Project Report

for

Project 13 - Skiplist

as part of VU on

Advanced Multiprocessor Programming

SS 2017

Philipp Paris

e1325664@student.tuwien.ac.at

1325664

Emmanuel Pescosta

e1326934@student.tuwien.ac.at

1326934

26 June 2017

1 Description / Strategy

A Skiplist is basically a collection of sorted list nodes, where each node has a link to one or more following nodes. These links form shortcuts which can be used to speed up the searching by skipping one or more nodes. In contrast to tree-based search structures randomized Skiplists don't require re-balancing, thus making the concurrent implementation easier.

The links are arranged in multiple levels. Level 0 is basically a linked-list of all inserted nodes sorted by value. The higher the level, the less nodes are interconnected meaning that the expected number of nodes on each level decreases exponentially. At the beginning and at the end of each level the `head` and `tail` sentinel nodes are placed. The `head` node is filled with the smallest and `tail` node with the biggest possible value. These sentinel nodes avoid some special cases in various Skiplist operations.

Nodes can be found by starting at the maximum level and travel along the nodes of the current level as long as possible, thus exploiting the shortcut links. Whenever the value of the inspected node is bigger than the desired value, the level has to be decreased by one and the lookup can go on further.

The randomized Skiplist uses coin flipping to probabilistically determine the height of a new node. The coin flipping is repeated until head is reached, thus the node will have a height of h with a probability of $\frac{1}{2^h}$. The height of the Skiplist, which is the maximum height of all nodes in the list, is $\mathcal{O}(\log n)$ with high probability. Given that the `insert`, `remove` and `contains` methods all use the internal `find` method to search for a node as well as its predecessors and successors, the maximum bounds of these methods is equal to the maximum bound of the `find` method because the remaining parts of these methods take constant time. Given that `find` either travels down the levels of a node (bounded by height $\mathcal{O}(\log n)$) or travels along on the nodes on one level, the `find` method is bounded by $\mathcal{O}(\log n)$ with high probability. Thus the `insert`, `remove` and `contains` methods are all bounded by $\mathcal{O}(\log n)$ with high probability.

2 Implementation

Each concrete Skiplist implements the `SkipList<T>` interface shown in Listing 1, providing methods for inserting, removing and searching of values.

Listing 1: Skiplist Interface

```
1  template <typename T>
2  class SkipList
3  {
4    public:
5      static_assert(std::is_integral<T>::value,
6                    "T must be an integral type");
7
8      virtual bool empty() = 0;
9      virtual size_type size() = 0;
10     virtual bool insert(const_reference value) = 0;
```

```
11      virtual bool remove(const_reference value) = 0;
12      virtual bool contains(const_reference value) = 0;
13      virtual void clear() = 0;
14  };
```

For simplicity all the implementations only support integral types and can only handle unique values, meaning that adding the same value twice will fail on the second time. Additionally each type must have a specified minimum and maximum value, because these values will be used by the `head` and `tail` sentinel nodes respectively.

2.1 Sequential

The `SequentialSkiplist` implements the algorithm described in [1] with two small optimizations. The current maximum level of all inserted nodes is cached so that the unused levels can be skipped immediately. Additionally the next-node pointers are stored in a static array within the node object itself, thus eliminating one indirection which would occur when using a dynamic datastructure such as `std::vector`, with the small downside that it always allocates `MaxHeight` next-pointer slots even if some slots are unused.

2.2 Concurrent

The `ConcurrentSkiplist` is based on the sequential one and uses a `std::mutex` to control the concurrent access of multiple threads. The implementation simply forwards all requests to the internal instance of a sequential Skiplist after successfully grabbing the lock, thus serializing the access to the internal `SequentialSkiplist`.

2.3 Lazy

The LazySkipList was based on the algorithm discussed in [1]. Instead of serializing all accesses as in the `ConcurrentSkipList`, it uses a `std::mutex` per node to serialize as little as possible.

