CS309 Project

Bhoomil Gohel: 190001008
Neel K. Parikh: 190001044
Yash Suvarna: 190001068
Parul Mogre: 190001037

A Practical, Scalable, Relaxed Priority Queue

November 15, 2021

Introduction

Priority queues are a fundamental data structure, and in highly concurrent software, scalable priority queues are an important building block. In many workloads like Dijkstra's Single-Source Shortest Path, a requirement of priority queue that each extract() returns the highest priority element can be relaxed, improving scalability by reducing accuracy.

We introduce ZMSQ, a scalable relaxed priority queue. It is the first relaxed priority queue that supports each of the following important practical features:

(i) relaxation accuracy that does not degrade as the thread count increases.

(ii) guaranteed success of extraction when the queue is nonempty.

In addition, ZMSQ significantly outperforms state-of-the-art prior algorithms.

Key Points

Priority queues are an important data structure for high-performance scalable systems. One of the most significant challenges in creating a scalable A priority queue is its strict sequential specification, which requires each extractMax() operation to return the highest-priority element in the queue. This creates a scalability bottleneck. Recent works showed that programs can tolerate when extractMax() returns a high-priority element that is not the highest-priority element, so long as there is a bound on the number of consecutive calls to extractMax() that do not return the highest-priority element.

Why Relaxation of Priority Queue is Acceptable:

(i) If a linearizable priority queue guarantees to order between an extractMax() and subsequent use of the returned value. That is if Thread T1 extracts element E1, and then T2 extracts E2, where E1 > E2 programs typically do not synchronize after the call to extractMax(), and thus T2 may use E2 before T1 uses E1.

(ii) In many graph algorithms, processing elements out of order still contributes to the forward progress of an application. If E2 is processed before E1, then either (a) E1's subsequent processing will not invalidate the work done with E2, or else (b) re-processing E2 will be quick because some of the total work on E2 will have been done already. As an example of the latter, consider Dijkstra's single-source shortest path algorithm: The work done processing elements out of order still advances the computation toward a solution.

How to increase Scalability and Achieve Relaxed Priority Queue:

Relaxed priority queues balance a decrease in the accuracy of extractMax() for an increase in scalability. ZMSQ, the first relaxed priority queue that supports the following practical features:

(i) the guaranteed success of extraction when the queue is nonempty,

(ii)relaxation accuracy that does not degrade as the thread count increases.

The ZMSQ algorithm uses several novel techniques to achieve the above features. It uses a small shared pool of high-priority elements for fast extraction, and periodically replenishes the pool from the main data structure. The use of a scalable shared pool without any thread-specific structures enables the algorithm to guarantee that it observes an empty queue only when indeed there are no elements in the queue. The shared pool is structured to support optional scalable low-latency consumer blocking, as well as non-blocking conditional extraction and/or spin-waiting. The data structures are managed such that a tunable level of relaxation is maintained (provided the queue contains enough elements) regardless of the number of threads and regardless of the input and use patterns.

Data Structures And Data Types:

ZMSQ uses the mound's structure but substantially improves the quality of a node's data versus the mound.

There are two goals of this:

(i)to ensure each node has enough elements

(ii)to ensure the elements at each node are close in value so that nodes near the root will have many elements that are close in priority.

ZMSQ also introduces an explicit mechanism for extracting many operations at once, a new synchronization strategy, memory safety, and support for blocking threads when the queue is empty.

What is Mound Data Structure:

The mound is a lock-free concurrent heap implemented as a binary tree of sorted lists. For every tree node Np with children Ncl and Ncr , Np .list .head ≥ max(Ncl .list .head, Ncr .list .head). To insert key k, a thread chooses a random empty leaf, and then does a binary search on the path from that leaf to the root (NR), stopping when it finds a node Nc with parent Np, for which Nc .list .head ≤ k and Np .list .head > k. It then inserts k as the head of Nc .list. ExtractMax() removes the head from NR's list and then checks if NR.list.head became smaller than the head of one of its children's lists. If so, it swaps the lists of NR and the child with the larger list head value. The swapping process recurses downward as necessary to restore invariants in every subtree. Relaxing the mound invariant at the root could transform the mound into a required relaxed priority queue.

