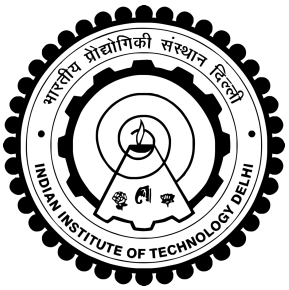


Indian Institute Of Technology Delhi
Dr. Smruti Ranjan Sarangi
Department of Computer Science and Engineering

A Simple Optimistic Skiplist Algorithm

COL818: Assignment 3

Author:
Shivam Verma, 2020CS50442

April 20, 2024

Contents

1 Introduction

We have implemented the paper "A Simple Optimistic skip-list Algorithm" in C++, authored by Yossi Lev, Maurice Herlihy and Victor Luchangco. The orginal paper can be found here.

Skiplists: A skiplist is a data structure that provides fast search, insertion, and deletion operations, similar to a balanced tree but with simpler implementation and typically better performance for certain operations. It consists of linked lists with multiple levels of nodes, where each level represents a subset of the nodes in the lower level.

The lowest level contains all the elements in sorted order. Each higher level is created by randomly 'skipping' elements from the lower level, forming shortcuts. This skipping mechanism allows for faster traversal during search operations, as it reduces the number of nodes that need to be visited.

The top level contains only a few elements, acting as an entry point to the lower levels. This structure resembles a staircase, where the lower levels have more elements and the higher levels have fewer elements.

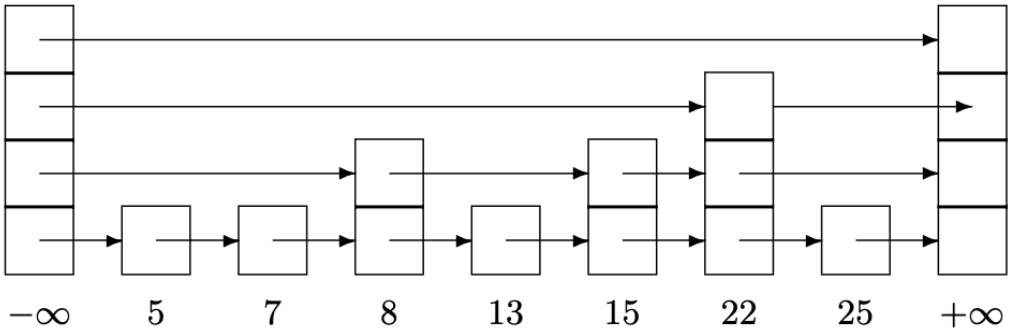


Figure 1. This is an example of Skiplist

We implemented the *add*, the *remove* and the *contains* operation.

1. **add(v):** adds v to the set and returns true iff v was not already in the set.
2. **remove(v)**: removes v from the set and returns true iff v was in the set.
3. **contains(v)**: returns true iff v is in the set.

The add() and remove() functions are lock-based whereas, contains(v) is wait-free.

2 Implementation

2.1 Node

This is how each node is initialised in our implementation of skip list.

```
1  struct Node;
2
3  struct lSentinel {};
4  struct rSentinel {};
5
6  struct Node {
7      int topLayer;
8      bool fullyLinked;
9      bool removed;
10     std::mutex lock;
11     std::vector<std::shared_ptr<Node>> nexts;
12     int value;
13
14     Node(int topLayer, bool fullyLinked, int value) :
15         topLayer(topLayer), fullyLinked(fullyLinked),
       removed(false), value(value) {
16         nexts.resize(topLayer + 1);
17     }
18 };
```