Linearizability

Additionally to the lock, each node contains two flags: `marked`, indicating that the node is virtually removed from the list and `fullyLinked`, which is set when the node is fully linked into the SkipList. These flags determine when a performed action (insert or remove) is visible to the other processes and therefore the moment when either the flag `marked` or `fullyLinked` is set to `true` is the linearization point of the respective method.

Progress Guarantees

Due to order in which the processes obtain and release the locks, it can be shown, that the implementation is deadlock-free: Each process acquires the lock of the predecessor from level 0 to max-level. Because of the way the nodes are linked in the SkipList, every process always acquires locks of nodes with higher keys first. So it cannot be, that two processes

want to acquire a lock which the other process holds and therefore the `LazySkipList` is deadlock-free.

While the insert and remove methods are blocking, the contains method of the `LazySkipList` is wait-free, which can be seen because it uses no locks and never has to restart it's execution.

Memory-Managed Version

In the standard implementations of the `LazySkipList` (and the `LockFreeSkipList`) the memory of nodes which are removed from the list is not deallocated. To fix this flaw, the `MMLazySkipList` uses `C++11` reference counted smart pointers to deallocate all removed nodes.

2.4 Lock-Free

The `LockFreeSkipList` [1] enables concurrent access without locks by using atomic `compareAndSet` operations for all list manipulations. Any list manipulations in the algorithm consists of changing the successor reference of a node while ensuring that the marked flag **and** the successor have not changed before. Because standard `compareAndSet` implementations only support to compare one value, we merge the reference and the marked flag into one `std::atomic_uintptr_t` by using one unused bit of the reference for our marked flag (see Listing 2).

Listing 2: AtomicMarkableReference

```
1  template <typename T>
2  class AtomicMarkableReference
3  {
4    public:
5      AtomicMarkableReference(T* ref = nullptr, bool marked = false)
6      {
7          value = ((uintptr_t)ref & ~mask) | (marked ? 1 : 0);
8      }
9
10     T* get(bool& marked)
11     {
12         uintptr_t tmp = value;
13         marked = (bool)(tmp & mask);
14         return (T*)(tmp & ~mask);
15     }
16
17     bool compareAndSet(T* oldRef, T* newRef, bool oldMarked, bool newMarked)
18     {
19         uintptr_t oldValue = ((uintptr_t)oldRef & ~mask) | (oldMarked ? 1 : 0);
20         uintptr_t newValue = ((uintptr_t)newRef & ~mask) | (newMarked ? 1 : 0);
21         return value.compare_exchange_strong(oldValue, newValue);
22     }
23
24   private:
25     std::atomic_uintptr_t value;
26     static const uintptr_t mask = 1;
27 };
```

Linearizability

As in the `LazySkipList` the linearization point for removing a list element is the moment when the marked flag is set to `true`. The linearization point for the insert operation is

defined in the `LockFreeSkipList` as the moment the node is linked in at the bottom level.

Progress Guarantees

The datastructure is lock-free because it is guaranteed that always some method call finishes in a finite number of steps: For the insert or remove operations, the only way this property is violated, if the `compareAndSet` methods fail, triggering a restart of the method. But this only happens if another process has successfully completed the `compareAndSet` method and thus made progress in its execution.

As in the `LazySkipList` the contains method is wait-free.

3 Experimental Setup

3.1 Saturn

Saturn is a shared memory parallel computer with 48 AMD CPU cores.

Table 1: Hard- and software configuration of Saturn

CPUs	4 AMD Opteron 6168 (12 cores, 1.9 GHz, 12 MB cache)
Main Memory	128 GB DDR3-1333 (currently only 120 GB available)
Operating system	Debian 4.7.6-1 (2016-10-07) x86-64 GNU/Linux
Compiler	gcc (Debian 6.3.0-6) 6.3.0 20170205

The hard- and software configuration of the test machine can be seen in Table 1. As compiler options we used `-O3`. We enabled `-DNDEBUG` and `-DCOLLECT_STATISTICS` while performing the benchmarks.

