RangeQueue: A Priority Queue algorithm

Abhay Shankar K: cs21btech11001 Kartheek Tammana: cs21btech11028
Rajapu Jayachandra Naidu: cs21btech11050

December 3, 2023

Abstract

This document is a report for the Final Project of the course CS5300: Parallel and Concurrent Programming, regarding a novel implementation of a lock-free data structure.

Introduction and Related Work

A priority queue supports two operations:

- `insert(key)`: Insert a key into the queue.
- `deleteMin()`: Delete and return the key with the lowest key from the queue.

Our implementation is based on the TSLQueue data structure proposed by Rukundo and Tsigas. In a TSLQueue, keys are stored in a binary search tree, and each node contains a `next` pointer to the node with the next lowest key. This allows for `insert` in log time, and constant time `deleteMin`, while still being parallelizable.

While the TSLQueue only stores a single key in each node, we generalize it to hold a range of keys per node, using a bitvector

Optimisations

The RangeQueue is an improvement over the TSLQueue in the following ways:

- Cache locality: Since each node represents a range of keys, the number of nodes in the queue is reduced. This reduces the number of cache misses.
- Fewer CAS operations:
  - The TSLQueue requires 2 CAS operations per insertion, because each insertion modifes a linked list and a tree. RangeQueue requires either 1 CAS operation, or 1 CAS and 1 FETCH_OR per insertion. This will be elaborated upon in the Algorithm section.
  - The TSLQueue requires 3 CAS operations per deletion, one to logically delete the node by marking a bit, and two to modify pointers. RangeQueue again requires fewer CAS's because not every deletion requires pointer rearrangement.
- Lesser memory footprint: We use bitvectors instead of storing the actual value.

Overview

A threaded tree is a tree in which every node has a pointer to its successorin an inorder traversal. Such a concept can be generalised from key-based nodes to range-based nodes trivially, as non-overlapping ranges have a total ordering.

Each node in the RangeQueue contains

- `base`: The value of the the least key that could be inserted into it. Must be divisible by `BUF_SIZE`.
- `next`: A pointer to the next node in the queue.
- `parent`, `left`, `right`: Pointers to the parent, left child, and right child of the node respectively. Standard BST terminology.
- `buf`: A bitvector of size `BUF_SIZE` that indicates whether the key at an index has been inserted.

- `ins`: A flag indicating whether the node is currently undergoing insertion. Identical to TSLQueue.

The RangeQueue has a `head` pointer that points to the first node in the queue. The `head` pointer is a dummy node that has a key of `-1` and a `next` pointer to the first node in the queue. The `head` pointer is used to simplify the implementation of the `deleteMin()` operation and address the ABA problem.

The RangeQueue also has a dummy root. The root has a `next` pointer to the actual root node of the tree.

Both the dummy nodes are used in TSLQueue, for ease of proof of correctness.

Algorithm

Insertion

- A thread descends the BST to find the node that contains the key to be inserted.
  - If such a node exists, it atomically sets the corresponding bit in the `buf` bitvector. If the bit is already set, returns duplication error.
    - If a node is logically deleted (i.e. `buf` is all zeroes), it's existence will not be reported, i.e. `InsertSearch` returns `existingNode = null`.
  - Otherwise, it allocates a new node with the appropriate bit set, and inserts it into the RangeQueue. This is identical to the TSLQueue insert, with an similar `Seek` struct.
    - Set `next` to next node in the linked list.
    - Atomically insert into the linked list by setting `next` pointer of the previous node to the new node.
    - Set `parent` pointer of the new node to the parent node.
    - Atomically insert into the BST by setting appropriate `left` or `right` pointer of the parent node to the new node.

Deletion

- Deletion uses the `head` pointer to access the minimal node, and resets the highest set bit in the `buf` bitvector.
  - This is a single atomic operation - upon failure, the thread simply retries delete.
  - If after deletion, the `buf` bitvector is all zeroes, the node is said to be logically deleted, unlike in TSLQueue where there is a specific tag on the `next` pointer to indicate the same. With some nonzero probability, the thread physically deletes the node. Physical deletion is identical to TSLQueue.
    - Atomically delete from the linked list by setting `next` pointer of `head` to the next node.
    - Atomically delete from the BST by setting `left` pointer of the parent node to the next node.

References

Rukundo, A., Tsigas, P. (2021). TSLQueue: An Efficient Lock-Free Design for Priority Queues. In: Sousa, L., Roma, N., Tomás, P. (eds) Euro-Par 2021: Parallel Processing. Euro-Par 2021. Lecture Notes in Computer Science(), vol 12820. Springer, Cham. https://doi.org/10.1007/978-3-030-85665-6_24