2.2 OptimisticSkipList

The OptimisticSkipList class with some of the member functions:

```
1  class OptimisticSkipList {
2
3  private:
4  std::shared_ptr<Node> head;
5  const int maxHeight = 128;
6
7  int findNode(int v, std::vector<std::shared_ptr<Node>>& preds,
       std::vector<std::shared_ptr<Node>>& succs) {
8      auto pred = head;
9      int found = -1;
10     for (int layer = maxHeight - 1; layer >= 0; --layer) {
11         auto curr = pred->nexts[layer];
12         while (curr->value < v) {
13             pred = curr;
14             curr = pred->nexts[layer];
15         }
16         if (found == -1 && v == curr->value) {
17             found = layer;
18         }
19         preds[layer] = pred;
20         succs[layer] = curr;
21     }
22     return found;
23 }
24
```

```
25  int randomLevel(int foo) {
26      std::random_device rd;
27      std::mt19937 gen(rd());
28      std::uniform_int_distribution<> dis(0, foo - 1);
29      return dis(gen);
30  }
31
32  bool okToDelete(std::shared_ptr<Node> candidate, int l) {
33      return candidate->fullyLinked && candidate->topLayer == l &&
        !candidate->removed;
34  }
35
36  void unlock(std::vector<std::shared_ptr<Node>>& preds, int
        highestLocked) {
37      std::shared_ptr<Node> prevPred, pred;
38      for (int layer = 0; layer <= highestLocked; ++layer) {
39          pred = preds[layer];
40          if (pred != prevPred) {
41              pred->lock.unlock();
42          }
43          prevPred = pred;
44      }
45  }
46
47  public:
48  OptimisticSkipList() {
49      auto h = std::make_shared<Node>(maxHeight, true,
        std::numeric_limits<int>::min());
50      auto t = std::make_shared<Node>(maxHeight, true,
        std::numeric_limits<int>::max());
51
52      for (int i = 0; i < maxHeight; ++i) {
53          h->nexts[i] = t;
54      }
55
56      head = h;
57  }
58  }
```

2.3 Add method

```
1  bool add(int v) {
2      int topLayer = randomLevel(maxHeight);
3      std::vector<std::shared_ptr<Node>> preds(maxHeight);
4      std::vector<std::shared_ptr<Node>> succs(maxHeight);
5      while (true) {
6          int found = findNode(v, preds, succs);
7          if (found != -1) {
8              auto nodeFound = succs[found];
9              if (!nodeFound->removed) {
```

```
10                  while (!nodeFound->fullyLinked) {}
11                  return false;
12              }
13              continue;
14          }
15          int highestLocked = -1;
16          std::shared_ptr<Node> pred, succ, prevPred = nullptr;
17          bool valid = true;
18          for (int layer = 0; valid && (layer <= topLayer); ++layer)
        {
19              pred = preds[layer];
20              succ = succs[layer];
21              if (pred != prevPred) {
22                  pred->lock.lock();
23                  highestLocked = layer;
24                  prevPred = pred;
25              }
26              valid = !pred->removed && !succ->removed &&
        pred->nexts[layer] == succ;
27          }
28          if (!valid) {
29              unlock(preds, highestLocked);
30              continue;
31          }
32
33          auto newNode = std::make_shared<Node>(topLayer, false, v);
34
35          for (int layer = 0; layer <= topLayer; ++layer) {
36              newNode->nexts[layer] = succs[layer];
37          }
38          for (int layer = 0; layer <= topLayer; ++layer) {
39              preds[layer]->nexts[layer] = newNode;
40          }
41
42          newNode->fullyLinked = true;
43          unlock(preds, highestLocked);
44          return true;
45      }
46  }
```

The successful add() operation is linearized at line 42, when *newNode -> fullyLinked* is marked True.

