Lock-Free Priority Queue Based on Multi-Dimensional Linked Lists

Jun Tao Luo [1] **Yumin Chen** [1]

Summary

For our course project, we investigated the performance of a concurrent lock-free priority queue implementation based on multi-dimensional linked lists (MDList), which guarantees a $O(\log N)$ worst-case for insertion and deletion operations. We augmented the implementation to support duplicate priorities which allowed us to benchmark its performance on a parallelized Dijkstra's Single Source Shortest Path (SSSP) algorithm, which is a more realistic workload, in addition to microbenchmarks. We demonstrated that this implementation of a concurrent lock-free priority queue scales well to high numbers of threads compared to a naive lock-based implementation as tested using OpenMP on GHC machines at CMU and Bridges2 machines at PSC. In our experiment using a parallel SSSP benchmark, we achieved up to 100% of speedup improvement compared to the coarse-grained priority queue with a global lock in proper high concurrent situation.

[1] Carnegie Mellon University, Pittsburgh, USA. Correspondence to: , Jun Tao Luo <jtluo@andrew.cmu.edu>, Yumin Chen <yuminc@adnrew.cmu.edu>.

1. Background

To put our newly gained knowledge from this course into practice, we decided to explore the implementation and performance of lock-free data structures. While there have been many implementation of lock-free trees, queues and lists in past years we did not find any implementations of priority queues so we felt this underexplored topic would be a worthwhile subject.

1.1. Priority Queues

Scalable concurrent priority queues, which are pivotal to topics such as the realization of parallelizing search algorithms, priority task scheduling and discrete event simulation, has been a research topic for many years (Zhang & Dechev, 2016). The two main operations on priority queues are **Insert**, which inserts an entry consisting of a priority and an optional value into the data structure, and **DeleteMin**, which removes the entry with the highest priority from the data structure. In sequential implementations, this can be achieved with binary search trees, binary min heaps, Fibonacci heaps and other similar approaches. However, these approaches do not transfer well to concurrent scenarios. Particularly challenging is the necessity for maintaining a consistent global data structure and ensuring all processors agree on a highest priority entry under sequential consistency.

1.2. Single Source Shortest Path

To contextualize the use of a concurrent priority queue we turned to a classical problem in computer science: single source shortest path. This approach is used a large variety of planning and optimization problems and is formulated as follows:

Definition 1.1. Given a graph $G(V, E)$, and a starting node v, compute the length of the shortest path between v and all other nodes.

For the purpose of this project, we restrict this problem to undirected edges with positive weights, which can be efficiently solved using Dijkstra's algorithm, which internally uses a priority queue to track which nodes to visit successively to update their distances. The pseudocode of the sequential version using node 0 as the source is reproduced in Algorithm 1.

Algorithm 1 Sequential Dijkstra

Input: Nodes $\{v_i\}$, Edges $\{e_{ij}\}$
Initialize visited = {}
Initialize dists[$|\{v_i\}|$] = 0
Initialize pq = PriorityQueue
pq.insert(0, v_0)
while pq not empty and $|\text{visited}| < |\{v_i\}|$ **do**
  dist, v = pq.DeleteMin()
  visited = visited $\cup$ v
  for $e_{vj} \in \{e_{ij}\}$ **do**
    if $v_j \notin$ visited and dists[v] + e_{ij} $<$ dists[v_j] **then**
      dists[v_j] = dists[v] + e_{ij}
      pq.Insert(dists[v_j], v_j)
    end if
  end for
end while

1.2.1. Parallelized Dijkstra's Algorithm

To adapt the sequential Dijkstra's algorithm to multiple workers we use the proposed algorithm from (Tamir et al., 2015) with a fine grained per node lock on distances and offers, which represent requests to update the distance of a node. We used a parallelized version of this algorithm to evaluate the correctness of our priority queue implementation and its efficacy on a realistic workload. The algorithm is similar to the sequential version except the while loop is run in parallel. The pseudocode is reproduced in Figure 1.2.1.

```
Graph (E,V,w)
done[1..TNum] = [false,..,false]
D[1..|V|] = [∞,..,∞]
Element[1..|V|] Offer =
                      [null,..,null]
Lock [1.. |V|]  DLock
Lock [1.. |V|]  OfferLock
```

```
relax(v,vd)
 lock(OfferLock[v])
   if (vd < D[v])
     vo = Offer[v]
     if (vo = null)
       Offer[v] = insert(v,vd)
     else
       if (vd < vo.key)
         publishOfferMP(v,vd,vo)
 unlock(OfferLock[v])
```

```
publishOfferMP(v,vd,vo)
  updated = changeKey(vo, vd)
  if (!updated and vd < D[v])
    Offer[v] = insert(v,vd)
```

```
publishOfferNoMP(v,vd)
  Offer[v] = insert(v,vd)
```

```
parallelDijkstra()
 while (!done[tid])
   o = extractMin()
   if (o ≠ null)
     u = o.data
     d = o.key
     lock(DLock[u])
     if(dist < D[u])
       D[u] = d
       explore = true
     else
       explore = false
     unlock(DLock[u])
     if (explore)
       foreach ((u,v) ∈ E)
         vd = d + w(u,v)
         relax(v,vd)
   else
     done[tid] = true
     i = 0
     while (done[i] and i<TNum)
       i =  i + 1
     if(i == TNUM)
        return
     done[tid] = false
```

Figure 1. Parallel Dijkstra's Algorithm

Note that we use publishOfferNoMP since we don't want to rely on priority queues having mutable priorities. To fix a livelock issue present in the given pseudocode, we modified the algorithm to reset elements of done to false at the end of each exploration, which occurs when all relax calls of an iteration are completed.

2. Approach

We chose to implement a version of concurrent lock-free priority queue based on Multi-Dimensional Linked Lists (MDList) inspired by the ideas of (Zhang & Dechev, 2016).

We mainly applied the CAS technique to implement the lock-free concurrent priority queue and provides two canonical APIs, **Insert** and **DeleteMin**. Note that our implementation considers smaller keys to be higher priority.

2.1. MDList Implementation

The priority of the each node on the priority queue is integer. The priority will be firstly mapped to a high dimensional vector using Algorithm 2 to uniquely locate the position of the node during insertion operation. The algorithm maps key in the range of [0, N) to vector coordinates by converting the integer key to a b-based number(b = $\lceil \sqrt[D]{N} \rceil$) and using each digit as an entry. For example, if the dimension of the MDList D is 8, the upper bound of the key N is 2^{32}, and the given key is 1000, the result vector would be $[0, 0, 0, 0, 0, 3, E, 8]$, which represents the key's location on the MDList.

Algorithm 2 Mapping from Integer to Vector

Input: int key
Output: int[D] k
int $basis \leftarrow \lceil \sqrt[D]{N} \rceil, quotient \leftarrow key, k[D]$
for $i \in (D, 0]$ **do**
  $k[i] \leftarrow quotient$ mod $basis$
  $quotient \leftarrow \lfloor quotient \div basis \rfloor$
end for
return k

The structure of the MDList follows two rules: 1), we define that the dimension of a node on the MDList is in the range of [0, D). A node of dimension d has no more than (D - d) children and each of the child node has a unique dimension in the range of [d, D)[Rule 1]; 2) a non-root node of dimension d with a vector coordinates $k = [k_0, k_1, ..., k_{D-1}]$ and its parent with coordinates $k' = [k'_0, k'_1, ..., k'_{D-1}]$, $k_i = k'_i, \forall i \in [0, d) \wedge k_d > k'_d$[Rule 2].

