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)
  - Abstracting hardware usage sacrificing 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 easier to write safe threaded code in Rust.
- Rust is a relatively new language.
- 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.

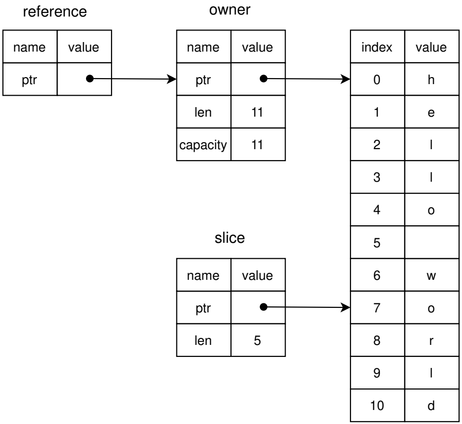
- Memory Management for High Complex Nested Objects
- Different Rust Memory Management Strategies
- Automated Reference Counting (Rc) vs. Reference
- Multithreading using Atomic Reference Counting (Arc)
- Arc vs. Deep Copy on Overall Performance of Big Data Processing

Different Rust Memory Management Strategies

Each one has different memory representation.

- ▷ Owner
- ▷ Reference
- ▷ Slice

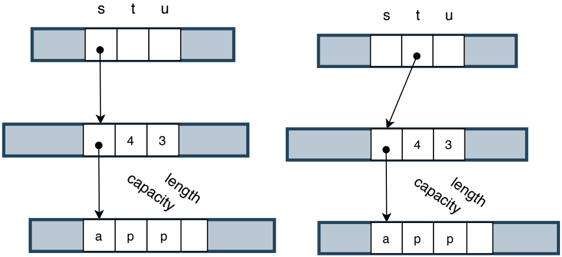
```
let owner = String::from("hello world");
let slice = owner[7:10];
let reference = &owner;
```



Each one may have different memory access time.

Ownership

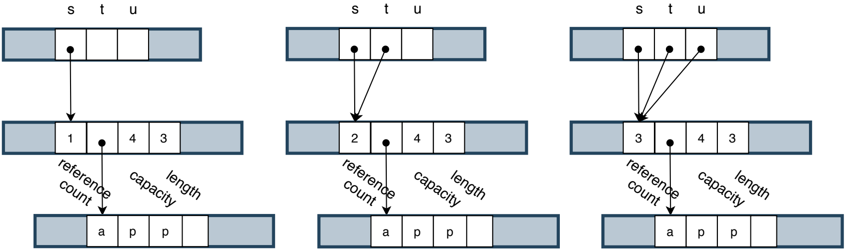
```
let s = "app".to_string();
let t = s;
let u = s;

```



- Each value is owned by single owner variable in Rust.
- Compile error: value "s" is dropped already.

Reference Counting

```rust
let s = Rc::new("app".to_string());
let t = Rc::clone(&s);
let u = Rc::clone(&s);
```

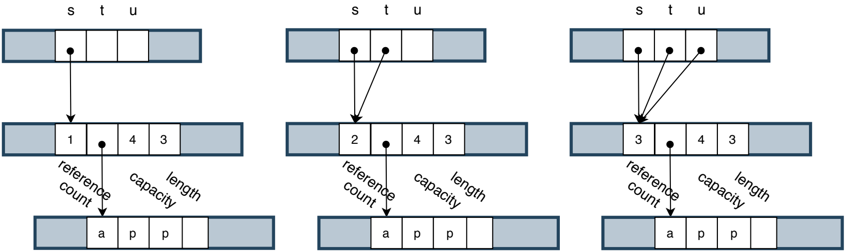


Advantage

- Sharing ownership
  - Each value is owned by single owner variable in Rust.
- Dynamic memory de/allocation
  - Rust usually determine lifetime of variable at compile time.

