Investigating priority scheduling algorithms

Project by Ido Kessler, Tal Leibovitch and Gilad Fudim.

Advanced Topics in Multi-Core Architecture And Software Systems, January 2019

Github: https://github.com/tnt22/KLSM-MQ_CAPQ-Galois

Introduction

Priority queues are data structures which store keys in an ordered fashion to allow efficient access to the minimal (maximal) key. They are a data structure for maintaining a set of keys (potentially stored along with some data) such that the minimum (or maximum) key can be efficiently accessed and removed.

In its simplest form, a priority queue supports:

1. insert() - Insertion of new keys.
2. delete-min() - Finding and deleting a minimal key.

*Priority queues may additionally support the deletion of arbitrary keys and decreasing (or increasing) the value of a key, as well as operations to meld and split queues.

Priority queues are essential for many applications, e.g., Dijkstra's single-source shortest path algorithm, branch-and-bound algorithms, and prioritized schedulers.

As the popularity of multicore processors and multiprocessor computing rises, the requirement for efficient concurrent data structures grows with it. Multiprocessor computing requires implementations of data structures that can be used concurrently and scale to large numbers of threads and cores. It turns out, however, that linearizable priority queues have so much overhead that they negate the benefits of parallelizing the algorithm. One approach for addressing this problem is to design more efficient priority queues. Another approach is to relax the priority queue guarantees. Because tasks can run in any order (maybe just less efficiently), we can consider priority queues that try to return some high-priority task instead of the highest-priority one. It turns out that this relaxation enables designing very lightweight and efficient priority queues, and there are many proposals in the literature.

While there are many priority queues and relaxed priority queues designs in the literature, there are insufficient insights into their relative strengths and weaknesses. In particular, the different designs have never been evaluated against each other. Our goal with this project is to implement a couple of priority queues in Galois and test them against each other.

k-LSM queue

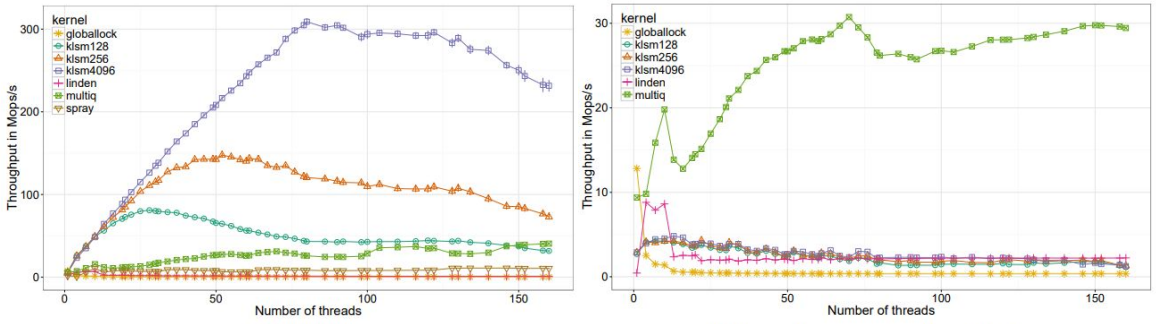
The k-LSM is lock-free concurrent priority queue built from a logarithmic number of sorted arrays of keys. It relaxes linearization requirements on insertions, thus allowing each thread to batch together up to k insert operations before it is required to linearize them wrt. insertions by other threads. The delete-min operation is also relaxed so that it may delete and return any of the k + 1 smallest keys visible to all threads. The parameter k can be configured at run-time, and can even be changed on a per-key basis. We have an implementation of the k-LSM queue in Galois[1]. However, it performs abysmally. Our goal is:

1. To understand why the performance is so poor.
2. To try and fix it, by improving either the algorithm or the implementation.
3. Measure it against others priority queues in a similar environment.

Why the performance is so poor

We will start by examining the paper[1]. We will notice that the priority queue is tested on is SSSP that ran on Erdős–Rényi random graphs[12] with 10000 nodes and edge probability 50%. This will

mean that there won't be a lot of collisions and there will be a lot of improvement for the algorithm time from the inherent build of the graph as opposed to real-world graphs that don't observe such high connectivity, a high number of neighbors for each node. This leads us to the conclusion that the priority queue wasn't properly tested at the beginning and it is showing good result only on very specific graphs which allows the k-LSM to work in good parallelism. We checked other papers to gain more insights into the performance of the k-LSM queue. In the article[2] we can see how the k-LSM benefits come to light only when all threads perform inserts and deletions together. As opposed to when there is a split workload, meaning when some threads perform only deletions and some perform only insertions, the queue performance falls significantly. In the figures we can see the difference between the two scenarios:



Uniform workload, uniform keys (32 bits).

Split workload. uniform keys (32 bits).

This paper reinforces our hypothesis that the queue wasn't properly tested in the original paper[1] and further analysis may be needed to better understand the queue weak points. From the [2] article we learned that the k-LSM priority queue seems superior to the other priority queues in specific scenarios: in uniform workload combined with uniformly chosen keys, throughput is almost 10 times that of other priority queues. However, in most other benchmarks its performance is disappointing. We hypothesize it is due to the different loads placed on its component data structures: whenever the extremely scalable D-LSM is highly utilized, throughput increases, and when the load shifts towards the S-LSM, throughput drops. This leads us to the first idea for improvement, mainly focusing on the improvement of the S-LSM, maybe to interchange the S-LSM with other shared data structure.

Fixing and improving

From the article[2] and our own benchmarks we came up with a couple of possible improvements to the k-LSM.

1. Dynamic K/Dynamic Queue - Which K for the k-LSM to choose? We observed that some priorities queues have advantages on particular graphs and benefits/drawbacks on different applications. This idea can be generalized: Which delta for the OBIM to choose? Which queue to choose for the app?

   Measuring time of each aspect of the data structure, will enable us to better understand the performance and which queue to use on each graph and application.

   As it is a high order parameter tuning job, a natural route will be building a machine learning model for the problem: given an application and graph characteristics, it returns the data structure and parameters(k,delta,...) with best-expected performance.

   > *This idea is good but it is general to all queues. It may require modifications to other queues as well and may be a stand-alone project on itself.*

2. Data Structure Hierarchy - Hierarchy of data structures. Instead of using only the shared and the distributed LSM, using a hierarchy of shared LSMs that benefit us by making the probability of contention smaller.

One option for such hierarchy is to use “per socket” storage between the global S-LSM, and the local one.. Making the probability of moving data between sockets smaller.

3. Workload Management - When the local queue is being filled up, send only the upper half to the shared LSM, leaving the better options for yourself. Benefiting from the locality of working with neighborhood of nodes on the graph.

    > *it might also suffer from leaving a single thread with all the “good” work, and the rest only with his “bad” work.*

4. Improving the weak link - Replacing the S-LSM to a more scalable data structure i.e. Skiplist or k-skiplist. The results of further benchmarking of the time the insert and delete operation each spend on the S-LSM and the local LSM reveal a massive load coming mostly from the S-LSM work (100 times more). To verify this idea and make sure we are taking the right direction, we first implemented a first “variant” of the idea just to check whether changing the S-LSM to some other type of queue will allow us better performance. Because S-LSM usage is by peeking, we decided implementing a naive “peek” by calling “delete_min” on the queue, and in case of not choosing it, adding it to the local queue instead. This, while making the k-LSM have a lot more bad work, provided performance improvement, which led us to take this direction.

Implementing the fixing and improving

Looking at various implementation inside the Galois, we chose to use the MultiQueue implementation for our shared data-structure. The reason, the relaxed behavior of the queue which we thought allowed easy implementation of a peek operation, and the already implemented “peek” operation that is used inside of it. Recent work on the multiqueue have also shown that the expectation of “regret” (bad work) while using such scheme is lower than expected[10].

The Contention Avoiding Concurrent Priority Queue (CA-PQ)

The Contention Avoiding Concurrent Priority Queue(CA-PQ)[9] is a relaxed concurrent priority queue with traditional/strict semantics under low contention but activates contention avoiding techniques that give it more relaxed semantics when high contention is detected. CA-PQ has two contention avoidance mechanisms that are activated separately: one to avoid contention in insert() operations and one to avoid contention in del-min() operations by removing items in bulk from the global data structure, which allows it to often serve del-min() operations without accessing memory that is modified by several threads. This way the CA-PQ solves the inherent sequential bottleneck in the head of the priority queues while achieving better performance than other solutions in the field. Our goal is:

1. To port, the code of the Contention Avoiding priority queue to Galios(cpp)
2. Measure it against others priority queues in a similar environment.

