The Contention Avoiding Concurrent Priority Queue*

Konstantinos Sagonas and Kjell Winblad

Department of Information Technology, Uppsala University, Sweden

Abstract. Efficient and scalable concurrent priority queues are crucial for the performance of many multicore applications, e.g. for task scheduling and the parallelization of various algorithms. Linearizable concurrent priority queues with traditional semantics suffer from an inherent sequential bottleneck in the head of the queue. This bottleneck is the motivation for some recently proposed priority queues with more relaxed semantics. We present the contention avoiding concurrent priority queue (CA-PQ), a data structure that functions as a linearizable concurrent priority with traditional semantics under low contention, but activates contention avoiding techniques that give it more relaxed semantics when high contention is detected. CA-PQ avoids contention in the head of the queue by removing items in bulk from the global data structure, which also allows it to often serve DELMIN operations without accessing memory that is modified by several threads. We show that CA-PQ scales well. Its cache friendly design achieves performance that is twice as fast compared to that of state-of-the-art concurrent priority queues on several instances of a parallel shortest path benchmark.

1 Introduction

The need for scalable and efficient data structures has increased with the number of cores per processor chip which has steadily increased for the last decade. Concurrent priority queues in particular are important for a wide range of parallel applications such as task scheduling [20], branch-and-bound algorithms [10], and parallel versions of Dijkstra's shortest path algorithm [18]. Typically, the interface of concurrent priority queues consists of an INSERT operation that inserts a key-value pair (called item from here on) to the priority queue, and a DELMIN operation that removes and returns the item with the smallest key from the priority queue. Strict (linearizable) priority queues require that the DELMIN operation always returns an item that had the smallest key of all items in the priority queue at some point during the operation's execution, while relaxed priority queues can return an item that was not the one with the minimum key.

Until quite recently, most research on concurrent priority queues has focused on strict priority queues, e.g. [2, 7, 12, 16–18, 21]. Still, even in the 1990's, there have been a few papers on parallel priority queues that consider more relaxed semantics [8, 15].

Inspired by the realization that the DELMIN operation induces an inherent sequential bottleneck in the head of strict priority queues, some recent papers have proposed relaxed priority queues for modern multicore machines [1, 13, 19, 20]. Even though all these proposals are successful in reducing the sequential bottleneck in the head of the priority queue, they all have a performance problem in that all DELMIN calls access memory

* Research supported in part by the Linnaeus centre of excellence UPMARC (www.upmarc.se).

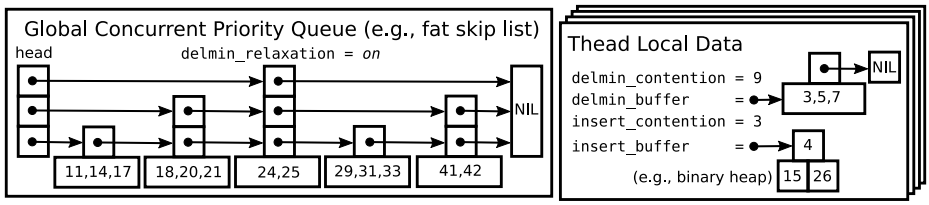


Fig. 1: The structure of a CA-PQ.

that is frequently written to by multiple threads. This is especially expensive on NUMA machines, as it causes data to be transferred between processor chips which in turn may cause long stalls in the processor pipeline and contention in the memory system.

In this paper, we describe a new concurrent priority called the *contention avoiding concurrent priority queue* or CA-PQ for brevity. CA-PQ does not have the performance problem mentioned above. Furthermore, CA-PQ differs from recent proposals in that it works as a strict priority queue when contention is low. Its semantics is relaxed only when operations frequently observe contention. Previously proposed relaxed priority queues have relaxed semantics even when this is not motivated by high contention. This is a problem because unnecessary use of relaxed semantics causes items with high priority to be ignored by DELMIN, which can cause unnecessary computations and performance degradation in some applications. Finally, in contrast to related work, CA-PQ has two contention avoidance mechanisms that are activated separately: one to avoid contention in DELMIN operations and one to avoid contention in INSERT operations.

Using a parallel program that computes the single source shortest paths on a graph, a benchmark which is representative for many best-first search algorithms that use priority queues, we compare CA-PQ's performance with that of other state-of-the-art concurrent priority queues. As we will see, CA-PQ's cache friendly design lets it outperform all other data structures with a significant margin in many scenarios. Furthermore, CA-PQ's adaptivity to contention helps it perform well across a multitude of scenarios without any need to manually tune its parameters.

We start by giving a high-level overview of CA-PQ (Section 2). We then describe its operations in detail (Section 3) and the guarantees that they provide (Section 4). Details of our implementation of the global CA-PQ component appear in Section 5. We then contrast CA-PQ with related work (Section 6), experimentally evaluate CA-PQ variants with other state-of-the-art data structures (Section 7) and conclude (Section 8).

2 A Brief Overview of the Contention Avoiding Priority Queue

As illustrated in Fig. 1, the CA-PQ has a global component and thread local components. When a CA-PQ is uncontended it functions as a strict concurrent priority queue. This means that the DELMIN operation removes the smallest item from the global priority queue and the INSERT operation inserts an item into the global priority queue.

Accesses to the global priority queue detect whether there is contention during these accesses. The counters delmin_contention and insert_contention are modified based on detected contention so that the frequency of contention during recent calls can be estimated. If DELMIN operations are frequently contended, contention avoidance

for DELMIN operations is activated. If a thread's delmin_buffer and insert_buffer are empty and DELMIN contention avoidance is turned on, then the DELMIN operation will grab up to k smallest items from the head of the global priority queue and place them in the thread's delmin_buffer. Grabbing a number of items from the head of the global priority queue can be done efficiently if the queue is implemented with a "fat" skip list that can store multiple items per node; see Fig. 1. Thus, activating contention avoidance for DELMIN operations reduces the contention on the head of the global priority queue by reducing the number of accesses by up to $k-1$ per k DELMIN operations.

Contention avoidance for INSERT operations is activated for a particular thread when contention during INSERT operations is frequent for that thread.The INSERT contention avoidance reduces the number of inserts to the global priority queue by buffering items from a bounded number of consecutive INSERT operations in the insert_buffer. When at least one of the delmin_buffer and insert_buffer is non-empty, the DELMIN operation takes the smallest item from these buffers and returns it.

3 Implementation

We will now give a detailed description of CA-PQ's implementation. First we will describe the implementation of the two operations, INSERT and DELMIN. We will then describe the general requirements for the global priority queue component.

3.1 Operations

The INSERT operation. Pseudocode for this operation can be seen in Algorithm 1. Items are inserted in the global priority queue (line 3) when contention is low or when the number of items in the thread-local insert_buffer equals its capacity. By initially setting the buffer's capacity to zero and setting it to a non-zero value when INSERT operations frequently observe contention, these two tests are folded into one; cf. line 2.

Algorithm 1: The INSERT operation

```
1 Function INSERT (pq, item)
2     if pq.local.insert_buffer.size == pq.local.insert_buffer.capacity then
3         contended = GINSERT(pq.global_pq, item);
4         if contended then pq.local.insert_contention += INS_CONT ;
5         else pq.local.insert_contention -= INS_UNCONT ;
6     else
7         INSERTBUFFERINSERT(pq.local.insert_buffer, item);
8     end
```

The INSERT operation on the global priority queue, called GINSERT, returns true if it observed contention during the operation and false otherwise. To estimate the contention level for INSERT operations in the priority queue, the thread local counter insert_contention is incremented by INS_CONT if contention was detected and is decremented by INS_UNCONT if no contention was detected (lines 4–5). In our implementation, INS_CONT is equal to two and INS_UNCONT is equal to one. As we will soon see, these values ensure that adaptation to contention in INSERT operations will eventually happen if more than one out of two INSERT operations are contended for a sufficiently long period of time. Finally, if the thread local insert_buffer has a size that is less than its capacity, the item is inserted into the insert_buffer (line 7).

