CiCTrie - C Implementation of CTrie

Daniel Solomon
ID:
Email: DanielSolomon94.ds@gmail.com

Or Karni
ID:
Email: orkarni@mail.tau.ac.il

***Abstract*—We ported high-level memory managed data structure *CTrie* to an unmanaged language *C*, using *hazard pointers* mechanism, while keeping the basic operations of the data structure. We compared our implementation against Scala's standard library *TrieMap* (*CTrie*) using similar tests as in the original article.**

***Index Terms*—CTrie, Hazard Pointers.**

I. Introduction

CTrie [1] [2] (or concurrent hash-trie) is a concurrent *thread-safe*, *lock-free* implementation of a hash array mapped trie. This data structure is consists of *key-value* pairs and it supports the following operations:

- **insert**: add a new *(key, value)* pair.
- **remove**: remove a *(key, value)* pair if it exists.
- **lookup**: find the *value* (if any) for a specific *key*.

In addition the **CTrie** data structure has a *snapshot* operation which is used to implement consistent *iterators*. In fact **CTrie** is the first known concurrent data structure that supports *O(1)*, *atomic*, *lock-free* snapshots.
CTrie aspires to preserve the space-efficiency and the expected depth of hash tries by *compressing* after removals, disposing of unnecessary nodes and thus keeping the depth reasonable. The **CTrie** implementation is based on single-word *compare-and-swap* instructions.

II. Implementation

We wanted to give a simple interface for using *CiCTrie* [3] which will be as close as possible to object oriented programming, therefore we used *Object Oriented C*; The user creates a *ctrie_t** struct using a "constructor" *create_ctrie* and use its functions passing itself as a "self object" (such as: `ctrie->insert(`**`ctrie`**`, key, value)`).
This way we could port easily the original *Scala* code from the original paper [1] and from the source code of the standard library itself [4].
In addition, *CiCTrie* supports only int keys and values, although if needed it can support any data structure as *keys* and *values*, without managing their memory. It can be implemented easily, if one will pass a *hash* function when calling `ctrie_create` and all keys and values will be `void*`.

A. Basic Operations

All basic operations were implemented; `insert, lookup, remove, clean, compress, to_contracted and clean_parent`; Where the latter one was implemented but it is not used since this optimization does not improve any performance at all.

B. Memory Reclamation

Hazard pointers are a mechanism which aims to solve the ABA and safe memory reclamation problems [5] [6]. Each thread maintains a list of *hazard pointers* to resources it currently uses. This list usually has a fixed size and is kept small. A used resource may not be freed or modified.
In the worst case scenario a specific thread holds 4 different pointers that were dynamically allocated, for example: When reaching a *TNode* we must clean it, the thread acquired 2 hazard pointers for the parent *INode* and its *MainNode* of type *CNode* and in `compress` function the thread acquires another 2 hazard pointers for each *INode* child and its *MainNode*.
In our implementation, each thread has a list of 6 *hazard pointers* in total - 4 "ctrie" *hazard pointers* and 2 "list" *hazard pointers*. The list *hazard pointers* is used for traverse a *LNode* and for using "temporary" *hazard pointers* in compress (hazard pointers which can be shortly overridden).
We implemented two lighter versions using 5 *hazard pointers* in total [7] for each thread (4 *hazard pointers* for *LNode* and "ctrie" *hazard pointers* and 1 "temporary" *hazard pointer*) and another version using 4 *hazard pointers* only (minimal as mentioned) [8]. This implementation is dividing the *hazard pointers* into 2 pairs. One pair is used for the CAS while the other is used for preparing the new node to be inserted. In each call, the current pair is determined by the level oddity - so it alternates. When inserting or removing a *SNode* or a *LNode*, the pair of the current inode and main node is the CAS pair, and the other 2 *hazard pointers* can be overridden freely. When cleaning a *TNode*, the parent *INode* and it's *MainNode* are the CAS pair, and the other 2 *hazard pointers* can be overridden freely.
Comparing both lighter versions to the 6 *hazard pointers* version we got better performance for the greedy 6 *hazard pointers* implementation (See *BENCHMARKING* section).

