Reagent Based Lock Free Concurrent Link List	**Spring 2012**
Ancsa Hannak and Mitesh Jain	April 28, 2012

1 Introduction

The most commonly used implementation of synchronization is blocking algorithms. They utilize locks to synchronize between concurrent processes. However, lock-based implementations can only express coarse-grain parallelism and do not scale to many cores.

Non-blocking algorithms deliver significant performance benefits over their blocking counterparts. The building blocks for such algorithm consist of *isolated* atomic updates to *shared states* and interactive synchronous *communication* through *message passing.* These complementary requirements of isolation and interaction, compounded with the performance considerations provide unique challenges in design of expressible and more importantly compassable abstractions. Reagents[3] provide a framework that abstracts away common design patterns in concurrent algorithms and still provide compassable fine-grained combinators.

The goal of our project is to implement a lock-free concurrent linked list data structure that represents the ordered set abstract data type. We develop a concurrent algorithm using reagents to express fine-grained parallelism in the linked list. We compare its scalability by measuring its raw throughput against the lock-based and lock-free implementations in Java.

2 Reagents

Reagents provide a basic set of building blocks for writing concurrent data structures and synchronizers. The building blocks include isolated atomic updates to shared state, and interactive synchronous communication through message passing. The building blocks also bake in many common concurrency patterns, like optimistic retry loops, back-off schemes, and blocking and signaling.

2.1 Combinators

Reagents are representation of computation as data. The computations being represented are fine-grained concurrent operations. A value of type Reagent [A,B] represents a function from A to B that internally interacts with a concurrent data structure through mutation, synchronization, or both(as a side effect). Each way of combining reagents corresponds to a way of combining their internal interactions with concurrent data structures. Memory is shared between reagents using the type Ref[A] of atomically-updatable references. Before introducing our algorithm, we give an overview of the combinators used in the algorithm.

1. **upd**: The *upd* combinator represents atomic updates to references. It takes an update function, which tells how to transform a snapshot of the reference cell and some input into an updated value for the cell and some output.

Although the upd combinator is convenient, it is sometimes necessary to work with shared state with a greater degree of control. To this end, we include two combinators, read and cas for working directly on Ref values. Together with the computed combinator described below, read and cas sufce to build update.

2. **read and cas**: If r has type Ref[A], then read(r) has type Reagent[Unit, A] and, when invoked, returns a snapshot of r. The cas combinator takes a Ref[A] and two A arguments, giving the expected and updated values, respectively. Unlike its counterpart for AtomicReference, a **cas** reagent does not yield a boolean result. A failure to CAS is transient, and therefore results in retry. The retry mechanism is abstracted away and can be changed without affecting the client code using the reagents.

3. **compute**: The reagents we have seen so far are constructed prior to, and independently from, the data that ows through them. Phase separation is useful because it allows reagent execution to be optimized based on complete knowledge of the computation to be performed. But in many cases the choice of computation to be performed depends on the input or other dynamic data. The computed combinator expresses such cases. It takes a partial function from A to Reagent[Unit,B] and yields a Reagent[A,B]. When the reagent computed(f) is invoked, it is given an argument value of type A, to which it applies the function f. If f is not dened for that input, the computed reagent issues a permanent (blocking) failure, similarly to the upd function. Otherwise, the application of f will yield another, dynamically-computed reagent, which is then invoked with (), the unit value.

$$
\begin{aligned}
&\textbf{upd} : \mathit{Ref}[A] \Rightarrow (A \times B \rightharpoonup A \times C) \Rightarrow \mathit{Reagent}[B, C]\\
&\textbf{read} : \mathit{Ref}[A] \Rightarrow \mathit{Reagent}[\mathit{Unit}, A]\\
&\textbf{cas} : \mathit{Ref}[A] \times A \times A \Rightarrow \mathit{Reagent}[\mathit{Unit}, \mathit{Unit}]\\
&\textbf{ret} : A \Rightarrow \mathit{Reagent}[\mathit{Unit}, A]\\
&\textbf{computed} : (A \rightarrow \mathit{Reagent}[\mathit{Unit}, B]) \Rightarrow \mathit{Reagent}[A, B]
\end{aligned}
$$

3 Linked list algorithm using reagents

A linked list is a data structure consisting of a group of nodes which together represent the ordered set abstract data type. Under the simplest form, each node is composed of a datum and a reference (in other words, a link) to the next node in the set. In our implementation, the list is ordered, which provides an efficient way to detect when an item is absent. The following figure depicts a linked-list whose elements are ordered.

