Shared Stack Implementation and Analysis Using Queue Delegation with Elimination (Final Project)

Esha Choukse (ec27876), Mike Thomson (mt29253)
Multicore Computing, Fall 2014

November 20, 2014

Abstract

Queue delegation and Elimination are two techniques that help us improve the performance of shared data structures, by reducing contention. We use both these concepts together to implement a shared stack and analyse the performance we get from it. We have used the C++ libraries published by Klaftenegger [1], and implemented elimination on top of it. We compare the performance of our implementation against the basic queue delegation, and MonitorT [2].

1 Introduction

Shared data structures between threads, always tend to slow down the execution due to the inherent sequentiality associated with them. A lot of implementations still use locks. Locks, because of their mutual exclusivity, make the execution of the critical sectoin completely sequential. Several techniques are used to alleviate this disadvantage. One of these techniques is Queue delegation. Queue delegation allows a thread to offload its task to a delegate thread, which is already operating on the shared structure. This allows the thread to carry on with its execution, till the delegate thread executes the queued task. Another technique is to use Elimination. Elimination tries to take advantage of operations on the shared data structure, which have reverse semantics, for example a push and a pop on a stack. This way, none of the two threads actually access the shared data structure, resulting in less contention. We use both these concepts together to implement a shared stack and analyse the performance we get from it. We have used the C++ libraries published by Klaftenegger, and implemented elimination on top of it. We compare the performance of our implementation against the basic queue delegation, and MonitorT.

2 Background

Let us first discuss the concepts of Queue delegation and elimination in detail.

2.1 Queue Delegation

Using locks on shared data structure is a very common practice. On doing so, each operation on the shared data structure by a thread, needs it to first acquire the associated lock. Even though fine grained locks perform better than coarse grained locks, there is still a lot of performance hit. Queue delegation is a method to reduce the hit by allowing threads to outsource their shared data structure operations to a delegate thread. This is done by pushing the task into a delegation queue. Once the task is in queue, the thread can carry on with its execution and does not need to wait on the result. The delegate thread reads tasks from the delegation queue, and executes them one by one, giving out the results to the associated threads. Note that since a single thread is executing the operations on the shared data structure, no locks are required. There are several ways to implement the Queue delegation. One of the main decisions to make is that whether we want a single thread to remain the delegate thread throughout the execution, or can any thread become a delegate thread. In the QD Lock libraries we have used, the second approach is implemented. The threads, upon reaching a shared data structure operation, try to acquire a lock, to become the delegate thread. If they fail to acquire the lock, they put the task into the delegation queue and leave. Instead, if they do acquire the lock, they execute their task, and all the tasks in the delegation queue, till the queue becomes empty eventually. Once this happens, the delegate thread gives up the lock and leaves. Upon contention, a thread may need to become the delegate thread for very long periods, affecting execution of that thread. To fix this, we have a limit on the number of tasks in the delegation queue, which we call delegation queue length. Once this limit is reached, the delegation queue stops accepting tasks. The choice of the delegation queue length is therefore an important factor in the performance.

2.2 Elimination

Elimination identifies the operations on a shared data structure, which have reverse semantics. Such operations can be made to collide outside the shared data structure, and return, without ever getting to access the actual structure. This helps reduce contention. The elimination uses an elimination array, which has multiple slots. Each slot can be in a waiting, busy or empty state. An operation, on finding an empty slot, changes it to waiting and waits. Another operation comes in, and on finding the slot waiting, makes it busy. In case these two operations are of reverse semantics, our work is done. We then return both the operations with the expected results. The choice of number of slots in the elimination array is an important one, since too big an array will lead to lesser collisions.

