Parallelism in Programming Languages and Systems

Sirshak Das

Saturday 29th April, 2017; 16:47

Final Project Report: Due 05/01/17

Help Optimal and Language-portable Lock free Data Structures(Linked List only) in golang

1 Introduction

My final project is based on the paper "*Help Optimal and Language-portable Lock free Data Structures*" presented in *45th International Conference on Parallel Processing, 2016 by Bapi Chatterjee, Ivan Walulya and Philippas Tsigas from Chalmers University of Technology, Gothenburg, Sweden.* This paper talk about the widely used technique of helping used for creating Lock free data structures. The paper presents a technique to create lock free data structures in **language portable** way. This is a important requirement as lot of existing solutions leverage language specific constructs like *bit stealing* and *runtime object inspection.*

Helping is technique where threads try helping other threads complete the concurrent operation in case they wait in queue to finish there operations. This technique has also been discussed in the book *The Art of Multiprocessor Programming* in Chapter 6 section 6.4. In the book helping is done by each threading announcing its action in global data structure and concurrent threads completing the actions of already executing threads.

2 Lock-Free, Language-Independence Data Structures and Helping

The essence of Lock – Free Synchronization is a thread assumes that its alone and attempts an operation which constitutes of atomic Step using hardware atomic primitives(CAS) wrapped around as library Call. Retries until not interfered by other operations but the problem of starvation exist. This is removed by wait free algorithms.

The question is Why lock free ? It removes the performance bottleneck of spinning on locks or overhead of scheduling threads which is one of primary bottlenecks in multiprocessor high performance environment were all the cores spin on locks waiting for one thread to

finish and yield access. Making such algorithms lock free makes increases more utilization of processors which in turn increases throughput.

The basic building block in lock-free data structures is CAS(Compare and Swap operation) which is Atomic. Usually if you have one CAS operation in concurrent procedure it succeeds or backtracks which is pretty simple. The trouble happens in case of multiple CAS operations in same procedure which makes back tracking complicated. This paper modifies the Linked List Data structure to support local back tracking by using `back` pointers. This brings us to a new term used called *concurrent obstruction*. When 2 partially completed operations block each other they cause what is called *concurrent obstruction*.

Language-Independence is one of the most important aspects of this design. This approach uses *splice nodes* instead of *bit stealing* or *dynamic object inspection or inheritance*. Splice nodes are special nodes with not `null/nil back` pointer and $-\infty$ key value. These nodes are used to indicate half completed CAS operation typically used for operations with multiple CAS. During a delete operation a splice node is inserted in front of the deleted node to indicate a partially completed delete.

Helping essentially means if concurrent threads face obstruction or need to perform extra steps while traversing logically removed or deformed nodes the pending steps of one thread is performed by the other hence helping.

3 Help Optimal Linked List

This paper discusses both the Help optimal Linked List and Binary Search Tree(BST). For the purposes of this project I have limited my work to Linked List. In order to strengthen the argument of language portability I have converted the existing implementation of this algorithm in `Java` to `golang`. This paper lists implementing the algorithms in `golang` as future work. I have also implemented another lock free version of Linked List called Harris Linked List. This project presents the comparison between these 2 algorithms in `golang` implementation.

I will be going over the overview of Help Optimal Linked List for more on the Harris Linked List one can refer (TODO).

3.1 Node Structure

```
type node struct {
  key *utils.Key
  next *node
  back *node
}
```

The node in case of Help Optimal Linked List has a extra back pointer which is `nil` for normal nodes and not `nil` for *splice nodes*.

3.2 Linked List Initialization

```
type HelpOptimalLFList struct {
  head *node
  headNext *node
```

```
  tail *node
  tailNext *node
}
```

This is the structure of linked list. These are sentinel nodes which are initialized at the creation of the linked list with values $-\infty_1, \infty_0, \infty_0, \infty_1$ respectively.