For simplicity, we further restricts to set of integers and thus comparison is an inexpensive < operation on integers. Since linked lists are the representation of a set, we do not allow two nodes to have the same data field. The *next* field is a reference to the next node in the list. In addition to the *regular* nodes, we have two special nodes: the sentinel *Tail* to mark the end of the list and the *Marker* node to indicate that the node predecessor to it has been marked for deletion. We maintain the invariant that the nodes are sorted in *data* order. Thus at any moment, the data fields of *regular* nodes, which do not have *Marked* nodes as successor are elements of the set. The *head* is a reference to the start of the list. The three methods *add, remove, contain* are implemented to add, remove, and search for data in the set. We maintain the following invariant for the data structure: If any node n was reachable at any moment from the head then at all time all node $m > n$ are reachable from n.

We will now explain the implementation of the concurrent linked list. The head reference is initial to the Tail node. At any moment, we use **read** combinator to extract the node that the reference points to. The read combinator is immediately executed.

1. **findNode** : This private function starts with a *curRef* Ref and uses the *read* combinator to immediately execute it[1] (*curRef.read ! ()*) to extract the node that it references. It then uses Scala pattern matching to deconstruct the instance of the node. A pattern like *case Node(d,r)* matches any instance of the node class binding d to its data field and r to its *next* field. In case the *curNode* has *Marker* node as its successor, it indicates that the the node has been logically marked for deletion by some thread. In such a case, the *findNode* method physically and atomically updates the curRef to point to the node referenced by the *Marker* . Notice that this preserves the invariance: all nodes greated that the curNode can be reached from the curNode even after physically delinking it. Since *findNode* is private and tail-recursive, Scala will compile it to a loop.

2. **add, Reagent[Int,Unit]** : The *add* reagent takes as input the data to be inserted. Unlike in stack, where all activity is focused on *head*, the operations on the linked list do not apriori have the Refs for the reagents to compute and in particular depend on the current state of the list. Thus for adding a new node in the list, we compute a dynamic reagent by calling the private function *addNode.* This uses findNode function to acquire the reference to the node,*predRef* and the node *curNode*, before which the new node must be inserted. In case the data already exists in the list, then it returns a constant reagent, which always succeeds and does not modify the state of the underlying concurrent data structure. Note that *add* is lock-free.

3. **remove, Reagent[Int,Unit]** : Similar to the *add* reagent, *remove* must also dynamically compute the reagent using the *compute* combinator. It calls the *findNode* to get the node to be marked for deletion and atomically update its *next* field to point to special *Marker* node. The *next* field of the Marker to point to successor of the current node. Notice that this updates need not be done atomically. Since we maintain the invariant, even if the successor node has been marked for deletion by a concurrent process, any future access will be able to reach nodes greater than current or its successor. Note that *remove* is lock-free.