2.4 Remove method

```
1  bool remove(int v) {
2      std::shared_ptr<Node> nodeToDelete;
3      bool isremoved = false;
4      int topLayer = -1;
5      std::vector<std::shared_ptr<Node>> preds(maxHeight);
```

```
 6      std::vector<std::shared_ptr<Node>> succs(maxHeight);
 7      while (true) {
 8          int found = findNode(v, preds, succs);
 9          if (isremoved || (found != -1 && okToDelete(succs[found],
       found))) {
10              if (!isremoved) {
11                  nodeToDelete = succs[found];
12                  topLayer = nodeToDelete->topLayer;
13                  nodeToDelete->lock.lock();
14                  if (nodeToDelete->removed) {
15                      nodeToDelete->lock.unlock();
16                      return false;
17                  }
18                  nodeToDelete->removed = true;
19                  isremoved = true;
20              }
21              int highestLocked = -1;
22              std::shared_ptr<Node> pred, succ, prevPred = nullptr;
23              bool valid = true;
24              for (int layer = 0; valid && (layer <= topLayer);
       ++layer) {
25                  pred = preds[layer];
26                  succ = succs[layer];
27                  if (pred != prevPred) {
28                      pred->lock.lock();
29                      highestLocked = layer;
30                      prevPred = pred;
31                  }
32                  valid = !pred->removed && pred->nexts[layer] ==
       succ;
33              }
34              if (!valid) {
35                  unlock(preds, highestLocked);
36                  continue;
37              }
38              for (int layer = topLayer; layer >= 0; --layer) {
39                  preds[layer]->nexts[layer] =
       nodeToDelete->nexts[layer];
40              }
41
42              nodeToDelete->lock.unlock();
43              nodeToDelete.reset();
44
45              unlock(preds, highestLocked);
46              return true;
47          } else {
48              return false;
49          }
50      }
51  }
```

The remove() operation, likewise calls *findNode* to determine whether a node with the appropriate key is in the list. If so, the thread checks whether the node is "okay to delete", which means it is fullylinked, notmarked, and it was found at its toplayer. If the node meets these requirements, the thread locks the node and verifies that it is still not marked. If so, the thread marks the node, which logically deletes it: that is, the marking of the node (lines 18) is the linearization point of the remove operation.

2.5 Contains method

The wait-free contains() method is implemented as:

```
1  bool contains(int v) {
2      std::vector<std::shared_ptr<Node>> preds(maxHeight);
3      std::vector<std::shared_ptr<Node>> succs(maxHeight);
4      int found = findNode(v, preds, succs);
5      bool f =  ((found != -1) && (succs[found]->fullyLinked) &&
       (!succs[found]->removed));
6      return f;
7  }
```

Contains() method returns true if and only if it finds a unmarked, fully linked node with the appropriate key.

3 Performance

We tested this C++ implementation of Skip-list algorithm in:

1. HPC with 18 cores (Haswell Nodes).
2. MacBook Air M1 with 8 cores (4 performance cores and 4 efficiency cores).

Different experiments were run and throughput in operations per millisecond was calculated for each experiment to test the performance.

In the following experiments, starting from an empty skip-list, each thread executes one hundred thousand (1,00,000) randomly chosen operations. We varied the number of threads, the relative proportion of add, remove and contains operations, and the range from which the keys were selected. The key for each operation was selected uniformly at random from the specified range.

Two set of ranges were selected, one with key range [1 to 20,000] and other with [1 to 200,000].

Note

Theoretically, the latter range ([1 to 200,000]) has low level of contention as compared to the former one ([1 to 20,000]), and therefore should have higher throughput. But the experiments we conducted and also in the original paper's results, the latter range had higher throughput. The reasoning for this is provided later.

For each key ranges we ran 3 experiments, where the relative proportion of the *add*, the *remove* and the *contains* were varied.

1. 9% add, 1% remove, and 90% contains operations → Case 1
2. 20% add, 10% remove, and 70% contains operations → Case 2
3. 50% add, 50% remove, and 0% contains operations → Case 3

Let there be n threads, each performing k operations. Let t be the total time (in milliseconds) taken between spawning each thread and the point where each thread successfully completes its task.

Then the net throughput of the system is given by $\frac{n \times k}{t}$ operations per milliseconds.

3.1 Results in HPC

Our HPC system had 18 cores (Haswell Nodes). The figure 2 shows the plot for *Number of Threads vs Throughput* for *key Range: 20,000* and figure 3 shows the same plot for *key Range: 2,00,000*.

