BOSTON UNIVERSITY

METROPOLITAN COLLEGE

Thesis

BIG DATA PROCESSING FOR MACHINE LEARNING TASKS WITH RUST

by

SHINSAKU OKAZAKI

B.S., Seikei University, 2018

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2020

Approved by

First Reader

Kia Teymourian, PhD
Professor of Computer Science

Second Reader

First M. Last
Associate Professor of ...

Third Reader

First M. Last
Assistant Professor of ...

Facilis descensus Averni;
Noctes atque dies patet atri janua Ditis;
Sed revocare gradum, superasque evadere ad auras,
Hoc opus, hic labor est. Virgil (from Don's thesis!)

Acknowledgments

Here go all your acknowledgments. You know, your advisor, funding agency, lab mates, etc., and of course your family.

As for me, I would like to thank Jonathan Polimeni for cleaning up old LaTeX style files and templates so that Engineering students would not have to suffer typesetting dissertations in MS Word. Also, I would like to thank IDS/ISS group (ECE) and CV/CNS lab graduates for their contributions and tweaks to this scheme over the years (after many frustrations when preparing their final document for BU library). In particular, I would like to thank Limor Martin who has helped with the transition to PDF-only dissertation format (no more printing hardcopies – hooray !!!)

The stylistic and aesthetic conventions implemented in this LaTeX thesis/dissertation format would not have been possible without the help from Brendan McDermot of Mugar library and Martha Wellman of CAS.

Finally, credit is due to Stephen Gildea for the MIT style file off which this current version is based, and Paolo Gaudiano for porting the MIT style to one compatible with BU requirements.

Janusz Konrad

Professor

ECE Department

BIG DATA PROCESSING FOR MACHINE LEARNING TASKS WITH RUST

SHINSAKU OKAZAKI

ABSTRACT

Existing big data processing tools are developed on A

Contents

1 Introduction **1**

1.1 Motivataion 1

1.2 Problem Description 3

2 Related Work **4**

2.1 Hadoop MapReduce to Spark 4

2.2 Resilient Distributed Datasets 5

2.3 Memory Management in Spark 5

2.4 Garbage Collection Tuning 7

2.5 Application to System Language 7

2.6 Region-Based Memory Management 9

2.7 Rust Memory Management 9

2.8 Serialization 11

2.9 Elements Copy and Insertion into Size-initialized Vector in Rust. 12

2.10 Access time to elements in vector 14

2.11 Access time to owned, borrowed, and sliced field of object 14

2.12 Possible Graph Structure in Rust 16

2.13 Experiment for Multithread 16

2.14 Note for next 17

2.15 LLVM 19

2.16 The existing Big Data tools 19

2.17 Experiment: Tree aggregate 19

2.18 Todos 20

2.19 Done 20

2.20 Time Line 20

3 Body of my thesis **21**

3.1 Some results 21

3.2 Experiment of Memory Allocation 22

3.3 Experiment of Memory Allocation for Different Element Type. 26

3.4 Assessment of different reference methods in Rust 32

3.4.1 Concept 32

3.4.2 Result 33

3.4.3 Discussion 33

3.5 Experiment for Merge-sort 34

3.5.1 Concept 34

3.5.2 Result 35

3.5.3 Discussion 35

3.6 Experiment for Tree-aggregation 37

3.6.1 Concept 37

3.6.2 Result 38

3.6.3 Discussion 38

4 Conclusions **41**

4.1 Summary of the thesis 41

A Proof of xyz **42**

A.1 Memory and Process in Operation Systems 42

A.2 Multi-threading and Parallelism 43

A.3 Memory management in Operation System 43

A.4 Demand Paging 44

A.5 Copy on Page 45

A.6 Threads and Concurrency 45

A.7 BLAS LAPACK 45

A.8 Netlib-Java 47
A.9 Create Java interface of CBLAS with JNI 47
A.10 Matrix Computation and Optimization in Apache Spark 48
A.11 Memory Management of each Linear Algebra Library 49

