

An Experimental Study of Memory Management in Rust Programming for Big Data Processing

Shinsaku Okazaki

Boston University

May, 2020

Motivation

Many of existing data-flow based Big Data Processing systems like Apache Spark and Flink have the following weaknesses!

- ▷ Using Java Virtual Machine (JVM)
 - ▷ Using additional computation
- ▷ Automated Memory Management
 - ▷ Generative Garbage Collection

Rust is a promising candidate for development of Big Data processing tools.

Rust

Rust is a "**system language**" which has unique memory management concept.

- ▷ A system language does not have Garbage Collection.
- ▷ Rust ensures memory safety.
- ▷ It is easy to write safe threaded code in Rust.
- ▷ Rust uses LLVM compiler infrastructure and provides high performance.

Problem Description

Our goal is to determine the magnitude of performance change regarding the following aspects.

- ▷ Memory Management for High Complex Nested Objects
- ▷ Different Rust Memory Management Strategies
- ▷ Automated Reference Counting (Rc) vs. Reference
- ▷ Multithreading using Atomic Reference Counting (Arc)
- ▷ Arc vs. Deep Copy on Overall Performance of Big Data Processing

Different Rust Memory Management Strategies

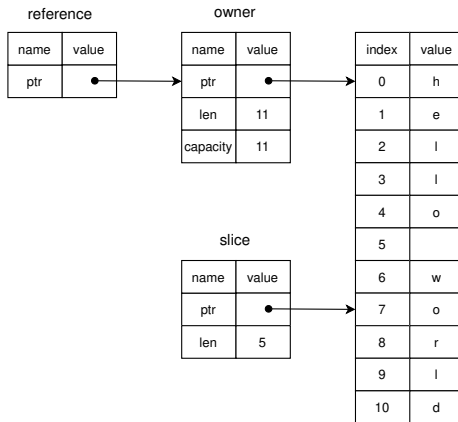
Each one has different memory representation.

- ▷ Owner
- ▷ Reference
- ▷ Slice

Different Rust Memory Management Strategies

Each one has different memory representation.

- ▷ Owner
- ▷ Reference
- ▷ Slice



Each one may have different memory access time.

Reference Counting

Advantage

- ▷ Sharing ownership
- ▷ Dinamic memory de/allocation

Disadvantage

- ▷ Need to check reference count
- ▷ Heap allocation

Complex Object - Memory Ownership and Borrowing

A Complex Object like a nested Customer Object like those defined in the complete Web-based shop.

Memory Ownership

```
1 struct CustomerOwned {  
2     key: i32,  
3     total_purchase: f64,  
4     zip_code: String,  
5     order: OrderOwned  
6     ... 10 more  
7 }
```

Memory Borrowing

```
1  
2 struct CustomerBorrowed<'a> {  
3     key: &'a i32,  
4     total_purchase: &'a f64,  
5     zip_code: &'a String,  
6     order: &'a OrderBorrowed<'a>  
7     ... 10 more  
8 }
```


Complex Object - Slicing and Reference Counting

CustomerSlice has slices of String instead of reference of String

Memory Slicing

```
1
2 struct CustomerSlice<'a> {
3     key: &'a i32,
4     total_purchase: &'a f64,
5     zip_code: &'a str,
6     order: &'a OrderSlice<'a>
7     ... 10 more
8 }
```

Atomic Reference Counting

```
1
2 struct CustomerRc {
3     key: Rc<i32>,
4     total_purchase: Rc<f64>,
5     zip_code: Rc<String>,
6     order: Rc<OrderRc>
7     ... 10 more
8 }
```

Multithreading with Atomic Reference Counting

Atomic Reference Counting

Advantage

- ▷ Sharing ownership
- ▷ Dynamic memory de/allocation
- ▷ Sharing among multithreads

