Multi-Lane FlowPools: A detailed look

Tobias Schlatter[1], Aleksandar Prokopec[2], Heather Miller[2], Philipp Haller[2], and Martin Odersky[2]

[1] Student, EPFL
[2] Advisors, EPFL

Abstract. FlowPools, proposed by [4] are a powerful way to express dataflow logic in highly parallelized applications. The original paper proposes two ways of implementing a FlowPool: Single-Lane FlowPools (SLFP) and Multi-Lane FlowPools (MLFP). While SLFPs showed decent performance overall, insertion operations do not scale. MLFPs solve this limitation as benchmarks discussed in [4] have shown. This report goes into the details of the implementation of MLFPs and lies out the benchmarking results from [4] to show that MLFPs may reduce insertion time by $49-54\%$ on a 4-core i7 machine with respect to comparable concurrent queue data structures in the Java standard library.

1 Introduction

The goal of this section is to briefly remind the reader of the semantics and programming interface of a FlowPool and the basic ideas behind the implementation of SLFPs to allow for better understanding of the implementation of MLFPs.

1.1 Programming Model

The following operations are supported by a FlowPool. For a proof of determinism, refer to [4].

Append (`<<`) Inserts an element in the FlowPool. Fails if the number of elements the FlowPool has been sealed with is reached.
Signature: `def <<(x: T): Unit`

Foreach Traverse elements in the FlowPool. Calls a closure `f` exactly once (asynchronously) for each element added to the FlowPool (until it is sealed). Returns future of number of elements in pool. Foreach is normally implemented using the more general primitive `aggregate`.
Signature: `def foreach[U](f: T => U): Future[Int]`

Aggregate Reduce elements in the FlowPool to a single value. Starts off with `zero` as initial value to aggregate on, calls `op` exactly once per element to add it to an aggregation, finally uses `cb` to combine multiple aggregations into a single one. Note that `op` is guaranteed to be executed only once per element, whereas `cb` may be called any number of times
Signature: `def aggregate[S](zero: =>S) (cb: (S, S) => S)`
`(op: (S, T) => S): Future[S]`

Builders Abstraction to allow garbage collection of elements that are no longer required. Allows insertion without reference to initial pool (and hence all elements). See [4] for details.

Seal Fixes the number of elements that will eventually be in the pool allowing callbacks to be freed once reached. The final size of the pool is required as argument in order to preserve the determinism of the model. Subsequent call to `seal` will fail iff the FlowPool has already been sealed with a different size or the number of elements in the FlowPool is larger than the seal size.
Signature: `def seal(size: Int): Unit`

Figure 1 shows the implementation of some common monadic operations on top of the given primitives.

```
def filter
  (pred: T => Boolean)
  val p = new FlowPool[T]
  val b = p.builder
  aggregate(0)(_ + _) {
    (acc, x) => if pred(x) {
      b << x
      1
    } else 0
  } map { sz => b.seal(sz) }
  p
```

```
def flatMap[S]
  (f: T => FlowPool[S])
  val p = new FlowPool[S]
  val b = p.builder
  aggregate(future(0))(add) {
    (af, x) =>
    val sf = for (y <- f(x))
      b << y
    add(af, sf)
  } map { sz => b.seal(sz) }
  p

def add(f: Future[Int], g: Future[Int]) =
  for (a <- f; b <- g) yield a + b
```

```
def union[T]
  (that: FlowPool[T])
  val p = new FlowPool[T]
  val b = p.builder
  val f = for (x <- this) b << x
  val g = for (y <- that) b << y
  for (s1 <- f; s2 <- g)
    b.seal(s1 + s2)
  p
```

Fig. 1. Some common monadic operations implemented using the FlowPool primitives (taken from [4])

1.2 Single-Lane FlowPool

SLFPs are implemented using a single linked-list of arrays of elements, where the last non-empty element does not hold actual data, but the state of the FlowPool, i.e. a list with all the callbacks and the seal state of the FlowPool. Changes to the SLFP are only allowed by CASing at this particular point, which hence serves as linearization point of the data structure.