Data Type:

We define TNode as a node in the ZMSQ tree, consisting of a set of values and a lock. To reduce latency and synchronization, a TNode caches its set min and max values, as well as its count of elements, in atomic variables that are only updated while holding the lock. TNodes are organized as binary trees. Conceptually, each field has left, right, and parent fields. In practice, the ZMSQ nodes field is an array of arrays ofTNodes. In nodes, the sub-array at position i stores 2i TNodes. This representation of a binary tree allows binary searches along the path from any node to the root. The remaining fields of ZMSQ are leaf Level, batch, targetLen, pool, and poolNext. leaf Level indicates the deepest level of nodes whose sub-array contains non-null values. batch and targetLen are user-defined parameters: batch sets an upper bound on the number of elements (in addition to the maximum) that can be produced by extractPool(), and targetLen defines the number of elements to try and store in each TNode. A set may hold at most 2 × targetLen elements. The pool is a reference to a set of up-to-batch elements, and poolNext is an atomic integer.

Tuning the ZMSQ (target length and MaxLength)

ZMSQ can be altered/tuned to give better results by performing memory reclamation and finding the best choice of 'batch' and 'targetLen'. The latter is largely independent of the thread count used in the execution and therefore the tuning of ZMSQ becomes simple and straightforward. A good value of 'batch' and a good ratio of 'batch' and 'targetLen' will give the expected results.

From our observations and experimentations, it was found that for batch = targetLen, the extractPool() would rarely return a full pool of batch elements. For larger batch values, accuracy suffers but provides better scalability.

There are two main findings based on the graphs and the experimental, one being that several choices delivered roughly the same performance and the second being the ease of finding good choices for better performance.

Atomic Data Type:

To avoid uncertainty about interrupting access to a variable, you can use a particular data type for which access is always atomic. Reading and writing this data type is guaranteed to happen in a single instruction, so there's no way for a handler to run "in the middle" of access.

Push()

The push function aims to have a high number of elements in each of the nodes such that each node has the least range of numbers in it.

The algorithm that we use here has two modes for insertion 1) Fast 2) Slow

a) Fast insertion

In this technique the node is inserted into a leaf node in such a way that it doesn't aggressively cause an increase in the tree height whenever the tree height is greater than 3 or the node size is less than the targetLength or the decided size for each node's list. Here the max for the chosen list is less than the value to be inserted and the list length is less than the target length.

b) Slow insertion

In this the value is inserted into a node for which the Node.max < Val or the Node.parent.max >Val. When the size of the set gets twice the size of the target length, we split the node into 2 halves and then merge the smaller half (the right half) list with the child.

These changes offer three main benefits. First, they reduce the cost of tree traversals, since the tree is more compact. Especially for pop(), which may need to migrate a set from root to leaf, this reduces the number of levels by 4–5. Second, less memory is required, since the number of T Nodes is reduced substantially.

Finally, these changes decrease the frequency with which our relaxed version (batch > 0) of pop() touches the root.

We are also inserting elements in such a way that reduces the range of values in a list.

We first inspect its parent's min: if Node.parent.min < Val, then we insert k into the parent, and move Node.parent .min into Node. (Note that Val may not be the smallest element in Node, and Node.parent .min may be smaller than some of the elements in Node.) This technique decreases the range of values in Node.parent, which improves quality. It may increase the range of values in Node. However, there will be more opportunity to improve the quality of the set in Node in subsequent insertions, since it cannot satisfy any pop() calls until Node.parent satisfies at least two.

Pop()

We have an auxiliary data structure which is used to obtain the max element. When the pool is empty, we replenish the pool with values from the mound. Concurrency is maintained across all operations by using atomic variables which are inherently synchronous in nature and do not require locks for reading / writing to them. Similarly locks are applied whenever we are trying to update the values in the mound. Whenever the required conditions for insertion are satisfied the values are updated and the locks are released.

The tryLock() method attempts to lock the Lock instance immediately. It returns true if the locking succeeds, false if Lock is already locked.