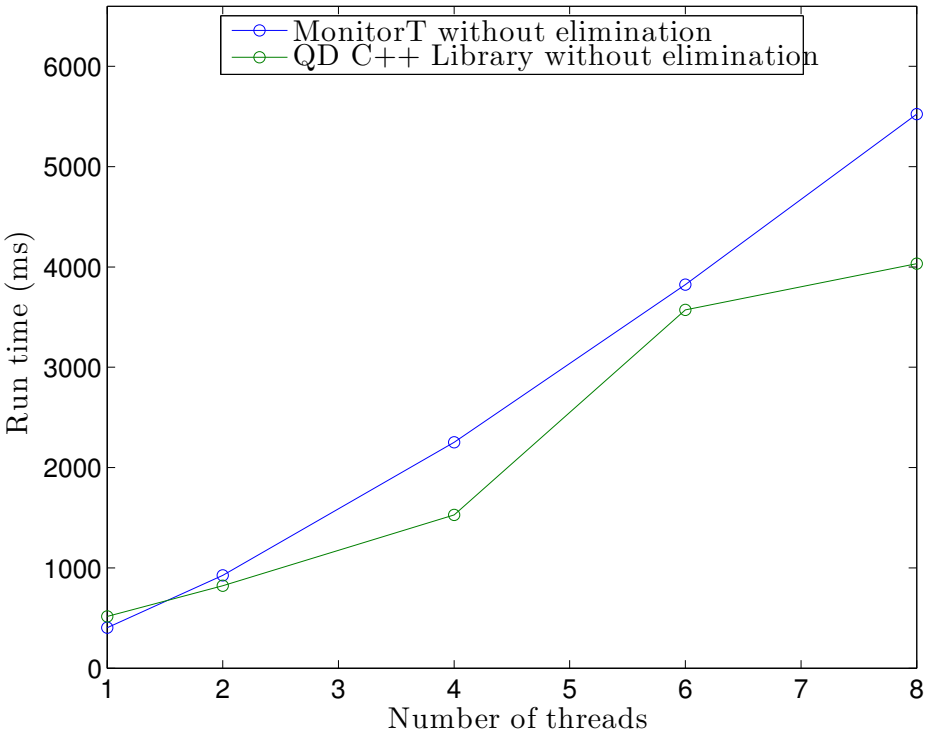


Figure 1: Baseline comparison of the MonitorT implementation (Java) and the QD Lock (C++), varying the number of threads, and without elimination for either implementation. Queue size = 256, push ratio = 0.5.

3 Results

4 Analysis

5 Conclusion

References

[1] D. Klaftenegger, K. Sagonas, and K. Winblad, "Delegation locking libraries for improved performance of multithreaded programs," in *Euro-Par 2014 Parallel Processing*, vol. 8632 of *Lecture Notes in Computer Science*, pp. 572–583, Springer International Publishing, 2014.

[2] W.-L. Hung, H. Chauhan, and V. K. Garg, "Activemonitor: Non-blocking monitor executions for increased parallelism," *arXiv preprint arXiv:1408.0818*, 2014.

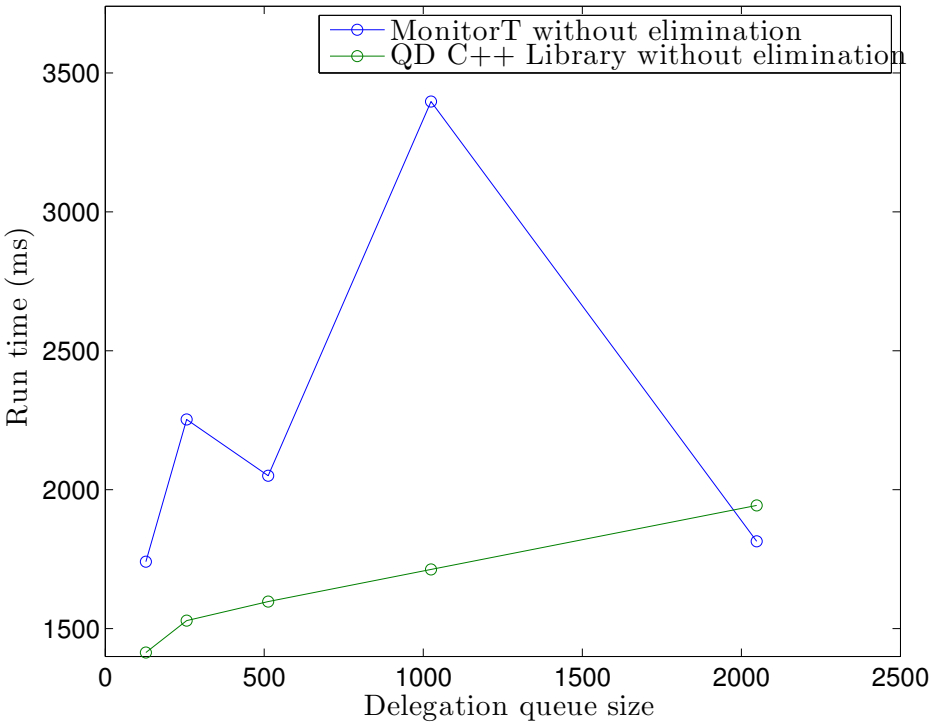


Figure 2: Baseline comparison of the MonitorT implementation (Java) and the QD Lock (C++), varying the delegation queue size, and without elimination for either implementation. Threads = 4, push ratio = 0.5.