Algorithm 2: The DELMIN operation

```
1  Function DelMin (pq, item)
2      switch SelectBufferWithSmallestKey(pq.local.delmin_buffer, pq.local.insert_buffer) do
3          case pq.local.delmin_buffer do
4              return DelMinBufferDelMin(pq.local.delmin_buffer);
5          case pq.local.insert_buffer do
6              pq.local.insert_buffer.capacity -= 1;
7              return InsertBufferDelMin(pq.local.insert_buffer);
8          otherwise do
9              contended, ret_val = GDelMin(pq.global_pq);
10             if contended then pq.local.delmin_contention += DELMIN_CONT ;
11             else pq.local.delmin_contention -= DELMIN_UNCONT ;
12             if pq.local.delmin_contention > DELMIN_RELAX_LIMIT then
13                 TurnOnDelMinRelaxation( pq.global_pq);
14                 pq.local.delmin_contention = 0;
15             else if pq.local.delmin_contention < DELMIN_UNRELAX_LIMIT then
16                 TurnOffDelMinRelaxation( pq.global_pq);
17                 pq.local.delmin_contention = 0;
18             end
19             if pq.local.insert_contention > INS_RELAX_LIMIT then
20                 pq.local.insert_buffer.max_size = MAX_INSERT_BUFF_SIZE;
21                 pq.local.insert_contention = 0;
22             else if pq.local.insert_contention < INS_UNRELAX_LIMIT then
23                 if pq.local.insert_buffer.max_size > 0 then
24                     pq.local.insert_buffer.max_size -=1;
25                 pq.local.insert_contention = 0;
26             end
27             pq.local.insert_buffer.capacity = pq.local.insert_buffer.max_size;
28             if ret_val is a buffer then
29                 pq.local.delmin_buffer = ret_val;
30                 return DelMinBufferDelMin(pq.local.delmin_buffer);
31             else return ret_val ;
32         end
33     end
```

The DELMIN operation. Pseudocode for this operation is displayed in Algorithm 2. If at least one of the thread local buffers is non-empty, the operation removes the smallest item from these buffers (lines 4 and 7). If an item is removed from the insert_buffer, the buffer's capacity is also decreased by one (line 6). This is done to ensure that DELMIN will fetch the minimum item from the global priority queue at least once in a given number of DELMIN operations performed by a particular thread.

If both buffers are empty, the GDELMIN operation is called on the global priority queue (line 9). This operation also returns an indication whether contention was detected during the operation in addition to the removed minimum item (if contention avoidance is turned off) or a buffer with the removed minimum items (if contention avoidance is turned on). (If the global priority queue is empty a special empty_pq item is returned.) After the call to GDELMIN, we record the contention by adjusting the delmin_contention variable (lines 10–11) in a similar way as was done for the insert_contention variable in the INSERT operation. In our implementation, the constants DELMIN_CONT and DELMIN_UNCONT are set to 250 and 1 respectively. These values ensure that adaptation to contention in DELMIN operations will happen if more than one out of 250 DELMIN operations are contended during a long period of time.

We then proceed to check if delmin_contention has reached one of the thresholds for turning on or off contention avoidance on the global priority queue (lines 12–17). The thresholds called DELMIN_RELAX_LIMIT and DELMIN_UNRELAX_LIMIT in

the pseudocode are in our implementation set to 1000 and -1000 respectively. Calling TURNONDELMINRELAXATION on the global priority queue will cause subsequent GDELMIN calls to delete up to k smallest items from the global priority queue and return these items in a buffer. Doing the reverse call, TURNOFFDELMINRELAXATION will cause subsequent GDELMIN calls to only remove and return the smallest item.

We then go on to check if one of the thresholds for changing the contention avoidance for INSERT operations has been reached (lines 19–25). In our implementation, the constants INS_RELAX_LIMIT and INS_UNRELAX_LIMIT are set to 100 and -100 respectively. Adapting to high contention for INSERT operations is done by setting the max_size value of the insert buffer to the constant MAX_INSERT_BUFF_SIZE (500 in our implementation) on line 20. When INSERT operations experience low contention we decrease max_size of the insert_buffer by one (line 24). We set the capacity of the insert_buffer to the max_size value of the insert_buffer on line 27.

Note that adaptation to contention in INSERT operations is done by only doing thread-local modification while adaptation to contention in DELMIN operations is done by changing the state of the global component. One could also implement DELMIN contention avoidance 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. We expect this alternative design choice to work equally well.

At the end of DELMIN's code, we check if the value returned by GDELMIN is a buffer of items or a single item (line 28). If the value is a buffer, we set it to be the thread local delmin_buffer and return an item from that buffer. Otherwise, if it is a single item, we simply return that item (line 31).

3.2 Global Concurrent Priority Queue Component

The requirements for the global priority queue are as follows. First, it should support linearizable INSERT and DELMIN operations. Second, it should also support a linearizable bulk DELMIN operation that returns up to the k smallest items from the priority queue in a buffer. Furthermore, all these operations need to be able to detect contention so as the contention avoidance mechanisms are activated. With these properties fulfilled, it is easy to see that the interface used for the global priority queue in Algorithm 1 and 2 can be implemented. The ability to turn off and on DELMIN relaxation can be implemented by associating a flag with the global priority queue. The GDELMIN operation simply needs to check this flag and use the bulk DELMIN functionality to return a buffer of items if the flag is on, or use the single-item DELMIN functionality to return a single item otherwise.

For the DELMIN contention avoidance to work as intended, it is crucial that the bulk DELMIN operations can remove and return the k smallest items much faster than doing k single-item DELMIN operations. To make this possible, our implementation of the global concurrent priority queue makes use of a skip list data structure with fat nodes; see Fig. 1. As every skip list node in our implementation can store up to k items, the bulk DELMIN operation can remove and return up to k smallest items with as little work as the single-item DELMIN operation needs to do in the worst case. A k value that is equal to or greater than the number of threads should be enough to eliminate most of the contention in DELMIN. Our implementation uses 80 as the value of k.

4 Properties

We will now state the guarantees provided by the CA-PQ. As some applications might not need the contention avoidance for both INSERT and DELMIN, we will first state and prove the guarantees of the CA-PQ variants derived by turning these features off.

First note that turning off the contention avoidance for both INSERT and DELMIN results in a strict priority queue. We call the data structure that results from turning off contention avoidance for INSERT operations CA-DM. To state the guarantee provided by CA-DM we first have to define a particular time period.

Definition 1. ***(Time period*** $\mathbf{TP(k, D_n)}$***)*** *Let an integer $k \geq 1$, $D_1, \ldots, D_n$ be the sequence of* DELMIN *calls performed by a thread T on a priority queue Q, and let $j = max(1, n - k + 1)$. Then $TP(k, D_n)$ is the time period that starts at the time D_j is issued and ends when the call D_n returns.*

We can now state and prove the guarantee that the CA-DM priority queue provides.

Theorem 1. ***(CA-DM DELMIN Guarantee)*** *The item returned by a* DELMIN *call D on a CA-DM priority queue Q is guaranteed to be among the $k \cdot P$ smallest items that have been inserted into the priority queue at some point in time t during the time period $TP(k, D)$, where P is the number of threads that are accessing Q and k is the maximum size of the buffer returned by the global priority queue that is used by Q.*

Proof: Let t be the linearization point of the latest GDELMIN call G (Algorithm 2, line 9) performed by the issuer of D before D's return. Note that t must then be in the time period $TP(k, D)$ as the number of items in the delmin_buffer decreases by one in every DELMIN call that does not get its item directly from the global priority queue. All items in the buffer returned by the call G are among the $k \cdot P$ smallest items in Q at the time of G's linearization point. To see this, note that no items in the global priority queue were smaller than the at most k items returned by G at G's linearization point and no more than $(P - 1) \cdot k$ items can be buffered in the delmin_buffers of other threads. □

We call the priority queue derived from CA-PQ by turning off contention avoidance for DELMIN CA-IN. The guarantee provided by CA-IN is arguably even weaker than that provided by CA-DM.

Theorem 2. ***(CA-IN DELMIN Guarantee)*** *At least one in every $m+1$* DELMIN *operations performed by a thread is guaranteed to be among the $m \cdot (P-1)+1$ smallest items in the CA-IN priority queue Q at some point in time during the operation's execution, where m is equal to MAX_INSERT_BUFF_SIZE and P is the number of threads that are accessing Q.*

Proof: At least one call D in every $m + 1$ DELMIN calls returns an item I from a GDELMIN call G since the capacity of the insert_buffer is decreased when items are removed from it (Algorithm 2, line 6). This item I must be among the $m \cdot (P - 1) + 1$ smallest items in the priority queue at the linearization point of G since there can be at most $m \cdot (P - 1)$ smaller items in the insert_buffers of other threads. □

The guarantee provided by a CA-PQ that has both contention avoidance for DELMIN and INSERT operations turned on is very similar to that of CA-IN.

