FAST THREAD-SAFE PRIORITY QUEUE

Priyankit Bangia, Taranpreet Kohli, Gabriel Tiongco

ABSTRACT

This report outlines the development of a high performing thread-safe priority queue for Java. Despite the thread-safe nature of Java's own PriorityBlockingQueue, it is evident from its implementation that its performance is heavily limited due to the locking strategy it employs. In this report, we propose an implementation of a pipelined priority queue which aims at providing finer locking granularity, and hence better performance. This report also outlines the methodology used to implement its structure, implementation details, testing, and evaluation, as well as future work that may be done to extend its functionality further. From our benchmark results, we show that our implementation is faster than PriorityBlockingQueue by a factor of 1.6-1.8 on average for a large number of threads.

INTRODUCTION

Priority queues are important and frequently used data structures. Common uses of priority queues range from scheduling algorithms at an operating system level to maintaining priorities of objects in user applications. Real-time systems must use a thread-safe implementation of a priority queue in multi-threaded environments. Research in this area has been extensive as performance has been a key consideration in such implementations.

A priority queue stores key/value pairs, with the key being the priority. Insert and DeleteMin/Max are the two main operations that are supported in a typical implementation. The Insert operation inserts a new key/value pair, and the DeleteMin/Max removes the head of the set which is maintained to be the value with the greatest priority at any given time. To support concurrent operations, Java provides the PriorityBlockingQueue implementation. In this implementation mutual exclusion caused by internal locking that is involved ensures safe concurrent access. However, the performance is degraded on high load as the entire data structure is locked to one thread exclusively each time. This means only one thread can make progress at a time.

There are two well-known thread-safe priority queues that address this problem. One of these includes a lock-free approach. In this, a lock-free skiplist guarantees progress of at least one one operation through its atomic primitives. Another approach researchers have proposed is the use of multiple locks in place of a global lock, known as a pipelined queue. This approach protects a smaller part of the data structure allowing for more than one operation to take place concurrently.

In this project, both approaches were analysed. As part of our project we implemented a version of a pipelined priority queue, because it conceptually seemed faster on small-moderate loads than the skiplist alternative. This report discusses the two alternative thread-safe priority queues, our implementation of a pipelined priority queue, and how it performs in comparison to PriorityBlockingQueue.

RELATED WORK

Sundel and Tsigas introduce a thread-safe priority queue using a lock-free algorithm [1]. It has been designed to be efficient in both pre-emptive and fully concurrent environments. The algorithm is based on a randomised skiplist data structure that makes use of common synchronisation primitives, like Compare and Swap (CAS) and Fetch and Add (FAA). The data structure consists of nodes with pointers to provide the links to the other nodes. The two synchronisation primitives are used when these links between nodes need to be updated during insert and delete operations. The skiplist data structure is essentially an ordered list that has randomly distributed short-cuts to improve search times. The balancing of the skiplist structure is probabilistic due to how it is constructed. Furthermore, the randomly distributed short-cuts gives it a probabilistic time complexity of O(logN) for insert, delete, and search operations.

The main benefit of this design is that there is no mutual exclusion (locking). Disadvantages of mutual exclusion include starvation, deadlock, and the degradation of the system's performance. Sundel and Tsigas have evaluated their algorithm and found that in both highly pre-emptive and fully concurrent environments their algorithm outperforms other lock-based implementations for 3 threads and more. Since there is also no contention on any nodes when others are blocked, the algorithm scales well to accommodate many threads performing operations on the queue. However, there are some limitations in using this skiplist-based algorithm. Since the short-cuts in the list are randomly distributed the worst case for insertions and deletions can be very inefficient, and there can be more overhead for potentially more comparisons made in these operations. It also uses multiple nodes for each value in a queue, so effects on memory usage need to be considered.