For both of the key ranges, the highest throughput is achieved in the case of 9% add, 1% remove, and 90% contains operations (blue plot). Since the *contains()* function is wait free, but *add()* and *remove()* function acquire locks, and this test experiment has low proportions of *add()* and *remove()* functions, it is expected to have high throughput, as can be seen in the plots.

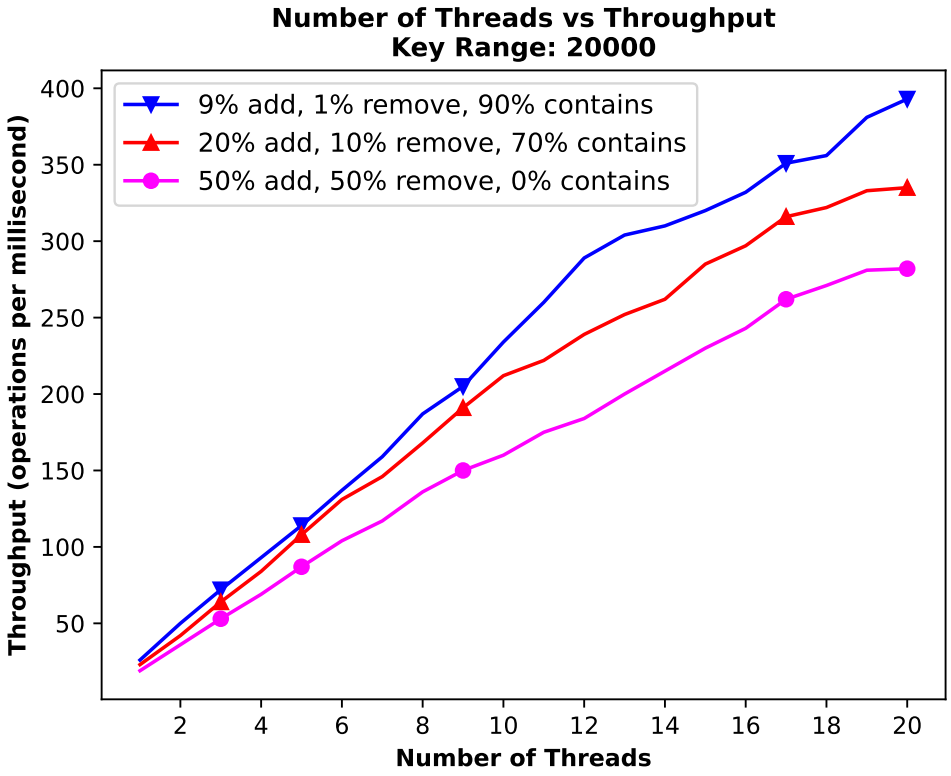


Figure 2. HPC: *Number of Threads vs Throughput* for *key Range: 20,000*

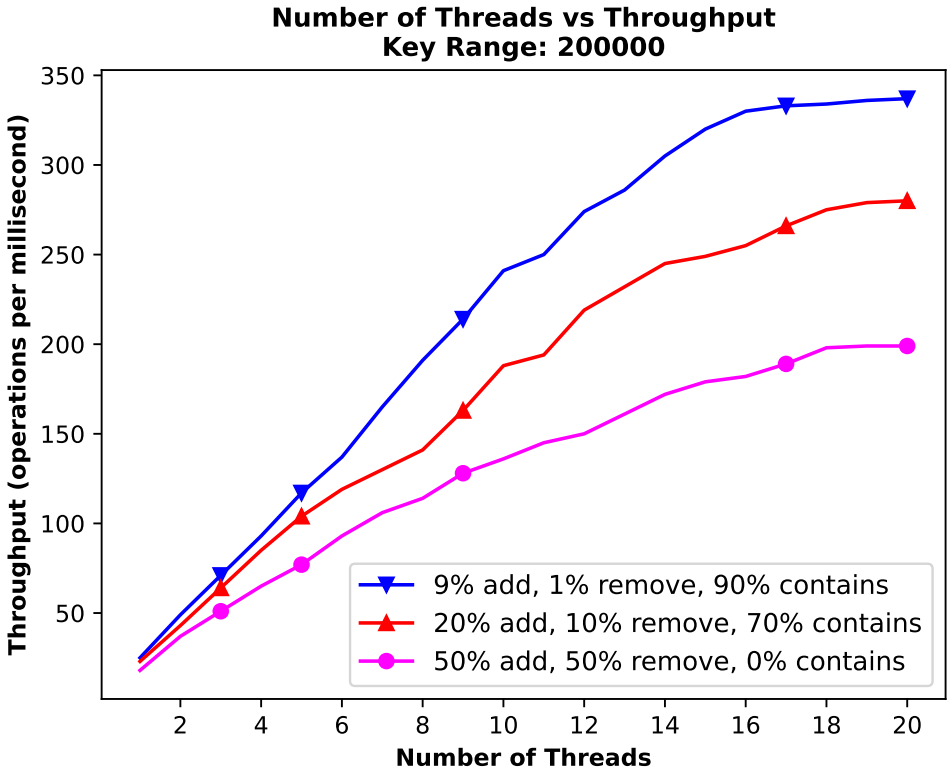


Figure 3. HPC: *Number of Threads vs Throughput* for *key Range: 2,00,000*

When we decrease the proportion of the *contains()* operations and increase the other two operations, there is an increase in contention. Multiple threads are fighting to acquire the lock and hence the net throughput of the system decreases as can be seen in the plots.

When we tested with 50% add, 50% remove, and 0% contains operations, we obtained least throughput. Since the add and remove operations require that the predecessors seen during the search phase be unchanged until they are locked, we expect that under high contention, they will repeatedly fail. Thus the throughput under such high contention is comparatively low.

On increasing the number of threads, the throughput also increases which is obvious as now more work can be done in parallel. But beyond a certain limit there is not much significant increase in throughput despite increasing the number of threads. The thoughput saturates when number of threads is increased beyond 18, as can be seen in the graphs. This is because our system only had 18 cores, which means more than 18 threads can't run simultaneously. The effects of context switching by the Operating System comes into picture here.