Theorem 3. ***(CA-PQ* DELMIN *Guarantee)*** *At least one in every $m+1$* DELMIN *operations performed by a thread is guaranteed to be among the $m \cdot (P-1)+1$ smallest items in the CA-PQ priority queue Q at some point in time during the operation's execution, where m is equal to k+MAX_INSERT_BUFF_SIZE, k is the maximum size of the buffer returned by* GDELMIN*, and P is the number of threads that are accessing Q.*

Proof: The proof is very similar to the proof of Theorem 2. The difference is that there is now also the delmin_buffer so that m becomes slightly larger. □

All priority queue variants mentioned above also support the property specified in the theorem below which is important for the termination of many parallel algorithms that employ concurrent priority queues.

Theorem 4. ***(DELMIN Deletes All)*** *Let S be the set of all threads that have issued operations on a priority queue Q and t be a specific point in time after which no* INSERT *operations are issued. If all threads in S issue a* DELMIN *operation after time t and all get the special item empty_pq as results, then all items that have been inserted into Q have been deleted and returned by* DELMIN *operations.*

Proof: An item that is inserted into Q and has not yet been deleted is stored in the global priority queue or in one of the thread-local buffers of threads in S. It is easy to see that all these locations must be empty if all threads in S issue DELMIN operations after t and get the empty_pq symbol as return value. □

5 Our Implementation of the Global Priority Queue Component

Our global concurrent priority queue is constructed from a contention adapting search tree (CATree) [14] using a skip list with fat nodes as backing data structure. We refer to the original CATree paper for a complete description of the CATree data structure and will here just briefly describe how we extended it to support the DELMIN operations. Figure 2 shows the structure of a CATree. The routing nodes are used to find the location of a specific item in the data structure. The actual items stored in the data structure are located in the sequential data structure instances in the last layer. These sequential data structures are protected by locks in the base nodes where they are rooted. Base nodes can be split and joined with each other based on how much contention is detected in the base node locks. As the smallest items in a CATree are always located in the leftmost part of the tree when depicted as in Fig. 2, the DELMIN operation first finds and locks the leftmost base node in the CATree. When the leftmost base node is empty it is joined together with its neighbor using the CATree algorithm for low contention adaptation until the leftmost base node is non-empty[1]. As depicted in Fig. 1, we reuse the fat skip list nodes as delmin_buffer and use a binary heap as insert_buffer.

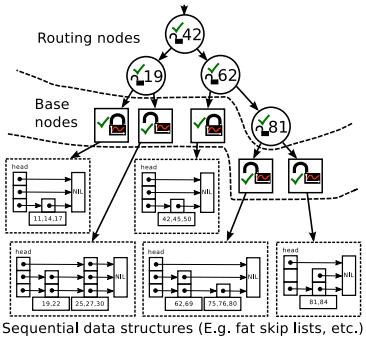


Fig. 2: The CATree data structure.

[1] The only difference between the low-contention join function described in the CATree paper [14] and the one used to create a non-empty leftmost base node is that the latter uses

Traditional locks are well known to give poor performance when they are contended [3, 6, 9]. Therefore, to improve the performance when base node locks in the CATree are contended we use a locking technique that we call delegation locking but that is also called combining in other places [3, 6]. More specifically we use a delegation locking technique, called queue delegation locking [9], when locking base nodes. Delegation locking lets the current lock owner thread help other threads perform their critical sections that are waiting to acquire the lock. By doing so the throughput of critical sections executed on a particular lock can be substantially increased because the current lock owner can keep the data protected by the lock in its private processor cache while helping critical sections from other threads. Queue delegation locking has the additional benefit compared to other locking algorithms that critical sections for which the issuing threads do not need any return value (such as the INSERT operation) can be delegated to the lock owner without any need to wait for the actual execution of the critical section. Linearizability is still provided as the order of the delegated operation is maintained by a queue. Contention in the operations is detected by checking whether another thread is holding the base node lock that the operation needs to acquire.

Memory management. The only nodes of the data structure that need delayed memory reclamation in our CA-PQ implementation are the routing nodes and base nodes in the CATree component. These nodes can be read by multiple threads concurrently so it is unsafe to reclaim these nodes before it is certain that no threads can hold references to them. To reclaim these nodes we use Keir Fraser's epoch based reclamation [4].

6 Related Work

Early attempts to construct concurrent priority queues, e.g [7], were based on heap data structures. More recent concurrent priority queues have often been based on concurrent skip lists as empirical evidence suggests that this design is more scalable than the heap based design [16]. Both the priority queue by Shavit and Lotan [16] and the one by Sundell and Tsigas [17] handle DELMIN by first doing a logical deletion of the node to be deleted by marking it before it is physically removed from the skip list. The skip list based priority queue by Lindén and Jonsson [12] (called **Lindén** from here on) also uses logical deletion before physical removal but achieves better performance and less memory contention by physically removing a prefix of logically deleted nodes in one go, in contrast to previous algorithms that physically remove one node at a time. Calciu *et al.* have explored the idea of using combining and delegation to speedup the DELMIN operation. Their data structure [2] uses a sequential skip list managed by a server thread for small keys and a concurrent skip list for larger keys to exploit the parallelism of INSERT operations. In a very recent work, Zhang and Dechev have proposed a concurrent priority based on multi-dimensional linked lists [21]. We consider all the above works on concurrent priority queues orthogonal to the main contribution of this paper which is a priority queue with more relaxed semantics.

Concurrent priority queues with relaxed semantics have also been proposed. The **MultiQueue** data structure by Rihani *et al.* [13] is created from $C \cdot P$ sequential priority

a forcing LOCK call instead of a TRYLOCK call to lock the neighbor. (This cannot cause a deadlock since no other code issues forcing lock calls in the other direction.)

queues protected by locks, where C is a constant and P is the number of threads using the priority queue. An INSERT operation in a MultiQueue selects one of the sequential queues at random and inserts in that queue. MultiQueue's DELMIN operation checks the minimum item in two of the sequential priority queues selected at random (without acquiring locks) and does the actual DELMIN in the one of these priority queues with the smallest key if that priority queue is successfully locked with a try-lock call. The process is retried if the try-lock call fails. The MultiQueue does not provide any guarantee, but an experimental evaluation suggests that DELMIN often returns an item with one of the smallest keys in the priority queue [13].

Alistarh *et al.* have created the **SprayList** which is a relaxed priority queue based on the skip list data structure [1]. SprayList relaxes the result of the DELMIN operation by "spraying" into a random position close to the head of the skip list. The SprayList guarantees that the item returned by DELMIN is among the $\mathcal{O}(P \log^3 P)$ smallest items with high probability, where P is the number of threads.

For scheduling purposes in a task-based parallel programming framework, Wimmer *et al.* have created relaxed priority queues that have different trade-offs between quality of the items returned by DELMIN and scalability [20]. Of these, the queue that seems to perform best is called Hybrid k. A later publication, also by Wimmer *et al.*, introduced the k**-LSM** priority queue [19]. k-LSM provides the structural guarantee that no more than $k \cdot P$ items might be skipped by DELMIN, where k is a configurable parameter and P is the number of threads. We will here focus on the k-LSM priority queue rather than Hybrid k because the implementation of the latter is optimized for a particular task-based parallel programming framework, making it difficult to compare with, and experiments by Wimmer *et al.* suggest that k-LSM performs slightly better than Hybrid k [19]. The k-LSM data structure is based on so called log-structured merge-trees (LSM) and consists of a thread local LSM component and a shared relaxed LSM component. INSERT inserts the item to the thread local LSM component. If this results in a block larger than a certain size, that block is merged into the shared LSM. DELMIN compares one of the k smallest items in the shared LSM with the smallest item from the local LSM and tries to remove the smallest of those items.

All the above relaxed priority queues (MultiQueue, SprayList, Hybrid k and k-LSM) utilize relaxations to avoid contention in DELMIN operations. However, in contrast to CA-PQ, they all access non-thread-local memory in every DELMIN operation. As this shared memory is written to by many threads frequently, many of these accesses induce cache misses. This can be expensive as it causes the core executing the thread to wait for data to be transferred from remote locations and causes contention in the memory system. On big multi-cores, especially on NUMA machines with several processor chips, getting data from remote locations can be several orders of magnitude more expensive than getting data from the same processor's cache. There are two reasons why CA-PQ can avoid the frequent remote memory accesses in DELMIN. Firstly, its DELMIN fetches a block containing several items from the global priority queue, i.e., it gets several items for a single cache miss (because several items can be stored on the same cache line). Secondly, the guarantees provided by CA-PQ are more permissive than those provided by SprayList, Hybrid k and k-LSM, which makes it possible to allow CA-PQ's DELMIN to often be performed without checking if other threads have changed the data structure.