Before accessing anything, a thread must first place a *hazard pointer* to ensure no one will free the desired memory. However, *hazard pointers* by themselves don't guarantee success. For example, if a thread reaches an *INode* and wishes to continue down to it's *MainNode*, it reads the pointer to the *MainNode*, places a *hazard pointer* and then accesses the node. But what if between the read and the placement of the *hazard pointer*, another thread changes the *CTrie* and frees the *MainNode*? The original thread would place a *hazard*

pointer on an invalid address and wouldn't know it's already freed. To solve this problem, we added a *marked* field to *INode*, *LNode* and *CNode*. The marked field indicates that this node has been removed from the *CTrie* and is scheduled for freeing. In places where a race might occur, the accessing thread must validate it's state after placing a *hazard pointer*. In the example above, the thread must validate that:

1) The *INode* isn't marked.
2) The *Inode* still points to the expected *MainNode*.

If the race occurs the thread must *RESTART* its operation. Any node that points to another node should have a *marked* field for race detection, the only nodes that points to others are *INode*, *LNode* and *CNode*.
Since *marked* property must be accessible for any thread and must be up-to-date any time, after marking a node (or multiple nodes), a *FENCE* operation is invoked.
Each thread holds a fixed size free list. The size is determined by the number of threads times number of *hazard pointers* that is used in the implementation (For example in our 6 *hazard pointers* implementation, with 4 threads, each thread has a free list of size 24). This size is needed in order to make sure no "dead locks" will happen, since when the free list is full, a thread must free at list one member of the list before continuing its operations, therefore when getting to the free phase, it must be guaranteed that at least one member of the free list does not occur in any *hazard pointers* lists of the other threads.

C. Snapshot

The *CTrie* data structure support a *O(1)*, *atomic*, *lock-free snapshot* operation. This feature is implemented by replacing the CAS operation with a GCAS operation. The GCAS simulates an atomic DCAS, which doesn't exist natively on most machines. Each *INode* is assigned to a generation. A snapshot increases the root *INode*'s generation number. Each thread that encounters an *INode* of and older generation than the root it has seen at the start of the operation, it copies the inode and it's main node, replaces the old inode with the copy and continues operating on the copy. This way, the *INode*'s of the original generation stay unchanged and thus the snapshot isn't affected. With the concern of memory management, the snapshot procedure is a bit more complicated. In our implementation, each *MainNode* also has a generation number. When a thread removes a node of an older generation from the trie, it doesn't free it because it "belongs" to a snapshot. However, freeing a snapshot is a complicated action. Because of the lazy copying of the nodes, it is unknown whether a node in the snapshot is still in the actual new generation *CTrie*, and so it mustn't be freed. Because of this, and the rarity of snapshot operations, which are less useful in *C* than they are in *Scala* because there are no iterators, we've decided that a snapshot cannot be freed, which creates an acceptable memory leak. The snapshot implementation [9] is implemented in a different branch because of it's inherent difference from the "clean" implementation.

III. Benchmarking

A. Experiments

We performed experimental measurements on a machine with 44 cores and 2 hyper-threads per core, 88 threads total. *Scala* version 2.12.3, JDK 8 and sbt version 1.0.0 configuration was used.
Since we are using integers as keys, there will be no *LNode*s at all, because a hash collision will occur only if the keys collide. Therefore, we changed the *hash* function to: $hash(x) = x/10$ (both in *CiCTrie* and in Scala) in order to create collisions of different keys.
Each benchmark was run for 20 iterations with 1, 2, 4, 8, 16, 32, 44, 66 and 88 threads per iteration. We measured *insert*, *lookup* and *remove* operations by generating a sample of 2 million numbers between 0 and 5 million. Each run starts from an empty *ctrie* instance, inserts the whole sample, lookups for each key and then removes all keys. Insertions, lookups and removes were time measured separately. All operations were divided equally by all available threads (for example, in configuration of 4 threads, each thread runs 500,000 inserts, lookups and removes). The same benchmark was performed for our *C* implementation and for the *Scala* implementation (See Fig. 3). The *Garbage Collector* was invoked between *Scala* operations and wasn't included in the measured time.
The first 6 benchmarks don't include the *Scala* and compare the 4 and 5 *hazard pointers* C implementations with the 6 *hazard pointers* C implementation (See Fig. 1, 2)
The final 3 benchmarks are called 90-9-1, 80-15-5, 60-30-10 consist of inserts, lookups and removes invoked in the respective ratio. 2 million operations on keys between 0 and 5 million are created and shuffled (See Fig. 4).