Another implementation that was investigated was the pipelined priority queue proposed by Bhagwan and Lin [2]. This algorithm aimed to optimise the common lock-based priority queue by presenting a new data structure called a Pipelined heap (P-heap). This approach was developed for use in high-performance network switches, where packets need to be transmitted in a *highest-priority-first* order. They believe that hardware implementations for supporting parallel algorithms for heap-based priority queues is expensive and complicated. Developing a solution that allows for the same parallelism while keeping the hardware complexity low would be beneficial.

The P-heap is based on a modified form of a standard binary heap. This data structure allows the enqueue and dequeue operations to be pipelined so that they will essentially execute in constant time. The P-heap data structure has two core components, a P-heap binary array and a P-heap token array. The binary array stores all the priority values in semi-sorted order. The token array is used to aid the pipelining of operations by storing the priority value of what is being inserted into the binary array. This data structure offers some benefits over a standard binary heap and has potential to outperform Java's PriorityBlockingQueue. Bhagwan and Lin claim that this architecture scales very well, however, contention for the root node may become an issue when there are a lot of threads. Also, since this P-heap is based on a conventional binary heap, it shares its O(logN) for insertions and deletions, and is O(1) for peek (looking at the value of the highest priority i.e. the root). Operations are pipelined, meaning each operation will at most be accessing two subsequent levels of the heap at any given time. Therefore, there can be at most H/2 operations taking place in the heap concurrently, where H is the height of the tree (representing number of levels in the token array).

IMPLEMENTATION

In this section, implementation details of our PipelinedPriorityQueue based on the works of Bhagwan and Lin are discussed. The approach used coincides with the algorithms and data structures discussed in their publication, but has been expanded to take advantages of

Java's runtime framework, including the use of generics and object-oriented programming. It was a conscious decision to implement a PipelinedPriorityQueue over a lock-free SkipList priority queue because the latter infers a data structure with probabilistic attributes. A SkipList has potential for, although with a small probability, to perform inefficiently on large inputs because of how the levels inside it are constructed.

As this priority queue is to be used in the context of a shared-memory system with several threads accessing the queue concurrently, it was important to include the use of locks and thread-safe data structures to preserve the validity and correctness of the data structure. Our implementation is of a strict priority queue where the operations of dequeue() or peek() always return the element with the highest priority, as opposed to relaxed implementations which do not guarantee this.

Class Structure

There are three primary classes in this implementation. Namely, they are BinaryArrayElement, TokenArrayElement and PipelinedPriorityQueue.

- A single instance of BinaryArrayElement<E> represents a single node in the heap (binary tree) of the pipelined heap used in the queue.
- A single instance of TokenArrayElement<E> represents a single level of the pipelined heap used in the queue. It stores the value of the node being inserted or removed from the heap, and is primarily used to aid the pipelining of operations in this queue.
- A single instance of PipelinedPriorityQueue represents the unbounded priority queue that internally pipelines common operations such as enqueue() and dequeue(). In addition to supporting, enqueue(), dequeue(), we extended the functionality of this queue by also implementing other methods in the BlockingQueue<E>, Queue<E> and Collection<E> interfaces.

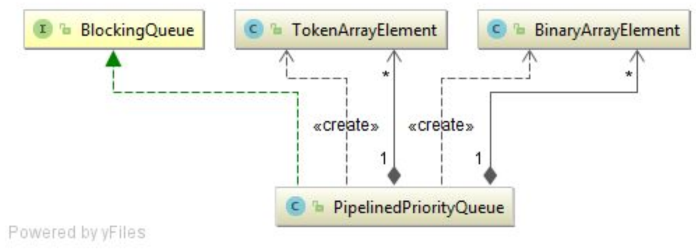

Fig. 1 - UML diagram showing the relationship and associations between the three primary classes in the PipelinedPriorityQueue (which implements Java's BlockingQueue interface.)

Algorithms

The algorithms that are employed for the main operations of enqueue and dequeue were developed through the guidelines provided by Bhagwan and Lin. These operations were broken down into enqueue, dequeue, local_enqueue, and local_dequeue. The local forms of the operations deal with a single level at a time and are called by the standard enqueue and dequeue until the respective operation is complete. The BinaryArray in which operations will be performed is denoted B and B[i] refers to a node in said tree. The TokenArray T has an element T[j] that stores the operation being performed at the jth level and its value. L denotes the number of levels in B and L_j a level J in B.