A Brief explanation of CA-PQ

Contention avoiding concurrent priority queue (CA-PQ) is an implementation of a priority queue which provides good performance as an answer to the rise in concurrent software implementation that requires scalable and efficient data structures.

CA-PQ behaves as a strict concurrent priority queue when contention is low and has relaxed semantics when contention is high. The contention is frequently observed by 2 special contention detection mechanism (one in “deletion” operations and the other in “insert” operations), thus minimising overlooking of high priority items often leading to a performance degradation.

This data structure operates by introducing a global component and thread local components, when contention is low, “delete” removes the smallest item from the global priority queue and “insert” adds an item to the global priority queue.

Both “delete” and “insert” detect contention by a set of counters (delmin_contention, insert_contention) with an adaptive delimiter to allow dynamic real time optimization to establish when contention avoidance mechanism should operate.

The CA mechanism for “insert” operations buffers (insert_buffer) a number of consecutive items insertions thus reducing the number of inserts to the global qa, as long as the buffer hasn’t maxed on its capacity.

The CA mechanism for “delete” operations, works by returning k smallest items in a buffer (delete_buffer) from the global priority queue and then selecting minimal item from them, this is done by implementing a skip list that stores multiple items per node as a backing data over a contention adapting search tree (CATree).

This reduces the contention of “delete” operations on the global pq by reducing the number of accesses by up to k - 1 per k operations (when delete_buffer and insert_buffer are empty, otherwise it returns the smallest item among them).

If the global pq is empty, it returns a special item as an indication (empty_pq) to the “delete” op.

Because CATree uses locks in order to allow access to the smallest items of the tree, and locks are devastating for concurrent performance as we've seen consecutively throughout the course, the writers of the data structures are using delegation locking as a means of dealing with this issue.

This mechanism lets a current lock owner thread help other threads by performing their critical sections, it is also favorable due to its ability to handle void functions well like “insert” operations, because there is no need to wait for critical section return value.

The linearizability of both single item “insert” and “delete” operations, as well as bulk “delete” operations is assured and justified in the article.

Ideas for improvement and further exploration of CA-PQ

1. Since adaptation is done by thread-local modifications for “insert” operations and by changing the global operations for “delete” executions, the article suggests CA for “delete” could also be achieved by only changing a thread local flag if the global priority queue exposes separate operations for deleting a single item and a buffer of items.

   The article suggests this alternative design choice should work equally well, but provides no proof for this claim.
2. During our review over the CA-PQ we came with the idea that the CA-PQ heap implementation could be very well replaced by a k-LSM implementation, which in turn provides a better cache utilization and cache management that might lead to a mild performance boost.
3. Routing nodes in the CATree can be read by multiple threads, so their reclamation is done using Keir Fraser’s epoch based reclamation.

   Unlike “delegation locking”, authors do not provide justification for this reclamation method and we think this lead might be worth exploring as well.

Porting the code to Galois

The original source code was available to us in C language. Since theoretically a valid C code is a valid C++ code, we built a wrapper in C++ to wrap the C code and make it a valid C++ code.

We’ve also added scripts that auto-compile the CA-PQ combined with the Galois package.

We rely on the correctness of the original source code of the CA-PQ as we didn’t make any changes to the logical operation of the code and only wrapped it to make it a valid C++ code.

The Benchmark Machine

Several queue implementations (k-LSM for example), rely heavily on hardware specifications such as a different number of sockets. These differences (bigger caches, number of sockets, number of cores,etc,) result in performance and time differences.

For our project, we chose to focus on the “rack-castor” as it has a higher count of cores as many other papers didn’t have the access to such kind of machine, we wanted to test the scaling of the queues to a higher count of cores and threads. The “rack-castor” is comprised of 4 Intel Xeon E5-4669 v4 CPUs that each has 22 cores and 44 threads (by hyperthreading)[8]. In conclusion, we have 88 cores and 176 threads between 4 sockets. In terms of memory, the machine has 528GB of RAM (we didn’t get the clock speeds).

Benchmarks and Results

To benchmark our old and new priority queues we use the following graphs. To better understand the performance of the queue on the different graphs we need to know the nature of the graph itself i.e. the number of nodes, the number of arcs, the average degree of the nodes etc. This information will later help us understand what are the weak and strong points of each priority queue. As we saw in the original article of the k-LSM[1] testing the queue on a single type of graph can be misleading and can lead us to wrong conclusions. Because of that, we used multiple graphs of multiple types and characteristics to better test and analyze each queue. Our set of graphs is comprised of both synthetic and real-world graphs, in total we have 10 graphs that we are going to test on our queues.

We tested the queues using Galois SSSP application. First, we tested the baseline performance of all the queues that were available to us:

1. k-LSM: with k=256,512,1028,2048,4096 (shades of purple).

   We saw that smaller K’s don’t achieve great results so we passed on benchmarking them.
2. OBIM: as a baseline performance, we used delta=4(lightslategrey).
3. Multiqueue: with c=1,4 that were already implemented in Galois(shades of red).
4. Spraylist[13](pink)

Then after development of the new queues we tested them as well(on the same machine):

1. Capq: the CAPQ we ported to Galois(orange)
2. Klsm256mq: A combined k-LSM with multiqueue with c=1.(teal)
3. Klsm256mq4: A combined k-LSM with multiqueue with c=4.(lightseagreen)
4. Klsm64mq4: A combined k-LSM with multiqueue with c=4.(springgreen)
5. Klsm4096mq4: A combined k-LSM with multiqueue with c=4.(limegreen)

Benchmark procedure: we run each queue on each graph for one time, if there wasn’t a timeout we run it for additional 10 runs to get statistically stable results.

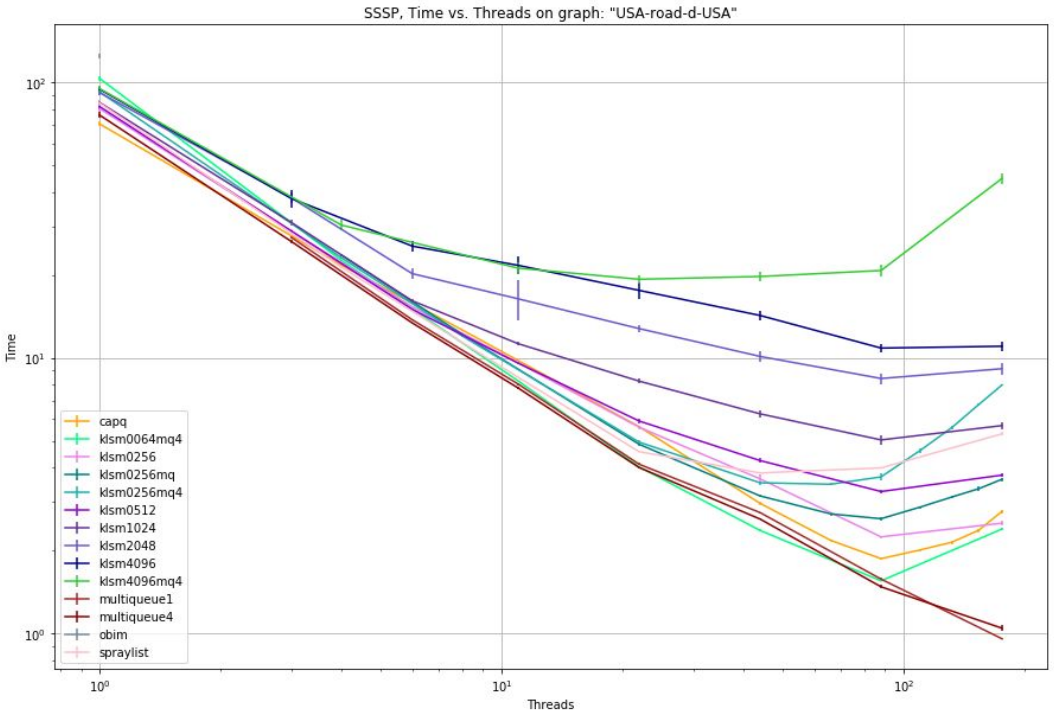
Analysis: we used the csv.py to extract the information and then used pandas and matplotlib to display the results. If a queue is not presented on the test results it means it either timed out or resulted an incorrect result. The Y-value is the mean time and the Y-error is the standard deviation. We used logarithmic scale on both axis to better visualize the scalability of the queues.