Disadvantage

- ▷ Need to check reference count
- ▷ Heap allocation
- ▷ Atomic operation

Impact of Memory Management on performance of Big Data Processing

▷ Merge-sort

- ▷ Many contiguous memory de/allocations

▷ Tree-aggregate

- ▷ Many intermediate HashMap like data structures

▷ K-Nearest-Neighbors (KNN)

- ▷ Document preprocessing

Experiment 1: Accessing Object with Different Variable Type

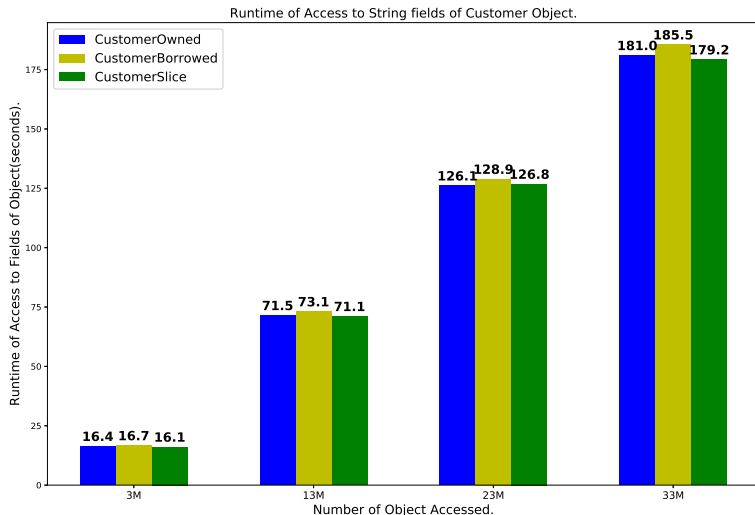
Question

- ▷ How much memory access time are different among different memory management strategies used in complex objects specification?

Evaluation

- ▷ **Construct Complex objects**
- ▷ With different memory management strategies: **Owner, Reference, Slice**
- ▷ **CustomerOwned** vs. **CustomerBorrowed** vs. **CustomerSlice**
- ▷ Measure access time of different **fields of complex objects**

Experiment 1: Accessing Object with Different Variable Type



Experiment 2: Assessment of different reference methods in Rust

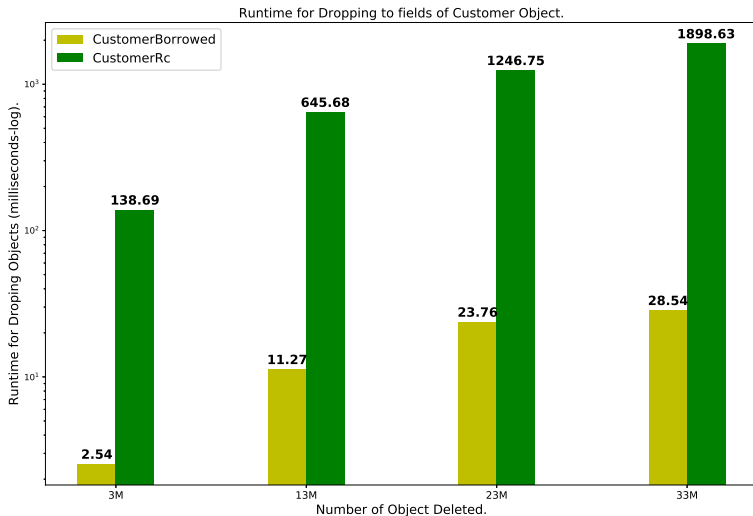
Question

- ▷ How much does Reference Counting hits performance?

Evaluation

- ▷ **Custruct complex objects**
- ▷ **Reference Counting** vs **reference**
- ▷ CustomerRc vs. CustomerBorrowed
- ▷ Measure time to **drop variables of complex objects**

Experiment 2: Assessment of different reference methods in Rust



Experiment 2: Assessment of different reference methods in Rust

Result

- ▷ Dropping Reference Counting is about **60 times slower** than normal reference.

Discussion

- ▷ Reference Counting needs some CPU cycles to check reference count.
- ▷ In complex objects, overhead is significant.

