Project Report
Concurrent Priority-Queues

Algorithms

We implemented 2 algorithms related to making the priority Queue data structure concurrent

- **Skip list based Lock-free algorithm**
- **Heap-based Fine-grained Locking algorithm (Supports Mutable Priorities)**

Application

We created 2 applications for comparing the performance of the Skip list based Lock-free approach with Heap-based fine-grained approach

1. Parallel SSSP run-time using these Priority Queues
   The reason why this metric is an interesting measure for the comparison is :
   a) The heap-based implementation supports ChangeKey operation, which better facilitates the needs of SSSP algorithm.
   b) Skip-List based implementation even though doesn't go well with SSSP algorithm (no support for ChangeKey operation) can still be used for SSSP with additional overhead in the algorithm. But the advantage here is that it inherently is lock-free and is faster than Heap-Based Implementation.

2. Core operations performance

   In this, we have inserted 1,00,000 nodes initially with keys chosen uniformly randomly in the integer range. And then each thread randomly decides whether to insert() a node or extractMin() with equal probability. Each thread repeats this 'k' times. Therefore N*k operations of which on expectation half of them are insert() and half are extractMin(). And we make sure the no of nodes in the Priority Queue doesn't deviate by much so that the comparison is fair.

   And with this process in mind, the is takenge time taken for insert() and extractMin() is calculated.

Design

1. Heap-Based Implementation
    a. This uses careful locking of individual nodes

    Deadlock Freedom

    Except for bubbleUp(), all the other operation acquire the lock to the nodes in the same order which is predetermined. bubbleUp() uses the tryLock() and releases the acquired lock if it fails to acquire the second lock on tryLock() thus avoiding deadlock.

    Linearizability

    CHAMP (the name of heap-based implementation) is linearizable. The following are the linearization points for each method.

    a) When there 2 or fewer nodes, the linearization point for peek(), extractMin(), insert() is when they acquired the root lock
    b) insert() and changeKey() which decreased the priority of e: the linearization point happens during the call to bubbleUp(e),,
    c) changeKey()

    Data Structure Invariants

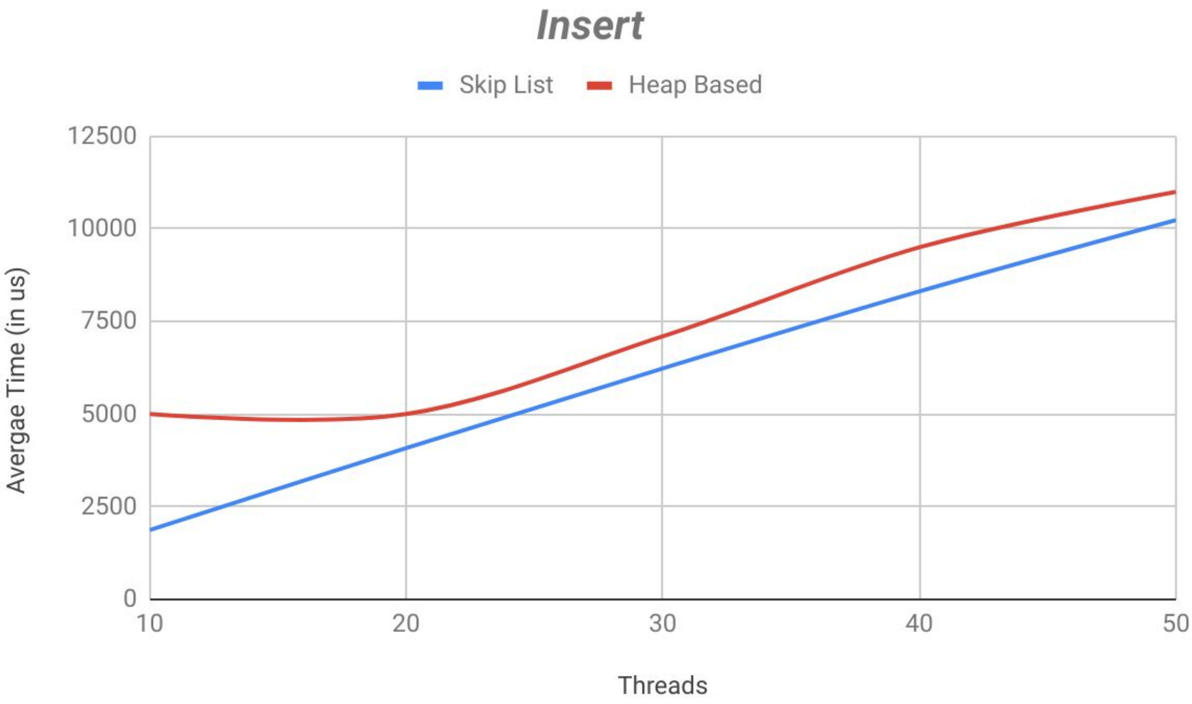
    a) no element in position
    b) The entries A[1].. A[Last] contain Non-NULL values
    c) The pos field of the nodes agrees with the actual position of the node in the heap.

