Karlsruher Institut für Technologie
Fakultät für Informatik

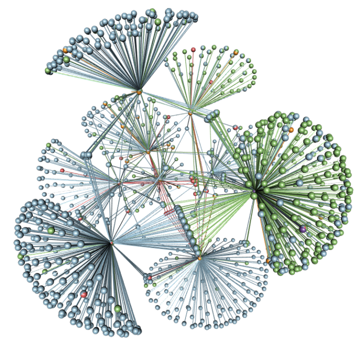
Lehrstuhl Algorithm Engineering
Prof. Dr. Peter Sanders

Master's thesis

Relaxed Priority Queues with Dynamic Quality

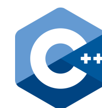
Description

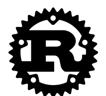
Priority queues are a fundamental data structure used in many algorithms, such as Dijkstra's algorithm or task scheduling. To utilize the parallel nature of modern hardware, these algorithms need to be parallelized. However, parallel priority queues are often a bottleneck due to inherent access conflicts. This problem can be mitigated by relaxing the priority queue semantics to allow out-of-order removals. Using a relaxed priority queue within a concurrent algorithm typically leads to additional work before finding the solution, but the improved scalability often leads to a lower execution time.
The MultiQueue[1] is the state-of-the-art relaxed priority queue. It uses multiple internal priority queues where the number of queues is proportional to the number of threads. Insertions distribute the elements among these queues randomly. Removals select two queues randomly and remove the element with the highest priority among them. The *quality* (degree of relaxation) of the MultiQueue scales with the number of internal queues and the number of sampled queues during removals, which both are fixed at runtime. This is problematic for applications where the degree of available parallelism changes dynamically, for example when traversing the graph on the right. A relaxed priority queue that can adapt its quality can dynamically balance between high scalability and high quality as needed.



Goal of the Thesis

The goal of this thesis is to develop and implement a variant of the MultiQueue that dynamically adjusts its quality based on the available parallelism. The `C++` implementation of the MultiQueue[2] can be used as a starting point. The simplest approach for the dynamic adjustment is the number of considered queues in the removal operation, but more advanced strategies should also be considered.
The data structure should be evaluated on a variety of benchmarks, including stress tests and real-world applications. A theoretical analysis of the runtime and quality complexity is also part of the thesis. We suggest building the implementation on top of the existing `C++` MultiQueue[2], but it can also be done in `Rust` if desired.





Requirements

- Solid foundation in (concurrent) algorithms and data structures
- Experience in `C++` or `Rust`
- Experience in parallel programming is a plus

[1] https://arxiv.org/abs/2107.01350
[2] https://github.com/marvinwilliams/multiqueue

Contact: Marvin Williams (williams@kit.edu) and Kåre von Geijer (kare.von.geijer@chalmers.se)
Informatikgebäude am Fasanengarten, Raum 206