If we increase the number of threads significantly beyond the number of cores, the performance will go down. In this case, most of the threads won't be getting any available CPU to run, since other threads have already acquired the CPUs. Thus, beyond a certain limit the throughput graph sinks down.

Reason for the Lower Throughput with the higher Key-Range:

On comparing the throughput between key range [1 to 20,000] and [1 to 2,00,000], one would expect that the latter case would have high throughput since there would be low contention when the key range is high.

But the plots show different behaviour. For a given number of threads the throughput in case of key range $= 20,000$ (figure 2) is higher than that of key range $= 2,00,000$ (figure 3).

The reason for the lower throughput with the higher key-range is simply because the set is much bigger, where the range of its keys is limited to 2,00,000 vs 20,000.

Note that even though we only ran 100k operations, out of which 70% are lookups that do not change the set size, these are 100k operations per-thread (and we ran with up to dozens of threads), so the total number of operations is quite high comparing to the range when the number of threads is high enough. With a 20:10 add/remove operations ratio, the set will quite quickly converge to a size of approximately 66% of the range, i.e. 13k for small range, and 130K with the higher range.

Despite the logarithm time access, a factor of 10 in size is substantial and result in longer time per operation, regardless of contention. The clock time per operation is going to be lower with the smaller set, hence the higher throughput.

3.2 Results in MacBook Air M1

Our MacBook Air M1 had 8 cores (4 performance cores and 4 efficiency cores).