For the insertion process, we divide it into two steps: node splicing and child adoption. At most two consecutive nodes are updated in the insertion process. Splicing involves pointing from the new node to the ancestor's old child and updating the ancestor's child pointer. Child adoption occurs when Rule 1 is violated after Step 1. If the dimension of a node increases from d to d', its children in the range $[d, d')$ must be adopted.

Algorithm 5. Pointer Marking

1: **const int** $F_{adp} \leftarrow$ 0x1, $F_{prg} \leftarrow$ 0x2, $F_{del} \leftarrow$ 0x1
2: **define** SetMark $p,\ m\ (p \mid m)$
3: **define** ClearMark $p,\ m\ (p\ \&\ \sim m)$
4: **define** IsMarked $p,\ m\ (p\ \&\ m)$

Figure 2. Pointer Marking

For the deleteMin operation, we apply logical deletion while maintaining a deletion stack to provide the information about the position of the next smallest node to reduce node traversal. Meanwhile, we also implement a rewind stack function to synchronized the insert and deleteMin operations. The stack rewind occurs only when the insertion threads notice the stack points to a position beyond the new node, which allows the insertion to move forward aggressively without blocking the deleteMin() operation.

We also applied the pointer marking technique in Figure 2 to mark adopted child and deleted nodes with three flags Fadp, Fprg and Fdel.

2.1.1. Data Structures

The structure of each node on the MDList is defined as follows(Algorithm 3). The descriptor object records the pending task of child adoption with information about the parent node and the range of the child to be adopted. A node in MDList contains a key-value pair, an array $k[D]$ of integers as the coordinate vector, an array of child pointers and a child adoption descriptor. To implement the pointer marking technique, we left shift the val by 1 bit and use the last bit as a Fdel flag. The dth pointer in the child array links to a dimension d child node. For simplicity, we allocate a child array of size D for every node while children at higher dimensions have less children. The version number helps us keep track of the proper timing for deletion stack rewind. The delete stack consists of a head pointer and an array of nodes of length D. The pointer at index $D-1$ points to the last discarded node, and the pointer at index $[0, D-1)$ points to the its parents at previous dimensions. All nodes in the stack form a path through which the next minimum node can be reached. The PriorityQueue object contains constant variables to indicate the MDList's dimension and size, a dummy head of the priority queue and a deletion stack.

Algorithm 3 Priority Queue Structures

```
struct Node
    int ver
    TKey key
    TVal val
    Node* child[D]
    AdoptDesc* adesc
    int k[D]
struct AdoptDesc
    Node* curr
    int dp, dc
struct Stack
    Node* head, node[D]
class PriorityQueue
    const int D, N
    Node* head
    Stack* stack
```

2.1.2. INSERT

In the Insertion operation(Algorithm 4), we firstly implement the inline function LocatePred to figure out the target insertion location by determining the newly inserted node's predecessor pred and successor curr and figuring out the new node's dimension dp and its child's new dimension dc. If there are pending child adoptions tasks for the predecessor and successor, we firstly finish the adoption by calling the finishInserting() function(Algorithm 5). Then we tried to insert the new node between pred and curr by applying the CAS technique. The CAS will fail in two cases: 1) another thread inserted a child into the desired child slot; 2) the child slot has been marked as invalid by parents. If it is the case 1, we retry the insertion from the predecessor. Otherwise, we retry the insertion from the head of the MDList. If the insertion into the target child slot succeed while the new node was inserted into a position before the last known deleted node, that cannot be reached by the subsequent deleteMin operations, we need to rewind the deletion stack(Algorithm 6). Figure 3 briefly illustrates how the insertion operation works. To insert a new node $(2, 0, 0)$ into a 3DList, we firstly locate the position to put into the new node starting from the root node $(0, 0, 0)$ with dimension 0. To obey Rule 2 mention in we increase the search dimension from 0 to 1 and iterate to root node's child node in dimension 1 $(1, 0, 2)$. Continually, we move the pointer to node $(1, 0, 2)$'s 1-dimension child and find the current node $2, 0, 1$ that violates the Rule2. In this way, we find the pred node $(1, 0, 2)$ and the cur node $(2, 0, 1)$. Then we fill in the new node, which takes over two children $(3, 0, 0)$ and $(2, 1, 0)$ from the old child $(2, 0, 1)$. The dimension of the old child increased from 0 to 2 because of the insertion. If node $(2, 0, 1)$ has children within the range of $[0, 2)$, they must be adopted. Figure 4 shows the scenario when we need to rewind the stack. The newly inserted node 4 was inserted before the last deleted node marked by the old stack, 5. We rewind the deletion stack to point to the closest deleted node before the newly inserted node, 3.

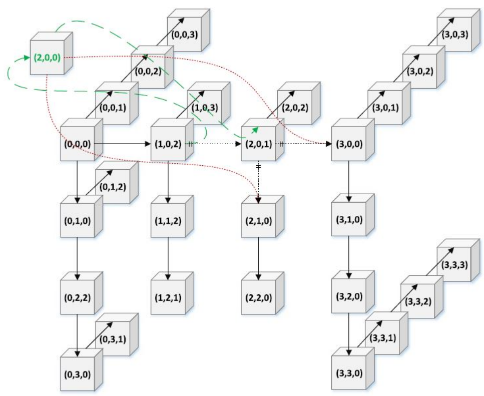

Figure 3. INSERT Operation in a 3DList

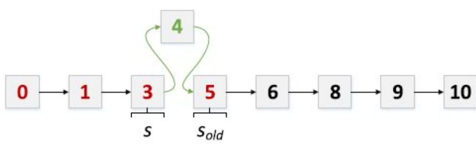

Figure 4. The Stack Rewind Scenario

2.1.3. DELETEMIN

In our implementation, we only implement the logical deletion shown in Figure 5. According to the Rules define in section 2.1, the next possible minimum will be the child node of most recently deleted node(the last entry of the stack) in dimension D - 1. This is because the top of the stack stack[D - 1] has the largest key among all the marked nodes and its smallest child should be assigned with the highest dimension. This is our starting point of search. We traverse the deletion stack from the top to see whether there is a node on the stack $stack.node[i]$ that has a unmarked child. Notice the location of the nodes on the stack is corresponding to their dimensions, we can easily get the dimensional range of their children, $[i, D)$. Figure 6 illustrates how the deletion stack helps with the deleteMin(). The red mark

