Supporting document for Hive proposal 2: evidence of std::list usage in open source codebases

Audience: LEWG, SG14, WG21
Document number: P3012R0
Date: 2023-10-12
Project: Supporting document for std::hive proposal 2
Reply-to: Matthew Bentley mattreecebentley@gmail.com

Background

std::list represents a convenient solution for many developers in the sense that it guarantees pointer/iterator stability to elements regardless of the insertion/erasure of other elements, unlike deque/vector. But unlike deque/vector, it's iteration and insertion performance are low (erasure performance is high) due to the individual dynamic allocation of each and every element. In some cases people may get around the insertion speed problem by using allocators, but they cannot generally get around the slow iteration problem caused by elements not being contiguous in memory. And in practice most codebases do not use allocators with std::list. In addition the memory overhead for each list element is high (2 pointers).

The common solution for vector of using a swap-and-pop for non-back erasure in vectors invalidates pointers to every back element swapped, and another common solution, that of a vector of pointers to dynamically-allocated elements, has the same performance issues as std::list – albiet with a slightly lower memory cost (1 pointer per element).

In any area where order of elements is unimportant but pointer/iterator stability is important, std::hive is a better solution than std::list or a vector of pointers, due to allocating elements in blocks rather than individually, and erasing via notation rather than via reallocation (ala vector) or deallocation (ala list) of elements. Currently, many open source codebases use std::list, for reasons which will be explored. One of those reasons is that they have to be readable by many contributors, so they're unlikely to use non-standardised containers. Were those instances to be replaced with std::hive, it would improve performance in those codebases, and in most cases improve memory use.

Hive vs std::list performance metrics

On average across ranges of elements from 10 to 1000000 in 1.1x steps (hence the averaged results are skewed toward smaller numbers of elements as these receive the most measurements), and across 5 types (1byte, 2byte, 4byte, 40byte and 490byte) the average performance characteristics are as follows (see the "raw performance benchmarks" on the plf::colony benchmarks page for sources and test setup):

- Insertion (singular): 520% faster than std::list (ie. 5.2x as fast) – speed increase is greatest for smallest type and decreases as type grows larger.
- Erasure: 71% faster – speed increase is greatest for large types and decreases as type gets smaller.

- Iteration after erasing 25% of elements: 50% faster - speed increase is greatest for large types and large numbers of elements, as the CPU is able to cache a certain number (~1000 on my test setup) of jump destinations – so once the number of elements gets above that number, hive outperforms list significantly (up to 218%).

Hence we can see why it would be advantageous to switch to hive where appropriate.

Open source project investigation

With that in mind I thought I would see how many modern open source projects are in fact using std::list versus vector or others despite it's performance issues. I downloaded the current trending C++ repositories on github (https://github.com/trending/cpp) as well as Libreoffice and QT-core. In total 13 large codebase repositories:
tensorrtx
SFML
libreoffice
latte-dock
json for modern C++
googletest
FreeCAD
fheroes2
ethersweep
dlib
ClickHouse
brpc
QT-core

Of the 13, 7 used std::list (projects which used std::list only in test/demonstration material rather than library code are not included in this result). Libreoffice used it extensively, even citing it's use in places as specifically for the purposes of retaining stable pointers to elements. In freecad and clickhouse it's use was ubiquitous. QT also used it considerably, the worst instance (in terms of performance) being a list of unique pointers to class instances which themselves contain pointers to graphics buffers.

Outside of libreoffice I could find no instances where it was documented as to why std::list was being used instead of more contiguous containers, so it is difficult to ascertain whether (a) order was important (b) pointer/iterator stability was important. However in both freecad and clickhouse they were extensively used to splice one std::list into another, or in some cases, a range from one list into another, or using splice to move a range around within a list.

Hive supports full-container splicing but not range-splicing. The latter can only be achieved via std::move'ing individual elements and erasing the originals, but with the caveat of losing pointer/iterator stability. Still, range-splicing was the exception rather than the norm, in these codebases at least.

Summary

- std::list slow, hive fast.
- Most devs do not accomodate std::list's slowness in insertion/erasure by using allocators.
- Allocators do not solve std::list's iteration slowness.
- Many open source frameworks and applications are using std::list, either for splice, iterator/pointer validity to elements regardless of insertion/erasure, or in ignorance of the performance costs.
- Where appropriate (unordered element scenarios, range-splice not needed), switching to hive (or providing hive as an alternative for new projects) will dramatically increase performance in these scenarios.
- Not having hive in the std:: will mean these projects will probably not use it, as the projects are designed to run on multiple platforms under multiple architectures and have many unfamiliar eyes looking at the code, hence a desire for use of well-understood std:: containers.
- ie. Not having hive in the standard == worse performance in open source projects.