Spin Lock

We use an atomic variable to implement a spin lock which is a non blocking lock in nature. We have three main methods in the spin lock

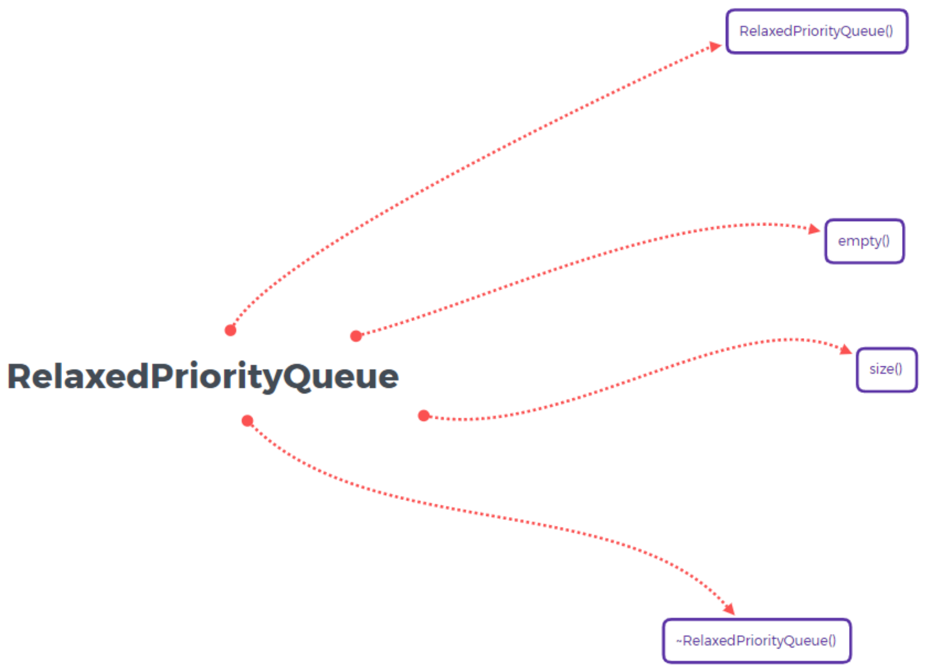
Lock : we use this to lock the spin lock

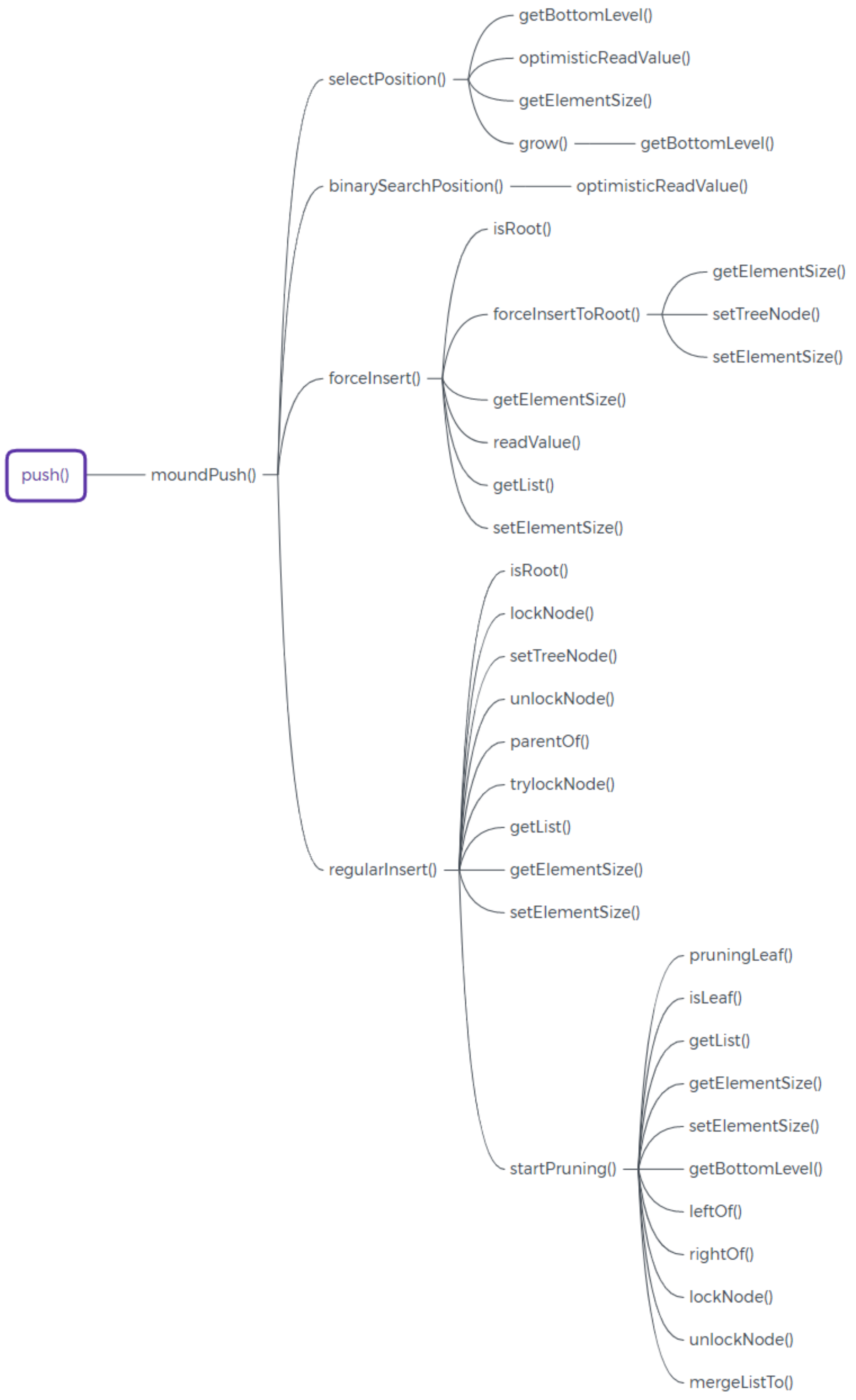
Try_lock : Function to check if the lock is free inorder to prevent cache misses

