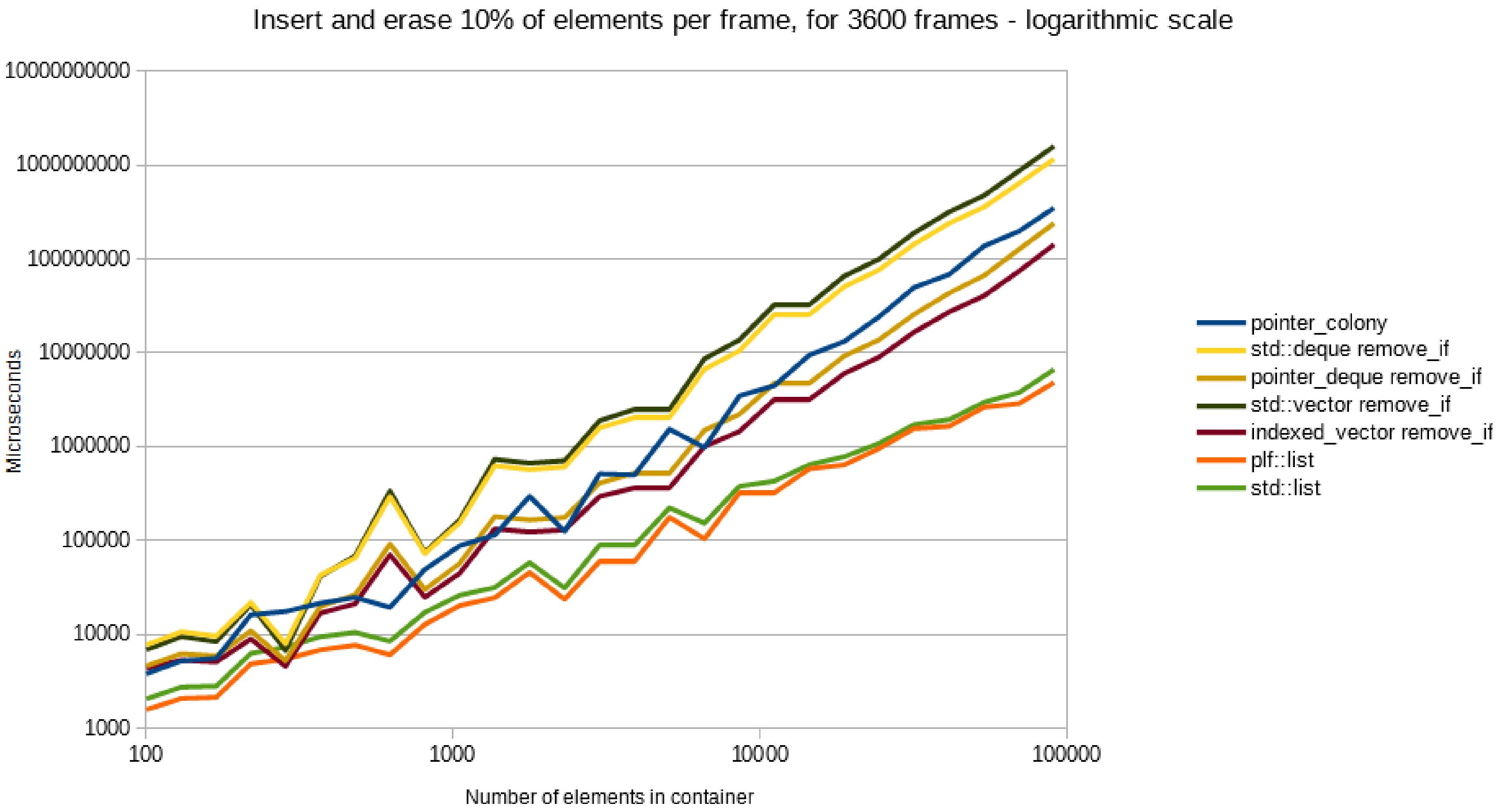
Ordered use-case: 10% per minute



Ordered use-case: 1% per frame



Ordered use-case: 10% per frame



Results summary

- Do use-cases for linked lists change? A little.
- For most part plf::list simply improves upon std::list's overall performance - thereby lowering the threshold at which a linked list becomes the better-performing container.
- When sorting medium-large structs, smaller numbers of small structs, and non-trivially-movable types, plf::list becomes preferable.
- If a programming scenario involves a fairly strong modification-to-iteration ratio you should use a linked list.

Could std::list be more like plf::list?

plf::list is mostly compatible with std::list. Main differences are:

- No partial splicing – full list only
- O(1) *amortized* single insert and erase instead of O(1)
- New functions: reorder, shrink_to_fit(), capacity(), reserve()
- Requirement for sort algorithm to be order-stable is deprecated
- Harder to make multithread-safe versions due to extra metadata (free-list heads etc would require mutexes/similar)

Proposal for std2::list possible, but not an amendment to std::list.

What I hope you've gotten from this talk:

- There are legitimate use-cases for linked lists purely from a performance perspective – intrusive and otherwise.
- In most cases another approach involving more contiguous containers will perform better, however:
- Sort, insertion, erasure, destruction and iteration performance ***can*** be largely improved by thinking outside the box.

Which leads me to my final point…

Ignorance is Bliss

- If you don't know anything, you're more likely to come up with things that other people don't know.
- Being indoctrinated in the 'way things are done best' typically ensures they will never be done better. eg. growth factors for std::deque.
- "Young people are the best at invention. They're the ones who run through a wall because they don't realize the wall's there."
- Ignorance has it's limits, but if you fill in too many of the blanks, there's nowhere the creative process to take hold.

“The British knew that for close tolerances in high quality ships, heavy machinery was necessary to cut metal accurately. Kaiser didn’t know this, and, anyway, he didn’t have heavy machinery. In his ignorance he told his workers to cut the metal using oxyacetylene torches. This turned out to be cheaper and faster than the traditional British methods. In his ignorance, Kaiser replaced riveting with welding, also cheaper and faster. ”

- Peter Drucker on The Value of Ignorance

www.plflib.org

- Matt Bentley, PacifiC++ 2017