Another major difference between CA-PQ and other relaxed priority queues is that CA-PQ only activates relaxations when this is motivated by detected high contention. As we will see in the next section, this makes it possible for CA-PQ to achieve high performance in a wide range of scenarios.

7 Experimental Evaluation

We evaluate the scalability and performance of CA-PQ and the variants CA-IN (INSERT contention avoidance turned off), CA-DM (DELMIN contention avoidance turned off) and CATree (the global priority queue component of our algorithm) in a parallel single-source shortest-path (SSSP) benchmark. The benchmark uses a parallel version of Dijkstra's algorithm using a concurrent priority queue; see Tamir *et al.* [18]. We note that we avoid the node locks used in this parallelization by updating the node weights in *compare-and-swap* loops. CA-PQ does not have a DECREASEKEY operation that changes the key of an item in the priority queue — such is also the case for the other concurrent priority queues that we compare against. Changing the weight of a key in the priority queue is therefore implemented by an INSERT operation and the other reference to the node that might exist in the queue is lazily removed when it is deleted by a DELMIN operation. As noted by Tamir *et al.* [18], this lazy removal scheme can induce some overhead over having a concurrent priority queue with a DECREASEKEY operation. To get a hint of how big this overhead might be, we include the sequential version of Dijkstra's algorithm that uses DECREASEKEY with a Fibonacci Heap [5] as priority queue as a base line. The overhead of not having DECREASEKEY operation seems to be quite low in many cases as the sequential Dijkstra has similar performance as the parallel SSSP algorithm using CA-PQ when using just one thread.

Data sets. We include results from running the SSSP benchmark on the California road network (called RoadNet from now on) and a social media network obtained from LiveJournal (called LiveJournal from now on) [11]. RoadNet is a relatively sparse network containing 1.95 million nodes connected to the source involving 5.5 million edges. LiveJournal is a more dense network containing 4.4 million nodes connected to the source and 68 million edges. As we do not have any natural weights for these networks we used two versions of these networks. A weight of one on all edges is used in the unweighted version. In the weighted version, a random weight from the range $[0, 1000]$ is assigned to each of the edges.

Data structures and parameters. We compare our priority queues to Lindén [12], MultiQueue [13], SprayList [1] and k-LSM [19]. Section 6 contains a description of these data structures. All implementations are those provided by their inventors except the MultiQueue which is implemented by the authors of k-LSM. We use the default parameters for SprayList as configured by its authors because the SprayList was evaluated in a very similar benchmark to ours [1]. To find a good value for the C parameter used by the MultiQueue, we ran the benchmarks with C equal to 2, 4, 8, 16, 32 and 64. We found that the values 8 and 16 gave the best performance and the difference between these two parameters was very small in all cases. We therefore use MultiQueue with $C = 16$. Similarly, to find a good value for the k parameter used by k-LSM we ran the experiments with k equal to 2^n for all integer values of n from 8 to 17. From this, we found that

$k = 2^{10} = 1024$ gave the best performance on RoadNet and that $k = 2^{16} = 65\,536$ generally gave the best performance on LiveJournal. We therefore show k-LSM with both $k = 1024$ (klsm1024) and $k = 65\,536$ (klsm65536).

Methodology. We show results from a machine with four Intel(R) Xeon(R) E5-4650 CPUs (2.70GHz, turbo boost turned off), eight cores each (i.e. the machine has a total of 32 physical cores, each with hyperthreading, which makes a total of 64 logical cores). The machine has 128GB of RAM and is running Linux 3.16.0-4-amd64. We compiled the benchmark which is written in C and C++ with GCC version 5.3.0 and used the optimization flag `-O3`. We have verified our results by running the experiments on a machine with four AMD Opteron 6276 (2.3 GHz, in total 64 cores)[2]. Threads are pinned to logical cores so that the first 16 threads in the graphs run on the first processor chip, the next 16 on the second, and so on. We ran each measurement three times and show the average and error bars for the minimum and maximum in the graphs. As a sanity check we compared the calculated distances against the actual distances after each run.

Results. The results from the SSSP benchmark are displayed in Fig. 3. The graphs show throughput $N \div T$ on the y-axis, where N is the number of nodes in the graph and T is the execution time of the benchmark in μs. We show throughput rather than time because this makes the scalability behavior easier to see. (The poor performance of some data structures would otherwise make the results unreadable.) The dashed black line shows the performance of the sequential Dijkstra's algorithm with a Fibonacci heap. The red line with legend Lock shows the performance of a binary heap protected by a lock.

RoadNet. Let us first look at the results for the RoadNet graphs shown in Fig. 3a and 3b. With RoadNet, none of the data structures manages to provide much increase in performance when more than one processor chip is utilized (after 16 threads). However, in the scenario with edge weight range $[0, 1000]$, CA-PQ archives a speedup of 11 compared to its single thread performance when running on 16 threads (remember that these 16 threads run on 8 cores with hyperthreading). It is clear from the worse performance of CA-DM (INSERT contention avoidance turned off) and CA-IN (DELMIN contention avoidance turned off) that both contention avoidance mechanisms are beneficial to achieving this performance in the relatively sparse RoadNet graph that gives high contention both in INSERT and DELMIN operations. The data structure that achieves the second best performance after CA-PQ in these scenarios is klsm1024. It is interesting to note that klsm1024 also buffers inserted items in a thread local storage.

To investigate the reason for the performance further, we show number of L2 cache misses (measured with hardware counters) divided by the number of nodes in the graph in Table 1. As the L2 cache is private to a core on this processor, more L2 cache misses is an indication of worse memory locality and more accesses to memory modified by several thread. Unsurprisingly, CA-PQ has the least amount of L2 cache misses in the RoadNet scenarios due to its cache friendly design.

In the sequential version of Dijkstra's algorithm each node is processed exactly once. In the parallel version, this is not always the case as the node with the smallest distance estimate is not always processed first. We can therefore use the number of nodes processed by the parallel algorithm as a measurement of how precise the DELMIN

[2] Results from the AMD machine and from additional scenarios as well as the benchmark code are available at http://www.it.uu.se/research/group/languages/software/ca_pq.

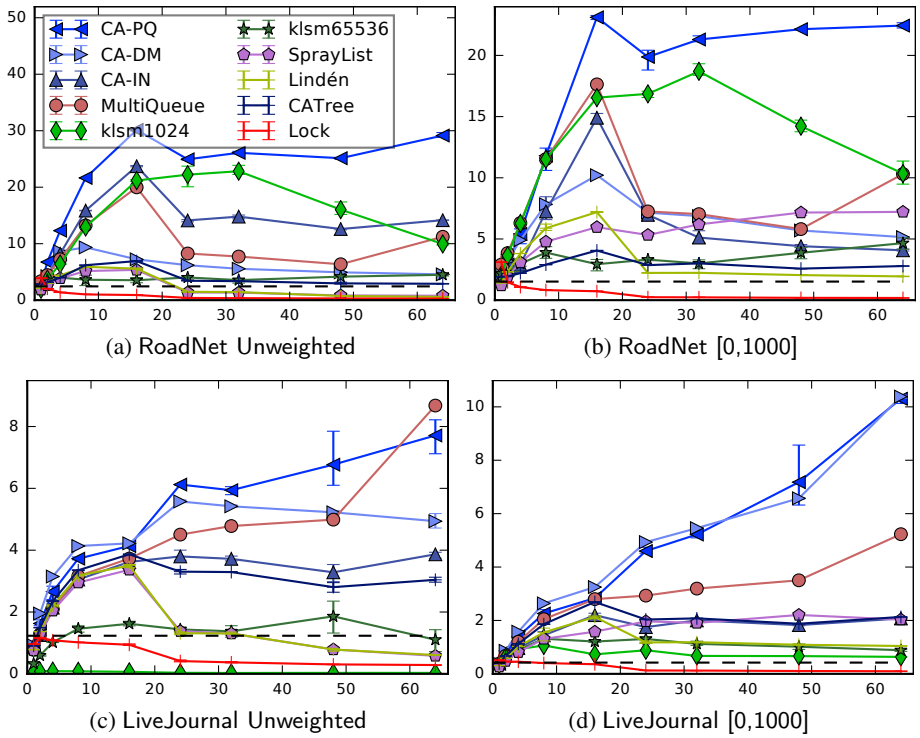


Fig. 3: Graphs showing results from the SSSP experiment. Throughput (# nodes in graph ÷ execution time (μs)) on the y-axis and number of threads on the x-axis. The black dashed line is the performance of sequential Dijkstra's algorithm with a Fibonacci Heap.

operation is (how far from the actual minimum the returned items are). In the column "waste" of Table 1 we show the number of nodes processed minus the number of nodes in the graph. We see that the strict priority queues CATree, Lindén and Lock all do a small amount of wasted work in both the unweighted and the weighted scenarios. CA-PQ, CA-IN and the k-LSMs all waste quite a lot of work considering that RoadNet only has 1.95 million nodes. However, as the contention on the priority queue is high in this scenario it can be less wasteful for the priority queue to be less precise in order to reduce the contention inside the priority queue. As CA-PQ only activates the relaxed semantics when high contention is detected, one can see it as opportunistic in the sense that it lowers precision and risks more wasted work in the application only when time and resources would be wasted anyway due to contention.