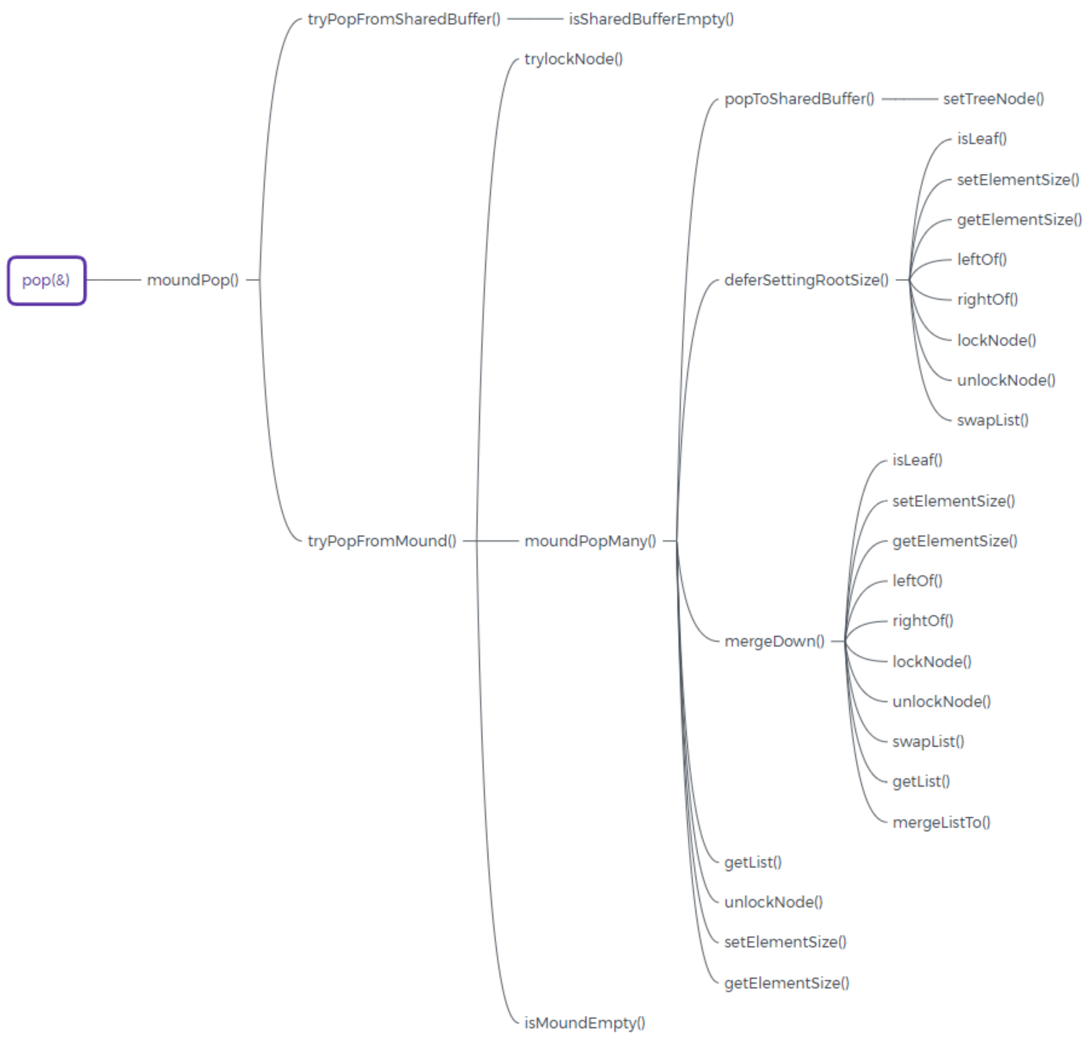
Unlock : using this we can unlock the atomic lock

```
// Custom spin lock implementation

// CS 309


#include <thread>

#include <mutex>

#include <atomic>
```

```
struct SpinLock
{
    std::atomic<bool> lock_ = {0};

    void lock() noexcept
    {
        for (;;)
        {
            // Optimistically assume the lock is free on the first try
            if (!lock_.exchange(true, std::memory_order_acquire))
            {
                return;
            }
            // Wait for lock to be released without generating cache
misses
            while (lock_.load(std::memory_order_relaxed))
            {
                // To reduce the contention between hyper threads
                __builtin_ia32_pause();
            }
        }
    }

    bool try_lock() noexcept
    {
        // Function to check if the lock is free inorder to prevent
cache misses

        return !lock_.load(std::memory_order_relaxed) &&

               !lock_.exchange(true, std::memory_order_acquire);

    }


    // unlock the lock

    void unlock() noexcept

    {

        lock_.store(false, std::memory_order_release);

    }

};
```