Algorithm 4 Concurrent Insert

> **Input:** TKey$\{key\}$, TVal$\{val\}$
> **nodeStack* stack** $\leftarrow$ **new Stack**
> **Node* node** $\leftarrow$ **new Node**
> $node.key \leftarrow key, node.val \leftarrow val$
> $node.key[0:D] \leftarrow$ **KEYTOCOORD**(key)[0:D]
> $node.child[0:D] \leftarrow$ NIL
> **Node* pred** $\leftarrow$ NIL
> **Node* curr** $\leftarrow$ **head**
> $dp \leftarrow 0, dc \leftarrow 0$
> $nodeStack.head = currNode$
> **while** true **do**
> $LOCATEPRED()$
>   **if** dc = D **then**
>   **break**
>   **end if**
>   FINISHINSERTING(pred, pred $\leftarrow$ dp, pred $\leftarrow$ dc)
>   FINISHINSERTING(curr, curr $\leftarrow$ dp, curr $\leftarrow$ dc)
>   FILLNEWNODE()
>   **if** $CAS(\&pred.child[dp], curr, node)$ **then**
>   FINISHINSERTING(node, node $\leftarrow$ dp, node $\leftarrow$ dc)
>   REWINDSTACK()
>   **break**
>   **end if**
> **end while**
> **inline function** LOCATEPRED()
> **while** $dc < D$ **do**
>   **while** $curr \neq$ NIL and $node.k[dc] > curr.k[dc]$ **do**
> $pred \leftarrow curr, dp \leftarrow dc$
> $curr \leftarrow CLEARMARK(curr \leftarrow child, Fadp|Fprg)$
>   **end while**
>   **if** $curr$ = NIL or $node.k[dc] < curr.k[dc]$ **then**
>   **break**
>   **else**
> $nodeStack.node[dc] \leftarrow curr, dc \leftarrow dc + 1$
>   **end if**
> **end while**
> **inline function** FILENEWNODE()
> $node.adesc \leftarrow$ NIL
> **if** $dp \neq dc$ **then**
> $node.adesc \leftarrow$ **new AdoptDesc**
> $node.adesc.curr \leftarrow curr$
> $node.adesc.dp \leftarrow dp, node.adesc.dc \leftarrow dc$
> **end if**
> $node.child[0:dp] \leftarrow Fadp$
> $node.child[dp:D] \leftarrow$ NIL
> $node.child[dc] \leftarrow curr$

Algorithm 5 Child Adoption

```
Input: Node* {n}, int {dp}, int {dc}
if n = NIL then
    return
end if
AdoptDesc* ad ← n.adesc
if ad = NIL or dc < ad.dp or dp > ad.dc then
    return
end if
Node* child, curr ← ad.curr
int dp ← ad.ap, dc ← ad.dc
for i ∈ [dp, dc) do
    child ← FETCHANDOR(curr.child[i], Fadp)
    child ← CLEARMARK(child, Fadp)
    CAS(&n.child[i], NIL, child)
end for
n.adesc ← NIL
```

Algorithm 6 Rewind Deletion Stack

```
inline function REWINDSTACK()
Stack* prevStack ← NIL
Stack* currStack ← stack
Stack* newStack ← new Stack
repeat
    if nodeStack.head.ver = prevStack.head.ver then
        if node.key ≠ currStack.node[D − 1].key then
            newStack.node[0, dp] ← nodeStack.node[0, dp]
            newStack.node[dp, dc] ← pred
        else if prevStack = NIL then
            ∗newStack ← ∗currStack
        else
            break
        end if
    end if
until CAS(&stack, currStack, newStack) or ISMARKED(node.val, Fdel)
```

of nodes indicates they have been logically deleted. And the stack recorded the latest deleted stack following by its parents at previous dimensions. So in the given 3DList and deletion stack, we firstly reads $s.node[2] = (1, 1, 3)$ and examines the dimension 2 child $s.node[2].child[2]$. Since the node $(1, 1, 3)$ has no child in our example, we back-track to $s.node[1] = (1, 1, 2)$ and examine its dimension 1 child $s.node[1].child[1] = (1, 2, 1)$. Because this node is unmarked, we marked it as deleted and then update the deletion stack to reflect the new deletion. If the node found by the current thread is deleted by some competing thread, we update the local copy of stack and retry the search from the D- 1 dimension.

Algorithm 6. Concurrent DeleteMin

```
function (DeleteMin)
    Node* min ← NIL
    Stack* s_old ← stack, s ← newStack
    ∗s ← ∗s_old
    int d ← D − 1
    while d > 0 do
        Node* last ← s.node[d]
        FINISHINSERTING(last, d, d)
        Node* child ← last.child[d]
        child ← CLEARMARK(child, F_adp|F_prg)
        if child = NIL then
            d ← d − 1
            continue
        void* val ← child.val
        if ISMARKED(val, F_del) then
            if CLEARMARK(val, F_del) = NIL then
                s.node[d : D] ← child
                d ← D − 1
            else
                s.head ← CLEARMMARK(val, F_del)
                s.node[0 : D] ← s.head, d ← D − 1
        else if CAS(child.val, val, SETMARK(val, F_del)) then
            s.node[d : D] ← child, min ← child
            CAS(&stack, s_old, s)
            if marked_node > R and not_purging then
                PURGE (s.head, s.node[D − 1])
            break
    return min
```

Figure 5. The Stack Rewind Scenario

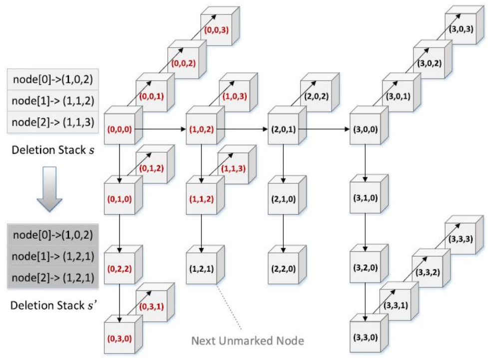

Figure 6. The Stack Rewind Scenario

2.2. Extensions and Modifications

Beyond the basic implementation of the priority queue described in (Zhang & Dechev, 2016), we needed to make a few modifications in order to support workloads such as the parallelized Dijkstra's algorithm described in Section 1.2.1.

2.2.1. Duplicate Priorities

The basic implementation of the MDList based priority queue assumes all keys are unique integers, similar to other previous skiplist-based algorithms referened in(Zhang & Dechev, 2016). A side effect of this assumption is that we cannot use 0 as a key since during initialization of the priority queue a dummy value with that key is inserted into the data structure. Additionally, the lack of support for duplicate keys is also problematic for our SSSP benchmark since the priorities represent the distances to the node stored in the value field, which can be duplicated. In fact, we may encounter multiple identical tuples if there are multiple paths to the same node with identical lengths in the graph.

To support such workloads, we partition the address space into two separate components: a key that is supplied to the **Insert** operation and a unique identifier that is assigned by the priority queue itself. For example, given a 32-bit key space, we use 19 bits to store the keys provided by the user and 13 bits to represent unique identifiers. The priority queue maintains an array of 2^{19} counters, one for each possible key the user can specify, and we atomically increment these values upon each insert call. The final key is composed of the user's key in the upper 19 bits and the unique ID in the lower 13 bits.

While this allows us to handle duplicate keys, it requires the user to specify how the key space is divided between IDs and raw keys and introduces another point of synchronization via an atomic fetch-and-increment operation. In our implementation the user can specify the type used for keys (e.g. uint for 32 bits or unsigned long for 64 bits) as well as the number of bits reserved for unique IDs via template parameters.

2.2.2. DeleteMin Return Values

Another modification we needed to make to the priority queue more simple to use is to return the priority, stored value and a flag indicating whether the values retrieved are valid instead of returning a pointer to an internal type Node. The flag is used to determine whether **DeleteMin** failed since the priority queue is empty. This required some modification of the algorithm in the reference paper since the value was stored as a pointer which sometimes is used to store temporary information that is unrelated to stored values.