B. Results

As is clearly visible, the *Scala* implementation's performance is better than the *C* implementation. We attribute this performance difference to *Scala* and *JVM* optimizations. However, it should be noted that the *C* implementation consumes much less memory (around two times less memory). Also, the gap between the implementation grows smaller as the thread count increases (except for inserts). This happens because the *Scala* implementation doesn't benefit very much from more than 32 threads, while the *C* implementation does, so it might stay improvable with more threads while *Scala* will not.
One anomaly we've noticed is a considerable slowdown between 1 and 2 threads. This phenomenon happened only on the benchmarking machine and we don't know it's cause.
Another interesting property is the unbridged gap between the implementations in the insert benchmark. A constant gap of about 1 second remained between the implementations even when the thread count increased. We've profiled the *C* implementation's performance in order to improve the performance. There wasn't an abnormal amounts of *RESTART*'s. The memory management and *hazard pointers*'s time consumption is not significant enough to cause the performance difference. When using a profiling tool, *perf* [10], no suspiciously time

consuming routine has arose. It appeared that simple memory reads and writes took most of the time.

As for the 5 *hazard pointers* implementation, we've had better performance with the 6 *hazard pointers* implementation. The overhead of the *hazard pointers* check in `scan` is probably lower than the `hp_list` management overhead (less *hazard pointers* require more sophisticated management).

Notice that both in *Scala*'s and *C*'s results, except to few number of threads the longest run and shortest run are very close to each other. The standard deviation for insert and lookup is smaller than the remove's most likely because the probability to be *RESTART*ed in remove is bigger (*CAS* races and *TNodes*) [All numbers are in nano-seconds] (See Table I and Table II).

TABLE I: Scala - Variance/Min/Max

Threads	Operation	Variance	Min	Max
1	insert	8.46e+14	1565517868	1667691751
2	insert	2.86e+15	863842838	1051936962
4	insert	1.02e+15	467942163	561266596
8	insert	3.52e+14	260197276	322273773
16	insert	6.46e+13	156279438	180283871
32	insert	1.18e+13	113353978	126449294
44	insert	7.25e+12	101493346	115129822
66	insert	1.45e+13	93313236	110542513
88	insert	1.93e+13	99354941	115267184
1	lookup	1.30e+16	1588477005	1967220906
2	lookup	1.68e+14	831415860	892782333
4	lookup	5.38e+13	415750029	443283217
8	lookup	2.57e+13	213251410	235864087
16	lookup	2.62e+12	109709780	115369755
32	lookup	1.41e+11	57900029	59351450
44	lookup	7.08e+13	44471729	84596094
66	lookup	2.14e+13	33457898	47158422
88	lookup	5.95e+12	38603780	48496281
1	remove	3.42e+15	2341652734	2558173043
2	remove	1.27e+15	1293776067	1417738473
4	remove	9.26e+14	648160830	768317981
8	remove	4.94e+14	360462934	445821495
16	remove	1.37e+14	199690864	250172785
32	remove	8.03e+13	121204469	162866851
44	remove	2.74e+13	119500160	136728323
66	remove	1.20e+13	101274083	114641203
88	remove	1.28e+14	104149189	153960945