push()
moundPush()
selectPosition()
getBottomLevel()
optimisticReadValue()
getElementSize()
grow()
getBottomLevel()
binarySearchPosition()
optimisticReadValue()
forceInsert()
isRoot()
forceInsertToRoot()
getElementSize()
setTreeNode()
setElementSize()
getElementSize()
readValue()
getList()
setElementSize()
regularInsert()
isRoot()
lockNode()
setTreeNode()
unlockNode()
parentOf()
trylockNode()
getList()
getElementSize()
setElementSize()
startPruning()
pruningLeaf()
isLeaf()
getList()
getElementSize()
setElementSize()
getBottomLevel()
leftOf()
rightOf()
lockNode()
unlockNode()
mergeListTo()


pop(&)
moundPop()
tryPopFromSharedBuffer()
isSharedBufferEmpty()
trylockNode()
popToSharedBuffer()
setTreeNode()
isLeaf()
setElementSize()
getElementSize()
leftOf()
deferSettingRootSize()
rightOf()
lockNode()
unlockNode()
swapList()
isLeaf()
setElementSize()
tryPopFromMound()
moundPopMany()
getElementSize()
leftOf()
rightOf()
mergeDown()
lockNode()
unlockNode()
swapList()
getList()
mergeListTo()
getList()
unlockNode()
setElementSize()
getElementSize()
isMoundEmpty()

Application

Dijkstra's Algorithm for Single Source Shortest Path(SSSP) problem

Dijkstra's Algorithm is used to find for a particular source vertex in a simple graph the shortest distance to every other vertex node. A typical implementation consists of using a minimum priority queue.

```
void dijkstra(int s, vector<int> &d) {
    int n = adj.size();
    d.assign(n, INF);
    d[s] = 0;
    priority_queue<pair<int, int>> q;
    q.push({0, s});
    while (!q.empty())
    {
        int u = q.top().second;
        int d_u = -q.top().first;
        q.pop();
        if (d_u != d[u])
            continue;

        for (auto edge : adj[u])
        {
            int v = edge.first;
            int len = edge.second;

            if (d[u] + len < d[v])
```

```
                {
                    d[v] = d[u] + len;
                    q.push({-d[v], v});
                }
            }
        }
}
```

We see from the implementation that, at each iteration, we find the vertex that currently has the least distance from the source vertex. We do this by using a priority queue where each pop operation gives a {dist, vertex} pair of the vertex having the least distance.

We also ensure that this vertex is not an outdated vertex by comparing dist with the current stored distance in d[vertex]. This is called lazy deletion.