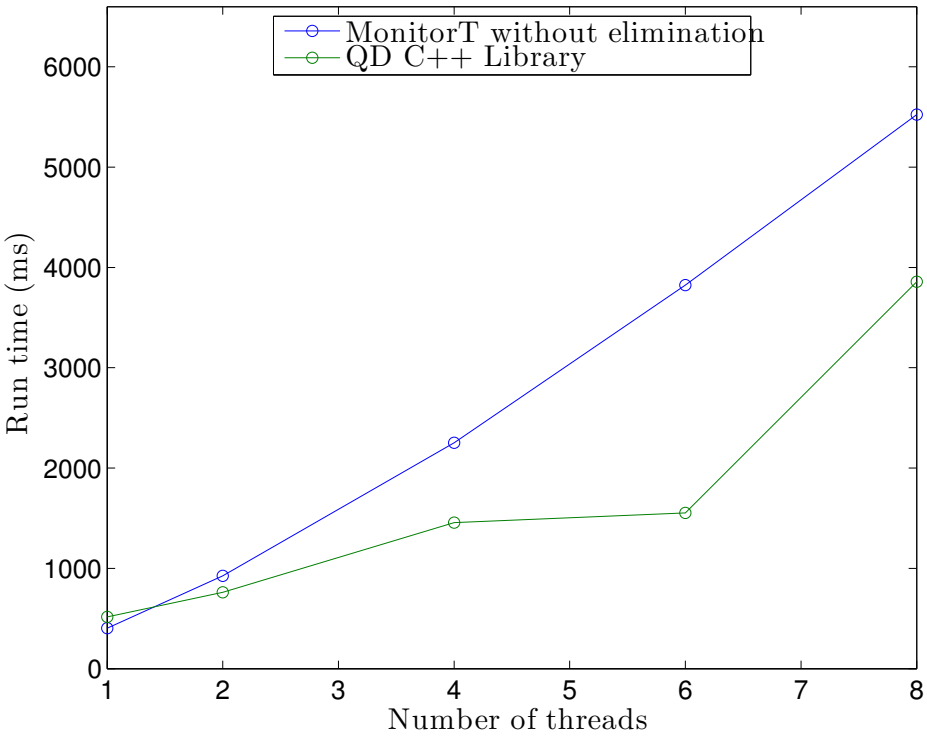


Figure 3: Comparison of the MonitorT implementation (Java) and the QD Elimination Lock (C++), varying the number of threads (elimination included for C++ implementation, not for Java). Queue size = 256, push ratio = 0.5.