TABLE II: C - Variance/Min/Max

Threads	Operation	Variance	Min	Max
1	insert	9.15e+14	3373524969	3476078641
2	insert	3.26e+17	3299984707	5128231007
4	insert	1.21e+17	1894273634	3016384274
8	insert	2.15e+16	1165826762	1650697393
16	insert	5.55e+15	923573895	1228303155
32	insert	2.76e+15	837945372	1021218363
44	insert	1.21e+15	815866044	963055571
66	insert	3.33e+14	805731658	874692640
88	insert	2.40e+14	805209930	867190137
1	lookup	2.09e+15	2554887913	2702243292
2	lookup	1.53e+18	1557068892	5305724781
4	lookup	7.64e+17	771727951	3518754762
8	lookup	2.21e+17	388718763	1884118200
16	lookup	9.10e+16	192139601	1287779436
32	lookup	5.24e+16	99774608	829316269
44	lookup	3.68e+16	74222433	766205832
66	lookup	1.81e+16	52388882	395748260
88	lookup	4.09e+15	59396617	244680555
1	remove	6.59e+15	7349148262	7619081115
2	remove	1.83e+18	6318573302	10773737757
4	remove	8.22e+17	3483085900	6357503854
8	remove	2.70e+17	1848544432	3557135601
16	remove	1.34e+17	946780919	2378249792
32	remove	9.41e+16	545096904	1531001275
44	remove	9.61e+16	424420636	1535618365
66	remove	2.41e+16	290965405	733644623
88	remove	1.82e+15	228875070	365224656

