Adaptive Priority Queue With Elimination and Combining

Christopher L. Taliaferro
Undergraduate
Department of Computer Science
University of Central Florida
Email: taliaferrodev@gmail.com

Tiger Sachse
Undergraduate
Department of Computer Science
University of Central Florida
Email: tgsachse@gmail.com

Ben Faria
Undergraduate
Department of Computer Science
University of Central Florida
Email: benfaria96@gmail.edu

Harrison W. Black
Undergraduate
Department of Computer Science
University of Central Florida
Email: harrison.w.black@knights.ucf.edu

1 Introduction

Priority queues are fundamental abstract data structures which can be implemented in a parallel fashion to assist in discrete event simulations and resource management [1]. Some of the current implementations of these parallel priority queues utilize heaps or skiplists to create the foundation, or main data structure to build these more abstract concurrent data structures. The skiplist implementation gives the ability to take advantage of the potential for parallelism of the add method [1].

2 Design, Concurrent Skiplist, Elimination and Combining

One skiplist-based implementation of the concurrent priority queue acts in a similar way to the LazyList and Lock-FreeList we have seen in the textbook, when a node is to be deleted it will be marked as such and then the thread will attempt to physically remove it, this implementation comes from Lotan and Shavit [1]. Another implementation of this concurrent priority queue comes from Hendler et al, this implementation introduced Flat Combining which batches together several operations and hand the batch to one thread to handle the execution of these operations [1]. This has also been taken and mixed in with a client server structure, where several client thread will create these operations to be handed off to a single server thread to then execute them. All these previous implementations have been steps in the right direction but they all have their flaws, the first implementation from Lotan and Shavit have the possibility of running into limited scalability when high thread counts are introduced due to contention on shared memory access. While the Flat Combining implementation reduces contention of the shared memory it introduces its own problem, which is the potential for the server thread to become a sequential bottleneck [1].

The implementations and their downsides lead to the implementation that is introduced and discussed in this paper, this being an implementation which uses a skiplist-based priority queue which employs an elimination algorithm. The skip list is used in this implementation because of its ability to provide both operations-batching and disjoint-access parallelism. [1]. As mentioned, part of this utilized the technique of batched operations from the Flat Combining implementation to improve the performance of the overall concurrent priority queue. As values are added into the priority queue they are subject to one of two paths, either the value is small enough to participate in elimination or it is too large and must be added into the skiplist normally. The second option mentioned is where the parallel portion of this data structure acts, high-values adds can be executed in parallel [1]. These two different paths that arise from adding to the skip list bring up the need for splitting the skip list into two different parts, being the sequential part handled by the server thread, and the parallel part [1]. The other side of this data structure comes from the remove min method which is where the batched operations comes into play, these requests can be batched and executed by a server thread using the combining/delegation paradigm. [1]. Designing the concurrent priority queue in this fashion allows for the reduction of contention and use of parallelism through elimination and high-valued adds [1].

As mentioned, the design of this concurrent priority queue is based on an underlying skip list which is split into two parts, sequential and parallel, these two parts are both complete skip lists. The priority queue supports both the remove min method and the add method, the sequential part of the skip list will serve the remove min requests as well as the low-valued add requests which are more likely to be removed soon, while the parallel part will cater to the large-valued adds which are less likely to be removed in the near future [1]. Early the two paths that can be taken on an add request were touched on, but now they will be broken down further. When a priority queue add request is made two things happen, first the value is determined to be either less than or greater than the last value in the sequential list and from there it is passed on to its next step based on the condition it meets. If it is less than the last value the priority queue will attempt a remove min request using the elimination array which will be discussed further soon, otherwise if it is greater than the last value it will be sent to be added to the parallel list, which in turn means it will be added in parallel with any other values that met this same condition. As mentioned the add request of small values will in turn cause the thread to attempt to execute a remove min request that may exist in the elimination array. If there is no such request in the array, then the add request itself gets added to the elimination array. It will then wait in the elimination array until it either eliminates or times out, the server thread will sequentially execute any operations that fail to eliminate [1].

The sequential part of the skiplist uses a move head method to bring in elements form the parallel part when it becomes empty, it can also use a chop head method to relink the sequential and parallel portions in the case that there havent been remove min requests for some time [1]. These methods allow for the underlying skiplist to adjust based on the current state of the priority queue, either bringing in more elements from the parallel portion or linking with the parallel portion to perform more efficient parallel add requests. Anytime these move head and chop head methods are called the currSeq and lastSeq pointers are adjusted to represent the start and end to the new sequential part of the list thus allowing for the priority queue add and remove min requests to react to those changes. The parallel part of the list uses a Single-Writer Multi-Readers lock to ensure that and priority queue add requests are not operating on buckets that are being moved to the sequential part by the move head method or interferes with the chop head method [1]. A skip list find method is used to determine whether any head operations have been executed and if the find method comes back clean a lock is held to avoid any head operations that take place after the clean find. This lock will then be released once a CAS fails while inserting a new value.