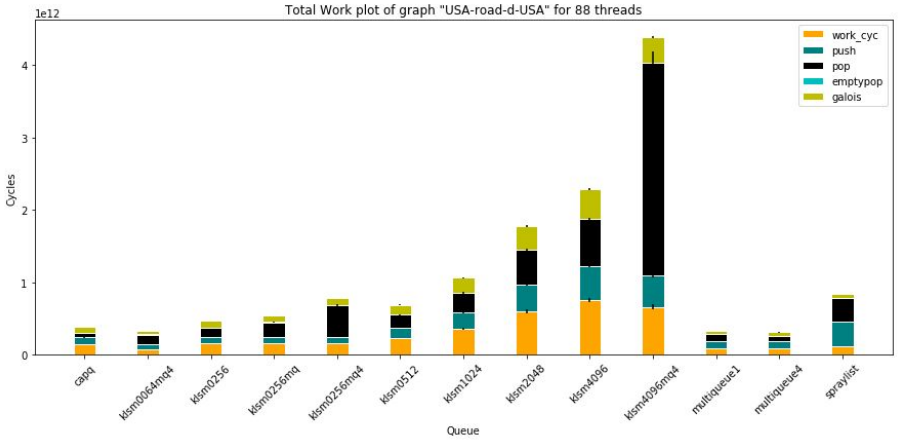
1. Road:

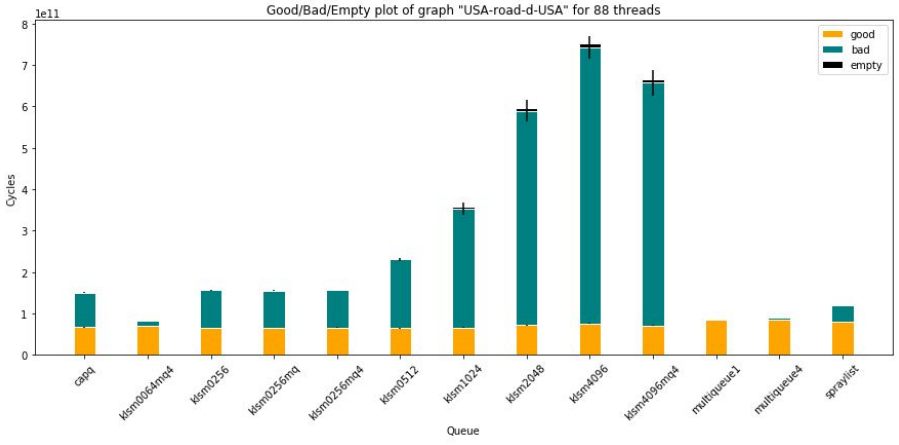
   The graphs are taken from DIMCAS[7]. These graphs are low density and fixed range of weights. As we discussed in class we would expect that the OBIM will struggle as the weights aren’t uniform.

a. USA-road-d-USA:

a distance graph of the USA roads. There are 23.9M nodes, 58.3M arcs.

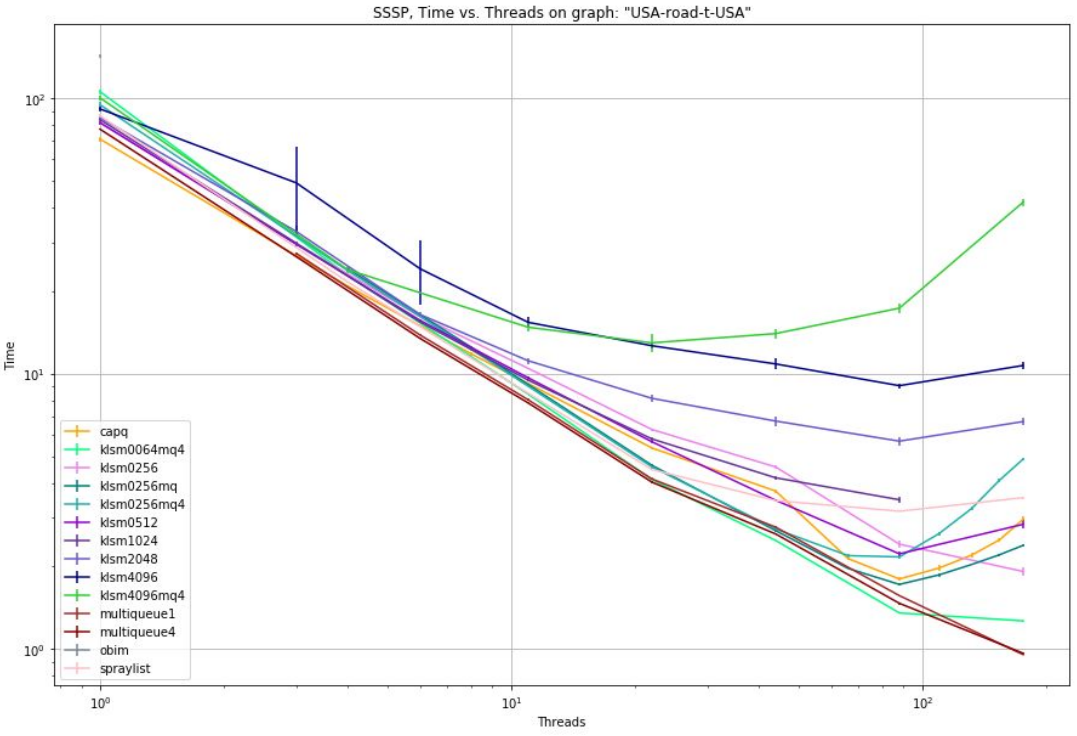


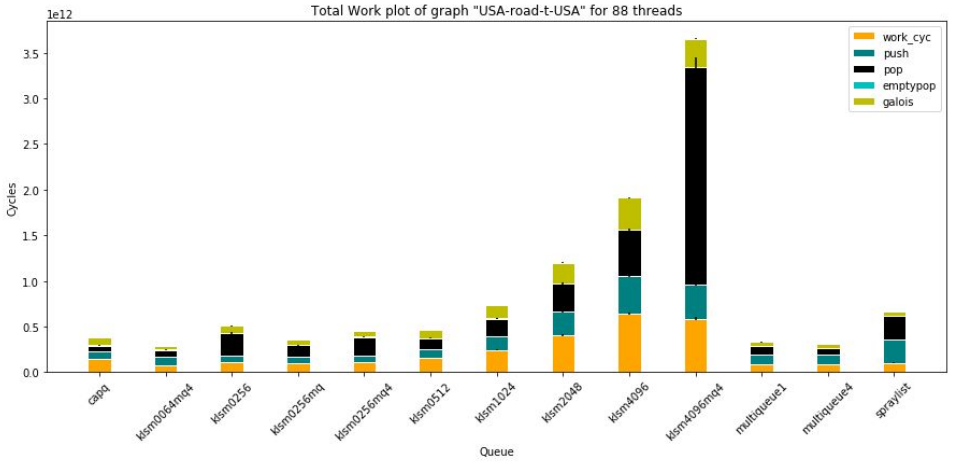


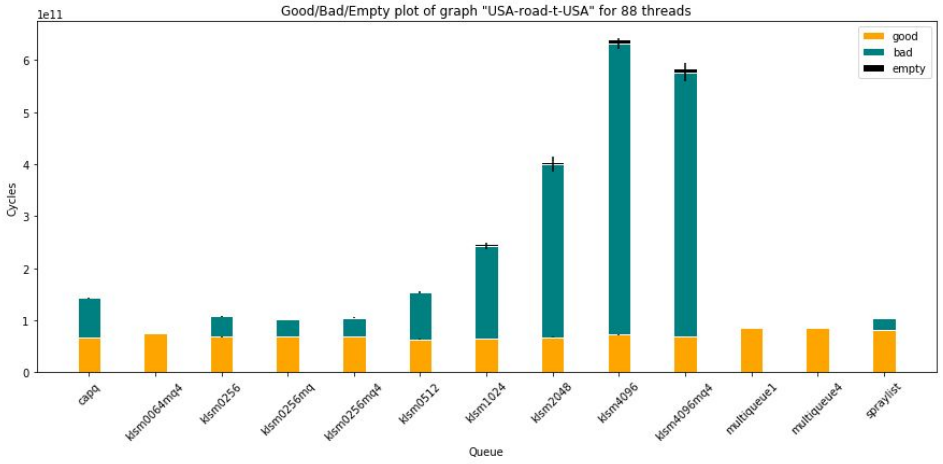


b. USA-road-t-USA:

a travel time graph on the USA roads. There are 14.1M nodes, 34.3M arcs.