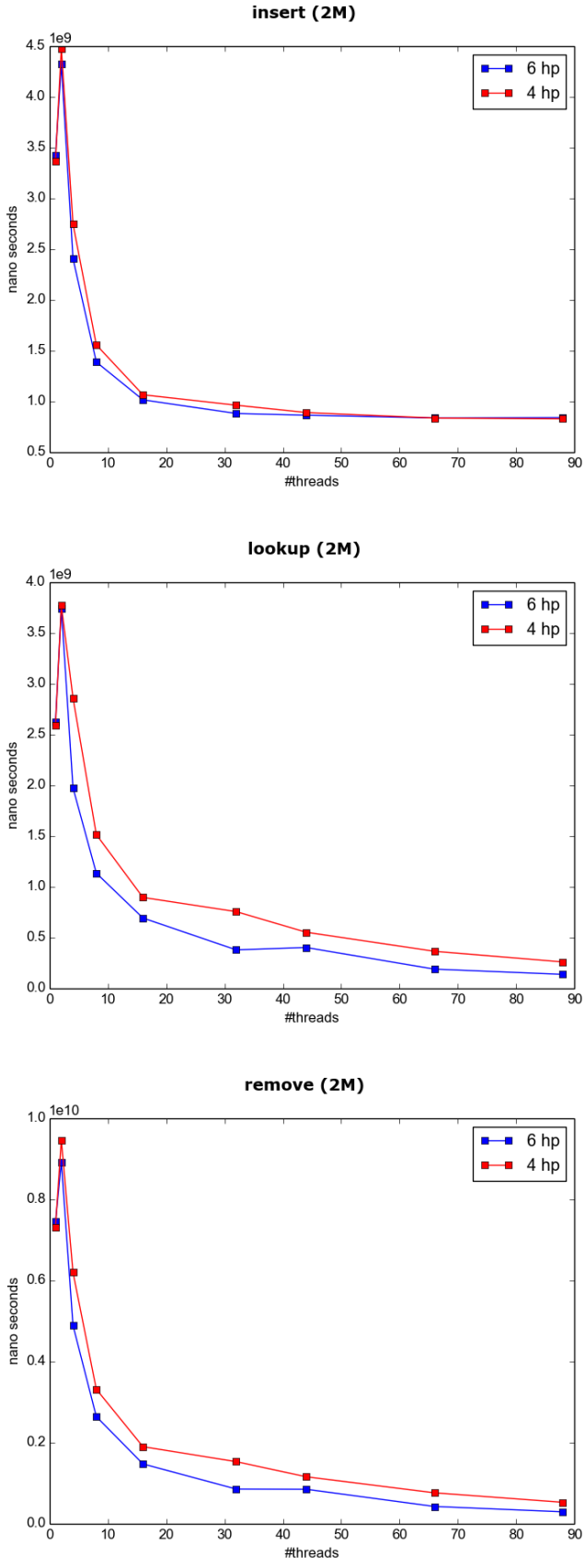
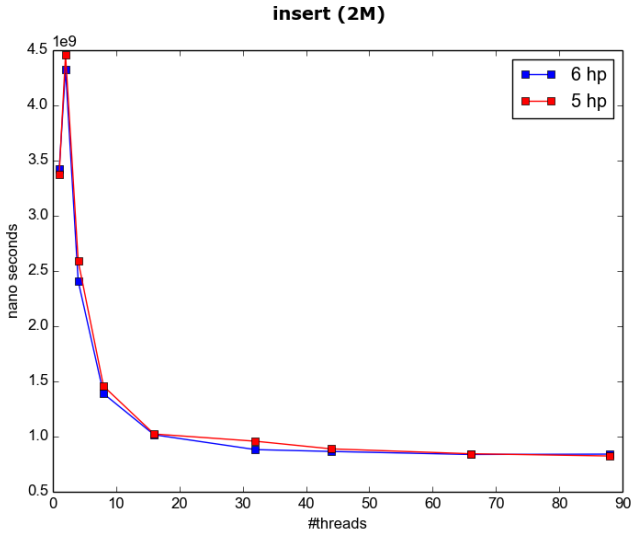
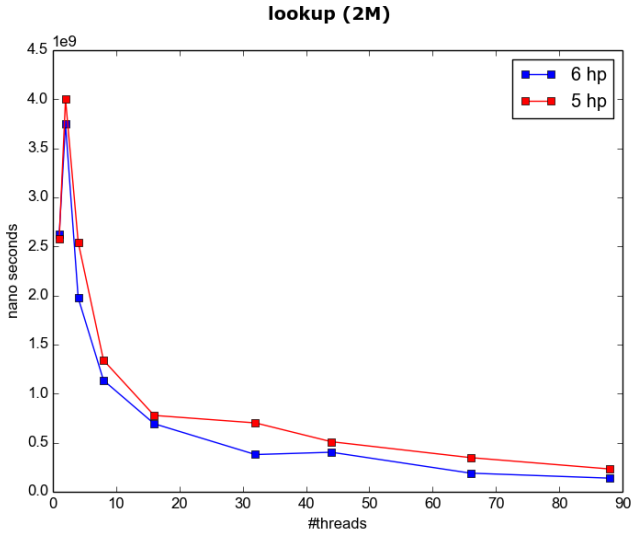


Fig. 1: hazard pointers implementations (6 vs 4)