With competing, concurrent writers, this approach does not scale nicely (see figure 5). Mainly due to cache contention (multiple CPUs often write to close memory locations) and CAS collisions.

2 Implementation

MLFPs take advantage of the lack of ordering guarantees in the FlowPool semantics to remove the limitation of SLFPs with respect to scaling by applying a simple but effective idea: Instead of having a single SLFP, we use one SLFP

(from now on called lane) for each processor (or thread). This means, that on one hand, every processor (or thread) gets its own lane to which it appends elements to, avoiding CAS collisions and cache contention. On the other hand, every callback or aggregation has to be registered on each of these lanes separately and completing the callback future has to be externally synchronized.

In the following, the implementation of the FlowPool operations in the MLFP are explained in detail. Please refer to figure 4 for pseudo code of the operations.

2.1 Callbacks

When calling `aggregate` on a MLFP, the implementation has to ensure that a copy of this callback is known to every underlying lane and that upon completion of the callbacks for every lane, the values are aggregated in the final result and the future is completed. Note that callback addition to a given lane is no different than in SLFPs.

The aggregation of each lane's final values is done using a FlowLatch, a construct that allows aggregating a given number of values into a single one using similar semantics as a FlowPool. A FlowLatch may supply a future that is completed with the aggregated value, once the expected number of values has been supplied. `aggregate` returns this future to the caller.

2.2 Seal

The `seal` operation is the only one whose complexity increases with multiple lanes, as the overall size guarantee has to be established over all lanes while preserving lock-freedom and linearizability, especially in the face of errors occurring while sealing.

To ensure the latter, a global seal state (stored in a location common to the whole MLFP) is required. The global seal may be in one of the following states (state transition graph in figure 2):

- `Unsealed` (U): No seal has yet been attempted, or all seals have failed.
- `Proposition(size)` (P): Sealing with given (total) size is being attempted. No other sealing operation may be attempted until this state is resolved to one of the other two. Threads that attempt to seal which then encounter a `Proposition` state, will help seal the FlowPool. (This is required to ensure lock-freedom)
- `Sealed(size, remain)` (S): The MLFP has been sealed with size `size`, where at the moment of the sealing, `remain` free slots were remaining. The latter value is required to distribute the remaining element slots amongst the lanes and will be explained in detail later.

Further, each lane holds its own seal state (similar to SLFPs) which may be one of the following. The state transitions are the same as for the global state (figure 2).

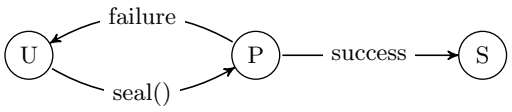

Fig. 2. State-transition graph of MLFP seal state

- `CallbackList` (U): The lane is unsealed. A seal procedure might have been started but is not yet completed. Insertion may be handled as in a SLFP.
- `SealTag` (P): A seal procedure has been started and any writer should resolve the tag by either helping to finish the sealing or replace it based on the global state.
- `Seal(size)` (S): This lane has been sealed with the given size.

In the following, the different parts of the sealing procedure are explained in detail.

Seal When a thread calls `seal()` it first checks the current state of the MLFP. If it is in the unsealed state, it tries to change the seal state to proposition and then continue with the sealing. If the MLFP is already in the proposition state, it helps to complete the current seal. If the MLFP is in the `Sealed` state, the call returns immediately, succeeding if the size in `Sealed` is equal to the proposed sealing size, failing otherwise.

Help Sealing Once the MLFP is in the proposition state, any thread attempting to seal (or attempting to insert an element as we will see later), will begin propagating the proposition state to each lane. It does this by placing a special token, `SealTag`, into the state location of the given lane, using a procedure similar to `seal()` on SLFPs.