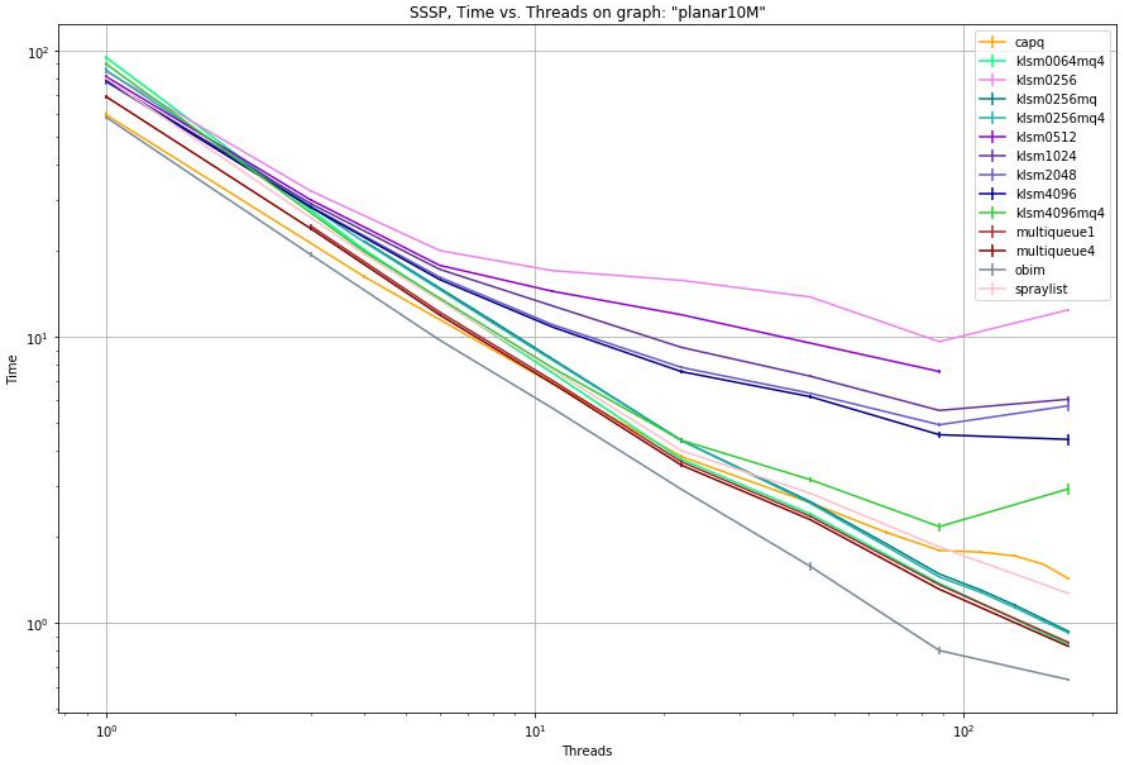


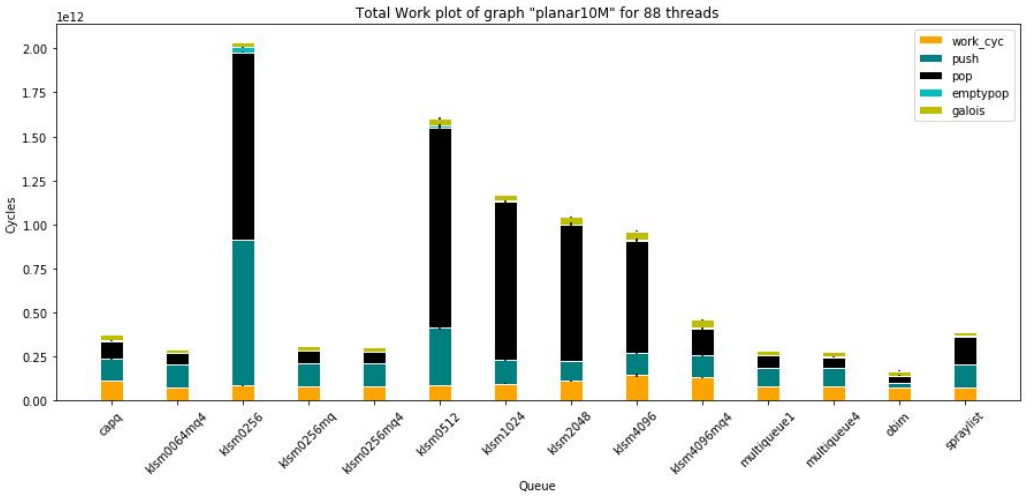


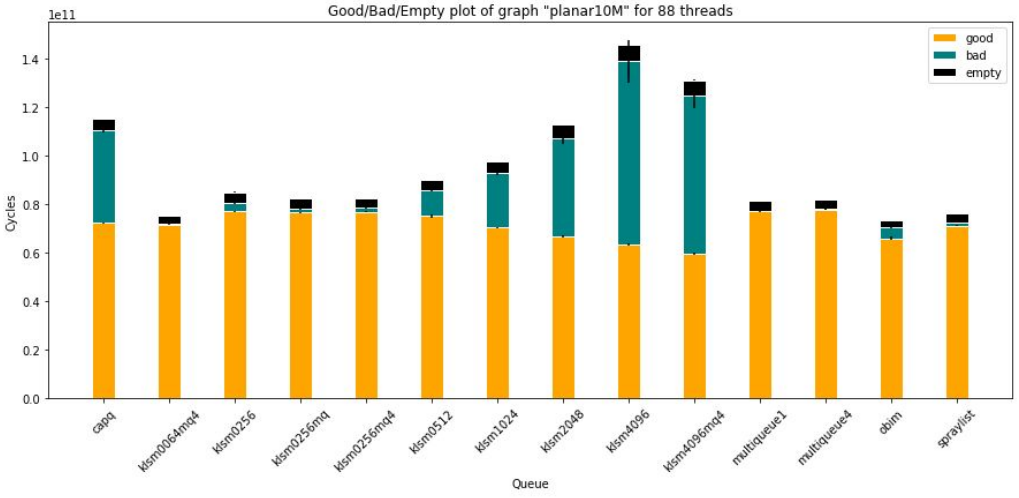


For both graphs we can see similar results. As expected the OBIM couldn't handle the weights and timed out. It is possible that further tuning of delta will result in better results. Interesting to see is that actually lower K's achieve better results and even lower K's than 256 may result in better times. Our newly created queues perform well until higher threads counts, where it seems their bad work affect their performance.

2. Planar10M:
   a planar graph is a graph that can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints. In other words, it can be drawn in such a way that no edges cross each other. Any planar graph with n vertices has m<3n-6 edges by Euler formula. Our graph is a planar graph with 10M nodes, 30M arcs and an average degree of less than 6[6].