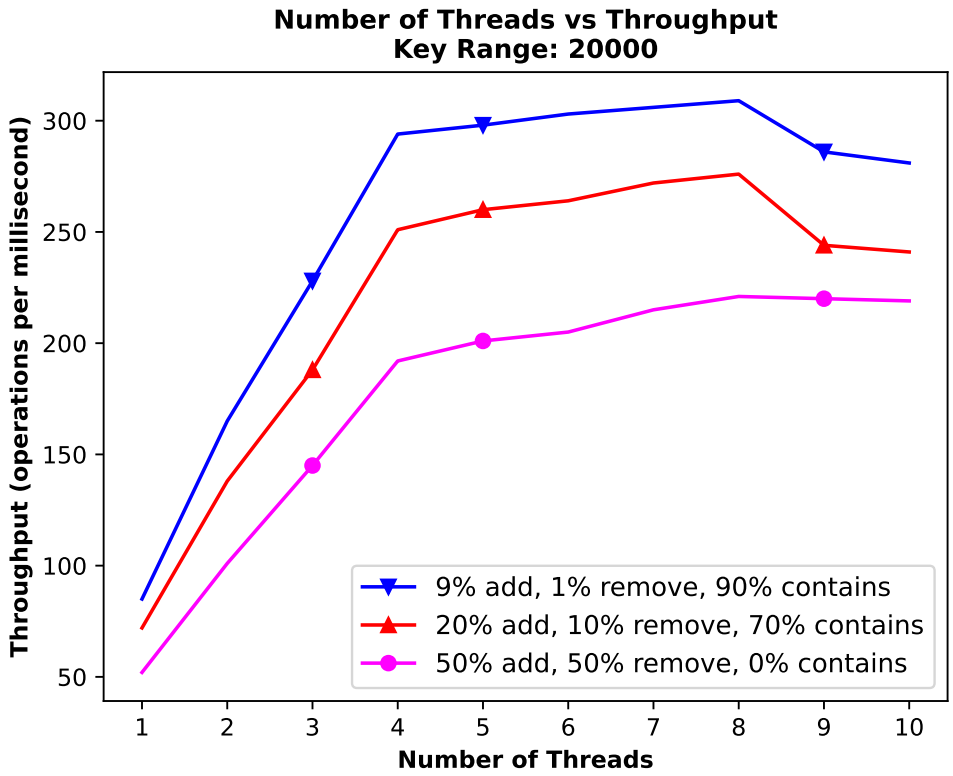


Figure 4. MacBook: *Number of Threads vs Throughput* for *key Range: 20,000*

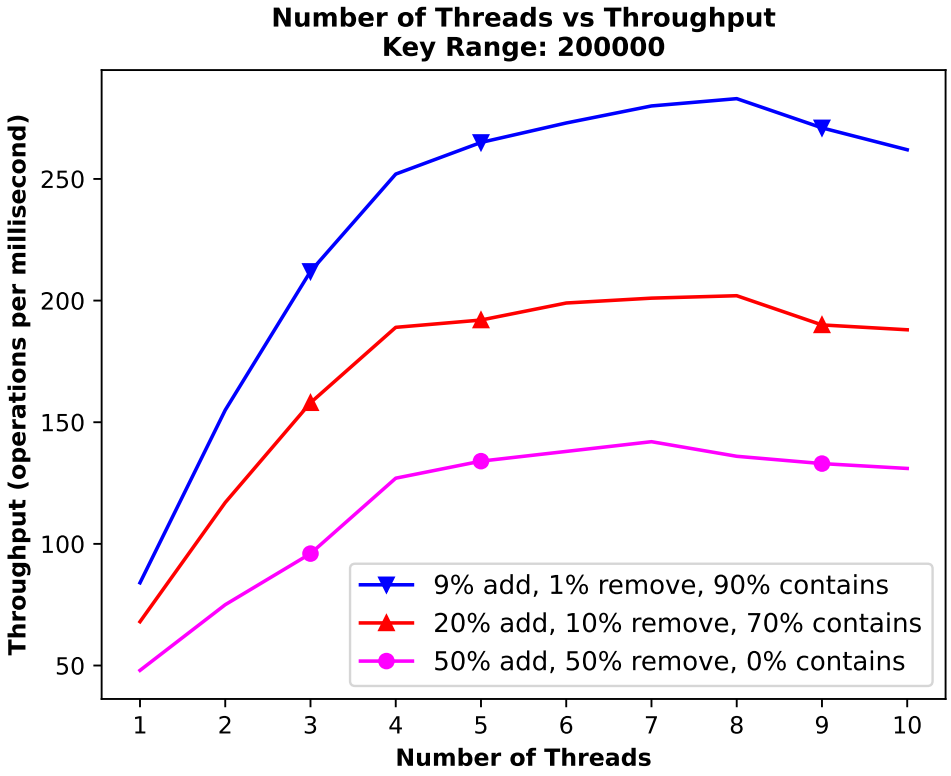


Figure 5. MacBook: *Number of Threads vs Throughput* for *key Range: 2,00,000*

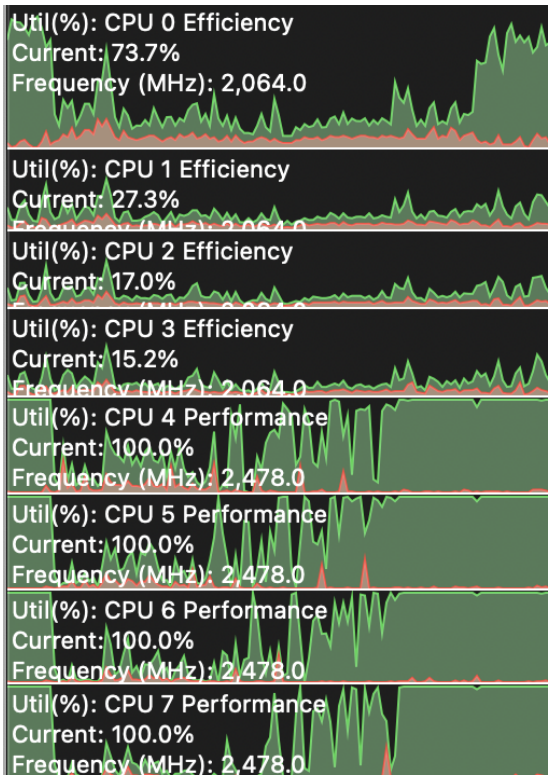
The trends observed while running experiments on MacBook are similar to the one obtained in HPC.

Please note that the throughput values relative to this case are significantly higher than those of the HPC system. This arises from the fact that the CPU in the MacBook machine is considerably more powerful and