Reference Counting

```rust
let s = Rc::new(vec!["lemon".to_string(),
                     "orange".to_string(),
                     "apple".to_string()]);
let t = Rc::clone(&s);
let u = Rc::clone(&s);
```



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.

```
struct CustomerOwned {
    key: i32,
    total_purchase: f64,
    zip_code: String,
    order: OrderOwned,
    ... 10 more
    }
```

- All fields are owners.

```
struct CustomerBorrowed<'a> {
    key: &'a i32,
    total_purchase: &'a f64,
    zip_code: &'a String,
    order: &'a OrderBorrowed<'a>,
    ... 10 more
    }
```

- All fields are references.
- 'a is lifetime specifier.
- Rust compiler cannot determine lifetime automatically.

Complex Object - Slicing and Reference Counting

```
struct CustomerSlice<'a> {
    key: &'a i32,
    total_purchase: &'a f64,
    zip_code: &'a str,
    order: &'a OrderSlice<'a>
    ... 10 more
    }
```

- Fields are mix of references and slices.
- Slice is obtained only for contiguously allocated data structure, such as String and Vec.

```
struct CustomerRc {
    key: Rc<i32>,
    total_purchase: Rc<f64>,
    zip_code: Rc<String>,
    order: Rc<OrderRc>
    ... 10 more
    }
```

- All fields are Reference Counting

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

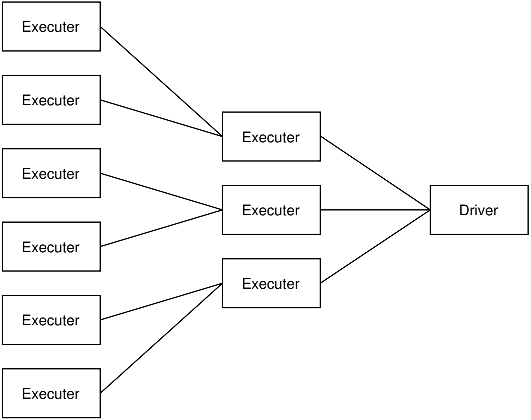
Merge-sort

- Many contiguous memory de/allocations
- Common memory usage pattern in sorting algorithms for Big Data Processing.
    - ex. External-sort

Impact of Memory Management on performance of Big Data Processing

Tree-aggtegate

- Many intermediate HashMap like data structures
- Many de/allocations of intermediate data
- Common memory and network usage pattern in Big Data Processing



K-Nearest-Neighbors (KNN)

- Text document preprocessing
- String manipulations
- An example of Machine Learning algorithm that requires Big Data processing.
- We are interested only in preprocessing phase.
- It is just matrix read operation after preprocessing phase.

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

Experimental Setting

Machine Specification

- Google Cloud Platform: n1-standard-8
- CPU : 8 cores
- RAM: 30 GB
- Standard persistent disk: 10 GB

Data Set

- Customer object
  - CustomerOwned
  - CustomerBorrowed
  - CustomerSlice
  - CustomerRc
- Wikipedia page data set
  - training set: 100×10^3 pages
  - testing set: 18×10^3 pages

Other Setting

- Run 5 times for each experiment
- Take average of runtime

Experiment 1: Accessing Object with Different Variable Type

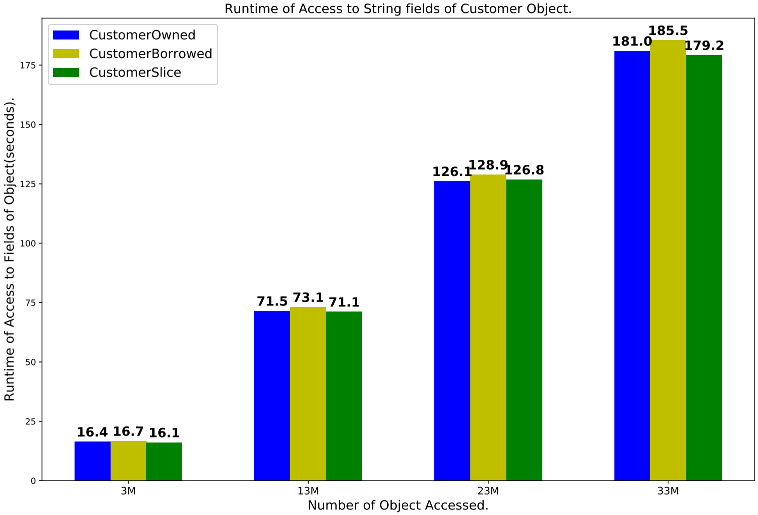
Question

- How much does selection of different memory management strategies differ memory access time?