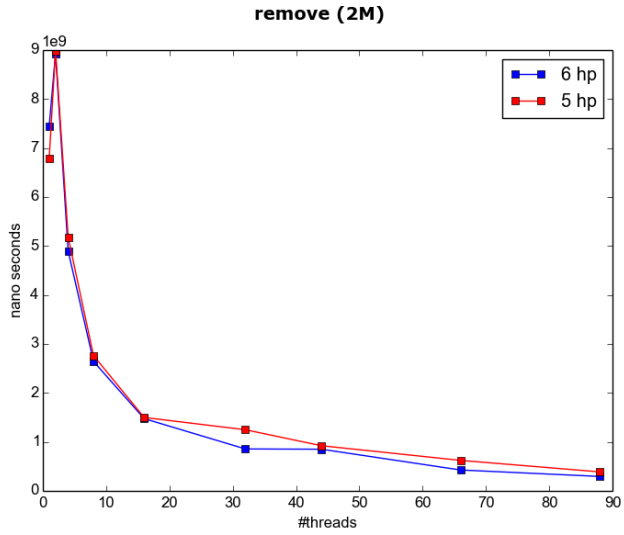
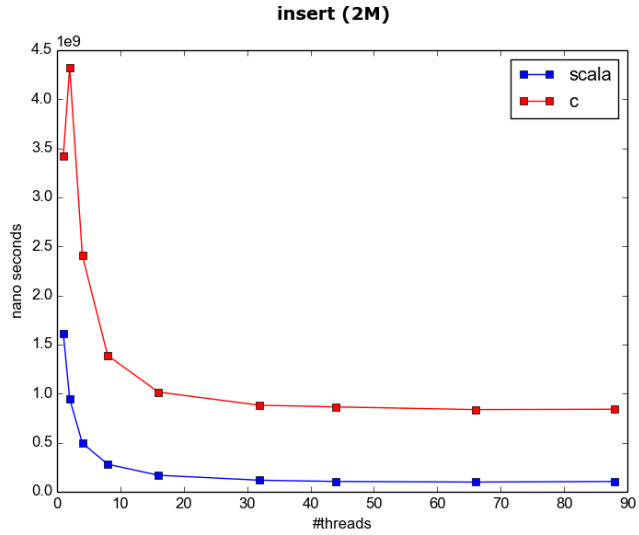
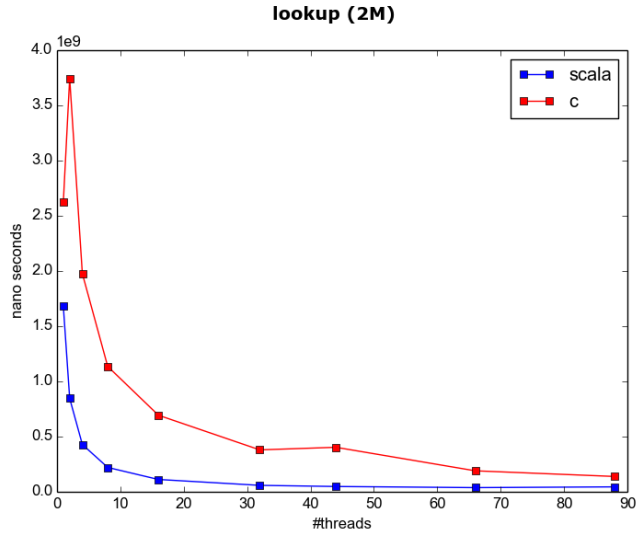


Fig. 2: hazard pointers implementations (6 vs 5)





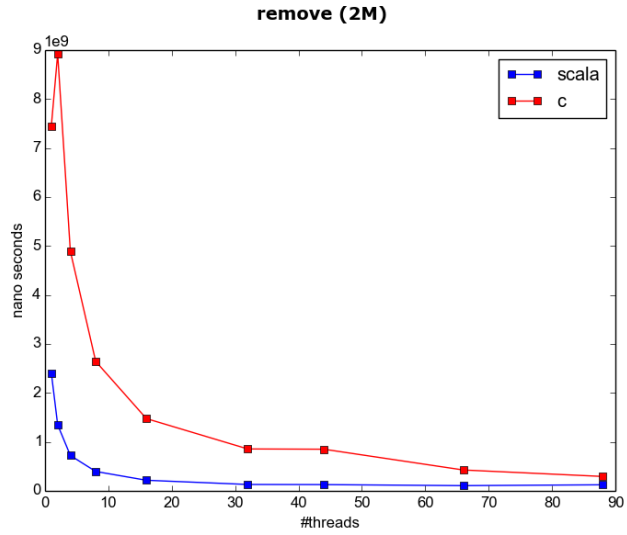


Fig. 3: C vs Scala implementations (isolated operations)