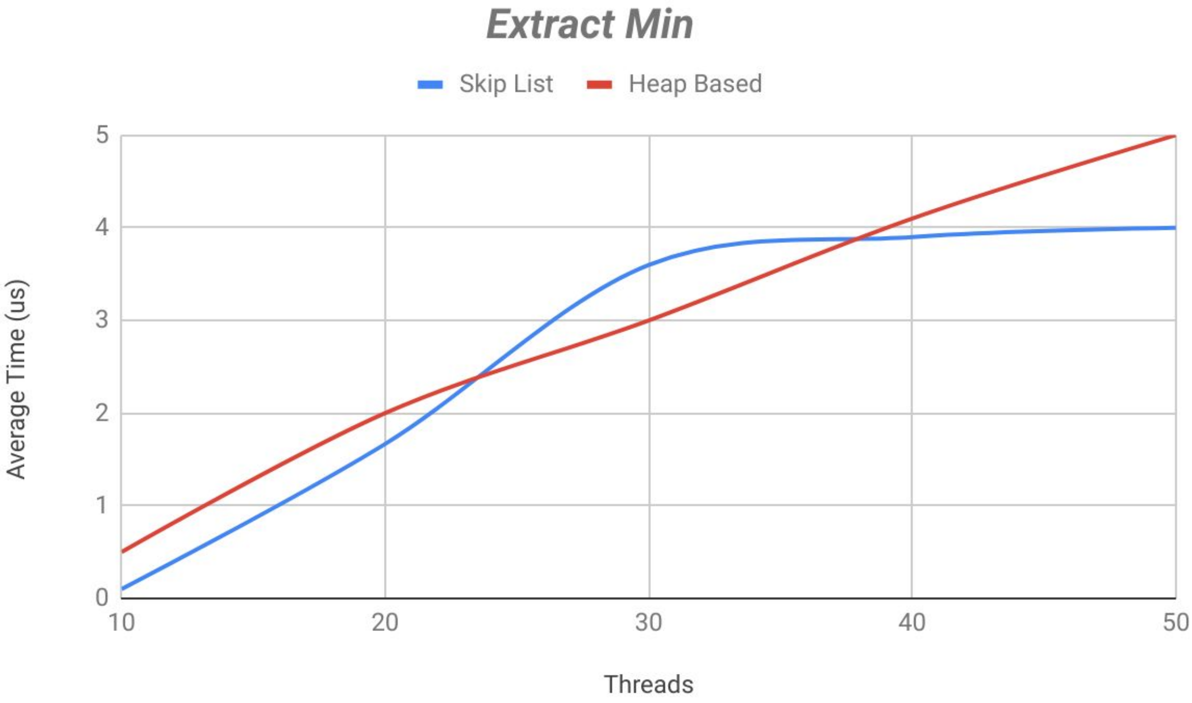
2. SkipList Based Implementation

- In Skip List based implementation, the algorithm lazily deletes the node and only deletes when no other thread is accessing the current node.
- For this, a reference counter is used with every node which keeps track of the outstanding references made to this node by other nodes.