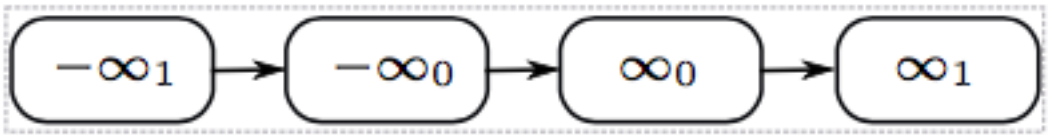


Fig. 3: Sentinel Nodes

3.3 Addition of Node

```
func (hoLFList *HelpOptimalLFList) Add(k *utils.Key) bool{
  pre := hoLFList.head
  suc := hoLFList.headNext
  cur := hoLFList.headNext
  nex := cur.next
  for true {
    //Search
    for cur.key.CompareTo(k) == true {
      if nex.back == nil {
        pre = cur
        suc = nex
        cur = suc
      } else {
        cur = nex.next
      }
      nex = cur.next
    }
    //isSplice Node
    if (nex.back != nil) {
      //Traverses Splice nodes and Helps removal
      for nex.back != nil {
        cur = nex.next
        nex = cur.next
      }
    } else if cur.key.Equals(k) == true {
      return false
    }
    //CAS p(k).nxt from s(k) to k

    if pre.casNext(suc, newNodeNext(k, cur)) == true {
      return true
    }
    //Local Back track
    suc = pre.next
    for suc.back != nil {
      pre = suc.back
      suc = pre.next
    }
    cur = pre
```

```
    nex = suc
  }
  //Dead Code
  return false
}
```

Add in linked list comprises of one CAS operation which is CAS on the next pointer of the previous node. The next pointer of the inserted node is local and not under contention so no need for CAS. As this list does not allow duplicate keys and the keys are ordered based on value in increasing order. The `add` operation first searches for appropriate position in linked list. While it searches for the node it also helps a concurrent delete which which has succeeded with CAS operation might inserted a splice node in front of a deleted node to indicate a partially completed deleted operation this can be removed by concurrent Add as indicated by the comments in code. Then a CAS is attempted on the previous nodes if successful `true` is returned else a local backtrack takes place to recompute the appropriate position.

3.4 Deletion of Node

```
func (hoLFList *HelpOptimalLFList) Remove(k *utils.Key) bool {
  pre := hoLFList.head
  suc := hoLFList.headNext
  cur := hoLFList.headNext
  nex := cur.next
  var marker *node
  mode := true
  nk := utils.NewKey()

  for true {
    for cur.key.CompareTo(k) == true {
      if nex.back == nil {
        pre = cur
        suc = nex
        cur = suc
      } else {
        cur = nex.next
      }
      nex = cur.next
    }
    if mode == true {
      //key not found or already logically removed
      if k.Equals(cur.key) == false || nex.back != nil {
        return false
      }
      marker = newNodeBack(pre, utils.NewKeyValue(nk.MinValue0))
      for true {
        marker.next = nex
        if cur.casNext(nex, marker) == true { //Logically Removing
          if pre.casNext(suc, nex) {
            return true
          }
          mode = false
          break
```

```
        }
        nex = cur.next
        if nex.back != nil {
          return false
        }
      }
    } else if nex != marker || pre.casNext(suc, nex.next) {
      return true
    }
    suc = pre.next
    for suc.back != nil {
      pre = suc.back
      suc = pre.next
    }
    cur = pre
    nex = suc
  }
  //Dead Code
  return false
}
```

The deletion of node is tricky cause it involves successful completion of 2 CAS operations. The remove operation inserts a *marker/splice* after the first CAS operation. Thus indicating that the node preceding the marker node has been marked for deletion. This is also termed as logically removing the node. With this local `mode` variable is also set which indicates partial completion of delete operation if the second CAS fails. This leads to backtrack and on return because the first CAS was successful in prior iteration this time only the second CAS is performed. This decision is based on the value of `mode`.