A thread that tries to insert an element and encounters a `SealTag` must resolve the tag before it can continue on (explained later). This ensures that, once a `SealTag` has been placed in a lane, its size does not change until the seal is finished, which makes it possible to calculate a snapshot of the total number of elements in the MLFP. Note that during this procedure, one might also find that the seal has been completed (when encountering a `Seal` in a lane). In such a case, helping can be stopped prematurely.

Note that each `SealTag` holds a reference to the global proposition object that caused its creation. This is required as a sentinel since other seal tags might still be present from a failed attempt to seal.

When a thread that succeeds in calculating the snapshot of the size, it will then try to change the MLFP's state to sealed (or to unsealed, if the number of elements is too big) and replace every occurrence of `SealTag` in the lanes by a `Seal`.

Resolution of seal tags When a writer encounters a `SealTag` it first checks the current global state. If it indicates, that the seal operation corresponding to the `SealTag` is still going on, the writer helps sealing as described above, then retries writing. If the global state indicates that the structure has been sealed, the writer calculates the remaining slots for this lane according to the following

formula and then seals this lane.

$$n_{seal} = n_{cur} + \frac{n_{remain}}{l_{total}} + 1_{\{l_{cur} < n_{remain} \bmod l_{total}\}}$$

If the writer finds that the current global state is unsealed or a different proposition than in the tag, it removes the tag and continues inserting normally.

Finalization Some procedures that are required to invoke the callback scheduling properly when sealing (i.e. ensuring the finalization procedure of each callback is called exactly once) have been omitted here for simplicity.

2.3 Insertion

Insertion into a MLFP happens the same way as into a SLFP, however, the lane in which the element is to be inserted has to be chosen first. For the choice of the lane we want to:

- Avoid collision between competing threads to a maximum.
- Make it possible to change the thread-lane assignment, once some lanes have been entirely filled (after a seal).
- Assure we can properly fill up the FlowPool entirely.

To achieve this, the following mapping is used at the beginning:

$$l_{cur} = T_{cur} \bmod l_{total} \qquad (T_{cur} : \text{Current thread ID})$$

When the first collision (i.e. attempted insertion into a full lane) happens, the following hashed mapping is used [1]:

$$l_{cur} = \text{rb}((T_{cur} + C) \cdot 9e3775cd_{16}) \cdot 9e3775cd_{16} \bmod l_{total}$$

where C is the (global) number of collisions so far and $\text{rb}(\cdot)$ inverses the byte order.

After a certain number of collisions (currently the number of lanes), the insertion switches to a linear search over all lanes to guarantee that either the FlowPool is filled, or an error is raised due to too many inserted elements.

Experimental results in figure 3 show that there is no use in having more than one lane per inserting thread and hence that the upper lane selection strategy is efficient. For details, refer to section 3.

3 Evaluation

This section comes mainly from [4] and is just repeated here for convenience.

Setup We evaluate our implementation (single-lane and multi-lane FlowPools) against the LinkedTransferQueue [2] for all benchmarks and the ConcurrentLinkedQueue [3] for the insert benchmark, both found in JDK 1.7, on three different architectures; a quad-core 3.4 GHz i7-2600, 4x octa-core 2.27 GHz Intel