efficient than the Haswell Nodes in the HPC system. The elevated Instructions per Cycle (IPC) value of the MacBook enables it to complete more work in fewer clock cycles, consequently yielding a higher relative throughput.

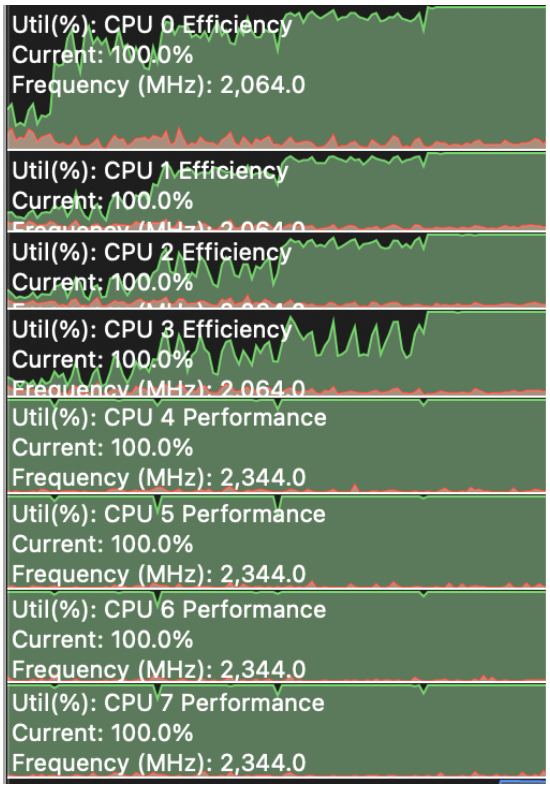
Another important trend observed in plots 4 and 5 is that the increase in throughput is not significantly pronounced beyond 4 threads. Up to the point where the number of threads is increased to 4, a notable increase in throughput is evident; however, beyond this threshold, although there is an increase, it is not as significant.

