# DD2443 H23

Group 7: Lukas Sebastian Beinlich, Aleksei Veresov

October 2023

## Compiling and Executing

All tasks can be compiled and executed using the following commands:

```
cd src/
javac *.java
java Experiment <scenario> [<valSeed> <opsSeed>]
```

Scenario can be either `local` or `dardel`. Values and operations seeds can be omitted, in this case current time in nanoseconds is used (and printed, for reproducibility).

In practice, it can be also useful to suppress error messages, this is done by redirecting the error stream. On Windows:

```
java Experiment <scenario> [<valSeed> <opsSeed>] 2>$null
```

On Linux:

```
java Experiment <scenario> [<valSeed> <opsSeed>] 2>/dev/null
```

## 1 Measuring time

This task is pretty straightforward, here is the code that measures the time for all scenarios, we simply loop over all cases and do the validation and time measurement, as well as warm up in one go:

```
for (Distribution values : new Distribution[] { new
    Distribution.Uniform(84, 0, MAX_NUMBER),
       new Distribution.Normal(84, 15, 0, MAX_NUMBER) })
{
    for (int[] distr : new int[][] { new int[] { 1, 1, 0 }, new int[] {
        1, 1, 0 } })
    {
        for (int num_threads : new int[] { 1, 2, 4, 8, 16, 32, 64, 96 })
        {
            int warmup = 0;
            for (int i = 0; i < warmup + 1; i++)
            {
                try
                {
                    // Create a standard lock free skip list
```

```
                    LockFreeSet<Integer> lockFreeSet = new
                        LockFreeSkipList<>();

                    // Create a discrete distribution with seed 42 such
                        that,
                    // p(0) = 1/10, p(1) = 1/10, p(2) = 8/10.
                    Distribution ops = new Distribution.Discrete(42,
                        distr);

                    // Run experiment with 16 threads.
                    long time = run_experiment(num_threads, MAX_NUMBER,
                        lockFreeSet, ops, values);
                    if (i == warmup) // after warmup
                        System.out.printf("Took %f ms to finish for %d
                            workers.\n", (time) / 1_000_000.0f,
                               num_threads);

                    // Get the log
                    Log.Entry[] log = lockFreeSet.getLog();

                    // Check sequential consistency
                    Log.validate(log, num_threads);
                }
                catch (Exception e)
                {
                    e.printStackTrace();
                    System.exit(1);
                }
            }
        }
    }
}
```

In this case the variable `MAX_NUMBER` is changed depending on if execution is on the PDC or locally.
The called function `run_experiment` simply executes a single iteration with the given amount of threads and given distributions and returns the total time the run took:

```
ExecutorService executorService = Executors.newFixedThreadPool(threads);
Task[] tasks = new Task[threads];
for (int i = 0; i < tasks.length; ++i)
{
    tasks[i] = new Task(i, count, list, ops.copy(i), values.copy(-i));
}
long startTime = System.nanoTime();
executorService.invokeAll(Arrays.asList(tasks));
long endTime = System.nanoTime();
executorService.shutdown();
return endTime - startTime;
```

The class `Task` is simply taking the distribution to sample the actions to take and then executing them the given amount of times, with the call function

looking like this:

```
for (int i = 0; i < count; ++i)
{
    int val = values.next();
    int op = ops.next();
    switch (op)
    {
    case 0:
        set.add(threadId, val);
        break;
    case 1:
        set.remove(threadId, val);
        break;
    case 2:
        set.contains(threadId, val);
        break;
    }
}
return null;
```

### Results

| Method | 1 Worker | 2 Workers | 4 Workers | 8 Workers |
|---|---|---|---|---|
| 10% Add, 10% Remove, 80% Contains | 24.788 ms | 31.232 ms | 40.175 ms | 56.725 ms |
| 50% Add, 50% Remove, No Contains | 45.607 ms | 61.779 ms | 72.323 ms | 95.276 ms |

All tables are for a uniform distribution from 0 to 100000, and we execute 100000 operations given the table above per thread on the skiplist.

As we can see, performance decreases slightly for more threads running at the same time. This is likely due to the underlying skiplist implementation, as with more threads we have more misses and have to "run more loops". This becomes especially apparent when we only use add and remove operations.

### Dardel

Due to maintenance, dardel results will be handed in later.