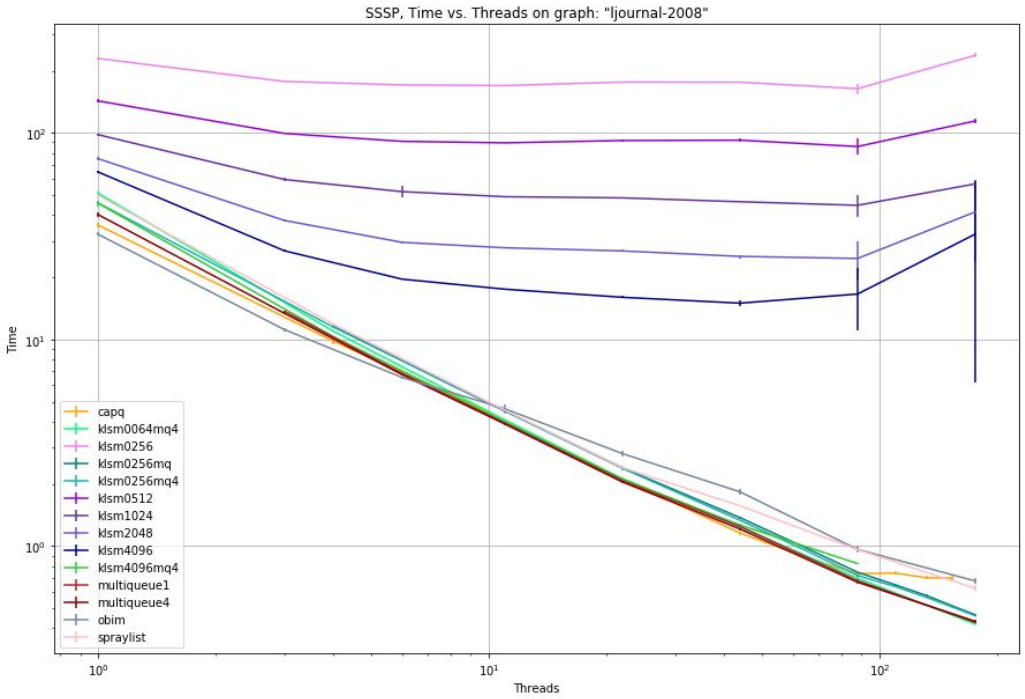






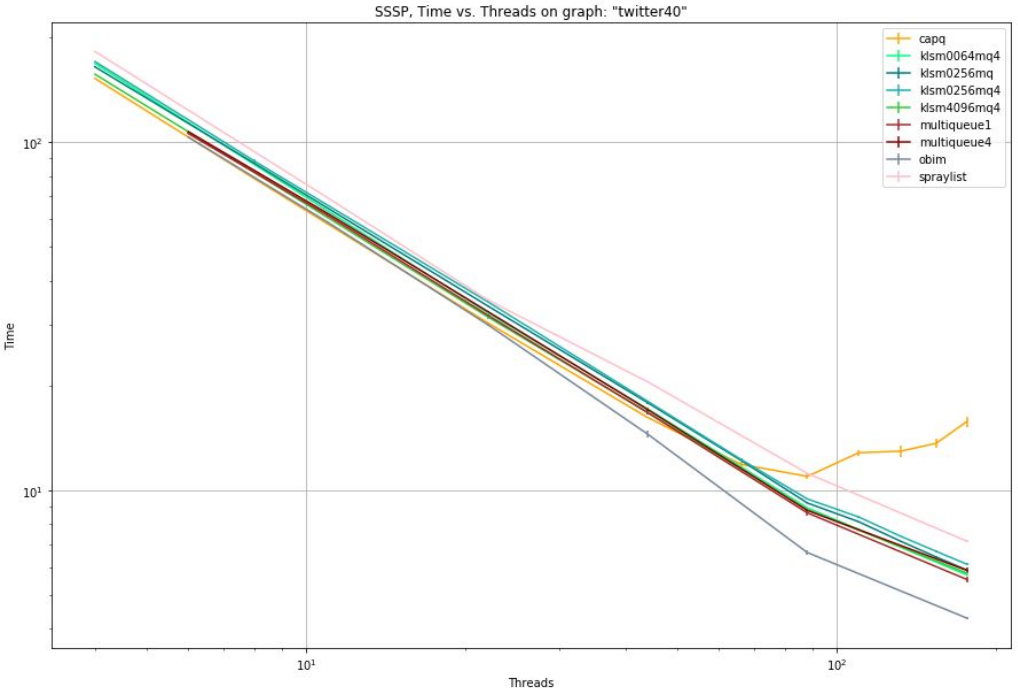
We can see that all of the queues managed to finish and the OBIM achieved the best results. This planar graph has many nodes with close distances that can be simultaneously, which is the best example of a graph where obim gain the most advantage. We can see that the multi queues variation and the klsm_mq variation also utilize this property.

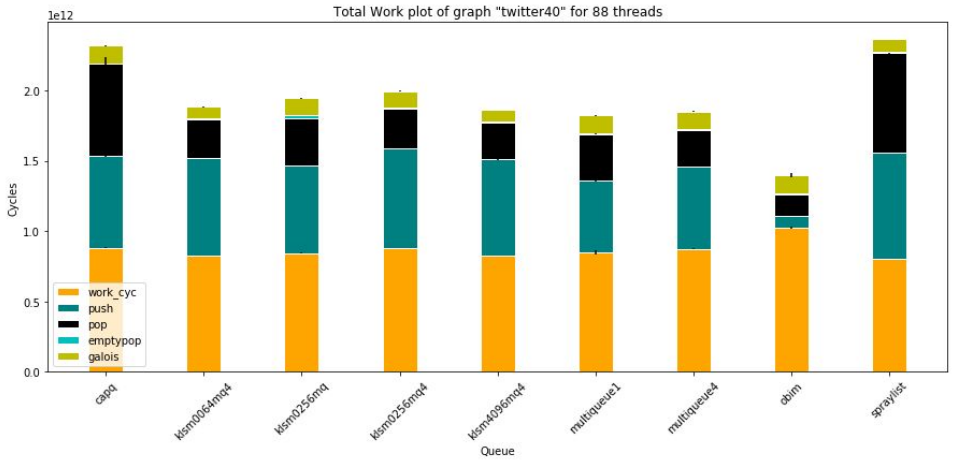
3. Ljournal-2008:
   LiveJournal is a virtual-community social site started in 1999: the gathering of data was made in 2008, nodes are users and there is an arc from x to y if x registered y among his friends. There are 5.3M nodes, 79M arcs and an average degree of 14.734.[5]

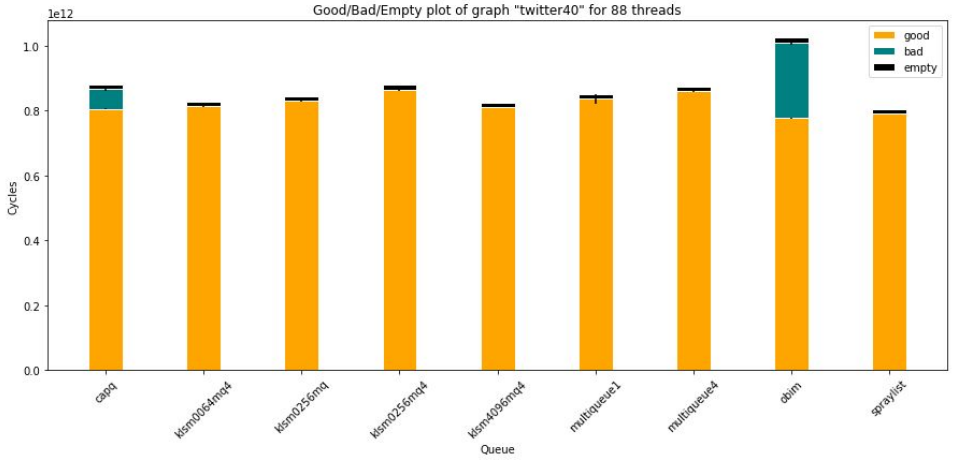


All the queues managed to finish but the k-LSM which had a hard time dealing with this graph. It is possible that contention points through the graph breakdown the queue ability to parallelize and it falls back to the shared queue, his weak point, resulting in poor performance.

4. Twitter40:
   crawled the entire Twitter site in 2009 and obtained 41.7M user profiles and an average degree of 70.506[4]. This the largest real graph that we had to test out, while from first examination the density seems sufficient and somewhat similar to “LJournal2008” for the k-LSM to finish.