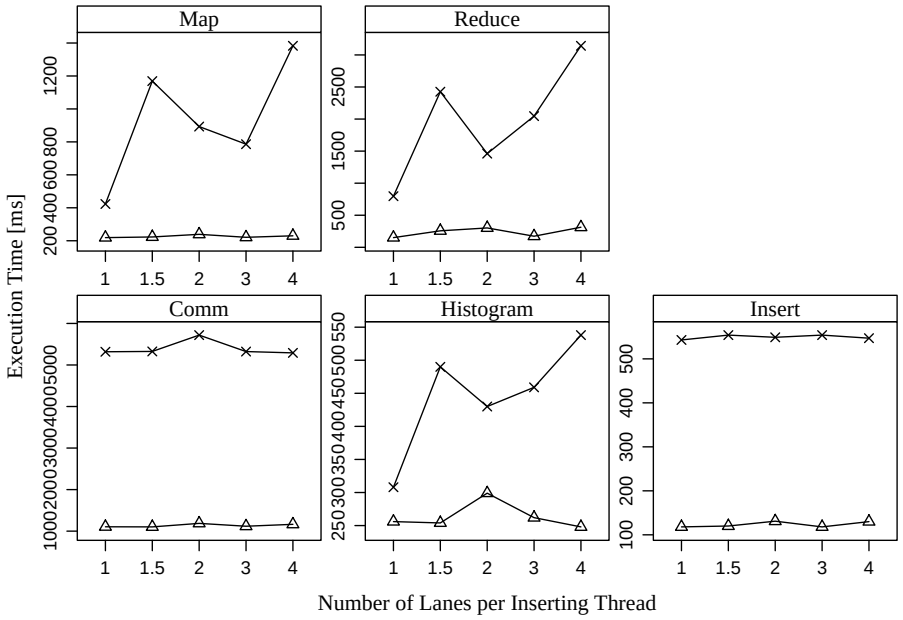

Fig. 3. Execution times with respect to number of lanes per inserting thread in MLFPs. × UltraSPARC T2, △ 4-core i7

```
def aggregate[S](zero: =>S)
                (cmb: (S, S) => S)
                (folder: (S, T) => S):
    Future[S] = {

  val aggregator = FlowLatch[S](zero)(cmb)
  aggregator.seal(laneCount)

  for (l <- lanes) {
    l.registerCB(folder, zero, aggregator)
  }

  aggregator.future
}

def seal(size: Int) {
  val cur_state = state
  cur_state match {
    case Unsealed =>
      val ns = Proposition(size)
      if (!CAS(state, cur_state, ns))
        seal(size)
      else if (!helpSeal(ns)) failure()
    case p: Proposition =>
      if (helpSeal(p)) failure()
      else seal(size)
    case Sealed(sz,_) if size != sz =>
      failure("already sealed")
    case _ => // Done
  }
}

def tryResolveTag[T](t: SealTag[T]) {
  cur_state match {
    case p: Proposition if (p eq t.p) =>
      helpSeal(p)
    case Sealed(_,rem) =>
      val sz = sealSize(cur_state)
      writeSeal(Seal(sz))
    case _ => revertTag()
  }
}
```

```
def helpSeal(p: Proposition): Boolean = {
  var sizes: Int = 0

  for (l <- lanes) {
    writeSealTag(l, p) match {
      case OnGoing(v)   =>
        sizes = sizes + v
      case Finished(success) => return success
    }
  }

  if (sizes <= p.size) {
    // Seal MUST succeed
    val remaining = p.size - sizes
    val ns = Sealed(p.size, remaining)
    if (CAS(state, p, ns))
      finalizeSeals(remaining)
    true
  } else {
    // Seal MUST fail
    CAS(state, p, Unsealed)
    false
  }

}

def append(x: T) {
  val index = {
    if (collisions <= 0)
      curTID % laneCount
    else if (collisions <= laneCount)
      hash(curTID, collisions)
    else
      findEmptyLane() // Fails if not found
  }

  if (!lanes(index).append(x)) {
    increment(collisions)
    append(x)
  }

}
```

Fig. 4. Multi-Lane FlowPool operations pseudo code

Xeon x7560 (both with hyperthreading) and an octa-core 1.2GHz UltraSPARC T2 with 64 hardware threads.

Benchmarks and Scaling In the *Insert* benchmark, Figure 5, we evaluate the ability to write concurrently, by distributing the work of inserting N elements into the data structure concurrently across P threads. In Figure 5, it's evident that both single-lane FlowPools and concurrent queues do not scale well with the number of concurrent threads, particularly on the i7 architecture. They seem to slow down rapidly, likely due to cache line collisions and CAS failures. On the other hand, multi-lane FlowPools scale well, as threads write to different lanes, and hence different cache lines on most occasions, meanwhile also avoiding CAS failures. This may reduce execution time for insertions up to 54% on the i7, 63% on the Xeon and 92% on the UltraSPARC T2.