2.2.3. Numeric Values

The next major modification is to allow arbitrary types for values associated with each key. In the basic MDList implementation, the value is stored as a void pointer. This is inconvenient since it requires values to be stored at specific memory locations without being overwritten throughout the entire execution of the program. Unfortunately for integer type values, this is not the case since often it is passed as an argument to the **Insert** function which marshalls the value via the stack, the space for which may be reused for other arguments in subsequent calls. Since we didn't want the priority queue to handle allocations and memory management for different value types, we opted for a design where the user explicitly specifies the type to be used for values, be it an integer type or a pointer type. This requires some modifications to the logic since the algorithm proposed in (Zhang & Dechev, 2016) involves a deletion flag that is co-located with the void pointer to the value which we resolved by reserving one bit in the value for such a flag.

2.2.4. Exclusion of Physical Deletion

Our reference paper (Zhang & Dechev, 2016) specifically disables physical deletion during its performance analysis in order to achieve parity with behaviours observed in other similar data structures such as TBBPQ which does not free memory until the termination of the object. Since we wanted to maintain functional parity with the reference paper, we decided to also support only logical deletion in our implementation.

2.2.5. Pseudocode Bugs

Throughout our implementation, we found many issues with the pseudocode provided by the reference paper (Zhang & Dechev, 2016) so our implementation differs from its description in several places. Though we won't be exhaustive, here are a few key differences:

- The loop described in the pseudocode for **DeleteMin** needs to be performed for all dimensions in $[0, D)$ instead of $(0, D)$ indicated.
- The **FinishInserting** call in **DeleteMin** should be implemented using the current and predecessor dimentions instead of d.
- When copying the stack, it needs to be a deep copy instead of a shallow copy implied by the pseudocode to prevent contamination of stack updates between concurrent threads.
- The initialization of the priority queue requires the insertion of a dummy node for the key 0. However, to support stack rewind correctly, this node needs to be marked as deleted, which was not clear from the pseudocode.

2.3. Unsuccessful optimizations

We attempted a few optimizations guided by our profiling results but either did not see any improvement or did not have time to thoroughly test them.

2.3.1. Allocation Pool

One of the bottlenecks in our **Insert** operation is the additional overhead of allocating space to store metadata for each new entry. For example, we need to allocate space for a new Node, AdoptDesc and Stack for each new entry. This is a significant overhead over the C++ std::priority_queue which stores a value directly without additional allocations. This is consistent with suggestions from discussions on other concurrent priority queue implementations which recommend reusing memory via an allocation pool. We decided to attempt a naive allocation pool where we pre-allocate a fixed number of Node, AdoptDesc and Stack and use an atomic counter to return an address in the pre-allocated pool. However, from our testing we did not see any performance benefit and we suspect the benefits of pre-allocating the memory is canceled out by the overhead of maintaining an atomic counter.

2.3.2. Physical Deletion

While we implemented physical deletion on a branch in our repository, we did not finish completely testing it and therefore do not know the performance impact it may potentially have. However, we suspect it will significantly improve certain scenarios such as the Mixed Microbenchmarks discussed later in this report. However, for other scenarios, the additional overhead of maintaining all the infrastructure to support physical deletion might be detrimental for performance.

2.3.3. Miscellaneous Optimizations

Finally we tried out a few simple optimizations and different settings for the dimension hyperparameter to see they had any impact on the memory pressure we observed in our microbenchmark profiling. For example, we tried to reduce the overhead of maintaining an atomic counter for each key to generate unique IDs and instead use a single counter for a bucket of keys. Also, we tried to use different types for keys such as uints instead of longs to reduce the memory used to store each element. Finally we tried to vary the dimension hyperparameter, trying a setting of 4 and 16. None of these attempts made any noticeable difference in performance metrics indicating that these are not the bottlenecks in our implementation.

2.4. Technologies used

We implemented the MDList based priority queue and parallel Dijkstra SSSP algorithm in C++14 compiled with gcc version 11.3.0 with optimization level O3. The parallelization is implemented using OpenMP. Benchmarking scripts to collect performance metrics and checking correctness are written in Python3.

For the benchmarking environments we collected data using GHC machines at CMU with Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz processor and 16 GB memory and PSC Bridges2 nodes with AMD EPYC 7742 64-Core Processor and 256 GB memory.

3. Experiments

As part of our experiments, we evaluated the correctness and performance of our MDList priority queue using two benchamrks: synthetic microbenchmarks and Dijkstra's SSSP.

3.1. Microbenchmarks

For the microbenchmarks evaluating raw priority queue performance on synthetic traffic, we mimicked the reference paper (Zhang & Dechev, 2016). In the paper, the performance is evaluated on three types of traffic: 100% Insert, 100% DeleteMin and a Mixed pattern consisting of 50% Insert and 50% DeleteMin. For the mixed pattern, the insertion and deletion operations are determined randomly for each iteration. All traffic patterns are run using 1M iterations in our microbenchmarks. The performance is measured by operations per second, as was done in the reference paper.

3.1.1. RESULTS

For full details on microbenchmark results see Appendix B.