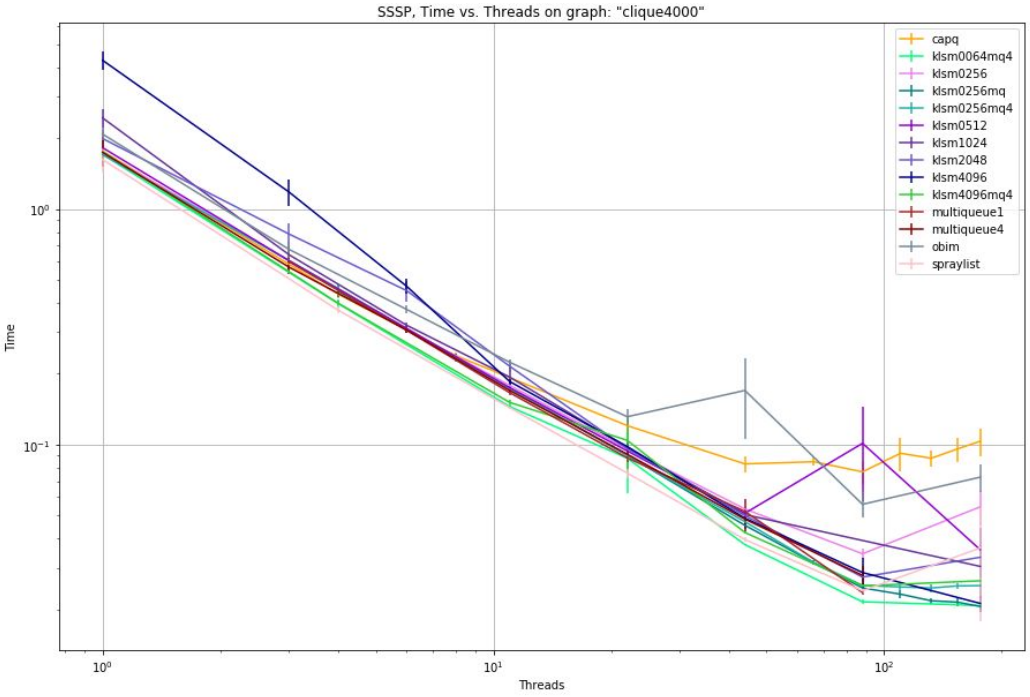


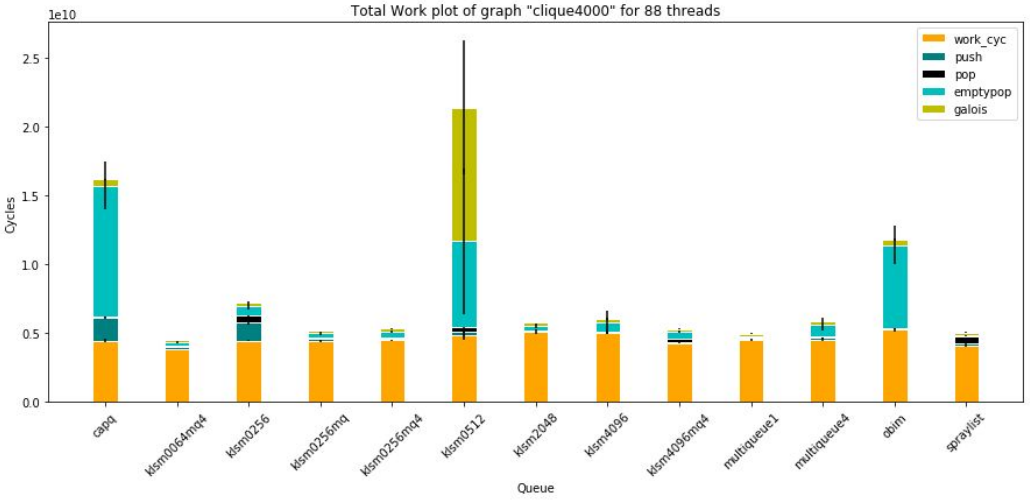
The k-LSM didn't finish and it probably due to some contention points through the graph and the size of the graph itself. It took all other queues a longer time to finish compared to the LJournal but all succeeded to parallelize except for the CAPQ who started to plateau and even had increased time at higher threads count probably because the higher contention from the number of threads.

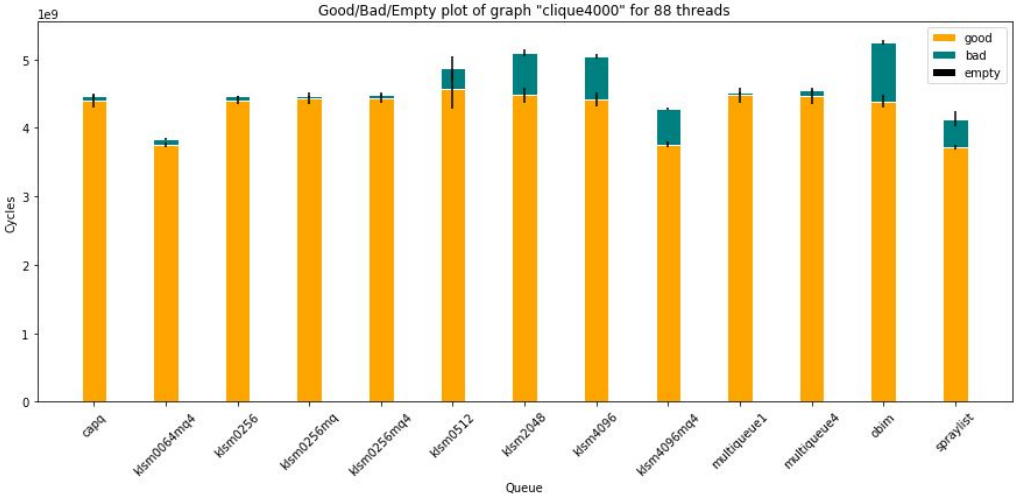
5. Clique4000:

   a clique is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent, that is, its induced subgraph is complete. In our graph, there are 4,000 nodes, 7.998M arcs and an average degree of 3999 (complete graph). It was generated using Galois tools, "clique.py".

   Due to similarity with the Erdos-Ryes, both very dense graphs. We will hypothesis that the k-LSM could benefit and shine in performance.


SSSP, Time vs. Threads on graph: "clique4000"
capq
klsm0064mq4
klsm0256
klsm0256mq
klsm0256mq4
klsm0512
klsm1024
klsm2048
klsm4096
klsm4096mq4
multiqueue1
multiqueue4
obim
spraylist
Time
Threads


Total Work plot of graph "clique4000" for 88 threads
1e10
work_cyc
push
pop
emptypop
galois
Cycles
Queue
capq
klsm0064mq4
klsm0256
klsm0256mq
klsm0256mq4
klsm0512
klsm2048
klsm4096
klsm4096mq4
multiqueue1
multiqueue4
obim
spraylist

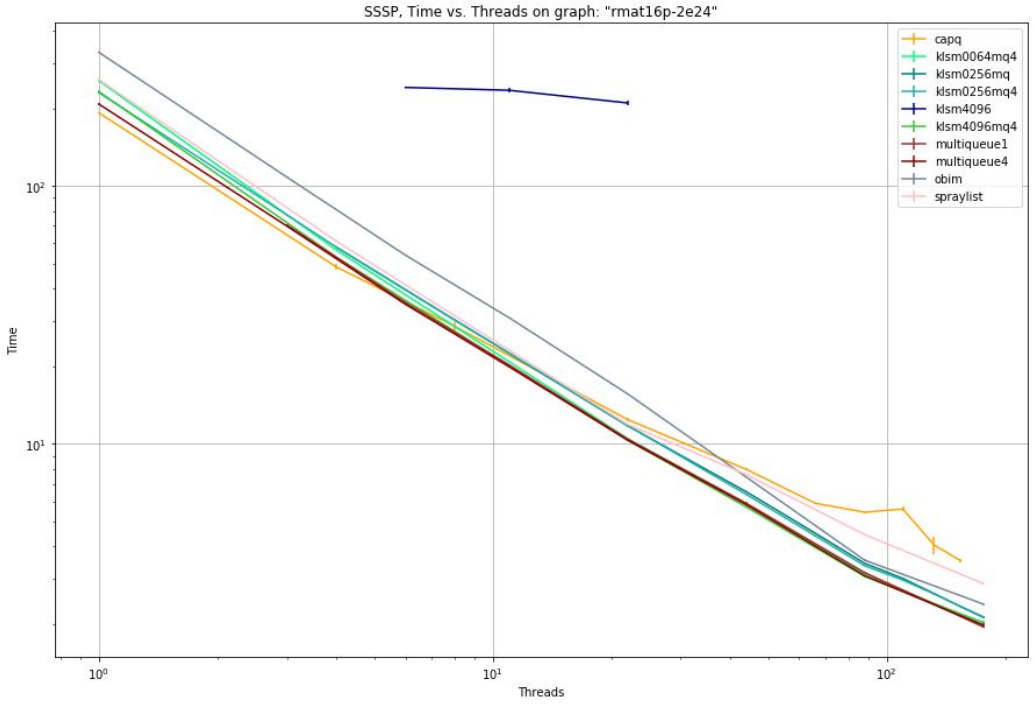

Good/Bad/Empty plot of graph "clique4000" for 88 threads
1e9
good
bad
empty
Cycles
Queue
capq
klsm0064mq4
klsm0256
klsm0256mq
klsm0256mq4
klsm0512
klsm2048
klsm4096
klsm4096mq4
multiqueue1
multiqueue4
obim
spraylist