3.5 Contains

```
func (hoLFList *HelpOptimalLFList) Contains(k *utils.Key) bool{
  cur := hoLFList.getNext(hoLFList.headNext);
  for cur.key.CompareTo(k) == true {
    cur = cur.next
  }
  return k.Equals(cur.key) && cur.next.back == nil;
}
```

Contains just searches for the node. The only check performed is if the node is in the middle of delete operation which can be checked by seeing if the following node is a *splice node*

4 Setup and Sanity Test

Prerequisites: `golang` installed
https://golang.org/doc/install
There are 2 ways of downloading the code.
1. Git
2. Tar Ball
Steps

4.1 Git

4.1.1 Git Clone

```
$ git clone https://github.com/rishdas/ConcurrentSetGo.git
Cloning into 'ConcurrentSetGo'...
remote: Counting objects: 204, done.
remote: Compressing objects: 100% (191/191), done.
remote: Total 204 (delta 93), reused 0 (delta 0), pack-reused 7
Receiving objects: 100% (204/204), 647.66 KiB | 1.15 MiB/s, done.
Resolving deltas: 100% (93/93), done.
Checking connectivity... done.
```

4.1.2 Submodules Checkout

```
$ cd ConcurrentSetGo/
$ git submodule update --init --recursive
Submodule 'ASCYLIB' (https://github.com/LPD-EPFL/ASCYLIB.git) registered for
    path 'ASCYLIB'
Submodule 'ConcurrentSet' (https://github.com/bapi/ConcurrentSet.git)
    registered for path 'ConcurrentSet'
Cloning into 'ASCYLIB'...
remote: Counting objects: 14812, done.
remote: Total 14812 (delta 0), reused 0 (delta 0), pack-reused 14811
Receiving objects: 100% (14812/14812), 4.70 MiB | 1.73 MiB/s, done.
Resolving deltas: 100% (10785/10785), done.
Checking connectivity... done.
Submodule path 'ASCYLIB': checked out '
    d546e6c48f14c7e158228e53593caef651fd5626'
Cloning into 'ConcurrentSet'...
remote: Counting objects: 176, done.
remote: Total 176 (delta 0), reused 0 (delta 0), pack-reused 176
Receiving objects: 100% (176/176), 3.78 MiB | 1.69 MiB/s, done.
Resolving deltas: 100% (71/71), done.
Checking connectivity... done.
Submodule path 'ConcurrentSet': checked out '6
    aeb950a5620c43c946000d28618715d597ee1f2'
```

4.2 Tarball

```
$ tar -xJf ConcurrentSetGo.tar.xz
$ cd ConcurrentSetGo/
```

4.3 Building and Running Sanity Test

Building the benchmark binary

```
$ pwd
<Working Directory>/ConcurrentSetGo
$ export GOPATH=<Working Directory>/ConcurrentSetGo
$ cd src/
$ cd benchmark/
$ ls
benchmark.go  benchmark-script.py
$ go build
```

```
$ ls
benchmark  benchmark.go  benchmark-script.py
```

Command Line Options:

```
$ ./benchmark -h
Usage of ./benchmark:
  -a string
        Available Algorithms  (default=HelpOptimalLFList) (default "HelpOptimalLFList")
  -d int
        Test duration in seconds (0=infinite, default=2s) (default 2)
  -i int
        Fraction of insert/add operations (default=50%) (default 50)
  -k int
        Number of possible keys (default=100) (default 100)
  -n int
        Number of threads (default=2) (default 4)
  -r int
        Fraction of search operations (default=0%)
  -t  Sanity check (default=false)
  -w int
        Go Runtime warm up time in seconds(default=2s) (default 2)
  -x int
        Fraction of delete operations (default=50%) (default 50)
```

4.3.1 Help Optimal Linked List Sanity

```
$ ./benchmark -a HelpOptimalLFList -t true
Sanity Test Complete
Traversal Test :
true
```

4.3.2 Harris Linked List Sanity

```
$ ./benchmark -a HarrisLinkedList -t true
Sanity Test Complete
Traversal Test :
true
```

5 Experimental Evaluation

As mentioned algorithms `HelpOptimalLFList` and `HarrisLinkedList` are implemented in `golang`. All the experiments were carried out in machine with following specification.

```
$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
```

```
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 94
Model name:            Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
Stepping:              3
CPU MHz:               1256.406
BogoMIPS:              6816.60
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
```

5.1 Benchmark Parameters

The performance is measured by *Mops/s*. Each experiment is run for *5 seconds*. The average over *6 trials* is then taken.