Enqueue

In order to enqueue an item to the BinaryArray B, we need to find an inactive node to insert the new value into (a free spot in the tree). We can do this by traversing through B until we find an inactive node. This can be done by following a path from the root to a leaf that has a capacity greater than 0. First, the enqueue operation writes the value to be inserted into

`T[1].value` and sets `T[1].position` to 1. Enqueue then calls the local_enqueue operation up to L times before it completes as we have to traverse from the root to a leaf before we can find an inactive node (top-down insertion approach). The local_enqueue operation takes as input j indicating level L_j in B and is used to compare nodes being traversed and inserting the value into an inactive node when found. First, the current node `B[i]` is checked to see if it is active. If it is not, then the `T[j].value` is inserted at this node and the enqueue operation completes. However, if `B[i]` is active then we check to see if the value we are inserting is greater than `B[i].value`. If so, we swap `B[i].value` with `T[j].value` then repeat the enqueue operation, percolating the old value of B[i] downwards. The next stage is checking which child of `B[i]` has the greatest capacity to determine which path to traverse down. Set `T[j+1].position` to the position of that child and move the value of `T[j]` to `T[j+1]`. At this stage, the capacity of `B[i]` is decremented by 1. This local_enqueue process is repeated up to L times to completion.

Dequeue

In the dequeue operation, the highest priority value from the heap is extracted, that is the value of the root node `B[1]`. At this point, `B[1]` is made inactive, capacity incremented, and its value is nullified. However, since the root node is inactive it may violate the P-heap property if it has any active children. The inactive node must be moved down until the P-heap property is satisfied throughout B. In local_dequeue both children are checked if they are inactive. If so the dequeue operation is complete. If not, the values of the children (`B[left(i)] and B[right(i)]`) are compared, to find the child `B[k]` with the largest value. `B[i]` is set to active, its value is set to `B[k].value`, and `B[k]` is set to inactive. `B[k]`'s capacity is incremented, and `T[j + i].position` to is set to `k`.

Concurrency

To handle concurrency, ReentrantLocks are used. These were used over synchronized access modifiers as they would enable us to lock at level granularity as opposed to global. Originally, we had attempted to implement a fine granularity locking scheme by making each BinaryArrayElement (each node in Fig. 2) have their own ReentrantLock. This was done with the idea of allowing multiple operations to proceed through different branches concurrently. This increases parallelism as shown in Fig. 2 when compared to level locking. Operations can safely proceed down either side of the branches even though they are on the same level. However, with initial testing on this approach we found that the increased number of locks resulted in a greater locking overhead. Although conceptually it provides greater parallelism, the completion times for tests were much slower than that of an implementation using locking at level granularity. Therefore, the level locking scheme that is proposed was used. This resulted in each TokenArrayElement having a ReentrantLock that a thread needs to acquire before it can perform any operation on the level it represents. Each time an operation takes place it needs to obtain the locks to two subsequent levels. Once an operation has processed through one level fully, it will unlock that level at `T[j]` but retain the lock on the level `T[j+1]` below. Before proceeding onto the next iteration of the operation, the lock on `T[j + 2]` representing the next level will be obtained. This process of locking and unlocking subsequent levels with a top-down approach is what allows the pipelining of operations through the P-heap. Other operations like peek involve just locking the root node while resizing B requires the locks of every level to be obtained.