## 2 Identify and validate linearization points

### 2.1 Log Entry

Our log entry is a simple class to hold several interesting attributes:

```
public static class Entry
{
    int threadId;
    Method method;
    String argument;
    boolean retval;
    long timestamp;
```

```
    public Entry(int threadId, Method method, String argument, boolean
        retval, long timestamp)
    {
        this.threadId = threadId;
        this.method = method;
        this.argument = argument;
        this.retval = retval;
        this.timestamp = timestamp;
    }

    public String toString()
    {
        return "Entry{" + "threadId=" + threadId + ", method=" + method
            + ", argument='" + argument + '\'' + ", retval=" + retval +
            ", timestamp=" + timestamp + '}';
    }
}
```

`Method` here is simply an enum with the possible values `ADD`, `REMOVE` and `CONTAINS`. To use this we first create the (incomplete) entry, for example like this in the add function:

```
Log.Entry entry = new Log.Entry(threadId, Log.Method.ADD, x.toString(),
    false, -1);
```

This sets the return value to a default of false (all three methods only return a boolean) and sets timestamp to -1 (which is a placeholder). As soon as we reach a (potential) linearization point we save the timestamp into the lock, in the global lock log for example like this:

```
lock.lock();
try
{
    c = pred.next[bottomLevel].compareAndSet(succ, newNode, false,
        false);
    entry.timestamp = System.nanoTime();
}
finally
{
    lock.unlock();
}
```

Right before the return of the function we then set the return value and add the entry to the list:

```
private void submitEntry(Log.Entry e)
{
    lock.lock();
    try
    {
        log.add(e);
    }
    finally
```

```
    {
        lock.unlock();
    }
}
...
entry.retval = true;
submitEntry(entry);
return true;
...
```

Later on, to validate the log we first sort the log by timestamps and then we iterate over the sorted log, while checking for discrepancies:

```
...
Arrays.sort(sortedLog, Comparator.comparingLong(entry ->
    entry.timestamp));
...
for (Log.Entry logi : sortedLog)
{
    int val = Integer.parseInt(logi.argument); // TODO think
    boolean resSeq;
    switch (logi.method)
    {
    case ADD:
        resSeq = seqSet.add(val);
        break;
    case REMOVE:
        resSeq = seqSet.remove(val);
        break;
    case CONTAINS:
        resSeq = seqSet.contains(val);
        break;
    default:
        throw new RuntimeException("Unknown method: " + logi.method);
    }
    i++;
    if (resSeq == logi.retval)
        continue;
    System.out.println(i + " " + logi.method.toString() + "(" +
        logi.argument + "): value of lock free (" + logi.retval + ") not
        matching sequential (" + resSeq + ")");
    wrong += 1;
}
```

## 2.2 Global log with locks

The code for this task is in `src/LockFreeSkipList.java`.

To get the linearization points we first have to get the linearization points of the find method, because if the object is already in the list the linearization point is as soon as we find that object.

```
private boolean find(T x, Node<T>[] preds, Node<T>[] succs, Log.Entry
```

```
    entry)
{
    int bottomLevel = 0;
    boolean[] marked = { false };
    Node<T> pred = null;
    Node<T> curr = null;
    Node<T> succ = null;
    retry: while (true)
    {
        if (entry != null)
        {
            lock.lock();
            try
            {
                pred = head;
                entry.timestamp = System.nanoTime();
            }
            finally
            {
                lock.unlock();
            }
        }
        else
        {
            pred = head;
        }
        for (int level = MAX_LEVEL; level >= bottomLevel; level--)
        {
            curr = pred.next[level].getReference();
            while (true)
            {
                succ = curr.next[level].get(marked);
                while (marked[0])
                {
                    if (!pred.next[level].compareAndSet(curr, succ, false,
                        false))
                        continue retry;
                    if (entry != null)
                    {
                        lock.lock();
                        try
                        {
                            curr = succ;
                            entry.timestamp = System.nanoTime();
                        }
                        finally
                        {
                            lock.unlock();
                        }
                    }
                    succ = curr.next[level].get(marked);
                }
                if (curr.value != null && x.compareTo(curr.value) < 0)
                {
```

```
                pred = curr;
                if (entry != null)
                {
                    lock.lock();
                    try
                    {
                        curr = succ;
                        entry.timestamp = System.nanoTime();
                    }
                    finally
                    {
                        lock.unlock();
                    }
                }
            }
            else
            {
                break;
            }
        }

        preds[level] = pred;
        succs[level] = curr;
    }
    return curr.value != null && x.compareTo(curr.value) == 0;
  }
}
```

If the list is empty, we define the linearization point at the time we grab the head, as the other checks afterward are not executed anymore. Otherwise, the linearization point is at the moment we grab the object, which we will check at the end to be the same as the given object.