3.2 Simulated Workloads

Various different workloads have been implemented to simulate different insert, remove and search uses. All the workloads support strong and weak scaling. By using strong scaling the number of items per thread will decrease when the number of threads is increased. By using weak scaling the number of items per thread will always be the same independent of the number of threads. Listing 3 shows the algorithm to determine the number of items per thread used in each workload.

Listing 3: Items Per Thread depending on the Scaling Mode

```
1 static long itemsPerThread(const BaseBenchmarkConfiguration& config)
2 {
3     switch (config.scalingMode) {
4     case Scaling::Weak:
5         return config.numberOfItems;
```

```
6       case Scaling::Strong:
7           return config.numberOfItems / config.numberOfThreads;
8       }
9   }
```

3.2.1 Interleaving Insert

The workload shown in Listing 4 simulates the insertion of interleaving values by multiple threads, causing that the predecessor and successor nodes of a node inserted by the current thread are always inserted by other threads. This benchmarks stresses the lazy and lock-free implementations by exploiting the `fullyLinked` property.

Listing 4: Interleaving Insert Workload

```
1  const auto items = itemsPerThread(config);
2  long number = config.initialNumberOfItems + threadId;
3  for (long i = 0; i < items; i++) {
4      list.insert(number);
5      number += items;
6  }
```

3.2.2 Interleaving Remove

The list will be prefilled with $ItemsPerThread * NumberOfThreads$ items. The workload shown in Listing 5 simulates the removal of interleaving values by multiple threads, causing that the predecessor and successor nodes of a node removed by the current thread are always removed by other threads. This benchmarks stresses the lazy and lock-free implementations by exploiting the `marked` property.

Listing 5: Interleaving Remove Workload

```
1  const auto items = itemsPerThread(config);
2  long number = threadId;
3  for (long i = 0; i < items; i++) {
4      list.remove(number);
5      number += items;
6  }
```

3.2.3 Mix of Insertions, Removals and Searches

This workload precomputes random numbers which are uniformly distributed between 0 and $ItemsPerThread * NumberOfThreads$. These random numbers will be passed on to the different Skiplist operations when invoking the work function. Each thread does either insert, remove or search for items. The number of inserting, removing and search threads can be varied when creating the workload.

3.3 Skiplist Statistics

To get a better picture about the performance and behavior of the Skiplist, performance counters have been added to the `insert`, `remove` and `contains` methods of all Skiplist implementations. These performance counters collect the number of retries, which is especially interesting for the lazy and lock free implementations, as well as the number of

successful and failed invocations. Based on these performance counters multiple different metrics can be evaluated, such as the average number of retries of `insert`, `remove` and `contains`.