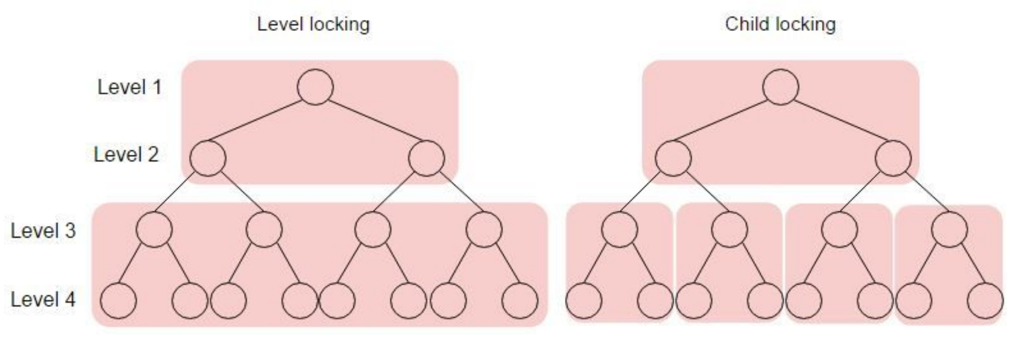

Fig. 2 - A comparison between coarse-grained level locking (left) vs fine-grained node-locking (right)

TESTING & EVALUATION

Testing involved intermediate steps between sequential implementation and parallel implementation of our algorithm. After a base sequential implementation we needed to ensure correctness before parallelising our code with locks. This made sure that concurrency problems discovered in the multi-threading testing stage were isolated from business logic. Extensive unit testing was performed to ensure both sequential and multithreaded operations kept the data structure in a consistent and correct state. Benchmark performance testing was also done for a range of input sizes and different number of consumers/producers to compare our implementations performance against the existing PriorityBlockingQueue implementation.

Sequential Correctness

For correctness testing on a sequential input, all methods inside the PipelinedPriorityQueue class were unit tested with JUnit for an extensive set of sequences and cases. The expected output and resulting state of the queue were examined and asserted for correctness after execution of these methods. For example, we would test the random insertion of integers that are within a specified range, then poll the queue to check if it gave us an ordered result for the size of the input. Other tests included more complex cases such as array resizing when the size of the input became larger than the size of the array, having mixed operations performed on the queue, and testing a combination of other methods.

Parallel Correctness

Once these tests were shown to pass, tests for concurrent access to the data structure were written. The multi-threading stage involved a test driven development strategy, as it would be difficult to conceptualise what is required from our algorithm in a multi-threaded environment without predefined tests for different cases. Threads were created with the specified operations in their run() method, through methods that were setup to generate different cases. All threads were started from the main thread, and the main thread waited on each of the threads to finish by using a join() on each thread. On completion, the sequence of events were able to be traced and also the same state correctness assertions were done on the resulting queue to ensure correctness of the concurrency handling. These tests were created with an aim to force errors on an implementation without concurrency handling. This was done by explicit interleaving of operations by sleeping threads for a short time, and simulating work, to allow the state changes caused by other threads while operating in the same space.

Performance

For performance testing, it was important we set Java's PriorityBlockingQueue as a benchmark for speedup analysis with our implementation. Stress tests were done for different cases such as number of total operations on the queue, type of operations, and number of threads sharing the work. Input size (number of operations on the queue) ranged from small input (512) to large input (524,000). The type of operations ranged from ordered input, reversed input, random input, and mixed operations (both poll() and put() operations). Each input size and type of operation were tested with a sequential run, a small number of threads (2), to a large number of threads (1000) on a dual core computer. The start and end time represented the total execution time for all specified cases to be completed from all the threads collectively. The same tests were run on our implementation, followed by analysis and evaluation of our implementation. Data from all tests were printed in CSV format and copied to a spreadsheet for analysis (spreadsheet with raw data is included in submission).

Evaluation

As the PriorityBlockingQueue (PBQ) implementation involves a global lock for each operation, our hypothesis was that it could struggle with a high number of threads having concurrent access. Our implementation with a finer granularity of locking allows more than one operation to have concurrent access to the data structure so conceptually it was expected to perform better and provide some speedup under high load. However, it was also noted that our implementation may be slower with sequential access due to the unnecessary locking for this case. Also it was expected to perform worse on a low number of threads as there would be low contention, not warranting the need for finer granularity locks.