As seen above, our hypothesis was right and the k-LSM queues achieve great results. We also can see that our newly created klsm64mq4 queue achieves great even better than the multiqueues results and can scale up with the threads.

6. RMAT:
stands for Recursive MATrix (RMAT) graph generator and may be the most commonly used model to generate realistic synthetic scale-free graphs due to its simple and fast mechanism. The basic idea behind RMAT is recursive quadrant selection on adjacency matrix for an edge generation, and RMAT repeats such an edge generation until a whole graph is generated. Conceptually, it partitions the adjacency matrix into four quadrants, which have probability parameters α, β, γ, and δ [3].

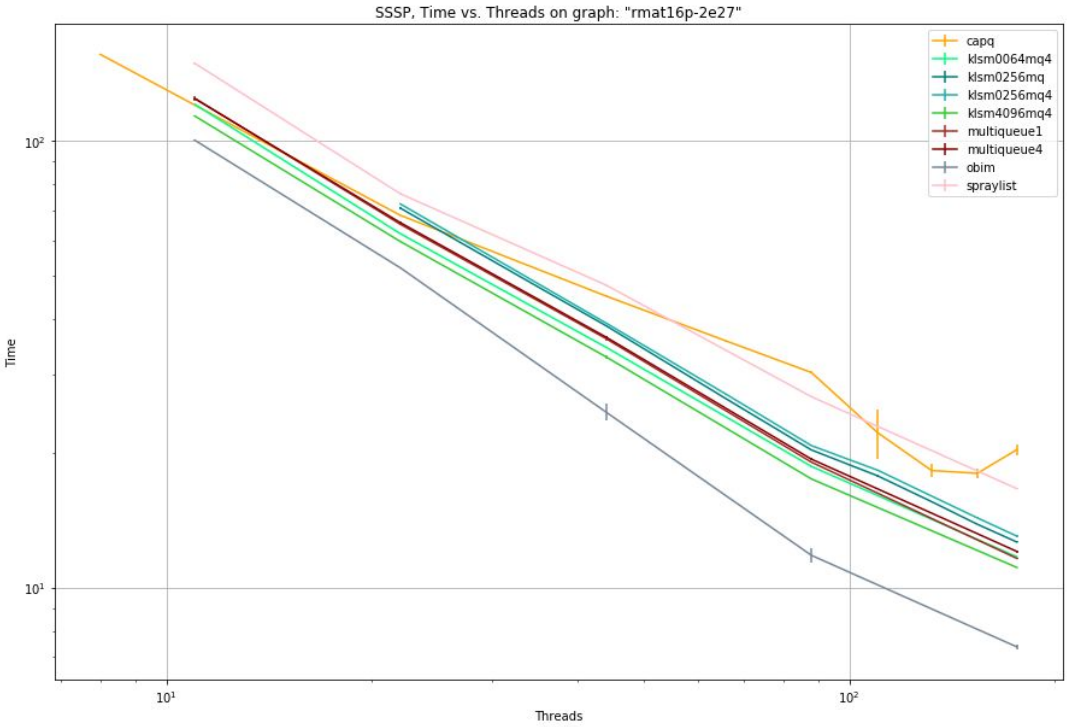
   a. Rmat16p-2e24:

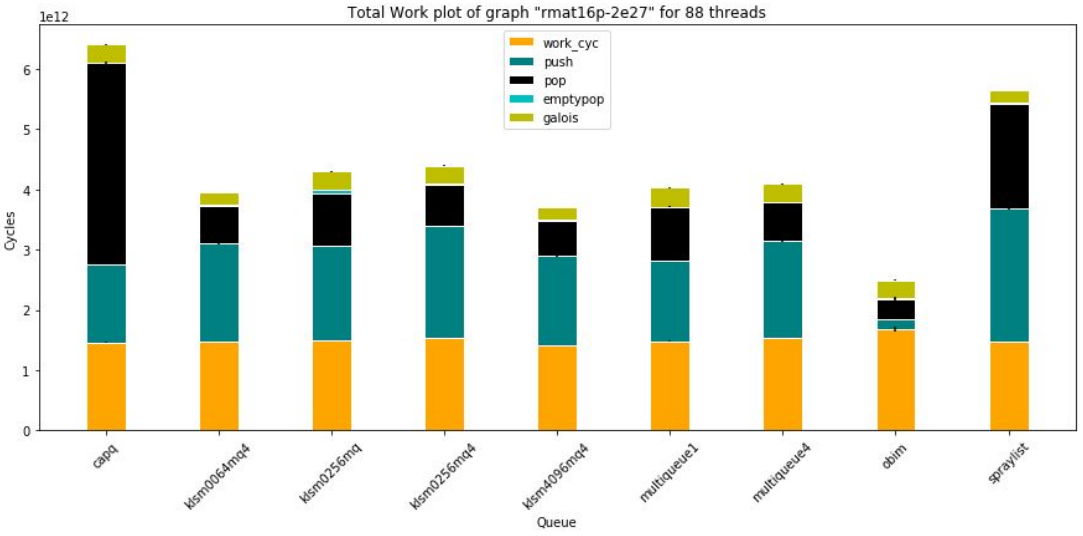
      α =0.5, β=0.1, γ=0.1, and δ=0.3



b. Rmat16p-2e27:

α =0.57, β=0.19, γ=0.19, and δ=0.05:



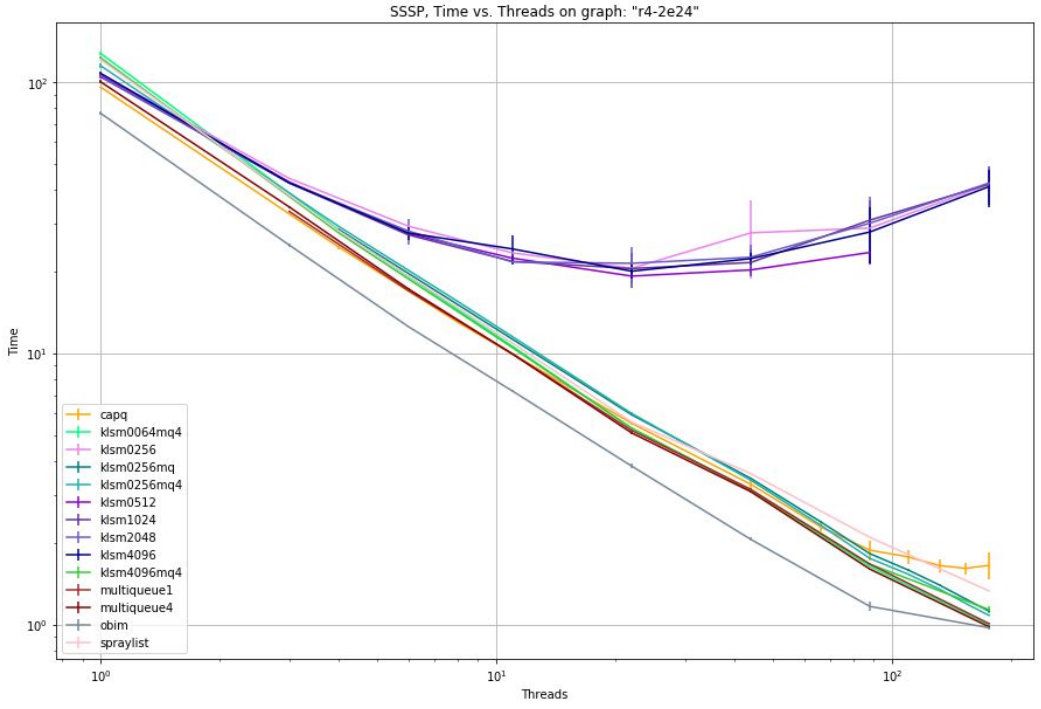


We can see that both results are pretty similar. All of the k-LSM's timed out and all other queues achieve relative the same times with an advantage to OBIM. We can see also that in second and third place are our newly created klsm4096mq4 and klsm64mq4 queues.