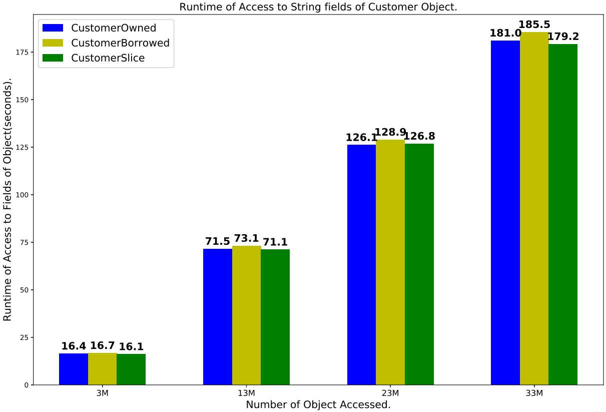
Evaluation

- **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 1: Accessing Object with Different Variable Type



Result

▷ No differences of memory access time among Customer objects using different memory management strategies.

Discussion

▷ Selection of different memory management strategies does not have impact on memory access time.

Experiment 2: Assessment of different reference methods in Rust

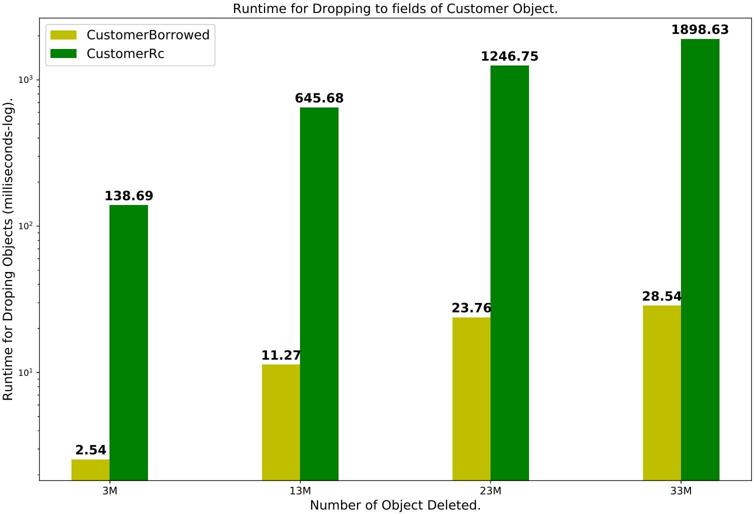
Question

- How much does Reference Counting hits performance?