Experiment 3: Merge-sort

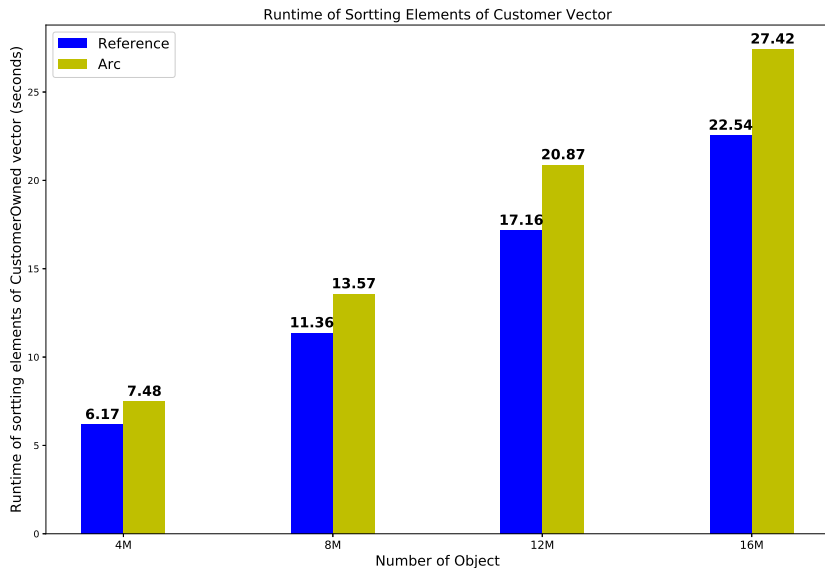
Question

- ▷ How much does sharing set of data with Atomic Reference Counting have impact on performance of merge-sort algorithms?

Evaluation

- ▷ Share **vector** of complex objects
- ▷ **Atomic Reference Counting (Arc)** vs **normal reference**
- ▷ Measure **runtime of merge-sort algorithms**

Experiment 3: Merge-sort



Experiment 3: Merge-sort

Result

- ▷ Arc are about **21% slower** than normal reference.

Discussion

- ▷ Atomic Reference Counting needs to check reference count.
- ▷ Atomic operations are more expensive than ordinal operations.

Experiment 4: Tree-aggregate

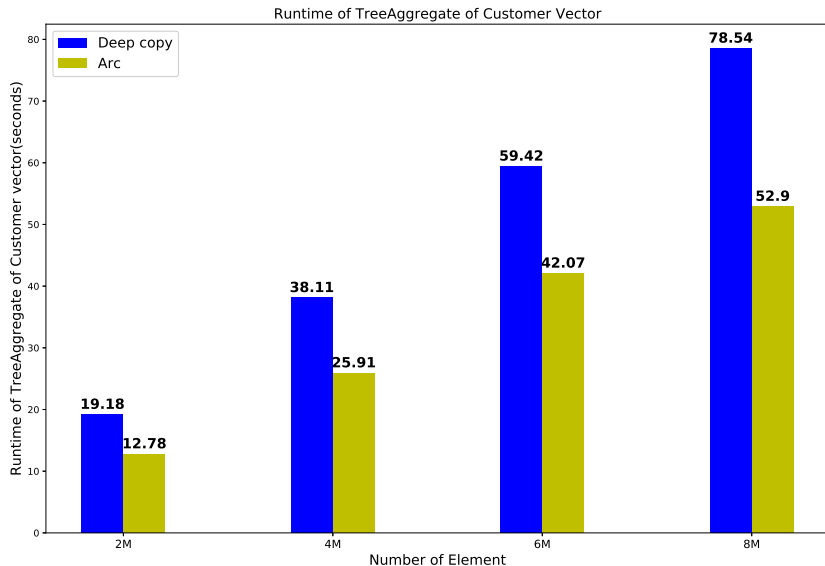
Question

- ▷ How much are runtime differences between sharing elements of data with Arc and deep-copying elements of data?