Once finding such a vertex, we try to relax all the neighbouring vertices of this vertex(u). Relaxation means trying to update the distance of a neighbour(v) from the source(s) by trying the path s -> u -> v. If it's possible that this path leads to a shorter distance than the currently stored dist[v] then we update dist[v] and also push a new {dist, vertex} pair.

After the while loop completes, we will have the distance of each vertex from the source in the distance vector.

Note: We use negative signs of distances while pushing in priority queue because the default implementation of the priority queue in std library's max priority, i.e. a pop operation extracts the highest priority element.

Parallelizing Dijkstra's Algo:

Looking at the implementation of Dijkstra algo, we can observe that we can try to parallelize the main portion of the algo, i.e. the relaxation step. Once we have determined the particular vertex(u) that we want to relax edges coming out of, we see that there is a particular number of edges that we want to relax, specifically "size(adj[u])", that is the number of neighbors of u.

We can distribute this task among a number of threads, where each thread will carry out the relaxation of edges independently and each thread gets some number of edges to relax. For instance, if we had 7 edges to relax and 3 threads to work with, the distribution may look like this:

1st thread -> edges {1, 2}

2nd thread -> edges {3, 4}

3rd thread -> edges {5, 6, 7}

This way we parallelize the work that would normally be done by one thread.

```
void relax(int i, int u, int npt)
{
    int sz = adj[u].size();
    int st = i * npt;
    int en = -1;
    if (i == NUM_THREADS - 1)
    {
        en = sz;
    }
    else
    {
        en = st + npt;
    }
    for (int v = st; v < en; v++)
    {
        int ver = adj[u][v].first;
        int len = adj[u][v].second;
        if (d[u] + len < d[ver])
        {
            d[ver] = d[u] + len;
            q.push({-d[ver], ver});
        }
    }
}

thread tharr[NUM_THREADS]; // Global
void dijkstra(int s)
{
    d.assign(n, INF);
    d[s] = 0;
    q.push({0, s});
    while (!q.empty())
    {
        pair<int, int> u;
        q.pop(u);
        int v = u.second;
        int d_v = -u.first;
        if (d_v != d[v])
            continue;
        int sz = adj[v].size();
        if (sz >= NUM_THREADS)
        {
            int v_per_thread = sz / (NUM_THREADS);
            for (int i = 0; i < NUM_THREADS; i++)
            {
                tharr[i] = thread(relax, i, v, v_per_thread);
            }
            for (int i = 0; i < NUM_THREADS; i++)
            {
                tharr[i].join();
            }
        }
        else
        {
            for (auto e : adj[v])
            {
                int to = e.first;
                int len = e.second;
                if (d[v] + len < d[to])
                {
                    d[to] = d[v] + len;
                    q.push({-d[to], to});
                }
            }
        }
    }
}
```

Time Taken Analysis:

Note: For analysis here, we use complete graphs with varying numbers of vertices. For instance, if n = 5, we will have 10 edges.

1. Serial implementation of dijkstra using std library priority queue-

n	time (microseconds)
10	31
100	454
1000	12855
2000	44440

2. Parallel implementation of dijkstra using Relaxed Priority Queue(customised for parallel tasks)-

2 Threads:

n	time (microseconds)
10	983
100	7037
1000	78198
2000	193914

4 Threads:

n	time (microseconds)
10	1883
100	12482
1000	103495

n	time (microseconds)
2000	230358

8 Threads:

n	time (microseconds)
10	2503
100	16175
1000	165881
2000	356802

16 Threads:

n	time (microseconds)
10	45
100	33146
1000	315120
2000	594514

Note: here (16 threads: n = 10) the parallel part of code doesn't get called bcz number of threads get more than number of neighbours.