Now, with a better understanding of the structure and functionality if the underlying skiplist and its parts, the next major aspect of this concurrent priority queue is its elimination algorithm. In this priority queue, a remove min request can always be eliminated but if an add request finds its way into the elimination array it can only be eliminated if its value is less than or equal to the lowest value of the sequential portion of the list. The algorithm works by using an elimination array in which elements are added to the array and a remove min request searches through the array until it finds a request to eliminate with or it finds an empty slot in the array [1]. Once it finds one of these two things it will execute a CAS to get the value it needs to return, or it will fall through to the sequential part of the list and perform a remove and return that value. There are cases where an add request can never be eliminated if its value never goes below the lowest value in the priority queue. This is where the sequential server thread comes into play, it collects add and remove requests that fail to eliminate and executes them sequentially on the skiplist [1]. This functionality is key to the reduced contention of this concurrent priority queue.

3 Linearizability

In this section we shall discuss the linearizability of our proposed method

3.1 Skiplist

When implementing the Skiplist version of the priority queue, linearization occurs when an element is added to the parallel section of the skiplist, with addParallel(T). When this element is added, compare and swap is invoked and the bucket for key T has its counter incremented, evoking linearization. If a thread inserts a new minimum value into our priority queue and the sequential part of the skiplist is empty, the thread then performs the required action of updating the queues minValue. If the sequential part of the list contains data, then the main thread synchronously updates minValue without linearization. Threads that succeed changing minValue linearize their operation at the point of the successful compare and swap. Some operations require the head of the list to be modified. The operations moveHead() and chopHead() execute while holding a lock. Linearization effectively occurs when the lock is released with lock.release().

3.2 Elimination

Each thread, either adding or removing, that finds the inverse operation in the elimination array must verify that the exchanged value is smaller than minValue. If so, the thread can compare and swap the elimination slot, exchanging arguments with the waiting thread. It is possible that the priority queue minimum value is changed by a concurrent add(). In that case, the linearization point for both threads engaged in elimination is at the point where the value was observed to be smaller than the priority queue minimum. The threads post their operation in the elimination array and wait for the main thread to process it. The main thread first marks the operation as in progress by compare and swapping in progress into the slot. It then performs the sequential operation on the skiplist and writes the results back in the slot, releasing the waiting thread. The waiting thread observes the new value and returns it. The linearization point of the operation happens during the sequential operation on the skiplist.

4 Evaluation

The author of the paper evaluated their results using the following benchmark. A thread randomly flips a coin with probability p to be an add() and 1 p removeMin(). The tests began after inserting 2000 elements into the priority queue. Our priority queue algorithm is designed for high contention scenarios, in which elimination and combining thrive. Therefore, it can incur a penalty at lower thread counts. Although, our priority queue can fully take advantage of both elimination and parallel adds, so it maintains peak performance over other algorithms such as the flat combining skiplist, the flat combining pairing heap, the lock free skiplist, and the lazy skiplist at higher thread counts.

5 Hardware Transactions

Regarding physical hardware transactions and the Adaptive Priority Queue, parallel threads are permitted speculatively execute critical sections. However, if two threads encounter a data conflict, one must roll back and re-execute its critical section. Results were evaluated using a Core i7-4770, running at 3.4GHz, with restricted transactional memory (RTM), and Hyperthreading enabled. Additionally, 8gb of RAM was shared and each core had 32KB of L1 cache [1].

The sequential and parallel skiplists both use the Single-Write-Multi-Readers lock to synchronize. This causes complications with the priority queue as this lock adds a lot of overhead. Since the problem was approached by making all operations transactional, this lead to too many aborts occurring with this implementation. This was resolved by having the server increment a timestamp any time a head-moving operation like moveHead() or chopHead() starts or ends. Two operations are used to read a timestamp and then find the corresponding insertion. If a head-moving operation occurs after starting the transaction, the timestamp will change, causing the transaction to be aborted due to a timestamp conflict. For the timestamp to be valid the find() operation must record all predecessors and successors of a new bucket at each position i in preds[i] and succs[i]. If the bucket already exists, then the counter inside the transaction is incremented and the operation completes. The more threads running, the more increase in the number of transactions per successful operations. Additionally, the percentage of threads that need more than 10 tries to succeed also increases with the amount of threads running. After ten tries, threads stop attempting the transaction path and the server completes it for them. The amount of times this happens is at most 10

References

[1] Calciu I., Mendes H., H. M. *The Adaptive Priority Queue with Elimination and Combining*. Kuhn F. (eds) Distributed Computing Lecture Notes in Computer Science, vol 8784. Springer, Berlin, Heidelberg.