The MultiQueue achieves very good precision according to the waste estimate but as each operation accesses at least one of the shared priority queues, it suffers from bad memory locality; see Table 1. Since communication between processor chips is more expensive than communication within the chip, the bad memory locality of MultiQueue becomes apparent first when more than one NUMA node is utilized; see Fig. 3a.

LiveJournal. We now go on to discuss the results from the graph LiveJournal that can be seen in Fig. 3c and 3d. As the LiveJournal graph is relatively dense there will

Table 1: **Waste and cache misses (64 threads).** The column *time* shows execution time in seconds, *waste* shows the number of nodes unnecessarily processed and the column *$miss* shows number of L2 cache misses divided by number of nodes in the graph.

Graph	**RoadNet**						**LiveJournal**					
Weights	**1**			**[0,1000]**			**1**			**[0,1000]**		
	time	**waste**	**$miss**	**time**	**waste**	**$miss**	**time**	**waste**	**$miss**	**time**	**waste**	**$miss**
CA-PQ	0.07	1730k	7.8	0.09	1927k	12.2	0.63	924k	30.1	0.47	353k	95.4
CA-RM	0.43	7k	14.8	0.38	11k	34.6	0.98	8	32.2	0.47	2k	94.1
CA-IN	0.14	2264k	8.2	0.48	2030k	27.3	1.25	1768k	37.0	2.34	714k	110.5
MultiQ.	0.18	8k	32.2	0.19	58k	36.1	0.56	39	63.4	0.93	2k	112.2
kl.1024	0.20	2498k	12.4	0.19	2222k	15.8	161.39	174	33980.3	7.63	3k	2538.5
kl.65536	0.44	28411k	82.5	0.42	26115k	105.6	4.76	688k	601.7	5.48	1857k	1192.7
Spray	2.51	134k	461.0	0.27	230k	88.3	8.33	41	314.9	2.39	7k	755.5
CATree	0.68	9	20.9	0.71	36	40.2	1.59	1	40.8	2.27	5	107.5
Lindén	3.39	206	108.4	1.01	252	114.6	7.96	21	142.6	4.64	0	353.1
Lock	7.06	210	39.7	11.02	490	59.0	17.01	54	62.4	49.73	86	163.4

be many priority queue items with the same distance (key) while running the parallel SSSP. This is especially true in the unweighted case (Fig. 3c). This can lead to a lot of contention in INSERT operations as the skip list based data structures (CA-*, SprayList, Lindén and CATree) all try to insert an item with the same distance in the same location. The MultiQueue however is excellent in avoiding contention and achieves the best performance in the unweighted LiveJournal (Fig. 3c). However, MultiQueue is tightly followed by CA-PQ as CA-PQ is also good at avoiding contention with its contention avoidance mechanisms and has good memory locality; see Table 1.

In the weighted LiveJournal scenario (Fig. 3d), where the contention in INSERT operations is not as high as in the unweighted case, CA-PQ and CA-DM are by far outperforming the other data structures. Some hints about the reason for this is given in Table 1: one can see that CA-PQ and CA-DM induces less L2 cache misses than the other data structures. However, we want to stress that the number of L2 cache misses is a course-grained measurement of memory locality. The cost of cache misses can differ depending on whether it is a read miss or write miss and whether the miss causes communication outside the chip or not.

From Table 1, we see that CA-DM generally does relatively little wasted work while CA-PQ is more wasteful which is natural as CA-PQ provides weaker guarantees than those provided by CA-DM. This also explains why CA-DM performs better than CA-PQ by a very small amount for most thread counts in the weighted LiveJournal scenario.

A note on denser graphs. We have also run experiments on randomly generated graphs that are more dense than the graphs used in the experiments we just presented. (Refer to `http://www.it.uu.se/research/group/languages/software/ca_pq` for the results of these experiments.) Dense graphs tend to give an access pattern on the concurrent priority queue with many more INSERT operations than DELMIN in the beginning of the run and then many more DELMIN than INSERT in the end of the run. CA-PQ is efficient in these kinds of scenarios because of its cache friendly DELMIN operation. For example, CA-PQ's execution time on a graph with 100 edges per node and

edge weights from the range $[0, 1000]$ is only about one third of the execution time of the second best data structure in this scenario (SprayList). The access pattern produced by denser graphs also explains why k-LSM performs badly with the LiveJournal graphs. When DELMIN operations are frequent and INSERT's are less frequent, most DELMIN calls will take items from the shared LSM, which induces contention and cache misses.

Usefulness of adaptivity. To investigate the usefulness of adaptively turning on the contention avoidance techniques we have run experiments where contention avoidance for both INSERT and DELMIN are always turned on (not shown in graphs to not clutter them). We found the performance of this non-adaptive approach to be similar to CA-PQ in scenarios where INSERT contention is high, but significantly worse in scenarios with low INSERT contention (e.g. LiveJournal weight range $[0, 1000]$). Thus, CA-PQ's ability to adaptively turn off and on the contention avoidance techniques is beneficial because it helps it perform well in a multitude of scenarios without any need to change parameters.

The global component. Finally, we comment on the performance of the strict priority queue that we developed as the global component of CA-PQ which is called CATree in Fig. 3 and Table 1. CATree beats the state-of-the-art lock-free linearizable priority queue by Lindén by a substantial amount in several of the scenarios and especially when more than one NUMA node is used. We attribute this good performance to the good memory locality provided by delegation locking and the fact that we use fat skip list nodes which increase locality and reduce the number of memory allocations.

A note on thread preemption. In our benchmark setup, thread preemption is uncommon since we use one hardware thread per worker thread. In setups where threads often get preempted or stalled for some reason, CA-PQ's buffering of items can be problematic, as small items can be stuck for a long period of time in the buffers of these threads. It remains as future work to investigate solutions for this problem, perhaps using a stealing technique similar to the one proposed by Wimmer *et al.* [19].

8 Concluding Remarks

We have introduced the CA-PQ concurrent priority queue that activates relaxed semantics only when resources would otherwise be wasted on contention related overheads and on waiting. CA-PQ has a cache friendly design and avoids accesses to memory that is written to by many threads when its contention avoidance mechanisms are activated, which contributes to its performance advantage compared to related relaxed data structures.

It would be interesting to investigate other strategies for adapting the relaxation. For example, one can experiment with a more fine grained adjustment of the relaxation than what is done in CA-PQ or consider relaxation based on feedback about wasted work from the application. However, the investigation of such strategies is left for future work.

References

1. D. Alistarh, J. Kopinsky, J. Li, and N. Shavit. The spraylist: A scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '15, pages 11–20, New York, NY, USA, 2015. ACM.