In order to spawn threads `go func(...)...` is used. As `golang` does not have a notion of `join`. `WaitGroups` and first class channels with `select` were used for synchronization. `Channels` in `golang` are communicating mechanism between threads which is used by `golang` to felicitate the Communicating Sequential Processes(CSP) Model.

Workload Distribution: This experiment considers 3 workload distributions

a. *write-dominated*: 0% CONTAINS, 50% ADD and 50 % REMOVE.

```
$ ./benchmark -a=HelpOptimalLFList -d=5 -i=50 -k=128 -n=2 -r=0 -x=50
$ ./benchmark -a=HarrisLinkedList -d=5 -i=50 -k=128 -n=2 -r=0 -x=50
```

b. *mixed*: 70% CONTAINS, 20% ADD and 10 % REMOVE.

```
$ ./benchmark -a=HelpOptimalLFList -d=5 -i=20 -k=128 -n=2 -r=70 -x=10
$ ./benchmark -a=HarrisLinkedList -d=5 -i=20 -k=128 -n=2 -r=70 -x=10
```

c. *read-dominated*: 90% CONTAINS, 9% ADD and 1 % REMOVE.

```
$ ./benchmark -a=HelpOptimalLFList -d=5 -i=9 -k=128 -n=2 -r=90 -x=1
$ ./benchmark -a=HarrisLinkedList -d=5 -i=9 -k=128 -n=2 -r=90 -x=1
```

Set Size: This experiment computes results for each work load across key ranges 2^7 2^9 and 2^{10}

1. $2^7 = 128$

```
$ ./benchmark -a=HelpOptimalLFList -d=5 -i=9 -k=128 -n=2 -r=90 -x=1
$ ./benchmark -a=HarrisLinkedList -d=5 -i=9 -k=128 -n=2 -r=90 -x=1
```

2. $2^9 = 512$

```
$ ./benchmark -a=HelpOptimalLFList -d=5 -i=9 -k=512 -n=2 -r=90 -x=1
$ ./benchmark -a=HarrisLinkedList -d=5 -i=9 -k=512 -n=2 -r=90 -x=1
```

3. $2^{10} = 1024$

```
$ ./benchmark -a=HelpOptimalLFList -d=5 -i=9 -k=1024 -n=2 -r=90 -x=1
$ ./benchmark -a=HarrisLinkedList -d=5 -i=9 -k=1024 -n=2 -r=90 -x=1
```

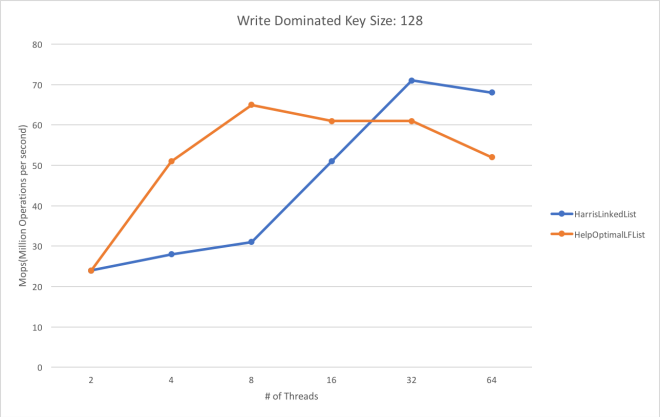
5.1.1 Interpreting the results

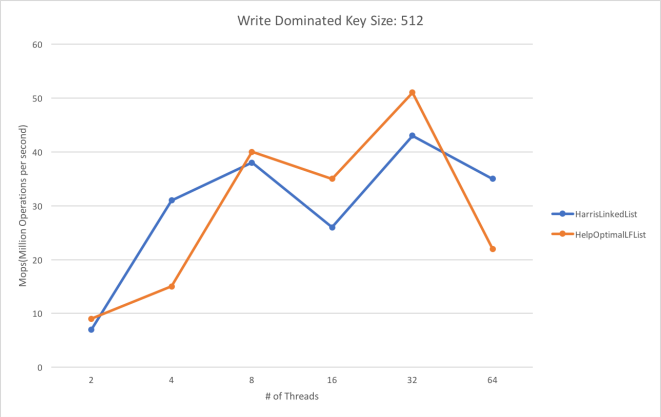
```
$ ./benchmark -a=HelpOptimalLFList -d=5 -i=50 -k=128 -n=2 -r=0 -x=50
128 50 50 0 HelpOptimalLFList 2 10713430
```