From our data that was collected we were able to ascertain that the PipelinedPriorityQueue (PPQ) implementation provided increasing speed up with higher load (above 2-4 threads). The overall time for test completion for operation type showed there was an average speedup of 1.8 for random, reversed, and ordered input, and 1.6 for mixed operations. These averages are broken down further in Figures 3 and 4, which show different aspects of the execution.

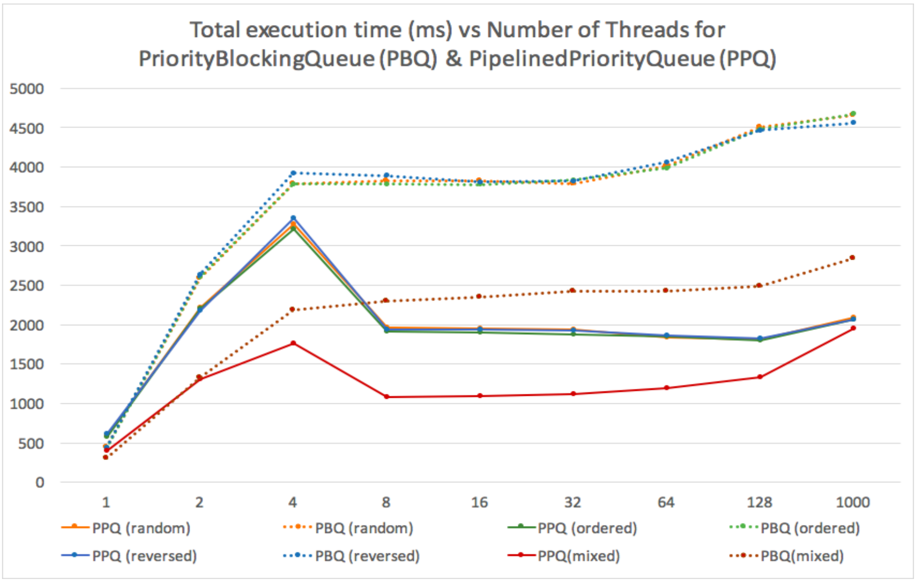

Fig.3 - Comparison of times for both implementations

The average execution time for all input sizes across all test types were calculated from the collected data. For all test types (random, reversed, ordered, and a mix of put and poll operations), PPQ performs better than PBQ on a high number of threads as expected. Both perform faster on a mixed operation test when compared to other test types due to the polling meaning there will be less total values to compare in the heap. When comparing the speed up factor with an increasing number of threads for all test types, we see an increasing trend. This is due to PPQ's ability to have finer granularity concurrency over PBQ's global lock as discussed. As expected, overall the PPQ does not provide significant speedup for a low number of threads (< 2), as in this case having a global lock may be more efficient. Here speedup is calculated as total time taken by PBQ divided by total time taken by PPQ.

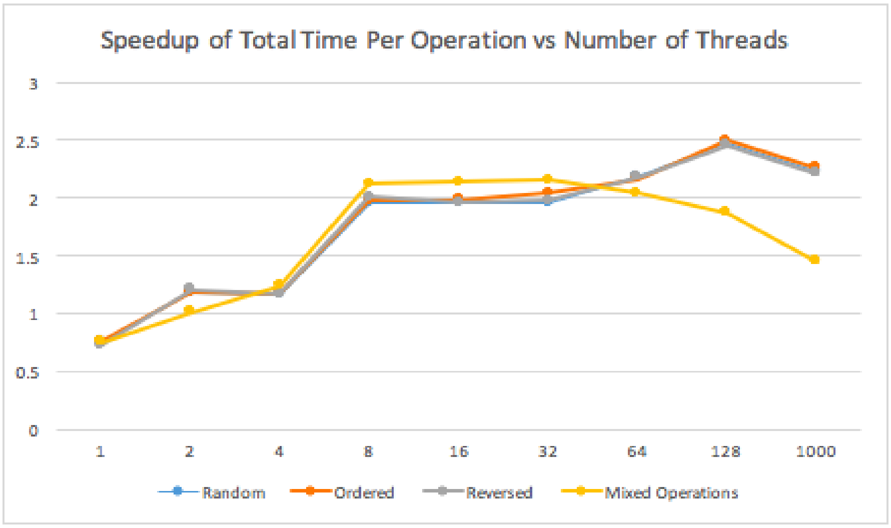

Fig.4 - Speedup achieved for the number of threads

METHODOLOGY

The process in the development of pipelined priority queue implementation used an agile and iterative approach, by determining goals and assigning tasks on a weekly basis. This approach of development ensured we remained flexible to changes. The process consisted of four stages: research, implementation, testing, and evaluation.