Evaluation

- ▷ Share **elements** of complex object
- ▷ **Atomic Reference Counting (ARC)** vs **Deep copy**
- ▷ Measure **runtime of Tree-aggregate algorithms**

Experiment 4: Tree-aggregate



Experiment 4: Tree-aggregate

Result

- ▷ Deep-copies are from **40 to 50% slower** than Arc.

Discussion

- ▷ Deep-copy of complex object is expensive.
- ▷ Performing deep-copy many times may lead significant overhead.

Experiment 5: K-Nearest-Neighbors (KNN)

Question

- ▷ What are better memory management strategies for common Machine Learning Algorithms?

Evaluation

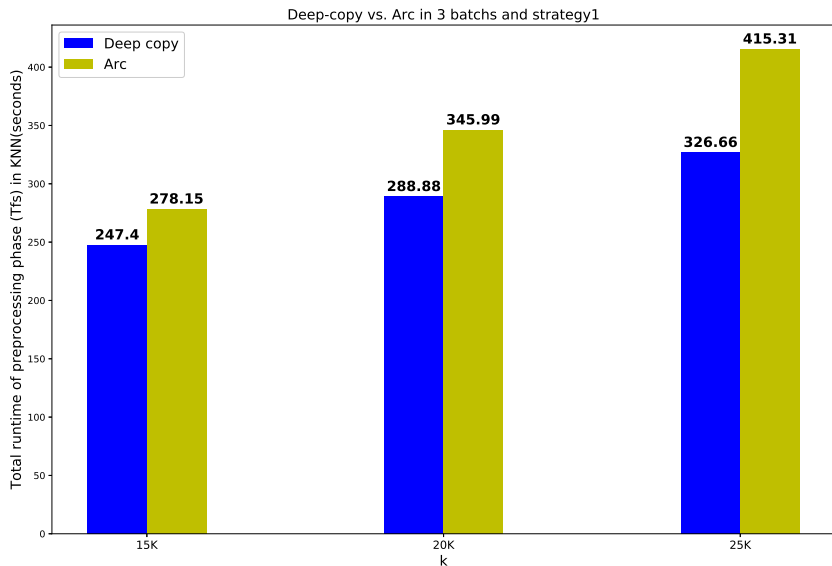
- ▷ Document classification on Wikipedia page dataset
 - ▷ train set: 100×10^3 pages
 - ▷ test set: 18×10^3 pages
- ▷ Preprocessing phase: calculating **Term-Frequencies (TFs)**
- ▷ String manipulation
- ▷ Measure runtime of **preprocessing time of KNN algorithms**

Experiment 5: K-Nearest-Neighbors (KNN)

Parameters

- ▷ Data Acquisition
 - ▷ **Atomic Reference Counting** (Arc)
 - ▷ **Deep-copy**
- ▷ Dimensions of feature matrices
 - ▷ 15K
 - ▷ 20K
 - ▷ 25K

Experiment 5: K-Nearest-Neighbors



Experiment 5: K-Nearest-Neighbors

Result

- ▷ **Arc** is **at most 38% slower** than deep-copy.

Discussion

- ▷ Deep-copying String is cheaper operation than Arc:
 - ▷ reference checking
 - ▷ atomic operation

Findings

- ▷ Use normal reference rather than Reference Counting whenever it is possible.
- ▷ Trade-off between **runtime performance** and **lifetime tracking**.
- ▷ **Avoid using Arc** when we can use reference.
- ▷ Use Arc instead of deep-copy, when **complexity of objects is large**.
- ▷ Use deep-copy, when **complexity of objects is small**, like String.

Conclusion

- ▶ We have seen that impact of memory memory Management is very large, Specially when working on High Complex Object structures and Large Volume of Data Set.
- ▶ Using a System Language like Rust is very promising to avoid using large CPU computation for memory management.
- ▶ By using Rust, writing memory-safe code in multithreading is fairly easy.