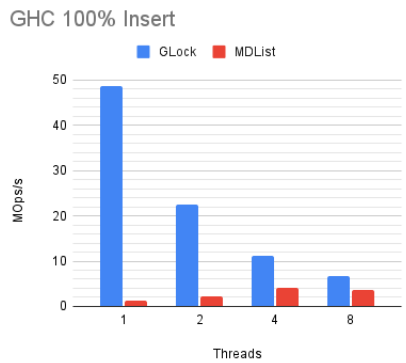

Figure 7. Microbenchmark on GHC - 100% Insert

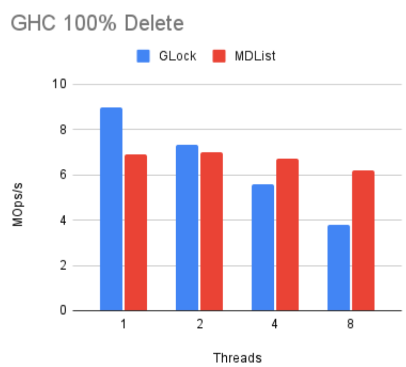

Figure 8. Microbenchmark on GHC - 100% Delete

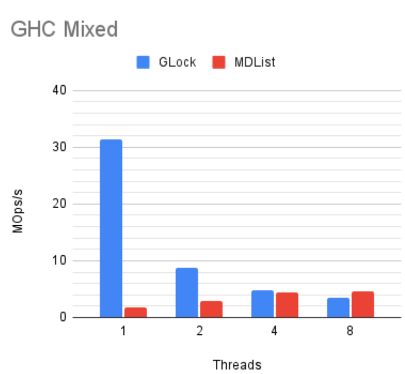

Figure 9. Microbenchmark on GHC - Mixed

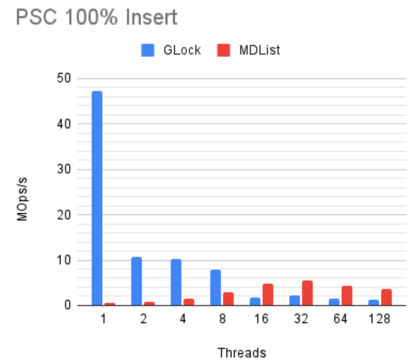

Figure 10. Microbenchmark on PSC - 100% Insert

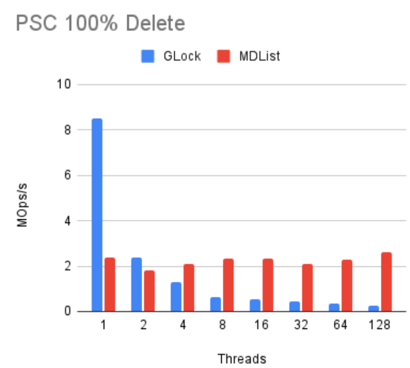

Figure 11. Microbenchmark on PSC - 100% Delete

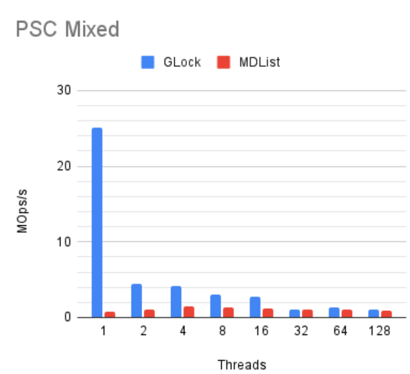

Figure 12. Microbenchmark on PSC - Mixed

3.1.2. DISCUSSION

In general, we see the common trend that coarse grain lock based concurrent priority queues generally performs well on 1 thread but quickly degrades as the number of threads increases. This clearly illustrates the limitations of a single lock under high contention where additional threads simply leads to more time spent waiting for the lock to be available and therefore a decrease in overall throughput as measured by operations per second. This effect is observed on both GHC and PSC machines.

In contrast, the performance of MDList concurrent queue generally scales well to higher number of threads. For the 100% Insert scenario on PSC, we observed an increase in performance up to 32 cores before additional cores lead to a decrease in performance. The MDList implementation outperforms the coarse grain lock for 16 cores or above. For the lower thread numbers (< 8), the MDList implementation performs poorly due to the high amount of overhead in its implementation to compute vector coordinates, managing flags and states, updating child pointers and tracking stack updates. For the intermediate number of threads (8 - 32), the overhead of MDList implementation becomes less significant and the benefits due to lock freedom, fewer number of nodes in each dimension and only requiring the modification of two consecutive nodes per insert becomes the dominant effect that helps it to continue improve its performance given more cores. Finally at extremely large number of threads (> 32), the MDList starts to see degraded performance and (Zhang & Dechev, 2016) suggests that this is due to the context switching overhead as beyond 64 threads, the executions are likely to be no longer fully concurrent given underlying hardware limitations. We observe a similar trend on GHC machines but due to the limited number of cores, we do not see any performance improvement of using MDList compared to a coarse grained lock implementation.

For the 100% Delete scenario, we see a relatively constant operations per second across all number of threads on PSC and GHC. This is expected as **DeleteMin** is an inherently a sequential bottleneck of a priority queue algorithm. However, we observe that the constant performance of the MDList priority queue represents an improvement over a coarse grained lock implementation starting at as little as 4 threads. This is because the lock free MDList implementation, while inherently sequential, does not suffer from lock contention between threads that limits the performance of the coarse grained lock implementation.

Finally, we observe relatively poor performance of MDList on Mixed traffic scenarios compared to the coarse grained lock implementation. However, this is a side effect of how the benchmarks are constructed. We followed the reference paper's approach in (Zhang & Dechev, 2016) which does not perform any physical deletion. On the other hand the coarse grained lock implementation uses a C++ priority_queue implementation that performs explicit physical deletion. As a result, the coarse grained lock implementation holds relatively few number of entries in its data structure during this benchmark compared to the MDList implementation which holds an ever increasing number of entries. As such, the MDList suffers from a large amount of entries and memory pressure since it only performs logical deletion. We found this effect too late during our analysis and profiling to include physical deletion in this microbenchmark and leave it as potential future work.

3.1.3. Analysis

To figure out the bottleneck of the MDList, we used VTune to measure the timing spend on each part of the code. One of our assumptions is that the time to create new node and new stack takes up most of the CPU time besides the busy time to add and delete nodes. As a result, we compare the CPU profile of the normal MDList and MDList with an allocator on PSC with 128 threads.

The results is shown as follows. From Figure 19 in Appendix D, the **DeleteMin** and operation new uses up most of the CPU time. To confirm our assumption that the overhead of creating more dynamic memory cause the downgrade of performance, we tried to preallocate the memory needed for the MDList node and stack creation with the Allocator.

We can see from the Figure 20 in Appendix D the CPU times for **DeleteMin** did decrease. However, the time taken by inserting increases with the cost of synchronization within the Allocator's atomic counters.

3.2. SSSP Benchmarks

To generate the inputs to our Dijkstra's SSSP benchmarks, we created a script to generate a graph with 64, 256, 1024, 4096 and 8191 nodes. To reduce the size of our input files, we then randomly generate non-zero weights for 5% of the possible edges between all nodes. As a baseline to compare our correctness and performance, we implemented a sequential version of Dijkstra using a priority queue to generate an output that consists of the distances to all nodes, which is used to judge correctness, as well as the time to run this sequential algorithm for all graph sizes, which is used to judge performance.

3.2.1. Parallelized SSSP

For a baseline to evaluate the performance of concurrent priority queues, we implemented a simple coarse grained lock concurrent queue that uses a single global lock to synchronize insertion and deletion operations. We evaluate its performance to set a baseline speedup for the parallelized SSSP benchmark. The performance is measured by its speedup compared to the sequential Dijkstra SSSP algorithm.

3.2.2. Results

Note that while we ran this benchmark for all generated graph sizes, only the 1024, 4096 and 8191 nodes scenarios are presented here as the performance smaller graphs are generally more noisy and their behaviour is similar to the 1024 node graph. For full details on SSSP results see Appendix A.