Usage of the inserted data is evaluated in the *Reduce*, *Map* (both in Figure 5) and *Histogram* benchmarks (Figure 6). It's important to note that the *Histogram* benchmark serves as a "real life" example, which uses both the `map` and `reduce` operations that are benchmarked in Figure 5.

The *Reduce* benchmark starts P threads which concurrently insert a total of N elements. The `aggregate` operation is used to reduce the set of values inserted into the pool. Note that in the FlowPool implementation there may be as many threads computing the aggregation as there are different lanes – elements from different lanes are batched together once the pool is sealed.

The *Map* benchmark is similar to the *Reduce* benchmark, but instead of reducing a value, each element is mapped into a new one and added to a second pool.

In the *Histogram* benchmark, Figure 6, P threads produce a total of N elements, adding them to the FlowPool. The `aggregate` operation is then used to produce 10 different histograms concurrently with a different number of bins. Each separate histogram is constructed by its own thread (or up to P, for multi-lane FlowPools). A crucial difference between queues and FlowPools here, is that with FlowPools, multiple histograms are produced by invoking several `aggregate` operations, while queues require writing each element to several queues– one for each histogram. Without additional synchronization, reading a single queue is not an option, since elements have to be removed from the queue eventually, and it is not clear to each reader when to do this. With FlowPools, elements are automatically garbage collected when no longer needed.

Finally, to validate the last claim of garbage being automatically collected, in the *Communication/Garbage Collection* benchmark, Figure 6, we create a pool in which a large number of elements N are added concurrently by P threads. Each element is then processed by one of P threads through the use of the `aggregate` operation. We benchmark against linked transfer queues, where P threads concurrently remove elements from the queue and process it. For each run, we vary the size of the N and examine its impact on the execution time. Especially in the cases of the Intel architectures, the multi-lane FlowPools perform considerably better than the linked transfer queues. As a matter of fact, the linked transfer queue on the Xeon benchmark ran out of memory, and was un-

Architecture	Elements	FlowPool $t[ms]$	P	Queue $t[ms]$	P	decr.
4-core i7	2M	17	8	35	1	51%
4-core i7	5M	44	8	87	1	49%
4-core i7	15M	118	8	258	1	54%
UltraSPARC T2	1M	23	32	111	4	79%
UltraSPARC T2	2M	34	64	224	4	84%
UltraSPARC T2	5M	62	64	556	4	88%
UltraSPARC T2	15M	129	64	1661	4	92%
32-core Xeon	2M	30	8	50	1	40%
32-core Xeon	5M	46	64	120	1	61%
32-core Xeon	15M	126	64	347	1	63%

Table 1. Execution times for insert benchmark for MLFP and concurrent linked queues, including execution time decrease percentage.

able to complete, while the multi-lane FlowPool scaled effortlessly to 400 million elements, indicating that unneeded elements are properly garbage collected.

Scaling in Input Size In figure 7 we can see that the Input, Map, Reduce and Histogram benchmark all scale linearly in the input size with any parallelism level. The Comm benchmark has not been tested for different sizes.

Multi-Lane Scaling By default, the number of lanes is set to the parallelism level P, corresponding to the number of used CPUs. However, since the implementation has to use hashing on the thread IDs instead of the real CPU index, we tested whether varying the number of lanes to $1.5P$, $2P$, $3P$ and $4P$ results in performance gain due to fewer collisions. Benchmarks have shown (see figure 3) that this yields no observable gain – in fact, this sometimes even decreased performance slightly.