[1]In the terminology of regents, we call this a reaction

4. **contains, Reagent[Int,Boolean]** : This is a simple reagent which uses the *findContain* private method to traverse the list. It ignores the nodes which have a *Marker* node as their successor and returns true if it finds a node, false if it reaches a node with a data field greater than the input or reaches the *Tail* node. Note that *contains* is wait-free.

```scala
final class LinkedList {
  //data definition
  private abstract class N
  private final case class Node(data : Int , next : Ref[N]) extends N
  private final case class Marker(next: Ref[N]) extends N
  private final case object Tail extends N
  private val head: Ref[N] = Ref(Tail)

  // access methods and helper function.
 // find the node with data d
  private def findNode(curRef: Ref[N], d: Int): (Ref[N], N) = {
    val curNode = curRef.read ! ()
    curNode match {
      case curNode@Node(_ , Ref(Marker(Ref(n)))) => {
   curRef.cas(curNode, n) !? (); // update and move on
   findNode(curRef, d)
      }
      case Node(x,r@Ref(_)) if x < d => findNode(r, d)
      case n => (curRef, n) //  x >= d (if n is the tail sentinal node x > d)
     }
  }

  // finds the node for deletion and logically (mark) delete it.
  private def markForDel (d : Int) : Reagent[Unit,Unit] = {
    val (predRef, curNode) = findNode(head, d)
    curNode match {
      // curNode still points to some Node and has not already been "marked"
      case Node(x, r@Ref(ov)) if x == d => r.cas(ov, new Marker(new Ref(ov)))
      case _ => ret(())
    }
  }
  val remove : Reagent[Int,Unit] = computed {
    (d : Int) => markForDel(d)
  }

  private def addNode(d : Int) : Reagent[Unit,Unit] = {
    val (predRef, curNode) = findNode(head, d)
    curNode match {
      // the data already exists in the set
      case Node(x, _) if x == d => ret(())
      case _ =>   predRef.cas(curNode, new Node(d, new Ref(curNode)))
    }
  }
  val add : Reagent[Int,Unit] = computed {
    (d: Int) => addNode(d)
  }

  private def findContain(curRef : Ref[N], d : Int) : Boolean = {
      if (d < 0) false
      else {
        curRef.read ! () match {
    case Node(_, Ref(Marker(r))) => findContain(r  , d)
    case Node(x, r) if x < d      =>  findContain(r, d)
    case Node(x, _) if (x == d)   => true // found
    case _                        => false // went past node
      }
      }
  }

  // search if input d is in Set.
  val contains: Reagent[Int,Boolean] = computed {
    (d: Int) => ret(findContain(head, d))
  }
}
```

4 Micro-benchmark and Experiments

The runtime compilation and garbage collection in managed systems induce complexity in performance measurements. The complex interaction of (1) architecture (2) JIT compiler (3) virtual machine (4) memory management and (5) application provide multiple parameters. We measure the steady-state runs to reduce the impact of the JIT compilation so that interaction is mainly limited between the application and the memory management system. But this does not guarantee that the JIT compiler behaves deterministically across each run. So we measure the covariance to estimate the variation in execution time. Before each run we run the garbage collector, and since the memory footprint of these benchmarks is small, we do not expect the heap size to grow large. To confirm this, we run the experiments by decreasing the probability of *remove* operation to an existing data in the set. This effectively allows the list to grow to larger sizes and thus have bigger memory footprints. We observe no significant difference in the raw throughput of the micro-benchmarks.

4.1 Micro-benchmarks

The micro-benchmarks we use try to emulate the real workload and corresponding access patterns for the concurrent linked-list. This is achieved by varying the ratio of concurrent accesses (of the data-structure) to the total work in an iteration. Each thread executes the same chunk of computation with some amount of variation around the mean number of iterations. This variation in number of iterations across threads assists in avoiding the states where all threads end up in perfect sync and therefore can potentially end up backing off in sync causing a live-lock.

We developed two micro-benchmarks to emulate light threads and heavy threads. The first benchmark (b1) has 3 phases of pure (local) work, each of them interleaved by an access to the concurrent linked list. (An add, a remove and contains, in this order.) For each node call to the *add* method, we make a call to *remove* method. Therefore, on an average the same number of nodes are added and removed from the queue. And for the data chosen uniformly at random, the expected length of the linked list in 0. This methodology ensures that the garbage collector, on an average has the same effect on individual measurements. The second micro-benchmark performs 1 phase of pure work and 1 access randomly chosen with equal probably among add/remove/contains) to the concurrent data structure.

```
// local work
def pureWork(work: Int, iters: Int) = {
  val r = new Random
  for (_ <-1 to iters) {
    Util.noop(r.fuzz(work))
    Util.noop(r.fuzz(work)) }

  //  Benchmark 1
  for (_ <- 1 to iters) {
    ll.add ! SomeIntData
    Util.noop(r.fuzz(work))
    whileFalse(ll.contain ! SomeIntData)
    Util.noop(r.fuzz(work))
    ll.remove ! SomeIntData
  }

  // Benchmark 2
  for (i <- 1 to iters) {
    r.next(3) match {
      case 0 => ll.add(SomeIntData)
      case 1 => ll.contains(SomeIntData)
      case 2 =>ll.remove(SomeIntData)
    }
    Util.noop(r.fuzz(work))
  }
```

4.2 Experimental Methodology

We now summarize the measurement methodology adopted from [3] framework.

1. **timePerWork** : Work local to threads is modeled with the *computed* function from the java.util.concurrent library. The two parameters, *work* and *iters*, determine the number of iteration of the inner and outer-loop respectively. For each call to this function, the inner loop iterates k times, where $|k - work| < \sigma$, for some σ. We measure the time required to execute one unit of pure work for each given value of *work* parameter.