Graphs and Analysis

Core Operations (Scenario 1):





Analysis:

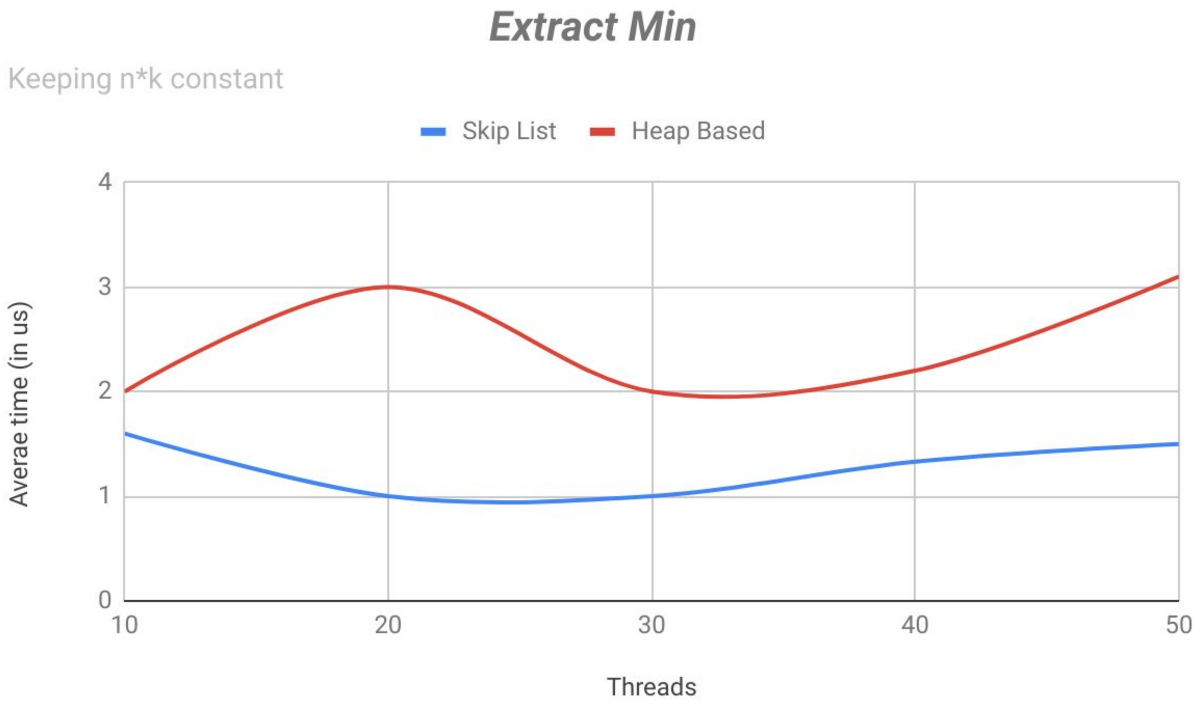
1. Here note that Skip-List performs better than Heap-Based Priority Queue. This is due to the fact that the overhead incurred by the parallelization is high in Heap-Based due to fine-grained locking.
2. As Skip-List Based implementation is lock-free and also offers randomized O(logN) complexity (due to the no of levels being kept equal to logN approximately in randomized skiplist), is faster than Heap-based implementation.
3. This is clear especially when the number of threads is higher.

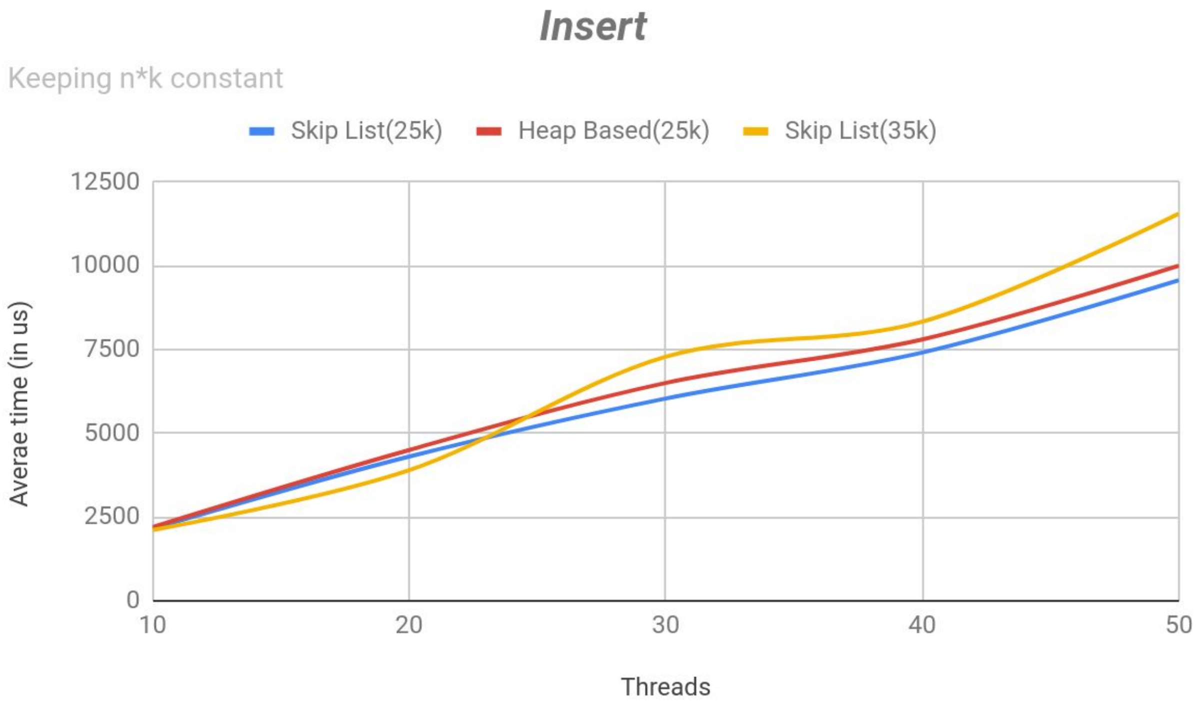
Analysis (extractMin):