Performance Gain As stated in the abstract, FlowPools – or more precisely multi-lane FlowPools – may reduce execution time by $49 - 54\%$ on 4-core i7. These figures have been obtained by comparing medians of execution times for insertions between multi-lane FlowPools and concurrent linked queues (which were always faster than linked transfer queues), where each structure was evaluated on its optimal parallelization level. The resulting data is shown in table 1.

Methodology All the presented configurations have been measured 20 times, where the 5 first values have been discarded to let the JIT stabilize. Aggregated values are always medians. The benchmarks have been written using `scala.testing.Benchmark` and executed through SBT[3] using the following flags for the JavaVM: `-Xmx2048m -Xms2048m -XX:+UseCondCardMark -verbose:gc -XX:+PrintGCDetails -server`.

4 Conclusion

In this report we have shown that a FlowPool as proposed by [4] can be implemented in a scalable way by using multiple single-lane FlowPools. Further,

[3] Simple Build Tool

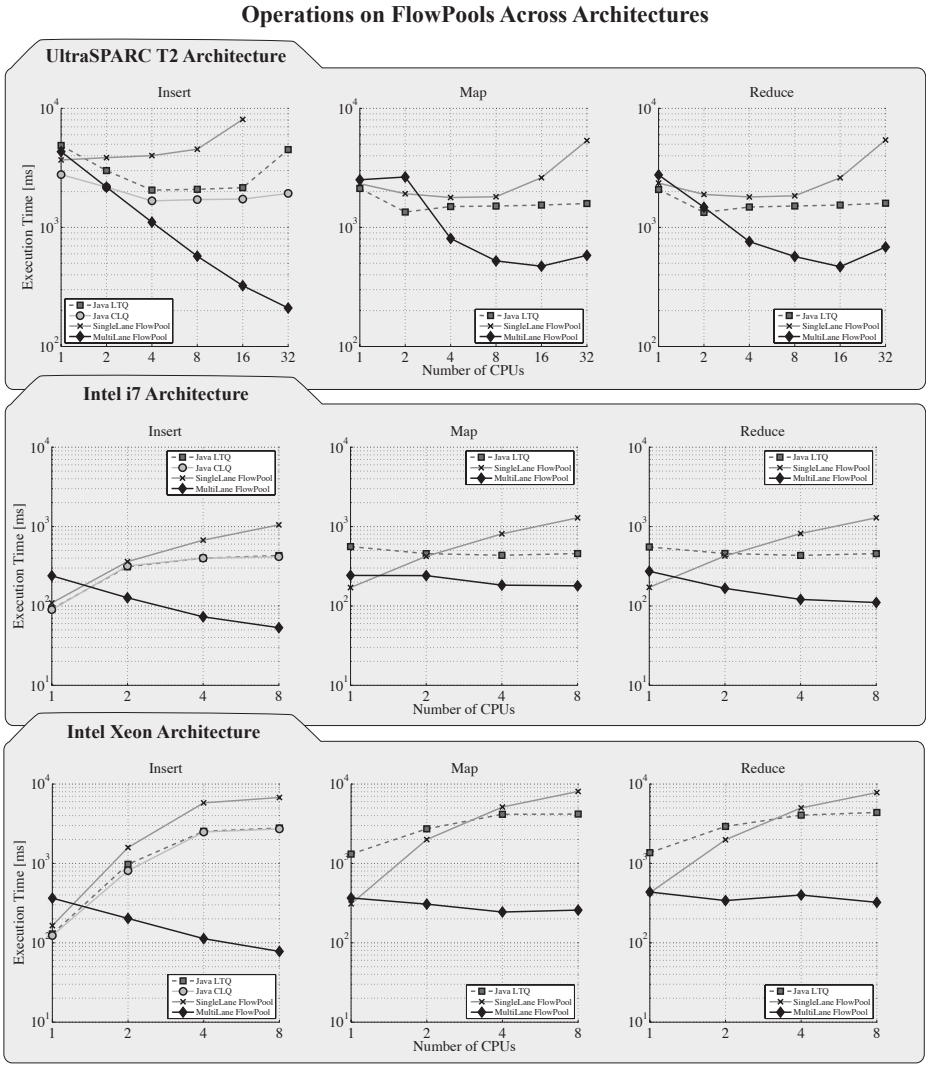

Fig. 5. Execution time vs parallelization across three different architectures on three important FlowPool operations; insert, map, reduce.