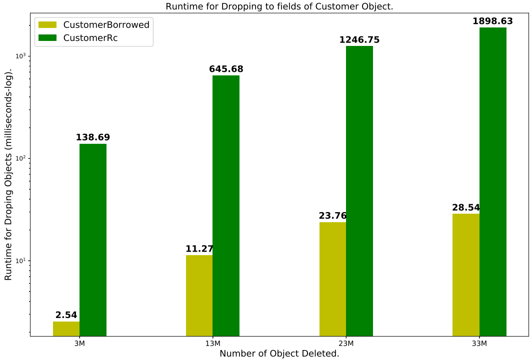
Evaluation

- **Construct 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

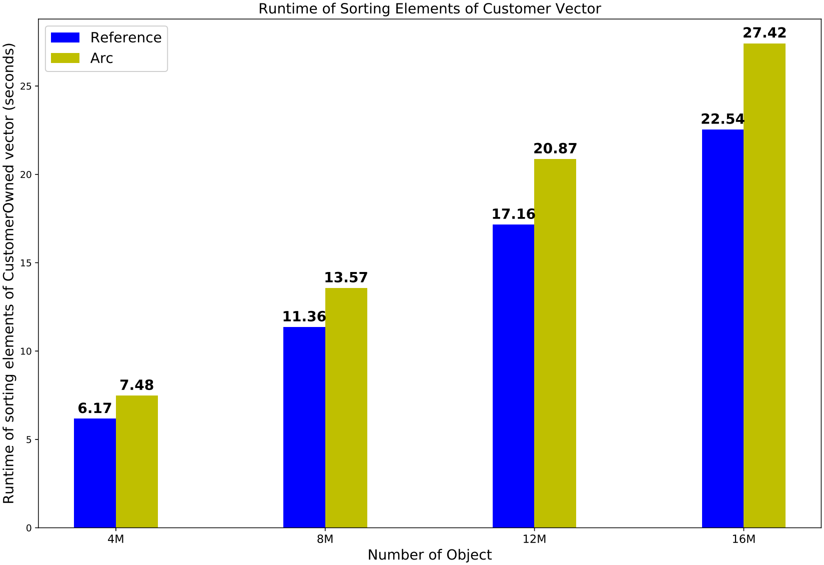
Question

- What is performance hit of Merge-sort algorithm using Arc vs, normal reference?

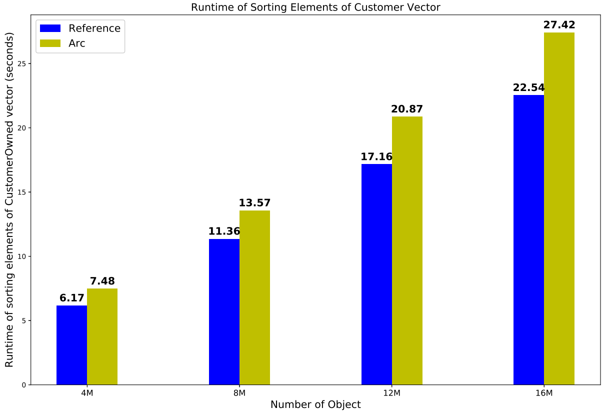
Evaluation

- Share **vector** of complex objects in multithreads
- **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

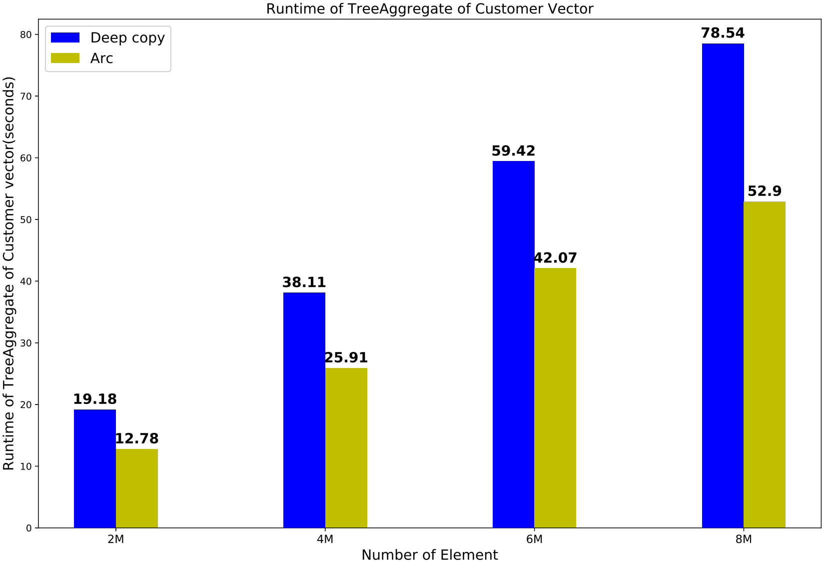
Question

- - What is performance hit of Tree-aggregate using Arc vs. deep-copy?