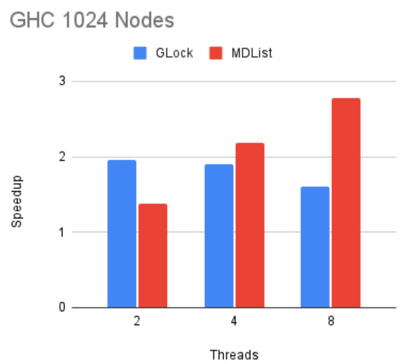

Figure 13. SSSP Benchmark on GHC - 1024 Nodes

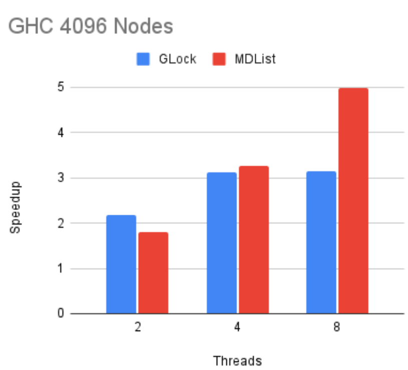

Figure 14. SSSP Benchmark on GHC - 4096 Nodes

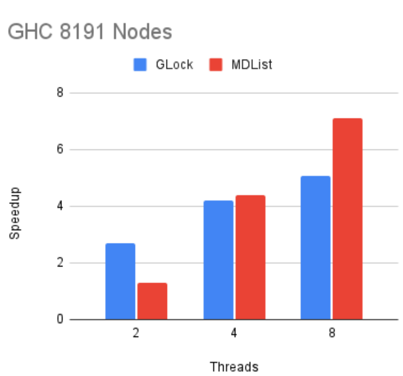

Figure 15. SSSP Benchmark on GHC - 8191 Nodes

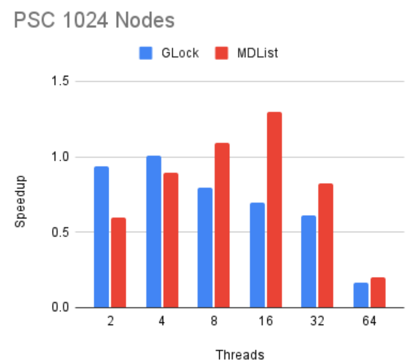

Figure 16. SSSP Benchmark on PSC - 1024 Nodes

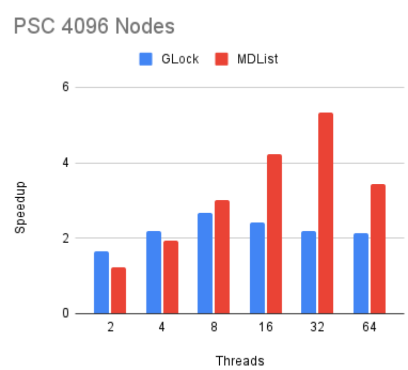

Figure 17. SSSP Benchmark on PSC - 4096 Nodes

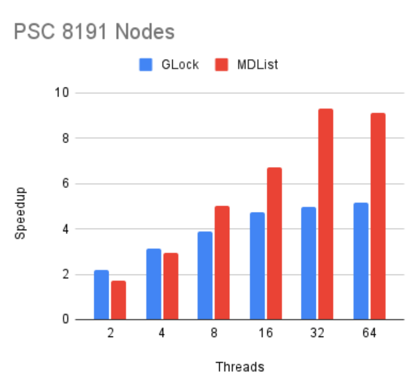

Figure 18. SSSP Benchmark on PSC - 8191 Nodes

3.2.3. Discussion

In general, we can see that the results on both GHC and PSC follow the same trend suggesting that our performance is relatively insensitive to machine architecture. Also, the coarse grain lock implementation performs better (i.e. higher speedup) than the MDList for scenarios with fewer threads and smaller number of nodes. This is due to the relatively high overhead of the MDList implementation. When given smaller problem sizes and few threads, the overhead are significant but for larger problem sizes and more threads, the concurrency benefits begin to dwarf the overhead. Overall, we see that MDList outperform coarse grain lock with for scenarios with ≥ 4 threads on GHC machines and ≥ 8 threads on PSC machines.

For coarse grain lock priority queue, we see that it scales poorly with additional threads under the parallelized SSSP workload. This is because it is quickly limited by the lock contention of the coarse grain lock. For example, the beyond 2 threads, the speedup increases much more slowly or even decrease.

For the MDList implementation we see poor speed up performance for when the number of threads is low. This is mostly due to the high overhead of the MDList implementation. For example, the structure of the node on the list is much more complicated than that on the coarse grain lock priority queue, containing information regarding the child nodes, the child adoption task and etc. In low concurrency situation, the large overhead limits the overall performance. However, when the thread number is higher, the MDList which supports concurrency without incurring higher synchronization cost outperforms the coarse grain lock implementation. As for the decreasing trend at the end of the graph, one factor is that the high competition on the MDList gives rise to more retry on CAS failure. We also notice that the larger the number of nodes, the better the speedup performance especially with higher thread numbers. This is because the larger number of nodes helps better utilize the multi cores sources.

3.2.4. Analysis

One surprising results we further analyzed was the superlinear speedup observed on the 8191 Nodes benchmark with 4 threads using the MDList priority queue implementation where we observed a speed up value of 4.38. From our profiling result, we conclude that this is mainly caused by two factors. First we see a slight reduction in branch misses of 0.35% compared to 1.16% in the sequential SSSP implementation. This is primarily because MDList priority queue does not rebalance the data structure on insert or delete. In contrast the sequential SSSP uses std::priority_queue which rebalances on insert and delete, which involves additional branching logic that are likely to be branch-misses. Also we see a reduction of cache miss rate from 44.49% in the sequential implementation to 38.29% in the MDList priority queue implementation. This is likely due to the fact that distributing the work among more threads reduces the working set for each thread, leading to higher probability that it will fit within the cache. This does not scale linearly with the number of threads because due to the random access patterns dictated by the priority queue, we can't partition the data required when running parallelized Dijkstra's algorithm into completely disjoint sets between the workers. Instead this is more of a probabilistic division where each worker likely examines a smaller portion of graph verticies and edges. For detailed profiling results see Appendix C.

We also had a suspicion that the CAS loop, while lock free, may lead to poor performance if it retries often, leading to a lot of wasted work. To test this theory we added diagnostics for successful and failed CAS counts and found that there is significant overhead in **DeleteMin**. For reference, we found that the MDList priority queue implementation running on 8 threads for the SSSP problem with 8191 nodes needed 36063 attempts to insert 34840 entries (a success rate of 96.6%) and 250915 attempts to delete 34848 entries (a success rate of 13.9%). This shows that CAS retries is a significant bottleneck for **DeleteMin**.

3.3. General Discussion

One of the main hyperparameters for this data structure is the number of dimensions. While we did not perform a full ablation study, we used the the findings from (Zhang & Dechev, 2016) to use a dimension of 8 given that our key space is 32 bits. This dimension seems to have the most stable and highest average performance across a wide range of thread numbers from 1 to 128.

While we are excited to be able to observe tangible performance improvements using a MDList based concurrent priority queue and adapt it to solve a practical problem like SSSP, we must clarify that the range of problems that this approach is applicable to may be narrow. In fact, many algorithm that uses a priority queue in its sequential version often uses a completely different approach in parallelized versions. This is because there is an inherent bottleneck in maintaining consensus of minimum entry in a priority queue. For example, more recent approaches for parallelized SSSP algorithms completely avoids using a priority queue (Srinivasan et al., 2018).

4. Further work

There are several avenues of exploration to further explore the topics visited in this project.

4.1. Concurrent Memory Allocation

We briefly explored this topic as we believe that our **Insert** is severely hindered due to its numerous allocations of small portions of memory. While our naive implementation of a memory pool did not yield any performance benefit, we believe that there are alternative malloc implementations that are worth evaluating such as (Evans, 2006).

4.2. Comparisons Against Alternative Implementations

We are aware of multiple other implementations of concurrent priority queues such as Intel's skip list based TBBPQ (Shavit & Lotan, 2000) and fine grain approaches such as (Hunt et al., 1996). Thought these are worth considering, they were evaluated as part of (Zhang & Dechev, 2016) so we felt there's little need to duplicate this effort.

4.3. Further Experimentation on MDList Bottlenecks

We unfortunately did not have sufficient time to use the results of our analysis to motivate additional experiments on optimizations to reduce the overheads and bottlenecks of the MDList based priority queue. Given additional time and resources, it would have been interesting to explore whether we can improve on the existing implementation using additional techniques learned in this class.

4.4. Physical Deletion

Though this feature is not required for correctness and we chose to not implement it for benchmark consistency reasons, we believe that for real workloads this will be highly impactful since it relieves memory pressure by periodically purging entries that have been deleted. It would be interesting to see if this feature would change any of the performance metrics we constructed.