2. We repeat the following procedure for each thread count and *work* ranging from 200 to 1000 units:

   (a) **Warmup**: Launch a list of threads in parallel, each executing the microbmk, and increase the number of iterations in steps of 1000 until the runtime is *1000 ms*. Before each run the garbage collector is kicked of the clean up the memory footprints of the previous iteration. This gives us the expected raw throughput , estimate time for concurrent operations in the bmk and number of trial iterations, *trialIters*.

(b) **Trials**: Time the benchmark for *trialIters*. If the variance of the runtime exceeds a threshold, then the trial is repeated.[2]

We measure the mean raw throughput(rTP) and mean concurrent operations throughput(copTP). We observe that copTP has larger deviation around the mean and less predictable to infer the scalability of the algorithm. We conjecture that this variation primarily arises from two factors: change in the timePerWork between the warm up run and the actual run and , secondary effect attributed to the processor.[3]

4.2.1 Observation and Inference

We used a 16 core Intel Xeon CPU 3.47Ghz and total cache (L1,L2 and L3) of 12MB to conduct our experiments. There are in total 2 sockets, 4 cores per socket and 2 threads per core. We compare the scalability of the following implementations of the concurrent linked-list[1].

- Coarse grain Lock based: This implementation locks the list as a whole
- Fine-grained synchronization: This implementation locks the individual nodes
- Lazy synchronization: In this implementation *add* and *remove* are blocking and contain is wait free.
- Lock free hand designed: In this implementation *add* and *remove* are lock free and *contains* is wait free.
- Reagents based: In this implementation *add* and *remove* are lock free and *contain* is wait free and implemented using reagents are explained above.

The amount of contention for the shared resource (nodes in the linked list) among threads is inversely proportional to the value of the work parameter.

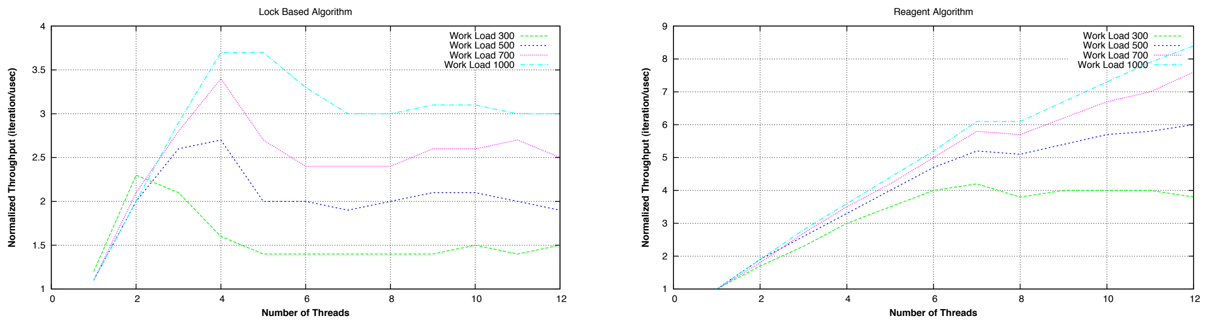
1. As the contention for data structure increases , the throughput of the lock based blocking algorithm decreases Figure 1a. The throughput does not scale beyond 4 cores. Thus the potential speed up with the multiple threads cannot be harnessed because the synchronization cost dominates the execution time.

   In contrast, the throughput of the reagent based implmentation scales Figure 1b fairly well until 6 cores even for high contention and for lower contention continues to scale uptil 12 cores.

2. At low contention Figure 3a and 3b the hand coded lock free implementation and the Reagent based implementation have very comparable raw throughput (17% upto 10 cores).