1. Clearly the extractMin operations of both implementation are O(logN). (due to the no of levels being kept equal to logN approximately in randomized skiplist, bubbleDown).
2. And hence again (lock-free property) Skip-List based implementation performs better.

Note: Each thread executes k operations (constant) and hence the graph is sub-linear increasing curve. If total work is constant the graph would be decreasing (Scenario 2)

Core Operations (Scenario 2):

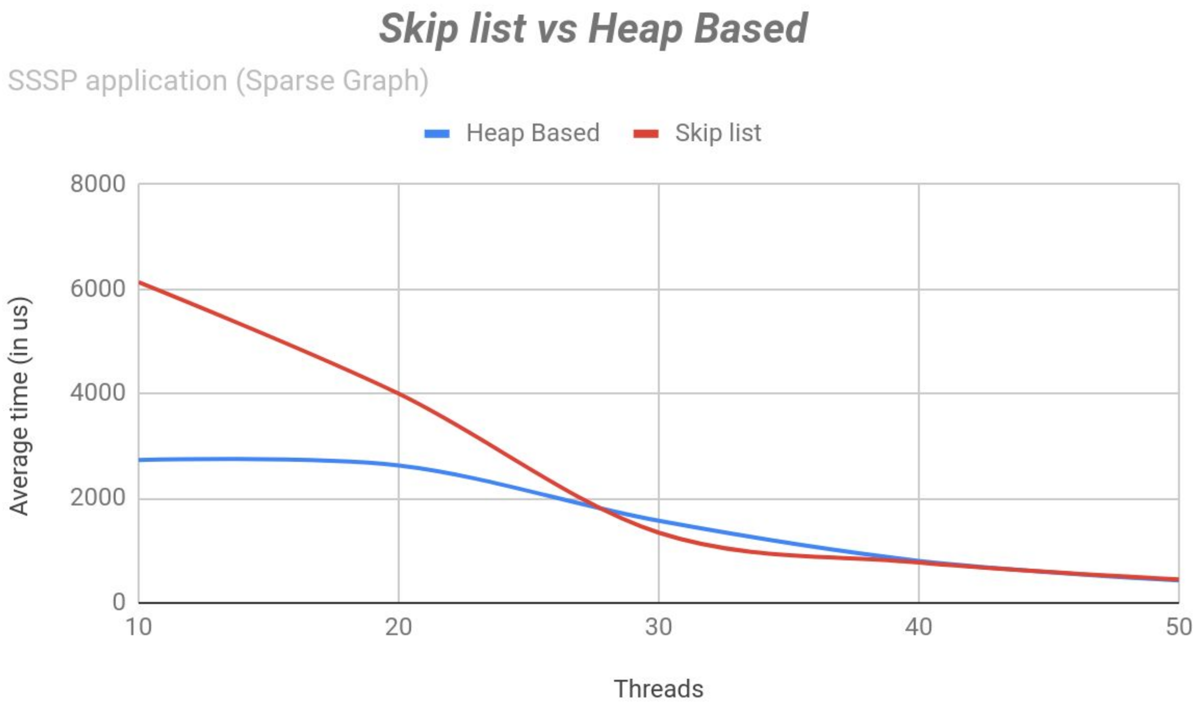




Observations:

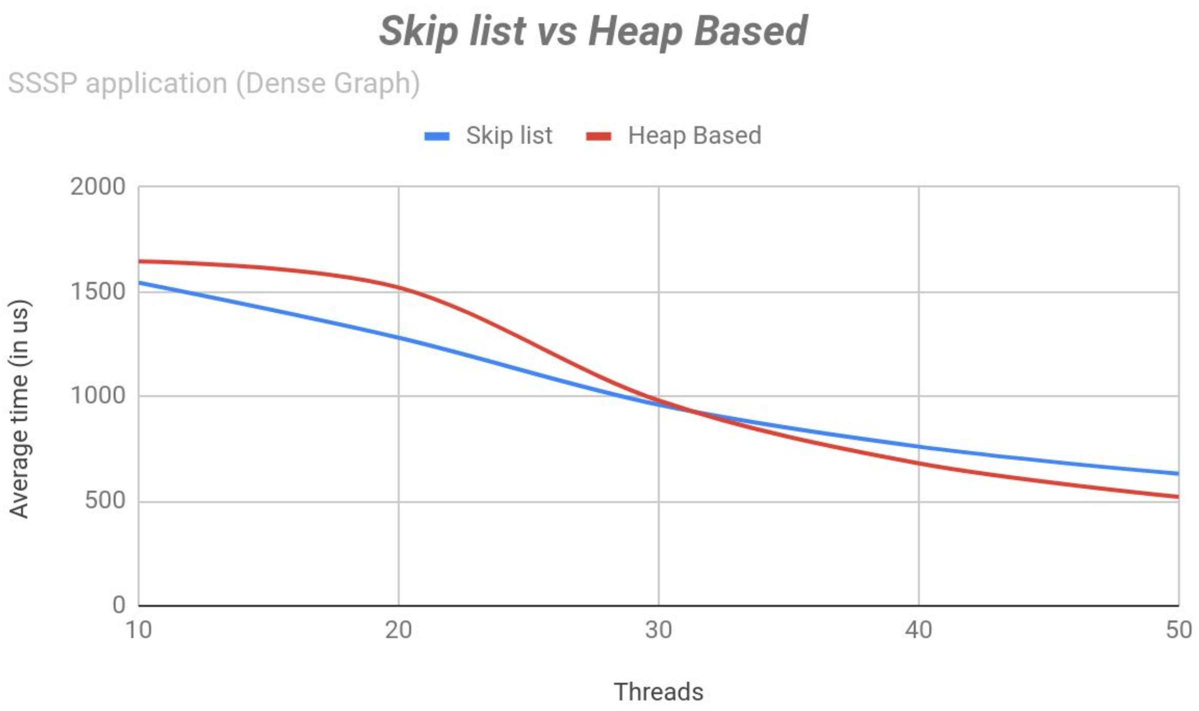
1. Skip lists based priority queue works better as it is lock free and lock free implementations inherently give a better performance
2. As expected when more number of instructions are processed, the performance degrades as shown by the blue and yellow color lines in the graph.
3. There is an increase in time with increase in number of threads, this can be explained by the extensive context switching which happen to accommodate all the threads.

SSSP Application:



Observations:

1. The Input Graph is a randomly generated graph with 400 Nodes and 400*80 edges.
2. The weights are randomly generated between 0 and 200.
3. A basic observation is that we see there is a decrease in running time of the algorithm with an increase in no of threads indicating parallelization.
4. With few threads, the Heap-Based Priority queue that leverages ChangeKey() operation still performs worse than Skip-List Based Implementation due to the inefficient core operations. As the no of threads increases, the Heap-Based Implementation slowly starts performing better than Skip-List implementation.



Analysis

- Lock free implementations perform better than lock based implementations
- This is because, lock free implementations gives more scope for parallelism.
- It was observed during the completion of the project, for lock free lazy deletion of a node from a linked list is the best method to gain performance.

Errors encountered

- We tried to implement this **Paper**, which had a very simple idea to physically delete the logically deleted nodes only after a certain threshold number of logical deleted nodes accumulate.
- Though simple in theory, the memory management required to realize the performance was very complex

Team Members:

Bhanu Prakash Thandu (CS15BTECH11037)
Abhinav Gupta (ES15BTECH11002)