5. Work Distribution

Yumin Chen and Jun Tao Luo divided work into roughly equal portions and would like to share the credit equally.

Table 1. Work Assignment

Task	Luo	Chen
Literature review	✓	✓
MDList data structures		✓
Coarse grained PQ	✓	
DeleteMin		✓
Benchmark scripts	✓	
Milestone report	✓	✓
Insert	✓	
Correctness checks		✓
Profiling and analysis	✓	
Final report	✓	✓
Poster	✓	✓

6. Conclusion

We are pleased that we are able to implement additional features for a MDList based concurrent priority queue and use it to solve SSSP problems correctly. We were also able to demonstrate that it scales well under high contention scenarios with many threads with both synthetic microbenchmarks and more realistic workloads compared to a naive coarse grained lock based priority queue on both GHC and PSC hardware.

References

Evans, J. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada*, 2006.

Hunt, G. C., Michael, M. M., Parthasarathy, S., and Scott, M. L. An efficient algorithm for concurrent priority queue heaps. *Information Processing Letters*, 60(3):151–157, 1996.

Shavit, N. and Lotan, I. Skiplist-based concurrent priority queues. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pp. 263–268. IEEE, 2000.

Srinivasan, S., Riazi, S., Norris, B., Das, S. K., and Bhowmick, S. A shared-memory parallel algorithm for updating single-source shortest paths in large dynamic networks. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pp. 245–254, 2018. doi: 10.1109/HiPC.2018.00035.

Tamir, O., Morrison, A., and Rinetzky, N. A heap-based concurrent priority queue with mutable priorities for faster parallel algorithms. In *OPODIS*, 2015.

Zhang, D. and Dechev, D. A lock-free priority queue design based on multi-dimensional linked lists. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):613–626, 2016. doi: 10.1109/TPDS.2015.2419651.

A. Detailed Parallelized SSSP Result

A.1. GHC Machines

A.1.1. GLOCK

-- Performance Table ---

Scene Name	2	4	8
bench-64	8.8e-05	0.000187	0.000387
bench-256	0.000359	0.000677	0.000395
bench-1024	0.001439	0.001488	0.001765
bench-4096	0.015637	0.010914	0.010777
bench-8191	0.055649	0.03562	0.029518

-- Speedup Table ---

Scene Name	2	4	8
bench-64	0.272727	0.128342	0.062016
bench-256	0.615599	0.326440	0.559494
bench-1024	1.963864	1.899194	1.601133
bench-4096	2.173435	3.113982	3.153568
bench-8191	2.694388	4.209433	5.079612

A.1.2. MDLIST

Scene Name	2	4	8
bench-64	9.3e-05	0.000145	0.000241
bench-256	0.000481	0.000394	0.000663
bench-1024	0.002044	0.001297	0.001019
bench-4096	0.018798	0.01044	0.00682
bench-8191	0.116867	0.034208	0.021115

-- Speedup Table ---

Scene Name	2	4	8
bench-64	0.258065	0.165517	0.099585
bench-256	0.459459	0.560914	0.333333
bench-1024	1.382583	2.178874	2.773307
bench-4096	1.807958	3.255364	4.983284
bench-8191	1.282997	4.383185	7.101113

A.2. PSC Machines

A.2.1. GLOCK

-- Performance Table ---

Scene Name	2	4	8	16	32	64
bench-64	0.00038	0.000424	0.000798	0.000998	0.0029	0.015268
bench-256	0.000739	0.000841	0.001232	0.001479	0.005017	0.014617
bench-1024	0.004475	0.004144	0.00528	0.006038	0.006879	0.025166
bench-4096	0.037754	0.028192	0.023348	0.025711	0.028318	0.029211
bench-8191	0.118094	0.08308	0.066503	0.054562	0.052378	0.050342

-- Speedup Table ---

Scene Name	2	4	8	16	32	64
bench-64	0.052632	0.047170	0.025063	0.020040	0.006897	0.001310
bench-256	0.441137	0.387634	0.264610	0.220419	0.064979	0.022303
bench-1024	0.934525	1.009170	0.792045	0.692613	0.607937	0.166177
bench-4096	1.643561	2.201014	2.657658	2.413403	2.191221	2.124234
bench-8191	2.194616	3.119535	3.897132	4.750027	4.948089	5.148206

A.2.2. MDLIST

-- Performance Table ---

Scene Name	2	4	8	16	32	64
bench-64	0.000495	0.000597	0.000888	0.001161	0.007756	0.015296
bench-256	0.001165	0.00097	0.001062	0.00135	0.00224	0.014903
bench-1024	0.007018	0.004686	0.003832	0.003232	0.005098	0.020723
bench-4096	0.05107	0.031908	0.02054	0.014684	0.011624	0.01802
bench-8191	0.152874	0.087819	0.051448	0.03851	0.027803	0.028424

-- Speedup Table ---

Scene Name	2	4	8	16	32	64
bench-64	0.040404	0.033501	0.022523	0.017227	0.002579	0.001308
bench-256	0.279828	0.336082	0.306968	0.241481	0.145536	0.021875
bench-1024	0.595896	0.892446	1.091336	1.293936	0.820322	0.201805
bench-4096	1.215019	1.944685	3.020983	4.225756	5.338180	3.443452
bench-8191	1.695324	2.951195	5.037533	6.729966	9.321692	9.118034

B. Detailed Microbenchmark Results

B.1. GHC Machines

-- Performance Table ---

Scene Name	1	2	4	8
PQGLock Insert	0.020509	0.0445174	0.0891181	0.149741
PQGLock Delete	0.111597	0.136812	0.179392	0.265326
PQGLock Mixed	0.031956	0.115131	0.209116	0.281722
PQMDList Insert	0.827034	0.435831	0.237808	0.130466
PQMDList Delete	0.144778	0.143114	0.149287	0.161638
PQMDList Mixed	0.578207	0.345395	0.228541	0.213554

-- Ops/s Table ---

Scene Name	1	2	4	8
PQGLock Insert	48.759 MOps/s	22.463 MOps/s	11.221 MOps/s	6.678 MOps/s
PQGLock Delete	8.961 MOps/s	7.309 MOps/s	5.574 MOps/s	3.769 MOps/s
PQGLock Mixed	31.293 MOps/s	8.686 MOps/s	4.782 MOps/s	3.550 MOps/s
PQMDList Insert	1.209 MOps/s	2.294 MOps/s	4.205 MOps/s	7.665 MOps/s
PQMDList Delete	6.907 MOps/s	6.987 MOps/s	6.699 MOps/s	6.187 MOps/s
PQMDList Mixed	1.729 MOps/s	2.895 MOps/s	4.376 MOps/s	4.683 MOps/s

B.2. PSC Machines

-- Performance Table ---

Scene Name	1	2	4	8	16	32	64	128
PQGLock Insert	0.0211886	0.0937253	0.0982313	0.125577	0.595807	0.466818	0.636549	0.75646
PQGLock Delete	0.117585	0.423217	0.779008	1.56822	1.78209	2.27517	2.74327	3.58381
PQGLock Mixed	0.0399167	0.221527	0.239356	0.32732	0.367156	1.00724	0.778583	0.910591
PQMDList Insert	1.89338	1.2012	0.65312	0.354061	0.205459	0.181712	0.2348	0.278263
PQMDList Delete	0.418828	0.552922	0.474337	0.431324	0.432042	0.418187	0.434104	0.382783
PQMDList Mixed	1.19834	0.889697	0.659059	0.788896	0.851559	0.912034	1.00867	1.17951