The first stage was done collectively as a group. This was so that each member of the group was knowledgeable about the problem space and could contribute ideas regarding the alternative priority queues researched. Research into two different approaches was done, one lock-based and the other lock-free, to gain insight into contrasting ideas for implementing a fast thread-safe priority queue. Through considering the implications of each and exploring implementation, we decided to implement a pipelined version.

GitHub issues were used to track issues and task assignments between members of the group. Most of our meetings were hosted remote through technologies like Google Hangouts. However, during more complex parts of the implementation and testing, in-person meetings were organised to discuss ideas, as well as employing a paired programming strategy. For task assignment, the components of the implementation were first broken down. These parts generally consisted of method stubs to be implemented for each of the classes in the project. Tasks were picked up by each group member and progress was updated on the issue tracker on GitHub so that it was clear who was working on what at any given time. The implementation and testing stages were often intertwined. At the start, the focus was on creating a working sequential implementation of the pipelined priority queue. This ensured that the implementation was correct before the introducing concurrency handling logic. The following stage was to extend implementation for parallelism and the testing of parallel execution of operations. To do this a test driven development approach was used. For the evaluation stage, the results from the performance tests against Java's PriorityBlockingQueue were collated and analyzed.

FUTURE WORK

There are still several areas that can be further investigated to enhance our implementation and further our research into a faster alternative for Java's PriorityBlockingQueue. Implementing a local_enqueue_dequeue operation that combines both an enqueue and dequeue can potentially enhance performance. It works by checking the operations that want to be performed on the heap and if two subsequent enqueue and dequeue operations are present it will combine it. This is done by extracting the value of the root node then replacing the root's value with the value to be inserted. This can reduce the number of operations to be performed on the heap, therefore reducing execution time. Lastly, in theory, lock-based algorithms have issues handling very high number of threads as we have seen with Java's PriorityBlockingQueue. This may make it worthwhile to investigate a lock-free alternative which scales better. The lock-free Skiplist algorithm's scalability is worthwhile comparing to the PipelinedPriorityQueue.

CONCLUSION

Our project aimed to investigate alternative implementations to Java's PriorityBlockingQueue that would perform better in highly concurrent environments. We developed an implementation for a PipelinedPriorityQueue based on the algorithm presented by Bhagwan and Lin. PipelinedPriorityQueue's performance was evaluated and tested against Java's PriorityBlockingQueue in different cases. It was found that speedups of 1.6-1.8 were achieved on a large number of threads, whereas there was no significant speed up observed for a low number of threads. The results from these benchmark tests aligned with theoretical explanations, as we found the pipelined priority queue to perform better than Java's PriorityBlockingQueue for high loads. There are still other areas that can be further investigated such as enhancing our current pipelined implementation and potentially implementing a lock-free approach to see how it would compare in performance to the others.

REFERENCES

[1] Sundell, H., & Tsigas, P. (2005). Fast and lock-free concurrent priority queues for multi-thread systems. Journal of Parallel and Distributed Computing, 65(5), 609-627.

[2] Bhagwan, R., & Lin, B. (2000). Fast and scalable priority queue architecture for high-speed network switches. In INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE (Vol. 2, pp. 538-547). IEEE.

CONTRIBUTIONS

Priyankit Bangia	Implementation, report, testing	33.3%
Taranpreet Kohli	Implementation, report, testing, class design, documentation	33.3%
Gabriel Tiongco	Implementation, report, testing	33.3%