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

Shinsaku Okazaki

May, 2020

Motivation

Development of modern Big Data processing tools has some problems!

- Generative Garbage Collection
- Stop-The-World
- Memory Safe impelementation

Rust can be a ideal 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 multithreading code in Rust.**

Problem Description

Do following concepts have impact on runtime performance of algorithms?

- **Complex object**
- **Different variable types**
- **Reference Count**
- **Multithreading**
- **Common Big Data algorithms**

Complex Object

In Big Data processing, data is represented by complex objects.

```
struct CustomerOwned {
    key: i32,
    age: i32,
    num_purchase: i32,
    total_purchase: f64,
    duration_spent: f64,
    duration_since: f64,
    zip_code: String,
    address: String,
    country: String,
    state: String,
    first_name: String,
    last_name: String,
    province: String,
    comment: String,
    order: OrderOwned
}
```

Different Variable Types

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**

Multithreading

Atomic Reference Counting

Advantage

- ▷ **Sharing ownership**
- ▷ **Dinamic memory de/allocation**
- ▷ **Sharing among multithreads**

Disadvantage

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

Common Big Data algorithms

- **Merge-sort**
- **Tree-aggtegate**
- **K-Nearest-Neighbors (KNN)**

Experiment 1: Accessing Object with Different Variable Type

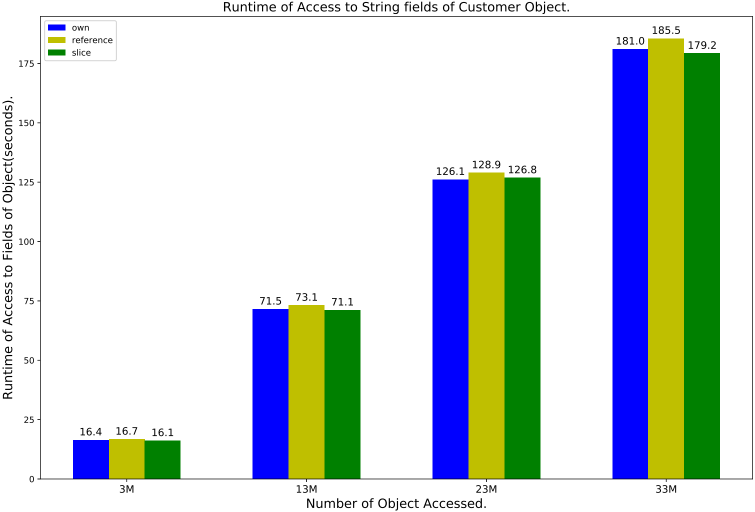
Question

- How much are memory access times different among different variable types used in complex objects?

Evaluation

- **Custruct complext objects**
- With different variable types: **Owner, Reference, Slice**
- Measure time to **access to fields of complex objects**

Experiment 1: Accessing Object with Different Variable Type



Experiment 2: Assessment of different reference methods in Rust

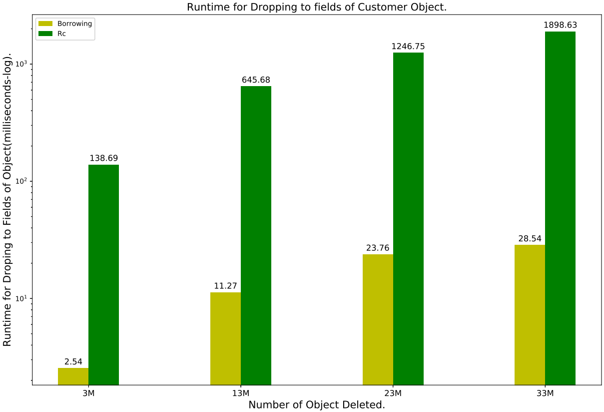
Question

- How much does Reference Counting slowdown time for dropping its variable?

Evaluation

- **Custruct complext objects**
- **Reference Counting** vs **reference**
- Measure time to **drop variables of complex objects**

Experiment 2: Assessment of different reference methods in Rust



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

Experiment 3: Merge-sort

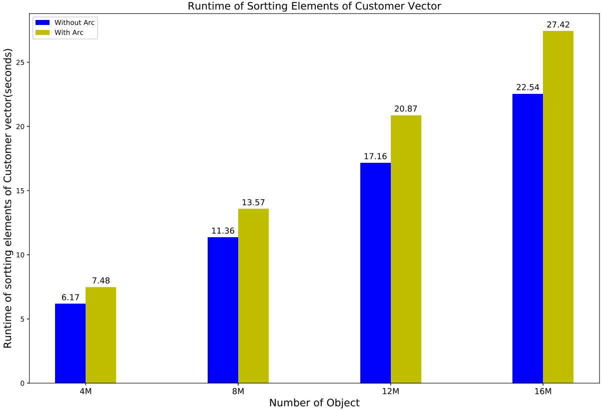
Question

- How much does sharing set of data with Atomic Reference Counting slowdown 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



Algorithms with Arc are about **21% slower.**

Experiment 4: Tree-aggregate

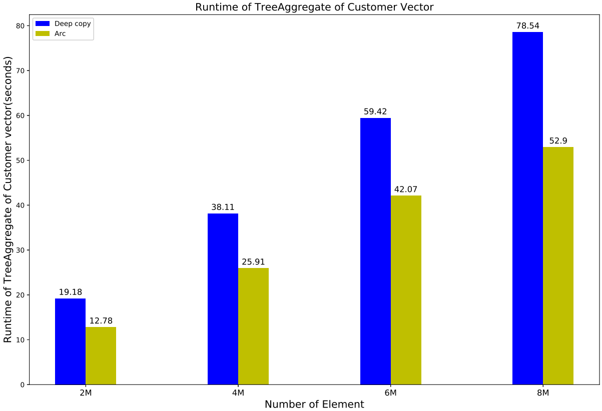
Question

- 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



Algorithms with deep-copy are from **40 to 50% slower.**

Experiment 5: K-Nearest-Neighbors (KNN)

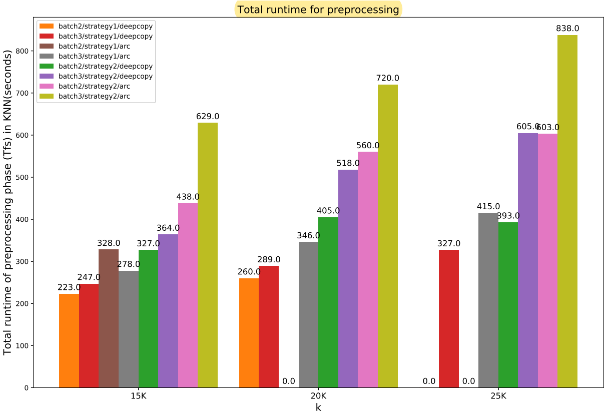
Question

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

Evaluation

- Document classification on Wikipedia page dataset
- Preprocessing phase: calculating **Term-frequencies (Tfs)**
- String manipulation
- **Atomic Reference Counting (Arc)** vs **Deep copy**
- **Frequency of memory de/allocation**
  - batch number
  - strategy
    - 1: keep intermediate objects in memory until owner is changed
    - 2: remove intermediate objects as soon as it is not needed
- Measure runtime of **preprocessing time of KNN algorithms**

Experiment 5: K-Nearest-Neighbors



- Algorithms with **Arc** are **at most 38 % slower** than deep-copy.
- Algorithms with **strategy 2** are **at most 85 % slower** than strategy 1.
- Algorithms with **3 batches** are **at most 40 % slower** than 2 batches.

Conclusion

- 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.