Proceedings of the 2018 Winter Simulation Conference
M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, eds.

OPTIMIZING DATA STRUCTURES FOR HIGHLY DYNAMIC CONTENT IN COLLECTIVE, ADAPTIVE SYSTEMS

Till Köster
Felix Hauptmann
Adelinde M. Uhrmacher

Institute of Computer Science
University of Rostock
Albert-Einstein-Straße 22
18059 Rostock, GERMANY

ABSTRACT

In discrete event simulation of collective, adaptive systems (CAS), it is necessary to store all the entities of the system in some data structure. However, collective adaptive systems, which are characterized by a high fluctuation of entities, pose a challenge for typical data structures. To address this problem we developed the sequential pile container and evaluated its performance based on a set of benchmarks and in comparison to the data structure Set and unordered_Set from the C++ template library and a recently developed data structure, i.e., plf::colony. The performance of plf::colony and the sequential pile proved overall superior in these benchmarks, and performed equally well in inserting, copying and iterating over all entities. Sequential pile outperforms plf::colony at deleting elements.

1 Methods

Well established data structures like set, list and unordered_set from the C++ standard template library meet the basic requirements for handling entities in discrete event simulation, such as a) dereference - every entity needs to have an identifier, that allows to dereference it, b) add/remove – it has to be possible to remove and add entities. Also, those operations shall not invalidate the identifiers used to dereference entities, and c) iterate – frequent iterations over all entities need to be supported. However, these data structures are not tuned to the simulation of collective, adaptive systems which are characterized by frequently adding and removing entities. This became apparent when investigating discrete simulations of cancer treatments, where there is a high volatility in the number of cancer cells, which grows (exponentially) over time and then quickly decreases when applying the treatment.

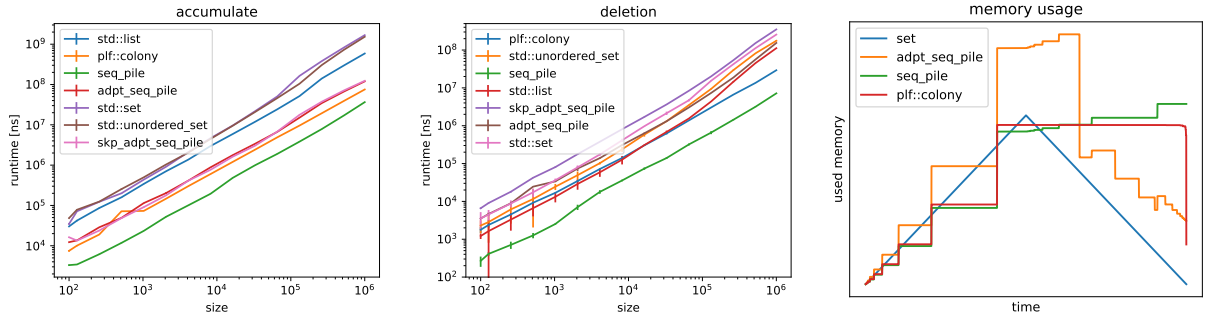
To address this problem we developed the *sequential pile* container. It consists of a data array and a bit array indicating whether the corresponding data in the data array is still valid. To insert an element, it is stored in the data array and true is stored in the bit array. If an element is to be inserted when the sequential pile is full, the whole data structure is copied to a larger (usually twice as large) chunk of memory (as done by standard C++ vectors) before inserting the element. To erase an element, the data remains in the data array, but the Boolean in the bit array is set to false, indicating the data is invalid. The index of the erased data is stored in a stack. When inserting a new element it will be stored at the first index in this stack, overwriting the invalid data.

In addition to the sequential pile, we tested two specially adapted variations: the adaptive sequential pile and the skip-enabled adaptive sequential pile. Both reduce their memory usage when the number of invalid entries in the data becomes too large, by reordering the elements and releasing memory thereafter. The skip optimization stores the distance to the next valid element with every entry in the sequential pile, increasing iteration performance. This skip list approach is similar to what is done by the plf::colony

datastructure, which originated in the gaming industry and solves a similar problem of storing changing amounts of data (Bentley 2017).

2 Results

We have benchmarked a selection of data structures across a broad range of sizes for different operations using the *Google Benchmark* (https://github.com/google/benchmark) framework and synthetic uniform workloads. We also tested across different fill rates, and found it has an impact on the result of the accumulation benchmark, where seq-pile shows a good performance at higher fill rates similar to plf::colony (Figure 1a), whereas the plf::colony and skip-enabled sequential pile perform better for lower fill rates. Omitting the skip lists in the sequential pile leads to significant improvement in the cost of deleting an element of almost one order of magnitude (Figure 1b). Furthermore, for deletion, the adaptive variant performs worse, because it has to restructure the elements in order to allow the release of memory. Similar, but less pronounced effects can be observed for insertion.



(a) accumulation time cost in dependence on container size.

(b) time cost of deleting 95% of the elements in dependence on container size.

(c) memory usage when filling and then emptying the container.

Figure 1: Measurements for different operations.

Finally, looking at the containers' memory usage (Figure 1c), we can see how the sequential style containers bulk-allocate memory. Interesting behavior is observed when starting to delete elements, where the sequential piles start to increase in size, because they have to remember the empty spots (plf::colony does so via the skip lists).

In conclusion, based on the time cost benchmarks, plf::colony and sequential pile are best with respect to inserting, copying and iterating. Sequential pile is faster at deleting elements. Both its other variants are performing slower in most benchmarks, but have the advantage of being able to release memory. Set and unordered_set do not perform well in most benchmarks.

ACKNOWLEDGMENTS

This work was supported by the DFG (German Research Foundation) UH66/13 "ESCeMMo".

REFERENCES

Bentley, M. 2017. "The Advanced Jump-Counting Skipfield Pattern". *The Computer Games Journal* 6(3):153–169.