In the add function itself the linearization point is either when we find the object, or when we insert the object in the list on the lowest level at the CAS-instruction:

```
public boolean add(int threadId, T x)
{
    int topLevel = randomLevel();
    int bottomLevel = 0;
    Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
    Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
    Log.Entry entry = new Log.Entry(threadId, Log.Method.ADD,
        x.toString(), false, -1);
    while (true)
    {
        boolean found = find(x, preds, succs, entry); // linearization
            point
        if (found)
        {
            submitEntry(entry);
            return false;
        }
```

```
        else
        {
            Node<T> newNode = new Node<T>(x, topLevel);
            for (int level = bottomLevel; level <= topLevel; level++)
            {
                Node<T> succ = succs[level];
                newNode.next[level].set(succ, false);
            }
            Node<T> pred = preds[bottomLevel];
            Node<T> succ = succs[bottomLevel];
            boolean c;
            lock.lock();
            try // linearization point
            {
                c = pred.next[bottomLevel].compareAndSet(succ, newNode,
                    false, false);
                entry.timestamp = System.nanoTime();
            }
            finally
            {
                lock.unlock();
            }
            if (!c)
            {
                continue;
            }
            for (int level = bottomLevel + 1; level <= topLevel; level++)
            {
                while (true)
                {
                    pred = preds[level];
                    succ = succs[level];
                    if (pred.next[level].compareAndSet(succ, newNode,
                        false, false))
                        break;
                    find(x, preds, succs, null);
                }
            }
            entry.retval = true;
            submitEntry(entry);
            return true;
        }
    }
}
```

The remove method has very similar linearization points, again in the beginning when the object is not found, we take the linearization point at the moment the object is not found. The other possibility is that we found the object, and we marked it, then the linearization point is at the moment we mark it on the lowest level (we execute the CAS):

```
public boolean remove(int threadId, T x)
{
    int bottomLevel = 0;
```

```
Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
Node<T> succ;
Log.Entry entry = new Log.Entry(threadId, Log.Method.REMOVE,
    x.toString(), false, -1);
while (true)
{
    boolean found = find(x, preds, succs, entry); // linearization
        point
    if (!found)
    {
        submitEntry(entry);
        return false;
    }
    else
    {
        Node<T> nodeToRemove = succs[bottomLevel];
        for (int level = nodeToRemove.topLevel; level >= bottomLevel
            + 1; level--)
        {
            boolean[] marked = { false };
            succ = nodeToRemove.next[level].get(marked);
            while (!marked[0])
            {
                nodeToRemove.next[level].compareAndSet(succ, succ,
                    false, true);
                succ = nodeToRemove.next[level].get(marked);
            }
        }
        boolean[] marked = { false };
        succ = nodeToRemove.next[bottomLevel].get(marked);
        while (true)
        {
            boolean iMarkedIt;
            lock.lock(); // linearization point
            try
            {
                iMarkedIt =
                    nodeToRemove.next[bottomLevel].compareAndSet(succ,
                    succ, false, true);
                entry.timestamp = System.nanoTime();
            }
            finally
            {
                lock.unlock();
            }
            succ = succs[bottomLevel].next[bottomLevel].get(marked);
            if (iMarkedIt)
            {
                find(x, preds, succs, null); // what does this do? ->
                    deletes marked objects
                entry.retval = true;
                submitEntry(entry);
                return true;
```

```
                }
                else if (marked[0])
                {
                    submitEntry(entry);
                    return false;
                }
            }
        }
    }
}
```

---

The contains method has multiple linearization points. First one is when we acquire the head, which is only not overwritten later on, when the list is empty (as all other linearization points will not be reached in this scenario). Otherwise, the linearization point is when we acquire the last element to check against the element to be found.

---

```
public boolean contains(int threadId, T x)
{
    int bottomLevel = 0;
    boolean[] marked = { false };
    Log.Entry entry = new Log.Entry(threadId, Log.Method.CONTAINS,
        x.toString(), false, -1);
    Node<T> pred;
    lock.lock();
    try
    {
        entry.timestamp = System.nanoTime();
        pred = head;
    }
    finally
    {
        lock.unlock();
    }
    Node<T> curr = null;
    Node<T> succ = null;
    for (int level = MAX_LEVEL; level >= bottomLevel; level--)
    {
        curr = pred.next[level].getReference();
        while (true)
        {
            succ = curr.next[level].get(marked);
            while (marked[0])
            {
                curr = succ;
                lock.lock();
                try
                {
                    entry.timestamp = System.nanoTime();
                    succ = curr.next[level].get(marked);
                }
                finally
                {
                    lock.unlock();
```

```
                }
            }
            if (curr.value != null && x.compareTo(curr.value) < 0)
            {
                pred = curr;
                lock.lock();
                try
                {
                    entry.timestamp = System.nanoTime();
                    curr = succ;
                }
                finally
                {
                    lock.unlock();
                }
            }
            else
            {
                break;
            }
        }
    }
    entry.retval = curr.value != null && x.compareTo(curr.value) == 0;
    submitEntry(entry);
    return entry.retval;
}
```