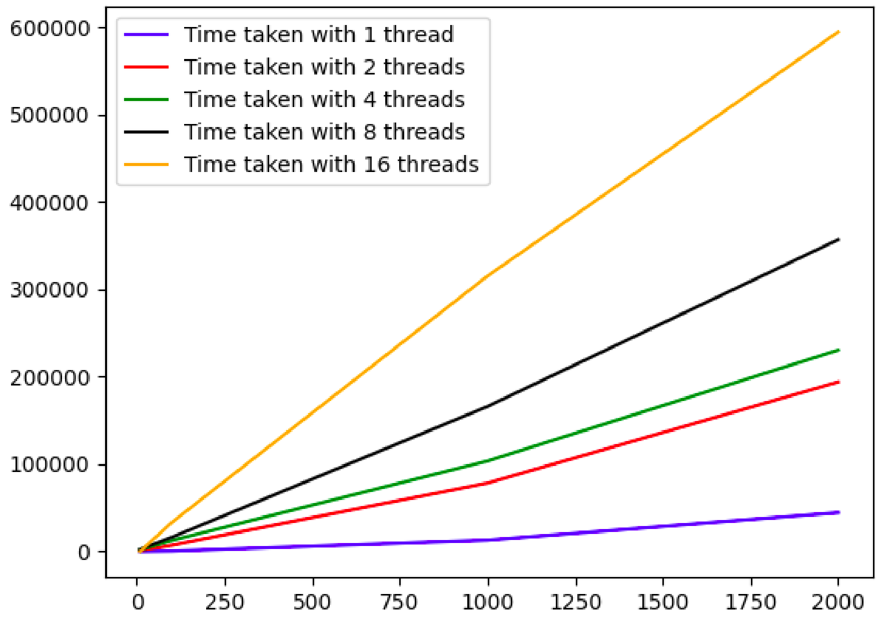
Speed Up:

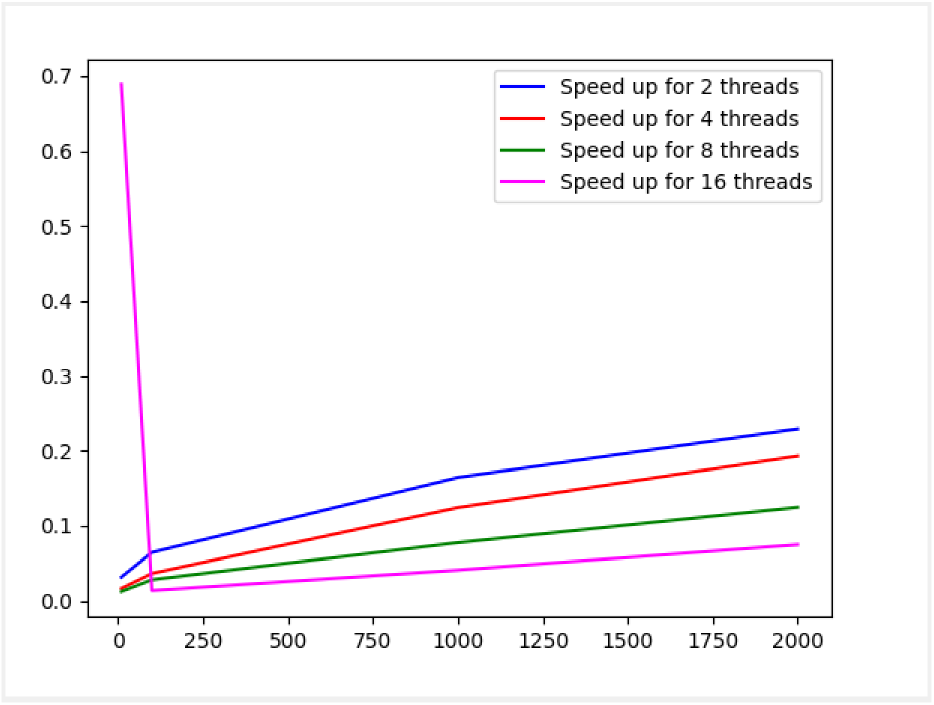
Speed up is defined as the ratio of performance of the task using the enhancement and the performance of the task without using the enhancement. It is one of the ways to measure the expected improvement after introducing the enhancement.

Speed up for ‘p’ threads = Time taken by 1 thread / Time taken by p threads

Plot:

Plots for: Time taken(y axis) and Speed up (y axis) vs Number of vertices (x axis)





Further Improvements:

The garbage collector attempts to reclaim garbage, or memory used by objects that will never be accessed or mutated again by the application. This will help with better locality of memory and thus faster access. Alignas can be used to align the memory element in RAM when working with a large dataset thus we can optimize the RAM usage.

Acknowledgment:

Tingzhe Zhou, Maged Michael, and Michael Spear. 2019. A Practical, Scalable, Relaxed Priority Queue. In Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019). Association for Computing Machinery, New York, NY, USA, Article 57, 1–10. DOI:https://doi.org/10.1145/3337821.3337911