2. I. Calciu, H. Mendes, and M. Herlihy. The adaptive priority queue with elimination and combining. In F. Kuhn, editor, *Distributed Computing: 28th International Symposium, DISC 2014. Proceedings*, pages 406–420, Berlin, Heidelberg, 2014. Springer.
3. P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, New York, NY, USA, 2012. ACM.
4. K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge Computer Laboratory, 2004.
5. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.
6. D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.
7. G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott. An efficient algorithm for concurrent priority queue heaps. *Information Processing Letters*, 60(3):151–157, 1996.
8. R. M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *J. ACM*, 40(3):765–789, July 1993.
9. D. Klaftenegger, K. Sagonas, and K. Winblad. Delegation locking libraries for improved performance of multithreaded programs. In *Euro-Par 2014 Parallel Processing*, volume 8632 of *LNCS*, pages 572–583. Springer, 2014.
10. V. Kumar, K. Ramesh, and V. N. Rao. Parallel best-first search of state-space graphs: A summary of results. In *AAAI*, volume 88, pages 122–127, 1988.
11. J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2016.
12. J. Lindén and B. Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In R. Baldoni, N. Nisse, and M. Steen, editors, *Principles of Distributed Systems: 17th International Conference, OPODIS 2013. Proceedings*, pages 206–220. Springer, 2013.
13. H. Rihani, P. Sanders, and R. Dementiev. Brief announcement: Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 80–82, New York, NY, USA, 2015. ACM.
14. K. Sagonas and K. Winblad. Contention adapting search trees. In *14th International Symposium on Parallel and Distributed Computing*, ISPDC, pages 215–224. IEEE, 2015.
15. P. Sanders. Randomized priority queues for fast parallel access. *Journal of Parallel and Distributed Computing*, 49(1):86–97, 1998.
16. N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *Parallel and Distributed Processing Symposium, 2000. Proceedings. 14th International*, pages 263–268, 2000.
17. H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. 17th International Symposium*, page 84, Apr. 2003.
18. O. Tamir, A. Morrison, and N. Rinetzky. A heap-based concurrent priority queue with mutable priorities for faster parallel algorithms. In *Principles of Distributed Systems: 19th International Conference, OPODIS 2015. Proceedings*, 2015.
19. M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas. The lock-free k-LSM relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '15, pages 277–278, New York, NY, USA, 2015. ACM.
20. M. Wimmer, F. Versaci, J. L. Träff, D. Cederman, and P. Tsigas. Data structures for task-based priority scheduling. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 379–380, New York, NY, USA, 2014. ACM.
21. D. Zhang and D. Dechev. A lock-free priority queue design based on multi-dimensional linked lists. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):613–626, 2016.

A Additional Results

This appendix contains results from an additional machine and two more graphs. The benchmark set up is the same as described in Section 7. The machines that we used are the following:

Sandy This is the same machine that we show results for in Section 7. It has four Intel(R) Xeon(R) E5-4650 CPUs (2.70GHz, turbo boost turned off), eight cores each (i.e. the machine has a total of 32 physical cores, each with hyperthreading, which makes a total of 64 logical cores) and 128GB of RAM. Sandy is running Linux 3.16.0-4-amd64 and we compiled the benchmark which is written in C and C++ using GCC version 5.3.0 and the optimization flag `-O3`.

Bulldozer This machine has four AMD Opteron 6276 CPUs (2.3GHz, in total 64 cores), 128GB of RAM and is running the same version of Linux as Sandy (3.16.0-4-amd64). On this machine the benchmark was compiled with GCC 4.9.2 and the optimization flag `-O3`.

In addition to graph instances with a weight of one on all edges (called unweighted) and the one with weights randomly taken from the interval $[0, 1000]$, we here also present results for graph instances with edge weights randomly taken from the range $[0, 1000000]$. The results for the two graphs described in Section 7 can be seen in Fig. 4 and 5. The results for Sandy are displayed on the left and the results for Bulldozer are displayed on the right for easy comparison.

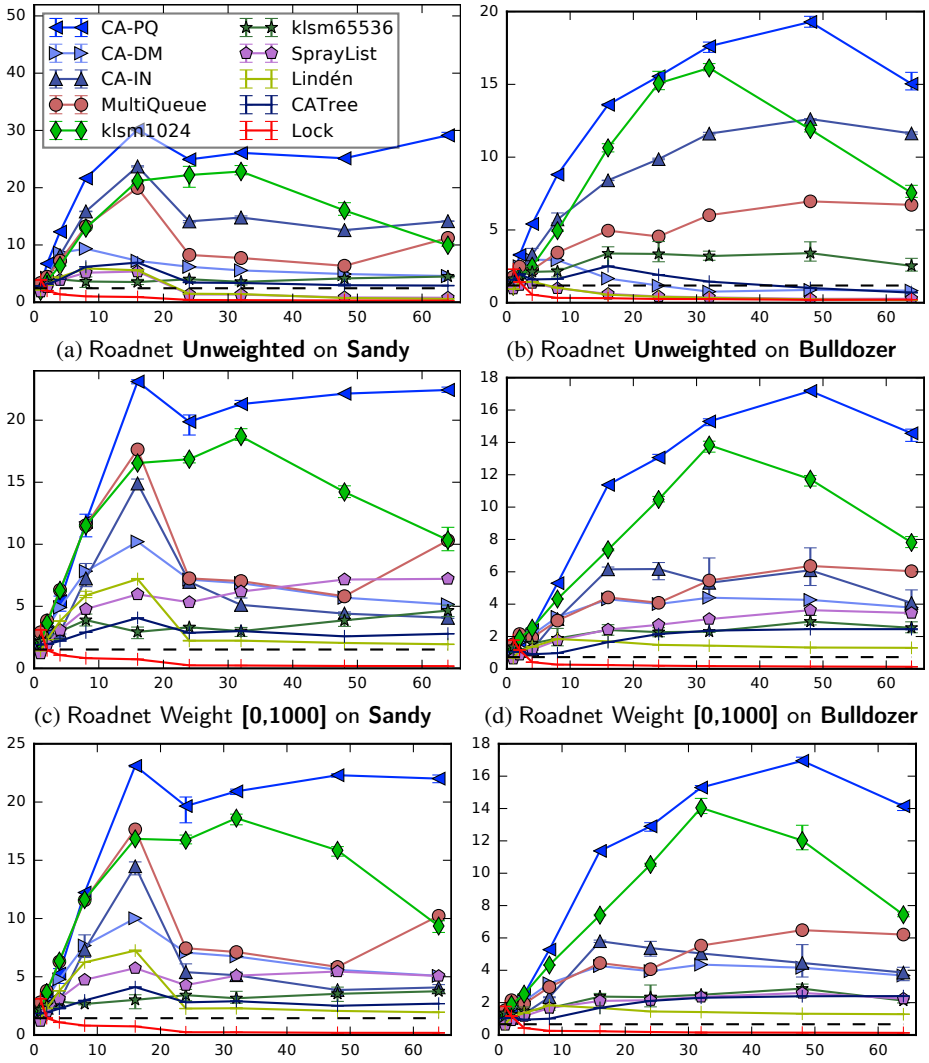
In addition to the realistic graphs RoadNet and LiveJournal, we here also present results for more dense randomly generated Erdős-Rényi graphs. The first of them has $1\,000\,000$ nodes and edge probability 0.0001 and the second one has $10\,000$ nodes and edge probability 0.5. Properties for all graphs are presented in Table 2. The results for the two randomly generated graphs are shown in Fig. 6 and 7.

L2 cache miss information and information about wasted work on the different graphs when running on the Sandy machine appears in Tables 3, 4, 5 and 6.

One can see that the performance of the sequential implementation becomes better compared to the parallel implementations the denser the graphs become. This is what one can expect as denser graphs lead to more node relaxations, so the benefits of having a priority queue with a DECREASEKEY operation become more apparent in denser graphs.

Table 2: # Nodes denotes the number of nodes that are reachable from the source and # Edges is the number of edges in the graph component involving these nodes.

Graph	**# Nodes**	**# Edges**	**# Edges ÷ # Nodes**
RoadNet	1950461	5502114	∼2.8
LiveJournal	4400347	68175771	∼15.5
$N = 10^6\ P = 10^{-4}$	1000000	100001086	∼100.0
$N = 10^4\ P = 0.5$	10000	49988838	∼4998.9