This observation can be explained by considering the fact that not all 8 cores of the MacBook machine are equally powerful. The first 4 cores are performance cores, typically featuring higher clock speeds designed to handle computationally intensive tasks that demand high processing power. The remaining 4 cores are efficiency cores, which operate at lower clock speeds and are optimized for low-power tasks and background processes that do not require significant computational resources.

Whereas in case of HPC we didn't observe any such distinction, since all the 18 cores are identical in performance.



(a) Running experiments with 5 threads



(b) Running experiments with 8 threads

Figure 6. CPU consumption in MacBook Air M1

The above figure shows the CPU consumption when the experiments were ran for 5 and 8 threads respectively. Note that first 4 cores are Performance cores and rest 4 are Efficiency cores.

Increasing the number of threads beyond 8 leads to some decrease in net throughput, due to the effects of context switching.

4 Conclusion

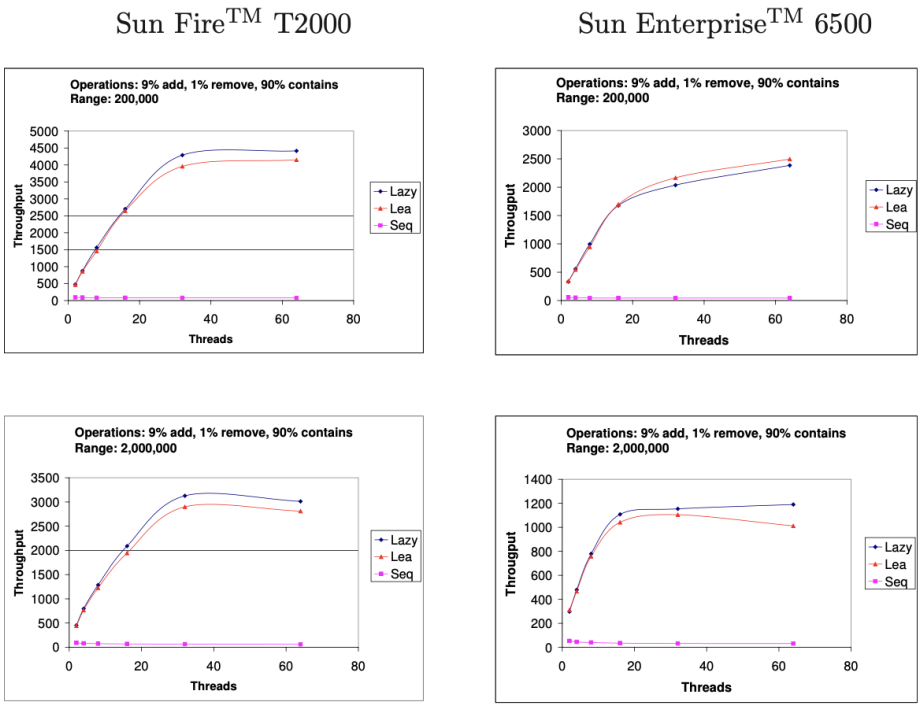


Figure 7. Throughput in operations per millisecond of 1,000,000 operations, with 9% add, 1% remove, and 90% contains operations, and a range of either 200,000 or 2,000,000.

The above figure is from the experiment results of the original paper. The trends observed in our implementation are similar to the original plots.