[2]We should have repeated the experiment until the measurements are in the confidence Interval

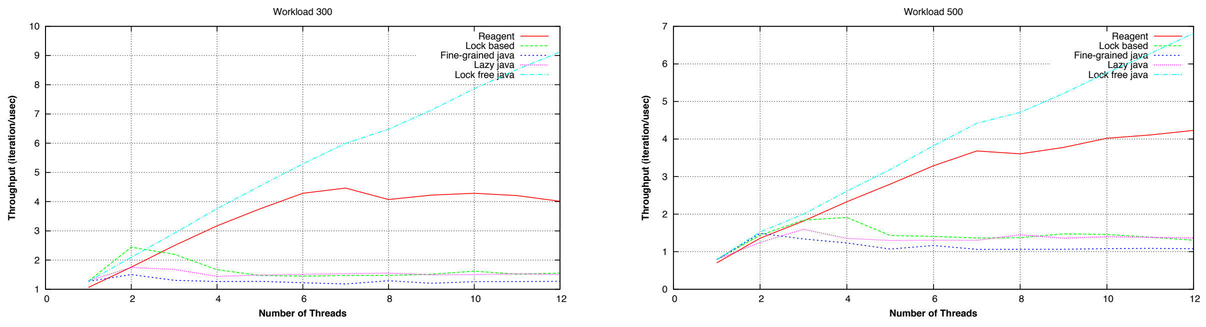
[3]primarily behavior of cache misses due to false sharing, as this perturbation increase with increase in number of parallel threads and small number of nodes.



(a) Locked Based

(b) Reagent Based

Figure 1: Normalized throughput for Reagents-based and Lock-based Algorithm. Increase in throughput on increasing pureWork indicates decrease in contention between threads



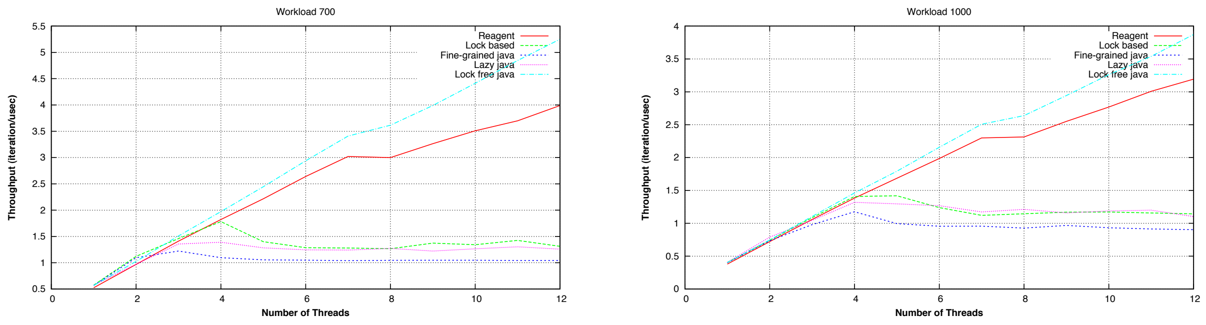
(a)

(b)

Figure 2: High Contention

3. There is a change (decrease) in the slope at thread count of 4 and 8. The first change can be attributed to the interaction of hyper-threading, virtual machine and JIT compilation. This leads to sharing of the CPU resource including the L1 cache . At thread count of 8, some thread must be scheduled on a different node. A cache line (synchronization) shared between two threads scheduled on different nodes incur the inter-node memory latency, which is considerably larger.

4. Throughput for the reagent is lower than the hand coded lock free implementation



(a) (b)

Figure 3: Low Contention

: One of the reason is that the extra layer of abstraction provided by the reagent adds an overhead and the JIT compiler is not able to completely optimize away the differences.

5 Conclusion and Future work

Reagents provide a fairly expressive framework for implementing fine-grained concurrent algorithm. Our experience as user of the framework was very positive. Also the performance of the algorithm compares fairly with the hand optimized lock-free implementation. Further work needs to be done to isolate the performance overheads of the abstraction layer and the impact of the garbage collection in benchmark which have larger footprints. Also hardware performance monitor must be used to measure the total number of CAS instruction executed to accurately measure the concurrent op throughput[2]. Also further sophisticated concurrent algorithm must be implemented.

Acknowledgment We greatly appreciate Aaron Turon for helping us understand the Reagent framework, being very responsive to our queries, and most importantly constantly encouraging us.

References

[1] M. Herlihy and N. Shavit, *The art of multiprocessor programming*, Morgan Kaufmann, 2008.

[2] P.F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind, *Using hardware performance monitors to understand the behavior of java applications*,

Proc. of the Third USENIX Virtual Machine Research and Technology Symp, 2004, pp. 57–72.

[3] Aaron Turon, *Reagents: Expressing and composing fine-grained concurrency*, 2012.