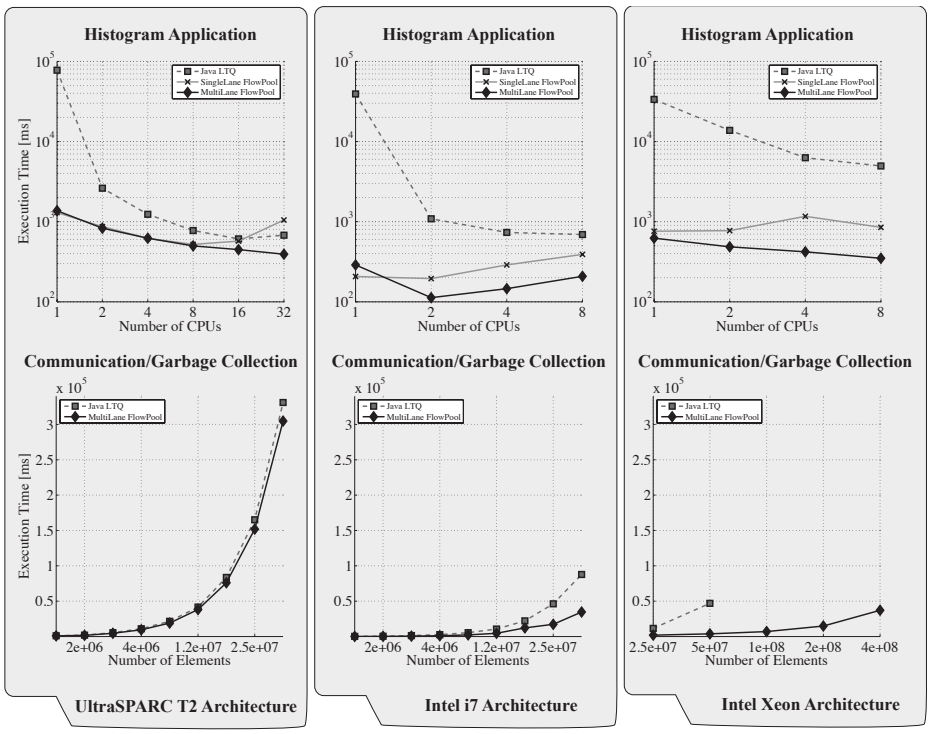

Fig. 6. Execution time vs parallelization on a real histogram application (top), & communication benchmark (bottom) showing memory efficiency, across all architectures.