(a) Roadnet **Unweighted** on **Sandy**

(b) Roadnet **Unweighted** on **Bulldozer**

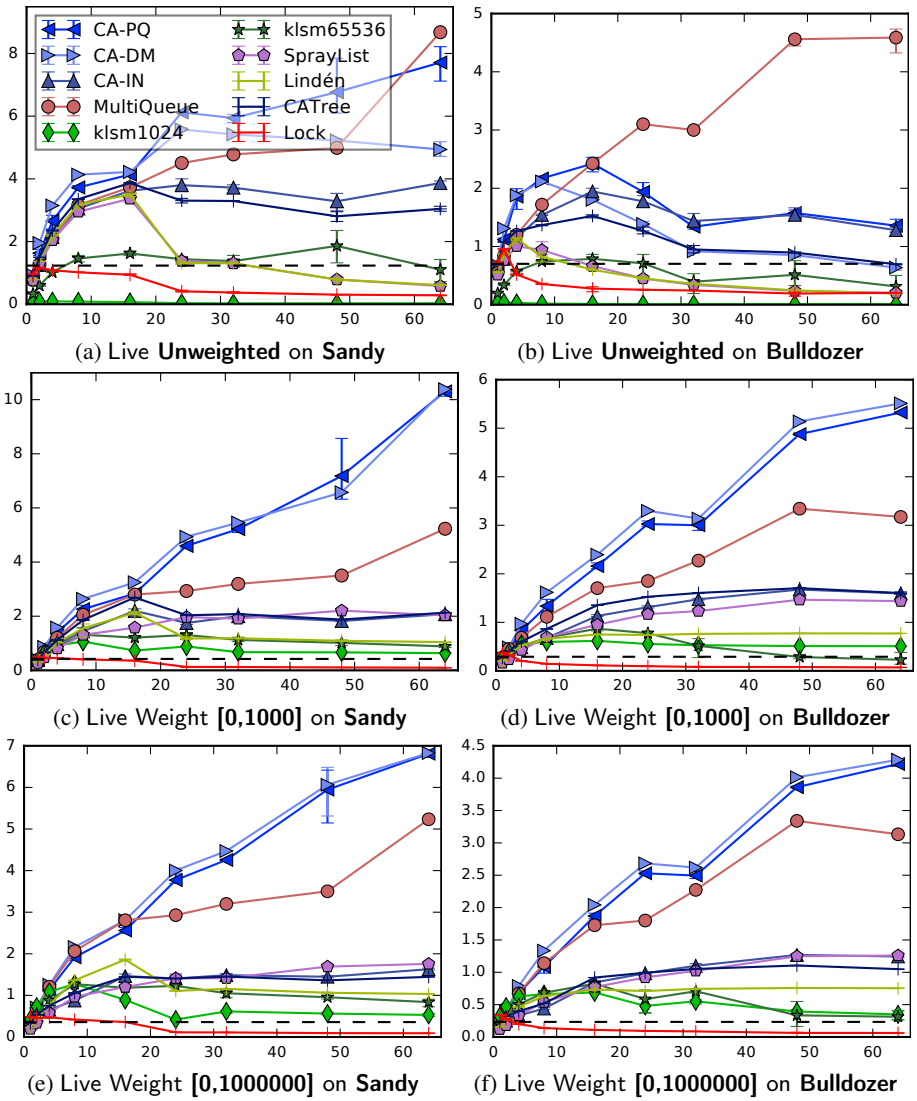
(c) Roadnet Weight **[0,1000]** on **Sandy**

(d) Roadnet Weight **[0,1000]** on **Bulldozer**

(e) Roadnet Weight **[0,1000000]** on **Sandy**

(f) Roadnet Weight **[0,1000000]** on **Bulldozer**

Fig. 4: Results from the **RoadNet** graph. Throughput (# nodes in graph ÷ execution time (μs)) on the y-axis and number of threads on the x-axis. The black dashed line is the performance of the sequential Dijkstra's algorithm with a Fibonacci Heap.



(a) Live **Unweighted** on **Sandy**

(b) Live **Unweighted** on **Bulldozer**

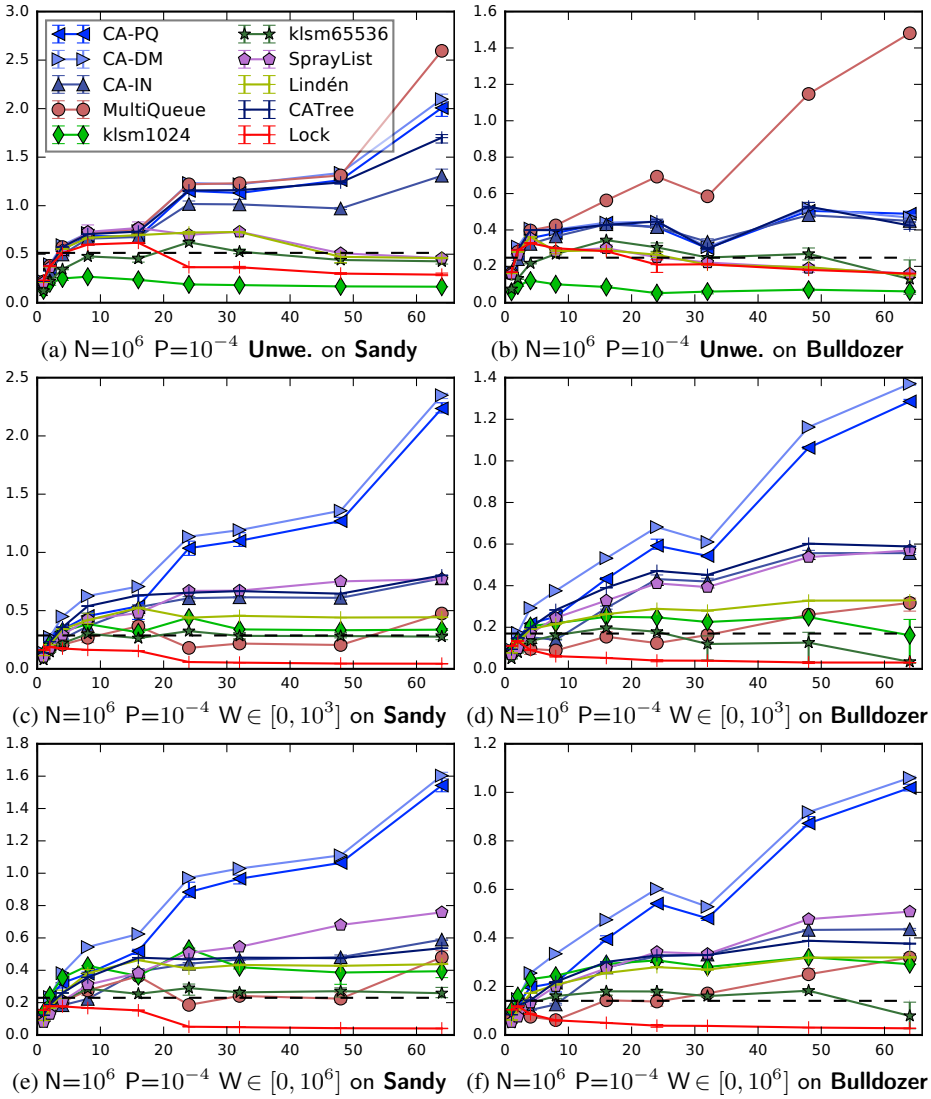
(c) Live Weight **[0,1000]** on **Sandy**

(d) Live Weight **[0,1000]** on **Bulldozer**

(e) Live Weight **[0,1000000]** on **Sandy**

(f) Live Weight **[0,1000000]** on **Bulldozer**

Fig. 5: Results from the **LiveJournal** graph. Throughput (# nodes in graph ÷ execution time (μs)) on the y-axis and number of threads on the x-axis. The black dashed line is the performance of the sequential Dijkstra's algorithm with a Fibonacci Heap.



(a) N=10^6 P=10^{-4} **Unwe.** on **Sandy**

(b) N=10^6 P=10^{-4} **Unwe.** on **Bulldozer**

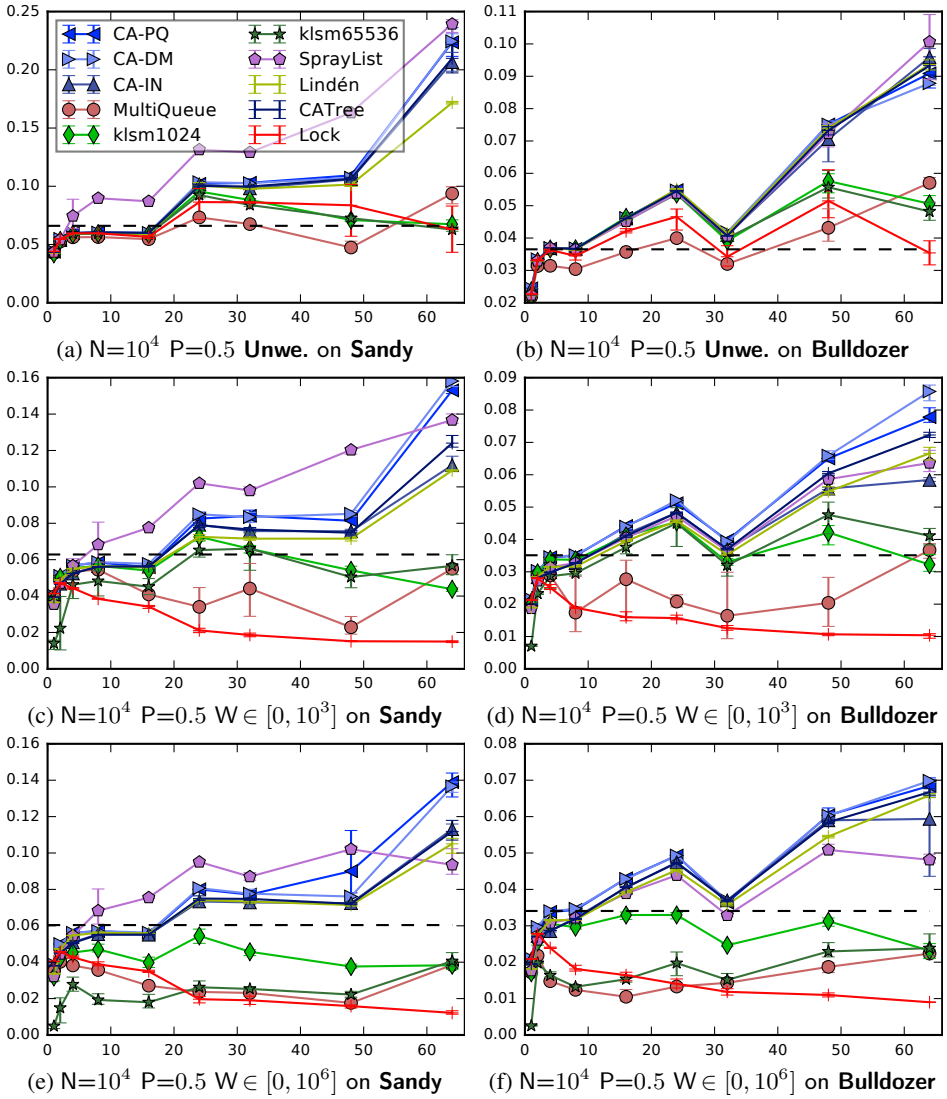
(c) N=10^6 P=10^{-4} W$\in [0, 10^3]$ on **Sandy**