-- Ops/s Table ---

Scene Name	1	2	4	8	16	32	64	128
PQGLock Insert	47.195 MOps/s	10.669 MOps/s	10.180 MOps/s	7.963 MOps/s	1.678 MOps/s	2.142 MOps/s	1.571 MOps/s	1.321 MOps/s
PQGLock Delete	8.504 MOps/s	2.363 MOps/s	1.284 MOps/s	0.638 MOps/s	0.561 MOps/s	0.440 MOps/s	0.365 MOps/s	0.279 MOps/s
PQGLock Mixed	25.052 MOps/s	4.514 MOps/s	4.178 MOps/s	3.055 MOps/s	2.724 MOps/s	0.993 MOps/s	1.284 MOps/s	1.098 MOps/s
PQMDList Insert	0.528 MOps/s	0.833 MOps/s	1.531 MOps/s	2.824 MOps/s	4.867 MOps/s	5.503 MOps/s	4.259 MOps/s	3.593 MOps/s
PQMDList Delete	2.388 MOps/s	1.809 MOps/s	2.108 MOps/s	2.318 MOps/s	2.315 MOps/s	2.089 MOps/s	2.304 MOps/s	2.612 MOps/s
PQMDList Mixed	0.834 MOps/s	1.124 MOps/s	1.517 MOps/s	1.268 MOps/s	1.174 MOps/s	1.096 MOps/s	0.991 MOps/s	0.847 MOps/s

C. Superlinear Speedup Profiling Results

C.1. Sequential Dijkstra

```
jtluo@ghc48:~/618Project$ perf stat ./dijk-release \
  -in graphs/bench-8191-init.txt -o logs/prof/out.txt
total time: 0.208802s

 Performance counter stats for './dijk-release -in graphs/bench-8191-init.txt -o logs/prof/out.txt':

            902.17 msec task-clock                #    0.996 CPUs utilized
                13      context-switches          #   14.410 /sec
                 0      cpu-migrations            #    0.000 /sec
            65,903      page-faults               #   73.050 K/sec
     4,155,049,136      cycles                    #    4.606 GHz
     9,875,077,959      instructions              #    2.38  insn per cycle
     1,966,852,513      branches                  #    2.180 G/sec
        22,888,711      branch-misses             #    1.16% of all branches

       0.905845294 seconds time elapsed

       0.794969000 seconds user
       0.107860000 seconds sys
```

```
jtluo@ghc48:~/618Project$ perf stat -e cache-references,cache-misses ./dijk-release \
  -in graphs/bench-8191-init.txt -o logs/prof/out.txt
total time: 0.208367s

 Performance counter stats for './dijk-release -in graphs/bench-8191-init.txt -o logs/prof/out.txt':

        44,904,780      cache-references
        19,980,102      cache-misses              #   44.494 % of all cache refs

       0.915987465 seconds time elapsed

       0.801012000 seconds user
       0.112141000 seconds sys
```

C.2. Parallel Dijkstra with MDList Priority Queue

```
jtluo@ghc48:~/618Project$ OMP_NUM_THREADS=4 perf stat ./pardijkMdlist \
  -in graphs/bench-8191-init.txt -o logs/prof/mdlist.txt
total time: 0.043833s

 Performance counter stats for './pardijkMdlist -in graphs/bench-8191-init.txt -o logs/prof/mdlist.txt':

            960.71 msec task-clock                #    1.151 CPUs utilized
                42      context-switches          #   43.718 /sec
                 0      cpu-migrations            #    0.000 /sec
           134,352      page-faults               #  139.847 K/sec
     4,406,037,456      cycles                    #    4.586 GHz
     9,603,341,725      instructions              #    2.18  insn per cycle
     1,938,623,691      branches                  #    2.018 G/sec
         6,704,793      branch-misses             #    0.35% of all branches

       0.834623575 seconds time elapsed

       0.800543000 seconds user
       0.160108000 seconds sys

jtluo@ghc48:~/618Project$ OMP_NUM_THREADS=4 perf stat -e cache-references,cache-misses ./pardijkMdlist \
  -in graphs/bench-8191-init.txt -o logs/prof/mdlist.txt
total time: 0.034470s

 Performance counter stats for './pardijkMdlist -in graphs/bench-8191-init.txt -o logs/prof/mdlist.txt':

        74,560,394      cache-references
        28,549,722      cache-misses              #   38.291 % of all cache refs

       0.813469502 seconds time elapsed

       0.797792000 seconds user
       0.148333000 seconds sys
```

D. VTune results

Insert Only without Allocator

```
Function                                                                    Module         CPU Time
--------------------------------------------------------------------------  -------------  --------
PriorityQueue<(int)8, (long)1000001, (int)100, (int)0, int, int>::deleteMin  micro           63.191s
operator new                                                                libstdc++.so.6   4.241s
PriorityQueue<(int)8, (long)1000001, (int)100, (int)0, int, int>::insert     micro            2.363s
func@0x1df54                                                                libgomp.so.1     1.555s
func@0x1dda4                                                                libgomp.so.1     1.060s
[Others]                                                                    N/A              0.900s
```

Insert and DeleteMin without Allocator

```
Function                                                                 Module         CPU Time
-----------------------------------------------------------------------  -------------  --------
operator new                                                             libstdc++.so.6   4.324s
PriorityQueue<(int)8, (long)1000001, (int)100, (int)0, int, int>::insert  micro            2.636s
__memset_avx2_unaligned_erms                                             libc.so.6        0.770s
func@0x1dda4                                                             libgomp.so.1     0.750s
func@0x1df54                                                             libgomp.so.1     0.671s
[Others]                                                                 N/A              0.200s
```

Figure 19. The VTune result without Allocator

Insert Only with Allocator

```
Function                                                                                     Module         CPU Time
-------------------------------------------------------------------------------------------  -------------  --------
PriorityQueue<(int)8, (long)1000001, (int)100, (int)0, (int)524288, int, int>::deleteMin  micro           49.101s
PriorityQueue<(int)8, (long)1000001, (int)100, (int)0, (int)524288, int, int>::insert     micro            4.316s
func@0x1df54                                                                                 libgomp.so.1     1.511s
func@0x1dda4                                                                                 libgomp.so.1     1.200s
operator new                                                                                 libstdc++.so.6   1.155s
[Others]                                                                                     N/A              1.597s
```

Insert and DeleteMin with Allocator

```
Function                                                                                  Module         CPU Time
----------------------------------------------------------------------------------------  -------------  --------
PriorityQueue<(int)8, (long)1000001, (int)100, (int)0, (int)524288, int, int>::insert  micro            4.313s
func@0x1df54                                                                              libgomp.so.1     0.868s
__memset_avx2_unaligned_erms                                                              libc.so.6        0.780s
_mcount                                                                                   libc.so.6        0.500s
operator new                                                                              libstdc++.so.6   0.169s
[Others]                                                                                  N/A              0.170s
```

Figure 20. The VTune result with Allocator