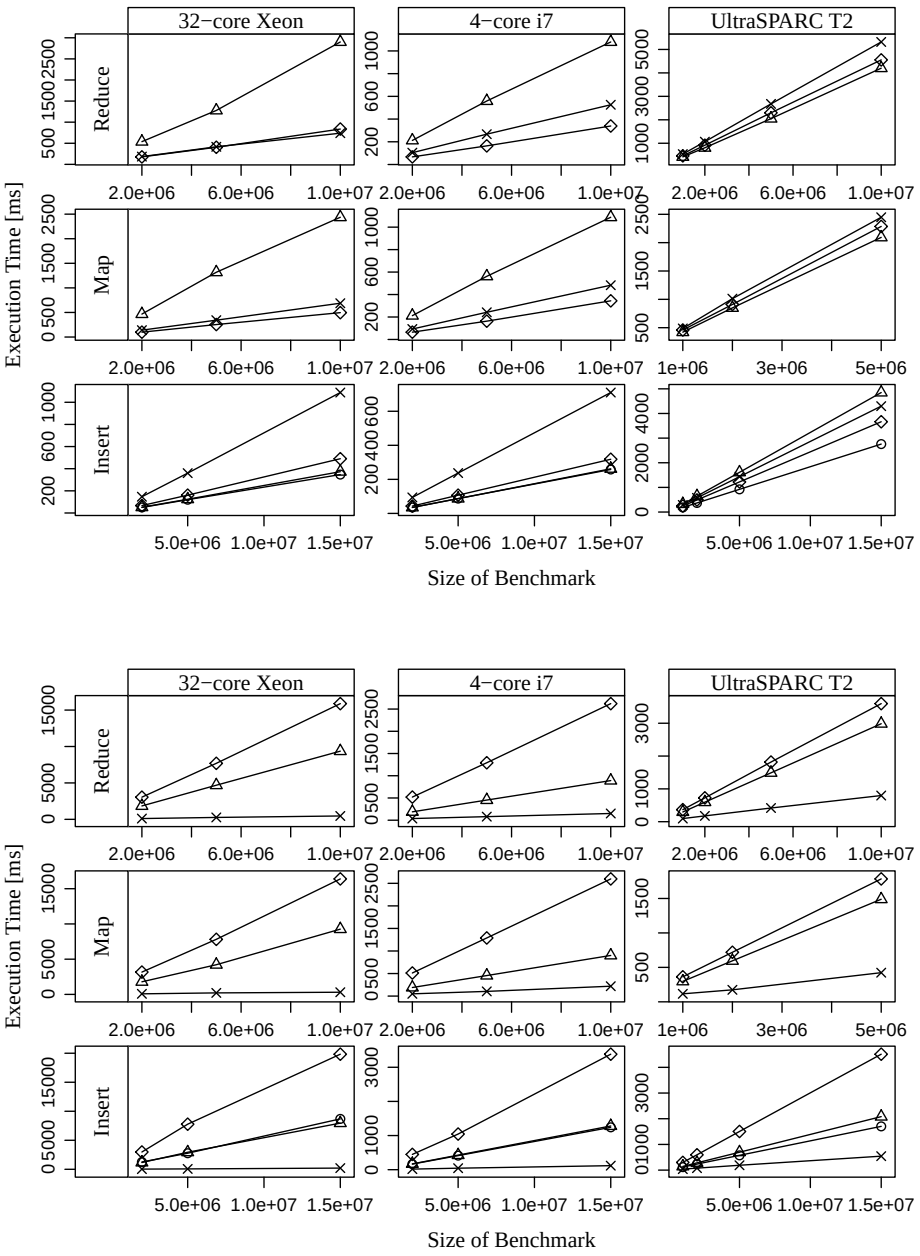

Fig. 7. Execution time vs benchmark size ($P = 1, 8$). ◊ SLFP, × MLFP, △ linked transfer queue, ○ concurrent linked queue

we have provided a detailed description including pseudo-code of how a multi-lane FlowPool is implemented, especially with respect to sealing – which had to be fundamentally changed – and the lane assignment rehashing strategy, that does not exist in single-lane FlowPools. We used and included benchmark results from the original paper to show that MLFPs scale nicely, may significantly reduce execution time with respect to comparable data structures and that the proposed rehashing strategy is efficient for a number of lanes equal to the number of inserting threads.

References

1. P. Bagwell. Byteswap hashing. See `http://www.scala-lang.org/archives/downloads/distrib/files/nightly/docs/library/scala/util/hashing/ByteswapHashing.html`.
2. W. N. S. III, D. Lea, and M. L. Scott. Scalable synchronous queues. *Commun. ACM*, 52(5):100–111, 2009.
3. M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.
4. A. Prokopec, H. Miller, T. Schlatter, P. Haller, and M. Odersky. Flowpools: A lock-free deterministic concurrent dataflow abstraction. In *Workshops on Languages and Compilers for Parallel Computing*, 2012. under review.