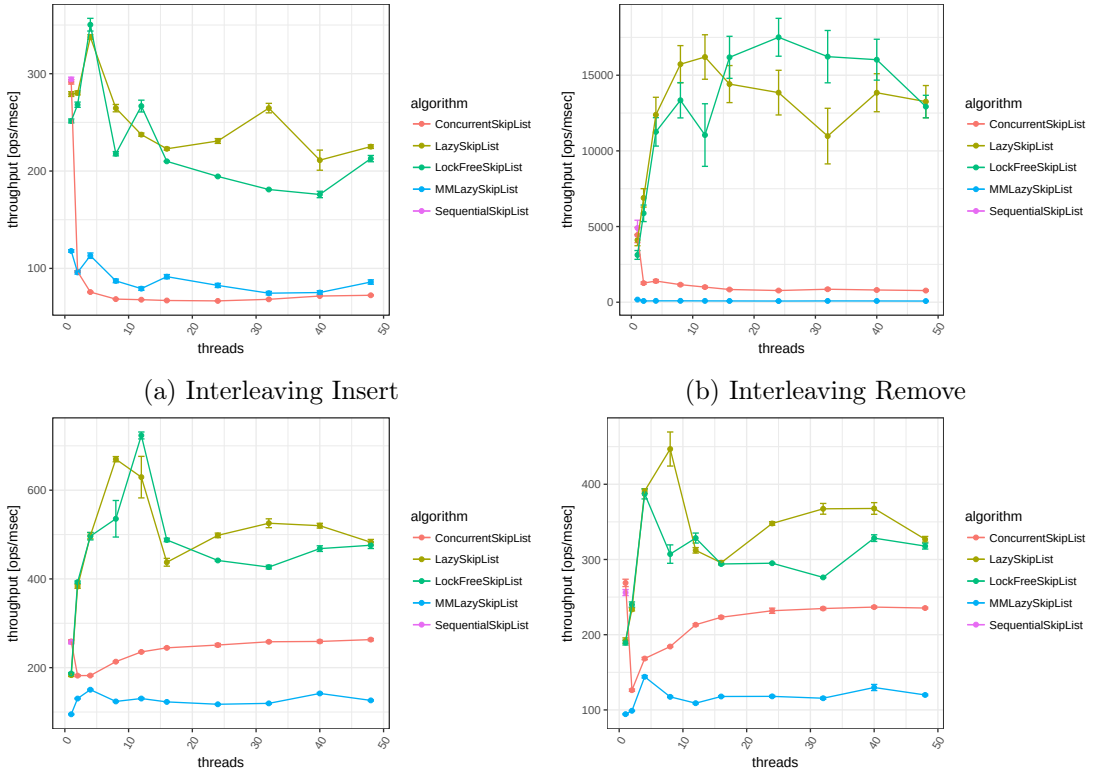
To minimize the possible influences on the benchmarking results, each thread collects the data in a thread-local statistics object, thus no locking and no atomics are required. At the end of each benchmark the thread-local statistics are aggregated by the benchmark suite to get the complete statistics.

The statistics collection can be completely turned off and thus doesn't add any overhead to production use.

4 Experimental Results

Each benchmark was repeated 30 times and the error bars of each plot show the 95% confidence interval. Each benchmark started with an empty list and inserted and/or removed 1000000 items. Strong scaling was used during benchmarking, thus the number of processed items didn't change with the number of threads. The maximum Skiplist height was 16. The benchmarks were performed using 1, 2, 4, 8, 12, 16, 24, 32, 40 and 48 threads. The execution time and Skiplist statistics of each repetition was collected and post-processed.

4.1 Throughput



(a) Interleaving Insert

(b) Interleaving Remove

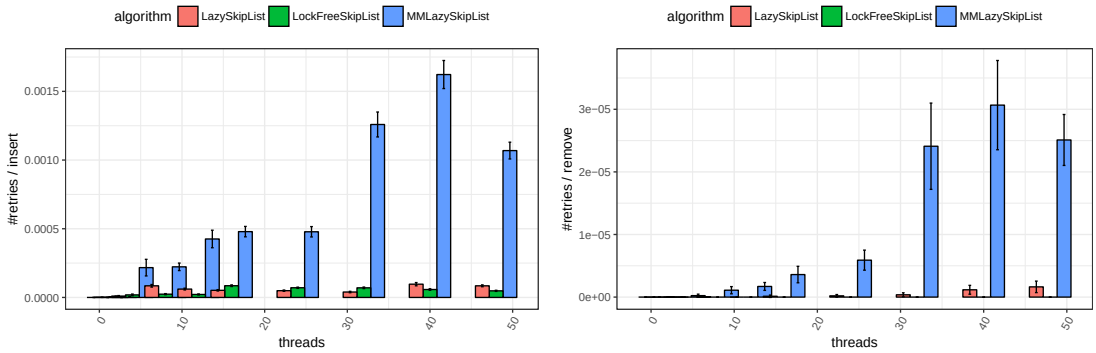
(c) 50% Insert, 20% Remove, 30% Contains