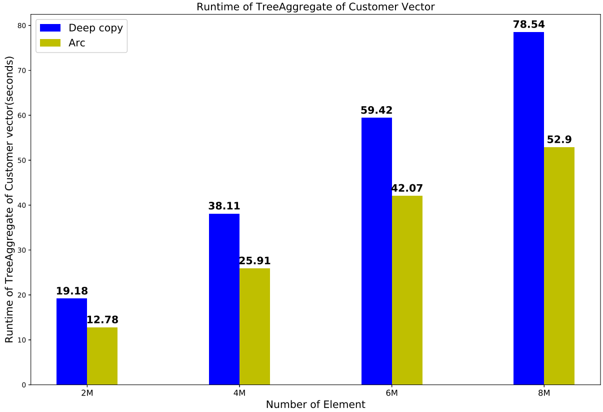
Evaluation

- Share **elements** of complex object in multithreads
- **Atomic Reference Counting (Arc)** vs. **Deep copy**
- Measure **runtime of Tree-aggregate algorithm**

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 to significant overhead.

Experiment 5: K-Nearest-Neighbors (KNN)

Question

- - What is performance hit of Machine learning algorithms using Arc vs. deep-copy?

Evaluation

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

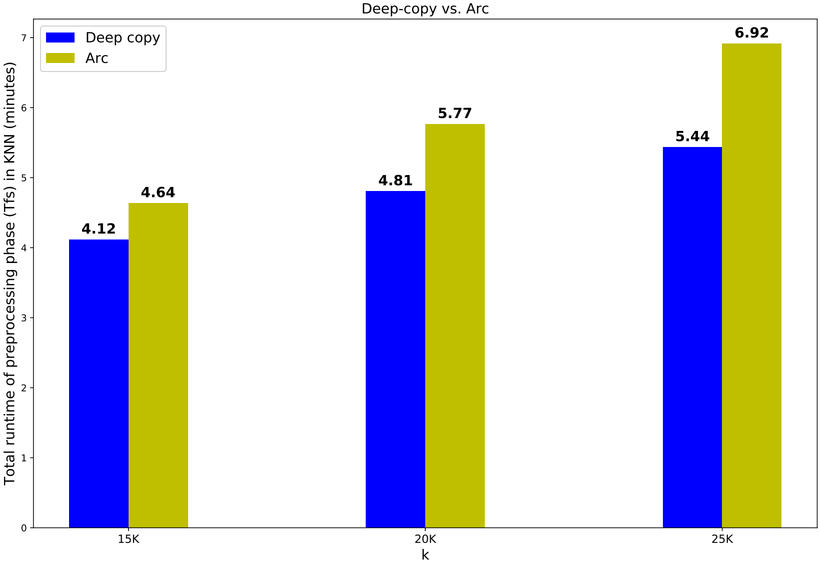
Experiment 5: K-Nearest-Neighbors (KNN)

Parameters

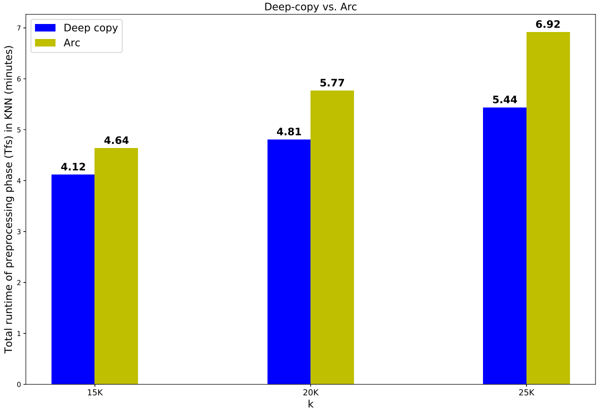
- Data Acquisition
  - **Atomic Reference Counting** (Arc)
  - **Deep-copy**

- Dimensions of feature matrices
  - 15K
  - 20K
  - 25K

Experiment 5: K-Nearest-Neighbors



Experiment 5: K-Nearest-Neighbors



Result

- **Arc** is **at most 27% slower** than deep-copy.

Discussion

- Deep-copying String is cheaper operation than Arc:
  - Relatively simple object
  - Contiguously allocated
- Overhead using Arc may become more significant as size of dat increases

Findings

- Use normal reference rather than Reference Counting whenever it is possible.
- **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, especially 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.
- When using Rust, writing memory-safe multithreaded code is fairly easy.