7. Random:
Random graphs that were generated using Galois tools, “random-graph.py”.

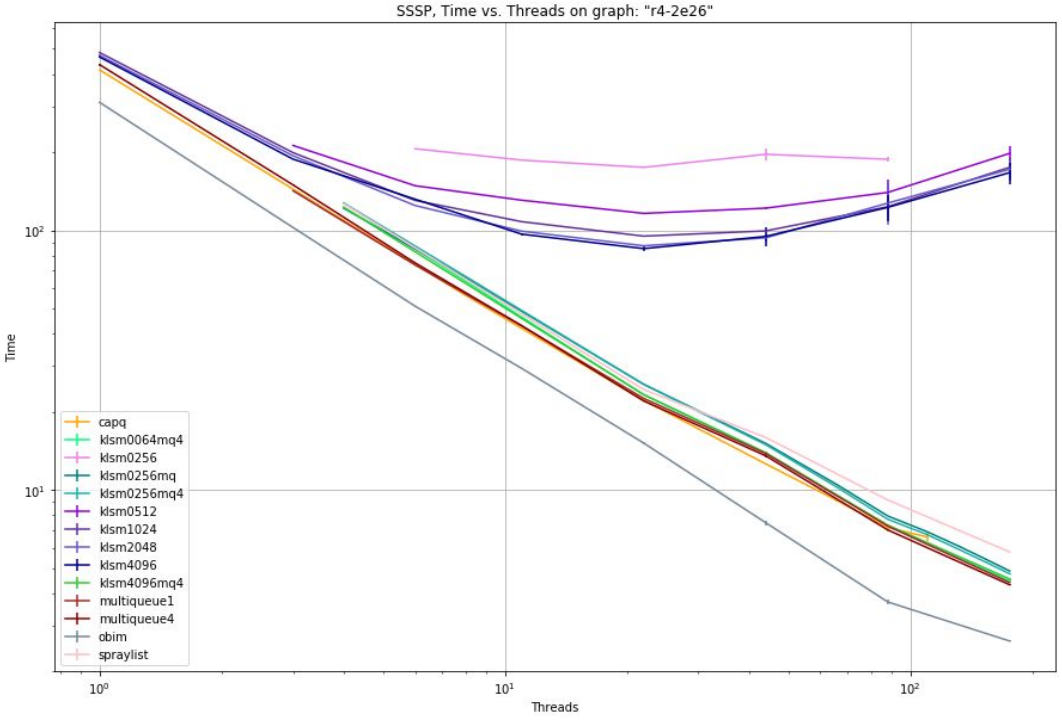
a. R4-2e24:

a random graph that has $2^{24} \approx 16.8M$ nodes and density 4. (density = (the number of edges) / (the number of nodes)).



b. R4-2e26:

a random graph that has $2^{26} \approx 67.1M$ nodes and density 4.



We can see that both results are pretty similar to each other. All of the k-LSM's perform poorly and all other queues achieve relative the same results with slight advantage to OBIM. The k-LSM does not succeed in achieving any parallelism because of his relynes of the shared data structure that is his weak point. This is fixed after changing his shared data structure type.

Conclusions

Contention avoidance priority queue:

Using both his contention avoidance scheme to reduce the insert and the delete_min time while highly in use, the contention avoidance priority queue achieve pretty good results with high scalability factor. We can see that all the graphs show pretty solid performance over all, meaning that the contention avoidance mechanism is dynamic enough not to hurt the performance in different graphs.

Overall the contention avoidance mechanism might be a good performance boost for many other queues, and might be an option for further research to implement those under different priority queues that does not scale too well, but as the code for the CAPQ is already amazingly optimized, the general conclusion is that the current implementation is not competitive enough if it does not win over less optimized priority queues with different designs.

k-LSM:

We see, as suspected, the k-LSM wasn't tested properly and in real life scenarios the performance is mildly disappointing. Our newly created priority queue, k-LSM combined with multiqueue, performance much better. This means our first hypothesis, that the shared queue is the bottleneck, is correct but we couldn't see a gain in performance compared to the MultiQueues. This is may be due to the lack of gain performance from allowing performing more bad work in the hope that it will lead for better times. As the local(distributed) queue is much faster as it not synchronized and sits higher in the memory hierarchy. Our idea can be generalized and tested out with other queues. Or improving older structures that could benefit from this architecture of shared and distributed queue that work together.

Benchmarks and Usage:

For an all-around,straight-out-of-the-box, free-configuration solution we will advise to use multiqueue, spraylist or the CA-PQ. Their results are stable through all the benchmarks and achieve relative good results without the need to find the right parameters. For a more optimized solution we will consider testing the application through our newly created k-LSM-MQ and OBIM using a gridsearch for parameter optimization for the specific application.

Future work:

We already suggested a few ideas for improving both the k-LSM and the CA-PQ. Furthermore, as a part of our research we encountered a similarly idea to ours that may be interesting to implement in Galois and to benchmark. It uses the same architecture of k-LSM, shared and distributed queues, but it is based on Skiplists instead of LSM[11], we tried to contact the researcher, Ashok Adhikari, but he didn't reach back to us.

In conclusion, we did succeed to better the performance of the k-LSM and NUMA machines may benefit more from the special queue architecture.

Summary

We integrated CA-PQ as priority queue in Galois. Benchmarked and investigated why the k-LSM present such unfavorite results in real graphs. We suggested few ideas on how to improve the efficiency of the queue, and implemented the idea we thought will have the most impactful results, swapping the inner shared data-structure which was being used (S-LSM) with a MultiQueue with "peek" operation we implemented. Benchmarking all of the above against known queues on real graphs. We compared the queues against each other and summarized our conclusions for our project and future work.

References

The following papers are the ones we are planning to base our work on.

1. k-LSM queue
   https://arxiv.org/pdf/1503.05698.pdf
2. Benchmarking Concurrent Priority Queues: Performance of k-LSM and Related Data Structures
   https://arxiv.org/pdf/1603.05047.pdf
3. TrillionG: A Trillion-scale Synthetic Graph Generator using a Recursive Vector Model
   https://infolab.dgist.ac.kr/~mskim/papers/SIGMOD17.pdf
4. What is Twitter, a Social Network or a News Media?
   http://www.ambuehler.ethz.ch/CDstore/www2010/www/p591.pdf
5. Ljournal-2008:
   http://law.di.unimi.it/webdata/ljournal-2008/
6. Planar graphs
   https://users.dcc.uchile.cl/~jfuentess/datasets/graphs.php
7. DIMACS
   http://users.diag.uniroma1.it/challenge9/download.shtml#benchmark
8. Intel Xeon CPU E5-4669 v4
   https://ark.intel.com/content/www/us/en/ark/products/93805/intel-xeon-processor-e5-4669-v4-55m-cache-2-20-ghz.html
9. Contention avoiding priority queue
   http://www.it.uu.se/research/group/languages/software/ca_pq/relax_prio_queue_paper.pdf
10. The Power of Choice in Priority Scheduling
    https://arxiv.org/abs/1706.04178
11. Non-blocking Priority Queue based on Skiplists with Relaxed Semantics
    https://digitalscholarship.unlv.edu/cgi/viewcontent.cgi?article=3935&context=thesesdissertations
12. Erdős–Rényi random graphs
    https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93R%C3%A9nyi_model
13. Spray List, a scaleable priority queue based on Skiplist
    http://groups.csail.mit.edu/mag/SprayList_full.pdf