**Results**

| Time | 1 Worker | 2 Workers | 4 Workers | 8 Workers |
|---|---|---|---|---|
| 10% Add, 10% Remove, 80% Contains | 52.414 ms | 221.050 ms | 471.637 ms | 1040.557 ms |
| 50% Add, 50% Remove, No Contains | 76.144 ms | 276.219 ms | 648.260 ms | 1487.103 ms |

| Total Wrong entries | 1 Worker | 2 Workers | 4 Workers | 8 Workers |
|---|---|---|---|---|
| 10% Add, 10% Remove, 80% Contains | 0 | 0 | 1 | 2 |
| 50% Add, 50% Remove, No Contains | 0 | 0 | 0 | 4 |

All tables are for a uniform distribution from 0 to 100000, and we execute 100000 operations given the table above per thread on the skiplist.

As can be seen in the tables, the number of wrong entries is very small, but we can not explain why there are some wrong entries. Checking the log itself, we noticed that it usually happens that a remove operation is after an add (so the timestamp is too late) but it is actually executed before the adding the same element again. This sometimes causes another afterward to be also wrong, but this is just because the list/set is not in the correct state, because the operations are flipped.

About the time it takes, we notice a very nonlinear increase at the first step, where the time it takes roughly quadruples instead of doubles, this is probably due to the added locks. For 4 and 8 threads the time increase is roughly as expected, but with 8 threads we are at the limit of cores of the pc, so the slight slowdown is probably because we reach the limit of the laptop.

## 2.3 Lock free (local) log

The code for this task is in `src/LockFreeSkipListLocal.java`.

In this section, each thread has its own local log, to which the entries are added. So there is no need for locks, otherwise the code is exactly the same as in the task above. At the end, all individual logs are concatenated into one big log and then sorted by timestamp (as described above).

**Results**

| Method | 1 Worker | 2 Workers | 4 Workers | 8 Workers |
|---|---|---|---|---|
| 10% Add, 10% Remove, 80% Contains | 42.817 ms | 52.681 ms | 64.118 ms | 93.479 ms |
| 50% Add, 50% Remove, No Contains | 68.929 ms | 82.509 ms | 101.236 ms | 147.273 ms |

| Method | 1 Worker | 2 Workers | 4 Workers | 8 Workers |
|---|---|---|---|---|
| 10% Add, 10% Remove, 80% Contains | 0 | 0 | 0 | 3 |
| 50% Add, 50% Remove, No Contains | 0 | 0 | 2 | 10 |

All tables are for a uniform distribution from 0 to 100000, and we execute 100000 operations given the table above per thread on the skiplist.

As we can see we have a few more errors, this might be due to the (described) granularity in the timestamps and maybe some unfortunate scheduling. For example add(1) and remove(1) is executed after add on a different thread, but in theory the linearization point (compareAndSet) for add is executed, then compareAndSet for remove is executed, then timestamp for remove is taken and then timestamp for add is taken. This would in the test show that remove is executed first, but in reality the add is executed first, then the thread is halted and the other the timestamp is only taken after some time. Timing is much faster and constant, as expected, this time, but we can also see the limit of the pc at 8 threads. For only removes and adds, the performance of the list is probably the driving factor for the slowdown.

## 2.4 Lock free global lock

The code for this task is in `src/LockFreeSkipListGlobal.java`.

In this method, instead of a `ArrayList` we simply use a `ConcurrentLinkedQueue` which stores the array entries. Otherwise, the code is the same as above.

| Method | 1 Worker | 2 Workers | 4 Workers | 8 Workers |
|---|---|---|---|---|
| 10% Add, 10% Remove, 80% Contains | 47.778 ms | 60.899 ms | 68.677 ms | 95.952 ms |
| 50% Add, 50% Remove, No Contains | 69.360 ms | 87.357 ms | 104.889 ms | 176.739 ms |

| Method | 1 Worker | 2 Workers | 4 Workers | 8 Workers |
|---|---|---|---|---|
| 10% Add, 10% Remove, 80% Contains | 0 | 0 | 0 | 2 |
| 50% Add, 50% Remove, No Contains | 0 | 0 | 0 | 2 |

All tables are for a uniform distribution from 0 to 100000, and we execute 100000 operations given the table above per thread on the skiplist.

As we can see fewer entries are wrong using a global lock free log. The timing is very similar to the local lock free log and slowdowns are probably only because of the underlying skip-list implementation.

## 2.5 Dardel experiments

Due to maintenance they will be handed in later.