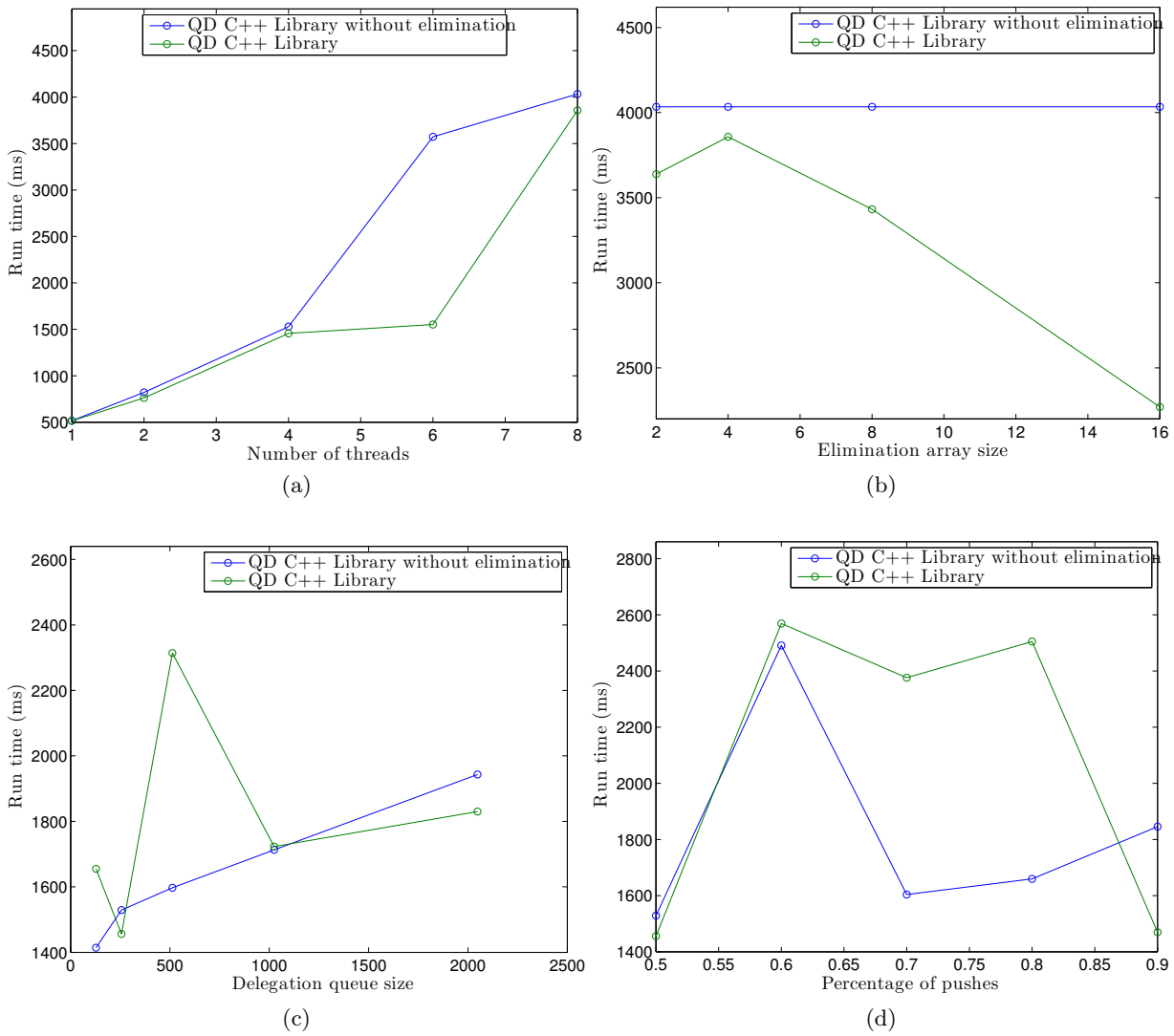


Figure 4: QD Elimination Lock versus QD Lock (both C++), while varying each parameter separately: (a) varying the number of threads (elimination array size = 4, queue size = 256, push ratio = 0.5); (b) varying the elimination array size (threads = 4, queue size = 256, push ratio = 0.5); (c) varying the delegation queue size (threads = 4, elimination array size = 4,push ratio = 0.5); (d) varying the push ratio (threads = 4, elimination array size = 4, queue size = 256).