(d) N=10^6 P=10^{-4} W$\in [0, 10^3]$ on **Bulldozer**

(e) N=10^6 P=10^{-4} W$\in [0, 10^6]$ on **Sandy**

(f) N=10^6 P=10^{-4} W$\in [0, 10^6]$ on **Bulldozer**

Fig. 6: Results from a randomly generated Erdős-Rényi graph with N=10^6 P=10^{-4}. Throughput (# nodes in graph ÷ execution time (μs)) on the y-axis and number of threads on the x-axis. The black dashed line is the performance of the sequential Dijkstra's algorithm with a Fibonacci Heap.



(a) N=10^4 P=0.5 **Unwe.** on **Sandy**

(b) N=10^4 P=0.5 **Unwe.** on **Bulldozer**

(c) N=10^4 P=0.5 W$\in [0, 10^3]$ on **Sandy**

(d) N=10^4 P=0.5 W$\in [0, 10^3]$ on **Bulldozer**

(e) N=10^4 P=0.5 W$\in [0, 10^6]$ on **Sandy**

(f) N=10^4 P=0.5 W$\in [0, 10^6]$ on **Bulldozer**

Fig. 7: Results from a randomly generated Erdős-Rényi graph with N=10^4 P=0.5. Throughput (# nodes in graph ÷ execution time (μs)) on the y-axis and number of threads on the x-axis. The black dashed line is the performance of the sequential Dijkstra's algorithm with a Fibonacci Heap.

Table 3: **Sandy RoadNet Scenarios (64 threads).** The column *time* shows execution time in seconds, *waste* is the number of nodes unnecessarily processed, and the column *$miss* shows number of L2 cache misses divided by number of nodes in the graph.

Weights	**1**			**[0,1000]**			**[0,1000000]**		
	time(s)	**waste**	**$misses**	**time(s)**	**waste**	**$misses**	**time(s)**	**waste**	**$misses**
CA-PQ	0.067	1729653	7.8	0.088	1926800	12.2	0.089	1900753	12.2
CA-DM	0.430	7099	14.8	0.382	11061	34.6	0.387	9770	35.1
CA-IN	0.139	2264220	8.2	0.483	2029996	27.3	0.482	2149061	26.6
MultiQueue	0.175	8021	32.2	0.191	57562	36.1	0.192	58402	36.1
klsm1024	0.198	2498244	12.4	0.191	2222479	15.8	0.210	2213296	14.7
klsm65536	0.438	28410677	82.5	0.425	26115089	105.6	0.524	33558557	97.5
SprayList	2.509	134462	461.0	0.272	229937	88.3	0.388	218571	89.8
CATree	0.678	9	20.9	0.705	36	40.2	0.730	36	40.3
Lindén	3.387	206	108.4	1.012	252	114.6	1.009	472	114.7
Lock	7.064	210	39.7	11.023	490	59.0	10.714	398	60.6

Table 4: **Sandy LiveJournal Scenarios (64 threads).** The column *time* shows execution time in seconds, *waste* is the number of nodes unnecessarily processed, and the column *$miss* shows number of L2 cache misses divided by number of nodes in the graph.

Weights	**1**			**[0,1000]**			**[0,1000000]**		
	time	**waste**	**$misses**	**time**	**waste**	**$misses**	**time**	**waste**	**$misses**
CA-PQ	0.631	923669	30.1	0.471	352507	95.4	0.712	466590	140.9
CA-DM	0.983	8	32.2	0.467	1870	94.1	0.710	2302	140.0
CA-IN	1.254	1768109	37.0	2.337	713527	110.5	2.976	895654	159.7
MultiQueue	0.559	39	63.4	0.927	2116	112.2	0.926	2207	112.2
klsm1024	161.391	174	33980.3	7.626	2518	2538.5	9.168	6245	1989.5
klsm65536	4.763	687844	601.7	5.481	1857467	1192.7	5.839	1664835	1729.4
SprayList	8.332	41	314.9	2.386	6637	755.5	2.759	6370	815.3
CATree	1.595	1	40.8	2.274	5	107.5	3.354	6	160.8
Lindén	7.957	21	142.6	4.644	0	353.1	4.688	0	382.7
Lock	17.014	54	62.4	49.727	86	163.4	53.965	88	179.9

Table 5: **Sandy Random N=1000000 P=0.0001 (64 threads).** The column *time* shows execution time in seconds, *waste* is the number of nodes unnecessarily processed, and the column *$miss* shows number of L2 cache misses divided by number of nodes in the graph.

Weights	**1**			**[0,1000]**			**[0,1000000]**		
	time	**waste**	**$misses**	**time**	**waste**	**$misses**	**time**	**waste**	**$misses**
CA-PQ	0.499	73549	161.9	0.447	70221	272.6	0.649	47067	369.0
CA-DM	0.477	0	145.8	0.426	301	264.4	0.624	149	362.1
CA-IN	0.765	69618	170.4	1.287	110164	295.4	1.696	133327	426.2
MultiQueue	0.385	0	184.4	2.106	928	691.2	2.091	752	727.6
klsm1024	6.055	34	6669.7	2.974	342	3074.3	2.556	498	1699.0
klsm65536	2.350	329032	1135.7	3.639	712214	2893.9	3.909	758414	3358.4
SprayList	2.176	8	499.0	1.298	2094	1791.5	1.319	1798	1577.1
CATree	0.588	0	152.2	1.246	3	287.7	1.863	6	404.5
Lindén	2.175	0	281.3	2.257	0	751.6	2.284	1	826.4
Lock	3.466	9	178.6	22.618	65	446.6	24.900	63	480.3

Table 6: **Sandy Random N=10000 P=0.5 (64 threads).** The column *time* shows execution time in seconds, *waste* is the number of nodes unnecessarily processed, and the column *$miss* shows number of L2 cache misses divided by number of nodes in the graph.

Weights	**1**			**[0,1000]**			**[0,1000000]**		
	time	**waste**	**$misses**	**time**	**waste**	**$misses**	**time**	**waste**	**$misses**
CA-PQ	0.045	2	1666.3	0.065	1672	2333.7	0.072	2067	2467.3
CA-DM	0.045	2	1667.4	0.063	595	2179.0	0.073	2042	2623.4
CA-IN	0.048	7	1721.9	0.089	3246	2569.8	0.088	120	2199.6
MultiQueue	0.107	269	3482.3	0.182	795	5724.7	0.260	2197	7591.9
klsm1024	0.149	135	2564.2	0.229	377	3115.9	0.261	11126	5962.3
klsm65536	0.159	326	2506.0	0.178	4281	3062.3	0.248	19403	6089.7
SprayList	0.042	22	2545.9	0.073	1690	4413.9	0.107	7698	6641.6
CATree	0.048	1	1695.4	0.081	70	2035.0	0.090	81	2232.9
Lindén	0.058	0	1832.2	0.092	13	2575.2	0.095	9	2718.0
Lock	0.170	4635	2786.3	0.667	115	2646.8	0.827	140	2915.7