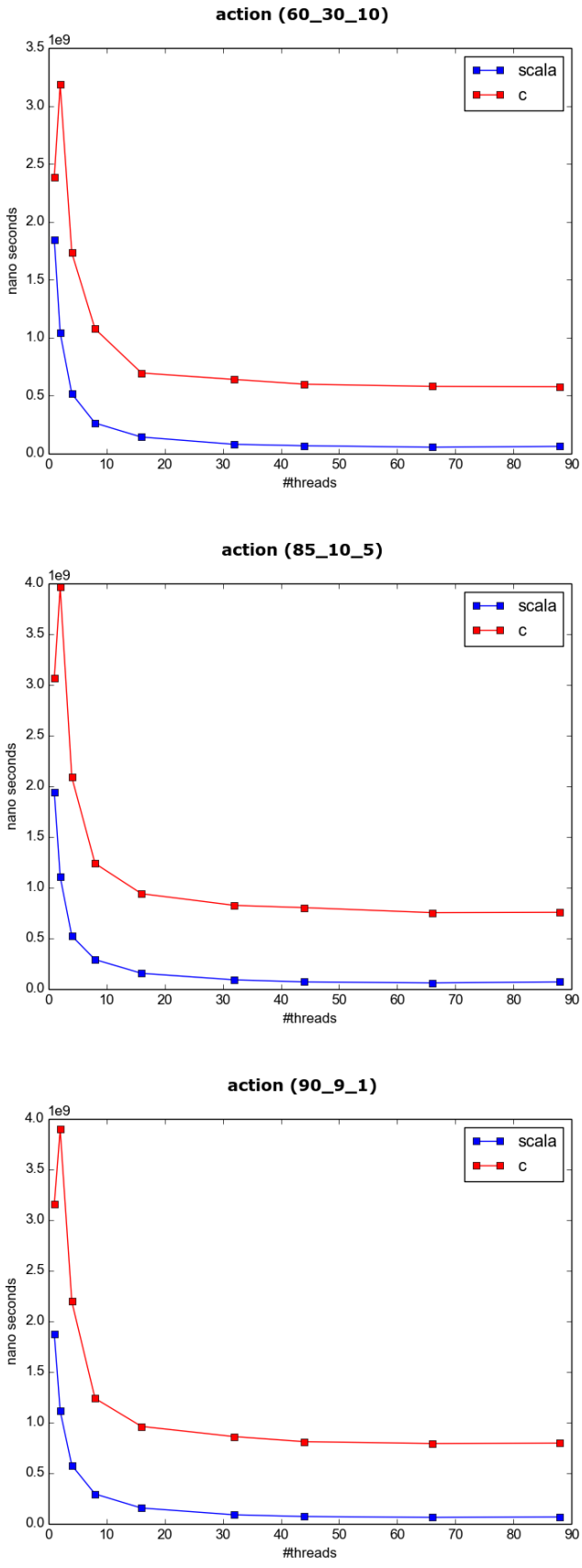


Fig. 4: C vs Scala implementations (heterogeneous operations)

REFERENCES

[1] Original article representing **CTrie** - https://axel22.github.io/resources/docs/ctries-snapshot.pdf.

[2] **CTrie** wikipedia reference - https://en.wikipedia.org/wiki/Ctrie.

[3] The **CiCTrie** on GitHub - https://github.com/DanielSolomon/CiCTrie

[4] TrieMap Scala source code - https://www.scala-lang.org/api/current/scala/collection/concurrent/TrieMap.html.

[5] Original article of hazard pointers - https://www.research.ibm.com/people/m/michael/ieeetpds-2004.pdf

[6] Hazard pointers wikipedia reference - https://en.wikipedia.org/wiki/Hazard_pointer.

[7] The **CiCTrie** with 5 hazard pointers - https://github.com/DanielSolomon/CiCTrie/tree/71f08706f92d5023823fd08de9300281022b8f98

[8] The **CiCTrie** with 4 hazard pointers - https://github.com/DanielSolomon/CiCTrie/tree/4hp

[9] The **CiCTrie** with snapshot - https://github.com/DanielSolomon/CiCTrie/tree/snapshot

[10] Perf wiki- https://perf.wiki.kernel.org/index.php/Main_Page.