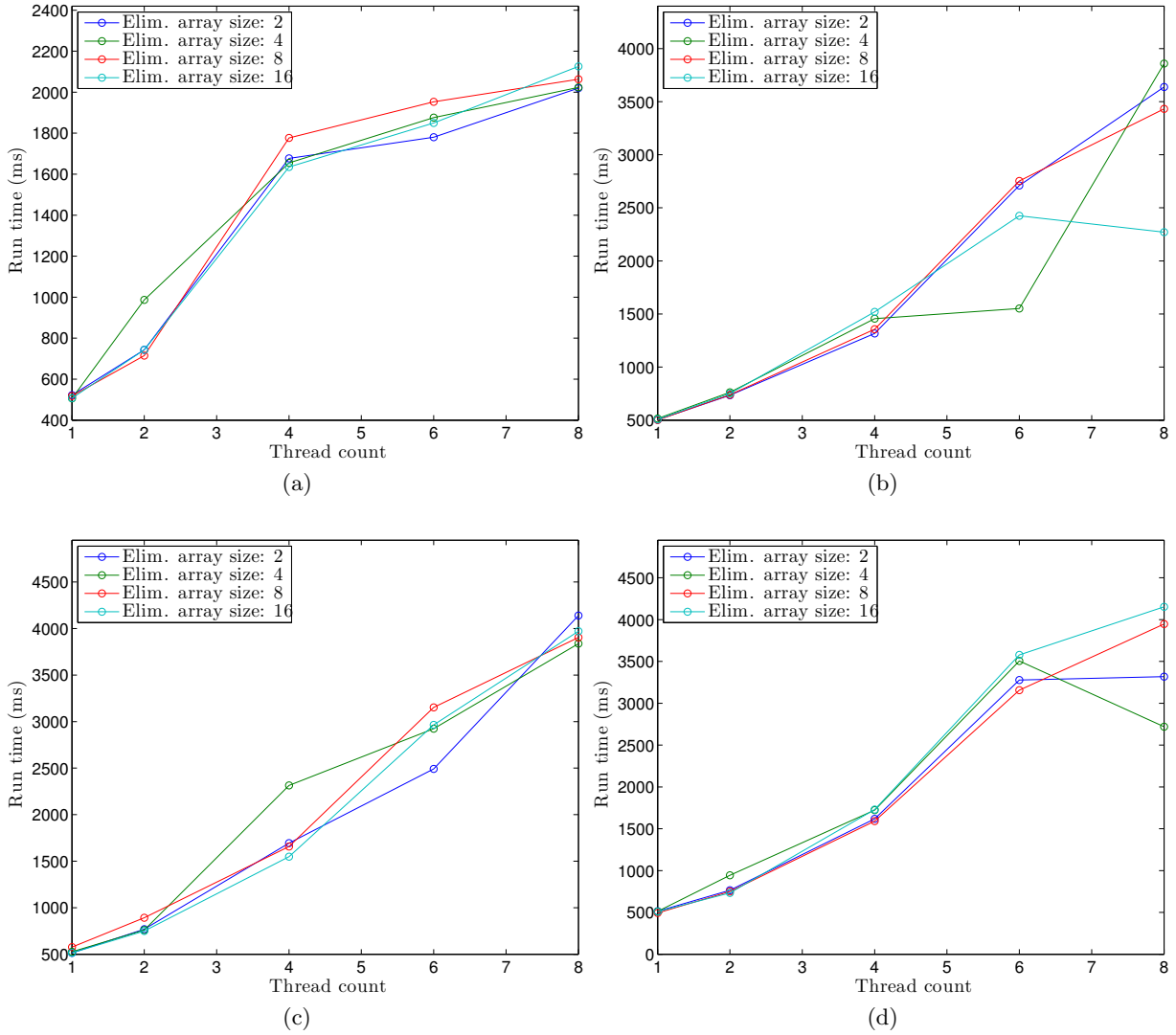
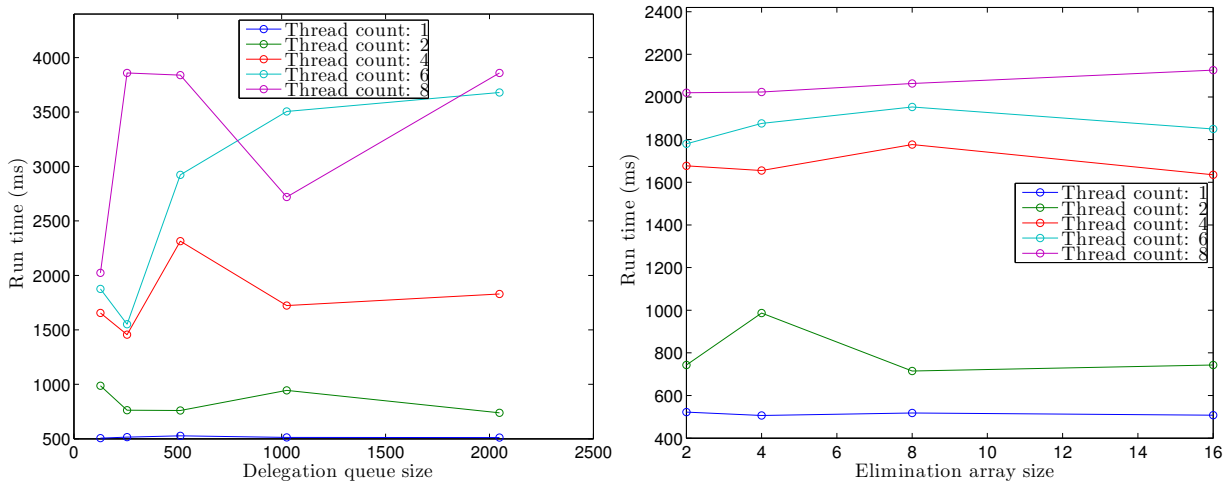


Figure 5: QD Elimination Lock performance versus thread count and elimination array size - each graph represents a different delegation queue size, while multiple elimination array sizes are plotted in each graph: (a) queue size = 128; (b) queue size = 256; (c) queue size = 512; (d) queue size = 1024. Push ratio = 0.5.



(a) (b)

Figure 6: (a) QD Elimination Lock performance versus delegation queue size and thread count. (b) QD Elimination Lock performance versus elimination array size and thread count. Push ratio = 0.5.