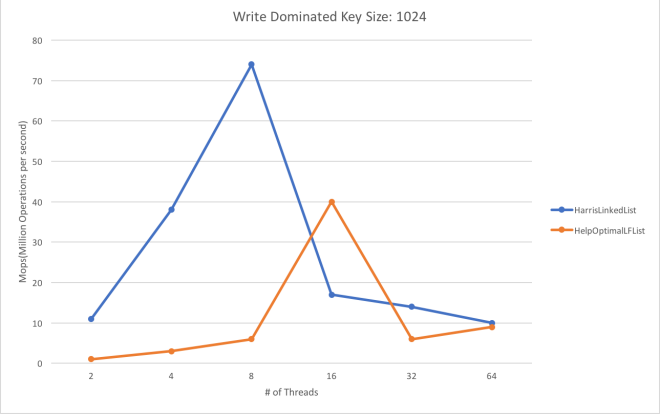
(Key Range) (% of Add) (% of Delete) (% of Contains) (Algorithm) (# of Threads) (Operation per Second)

6 Experimental Results

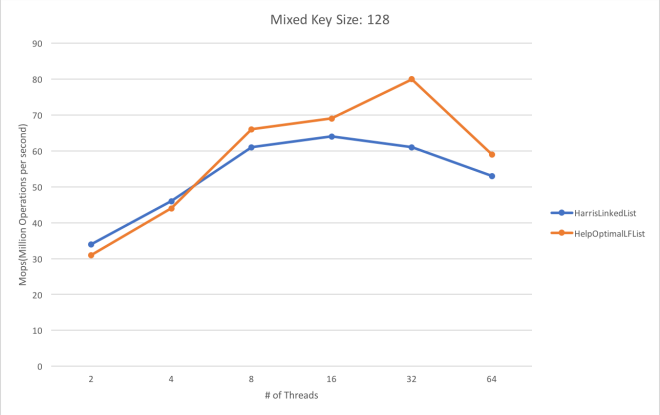
6.1 Write Dominated Workload Experiment Results for different key sizes