(d) 70% Insert, 30% Remove

Figure 1: SkipList Throughput of Different Algorithms under Different Workloads

4.2 Performance Counters

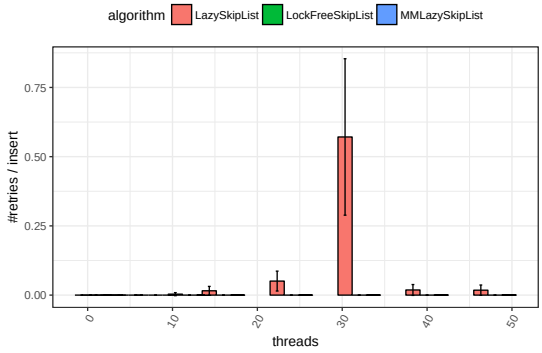
Interleaving Insert/Remove



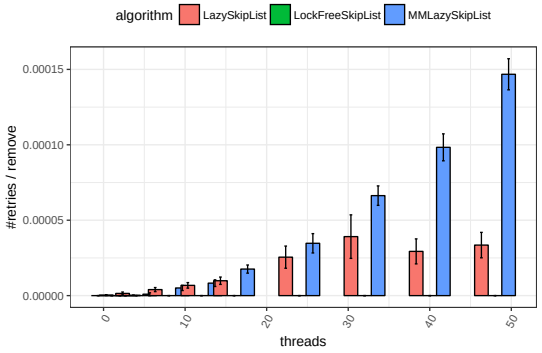
(a) #Retries per Insert

(b) #Retries per Remove

50% Insert, 20% Remove, 30% Contains

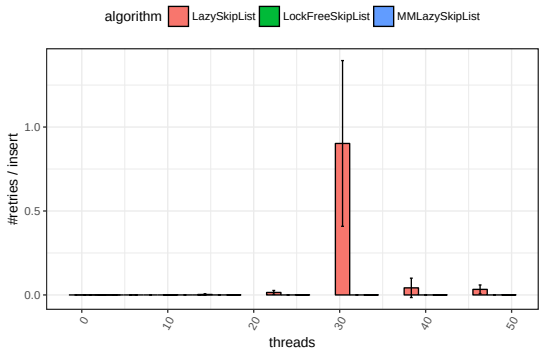


(a) #Retries per Insert

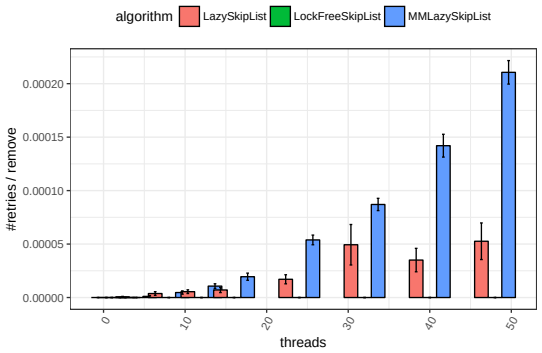


(b) #Retries per Remove

70% Insert, 30% Remove



(a) #Retries per Insert



(b) #Retries per Remove

4.3 Discussion

The plots in Section 4.1 show the achieved throughput dependent on the number of threads and the used workload. As expected the `ConcurrentSkipList` performs worse than the `Lazy-` and the `LockFreeSkipList` due the complete serialization of all executed operations, while the `Lazy-` and `LockFreeSkipList` perform similarly. Surprisingly the `MMLazySkipList` performs even worse than the `ConcurrentSkipList` in most scenarios, which is probably because of the higher than expected overhead of the atomic smart-pointer operations and the high number of retries.

As seen in Section 4.2 the number of retries per operation increases with an ascending number of threads. This is expected because the more threads are used the more contention happens. `MMLazySkipList` performs much more retries than all the other implementations. We assume that the additional atomic smart-pointer overhead may be the reason, because

more time elapses between the start of the operation (loop) and the validation of the nodes. So it's more likely that another thread causes an invalidation and thus a retry. `LockFreeSkipList` doesn't perform any retries during removal as seen in Figure 2b and only a few during insert as seen in Figure 2a, thus the relatively small number of retries in the mixed-workload benchmarks. Interesting is the particular high number of retries during insert of the `LazySkipList` in the mixed-workload benchmark when using 30 threads as seen in Figures 3a and 4a. We haven't found a reason why this happened.

References

[1] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914.