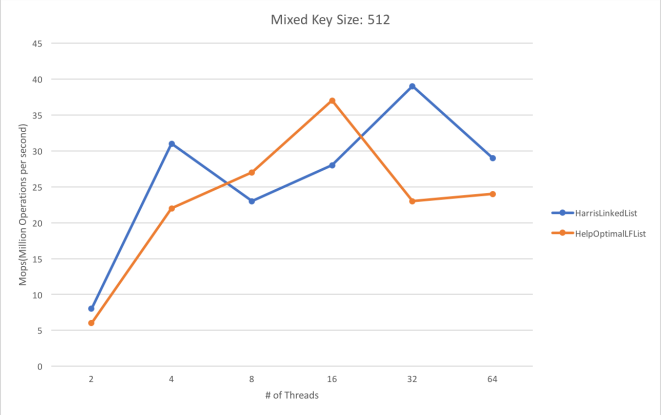


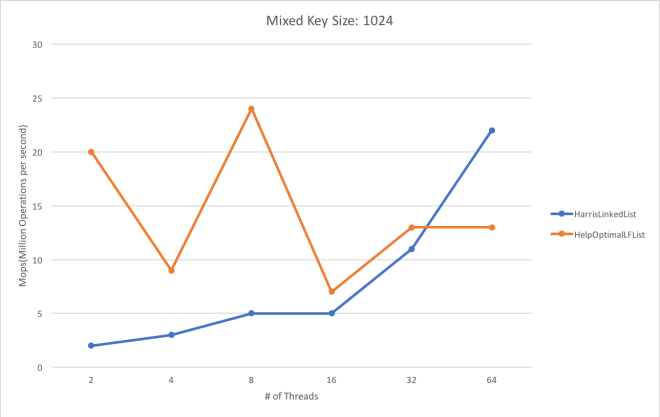




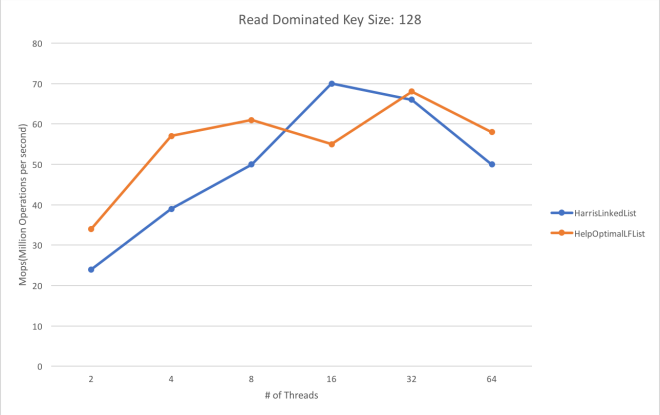
6.2 Mixed Workload Experiment Results for different key sizes

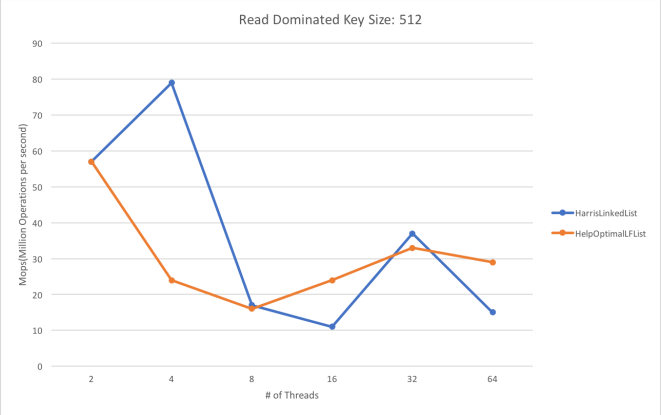


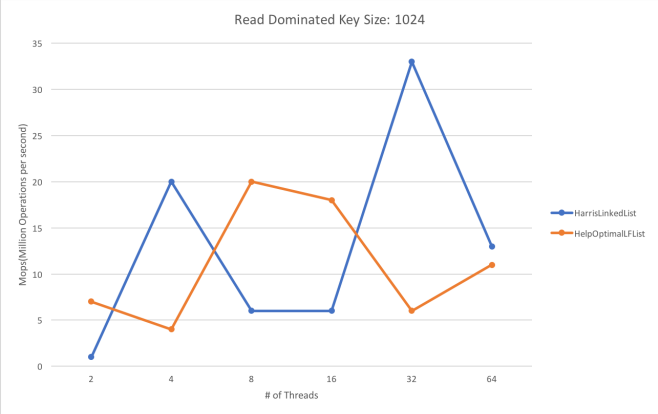




6.3 Read Dominated Workload Experiment Results for different key sizes







7 Conclusion

Implementing Help Optimal Linked List in `golang` itself strengthens the argument presented in the paper which is this technique being language portable. As far as the efficiency is concerned the results are mixed compared to standard lock free data structures like the Harris Linked List. As you can see its more or less comparable in case of *Write Dominated* and *Mixed* workload but performs poorly in case *Read Dominated* workload. This is something which should be analyzed. But the more or less comparable results should compel for further research on Language Portable Lock Free Data Structures.

8 References

1. [Harris Linked List] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in 15th DISC, 2001, pp. 300–314.
2. [Help Optimal Linked List] Chatterjee, B., Walulya, I. and Tsigas, P. (2016) Help-optimal and Language-portable Lock-free Concurrent Data Structures. Technical report - Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, no: 2016-02 ISSN: 1652-926X.