References **51**

Curriculum Vitae **52**

List of Tables

3.1 Absolute disparity error per pixel for the test data from Fig. A·1 and different parameter values. In each experiment one parameter is adjusted while other parameters are unchanged. 22

List of Figures

2·1 Java Heap Structure 7

2·2 Java Garbage Collection 8

2·3 Runtime of elements copy from one vector and insertion to the other vector. 13

2·4 Runtime of elements copy from one vector and insertion to the other vector. 14

2·5 Representation of Customer object in Rust. 15

3·1 Assignment of single-view intensities to RGB components: (a) view #1; and (b) view #2. 21

3·2 Memory allocation of ArrayList in Java 24

3·3 Memory allocation of Vector in Rust 25

3·4 Representation of Customer object in Java. 27

3·5 Representation of Customer object in Rust. 27

3·6 Memory allocation of Java ArrayList 28

3·7 Memory allocation of Rust Vector 29

3·8 Difference of Memory allocation of Java ArrayList between Non-initialization and Initialization 30

3·9 Difference of Memory allocation of Rust Vector between Non-initialization and Initialization 31

3·10 Runtime for Droping to fields of Customer Object 33

3·11 Representation of Source Vector 35

3·12 Runtime of Sortting Elements of Customer Vector 36

3·13 Runtime of Tree-aggregate algorithm 39

3·14 Aggregation function with Arc 40

3·15 Aggregation function with deep-copy 40

A·1 Integration of Native Methods 48

List of Abbreviations

The list below must be in alphabetical order as per BU library instructions or it will be returned to you for re-ordering.

CAD	.....	Computer-Aided Design
CO	.....	Cytochrome Oxidase
DOG	.....	Difference Of Gaussian (distributions)
FWHM	.....	Full-Width at Half Maximum
LGN	.....	Lateral Geniculate Nucleus
ODC	.....	Ocular Dominance Column
PDF	.....	Probability Distribution Function
$\mathbb{R}^2$	.....	the Real plane

Chapter 1

Introduction

1.1 Motivataion

Importance of cluster computing tool for Big Data Analysis has been increasing as amount, value, use of data has increased. Recently, almost all businesses stand on data, from web marketing analysis to factory automations and leverage of data is ubiquitous, because there are many open source tools to analyze data and cloud computer infractrustures which can support computation for massive amount of data. The improvement of accessability to these technologies to deal with big data democratized data driven businesses by eliminating significant amount of initial investment.

However these technologies do not come with free; we need to pay for it. Ususally, user needs to pay depending on use of computational resources. If your process of data analysis is too long or need to use a number of cluster with high speck specification, the cost will end up significantly hight. To address these problems, the quality of analysis tool is critical. If the tool can optimize the runtime performance and usage of computational resources, the cost for runing the businesses can be efficient.

Multiple cluster computing analysis tools have been developed, such as Hadoop MapReduce, Spark, and Flink. These tools have brought reliable and scalable ways to deal massive data. These has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing.

These tools are constructed on top of Java Virtual Machine (JVM). JVM abstracts hardware and memory management from the developper so that the development is fairly easy. In addition, Java or Scala compiled code is platform-independent, which can run on any ma-

chine with JVM. However, these advantages may be really critical weakness when it comes to processing big data. JVM abstract away most detail regarding memeory management from the system designer, including memory deallocation, reuse, and movement, as well as pointers, object serialization and deserialization. Since managing and utilizing memory is one of the most important factors determining Big Data systems performance, reliance on a managed environment can mean an ordeer-of-magnitude increase in CPU cost for some computations. This cost may be unacceptable for high-perfomance tool development by an expert.

To overcome these problems, one can use programming languages with more contoral on hardware, system languages, for development of Big Data tools. For example, C++ is a general-purpose, statically typed, compiled programming language which supports multiple programming paradcigm. It is also a system language which gives full control over hardware. There are several researches or projects where developers and researchers implement Big Data tools with this language. These tools shows significantly better performances than those developed with application languages. Although the evidence of the advantage of building high speed computational tools with C++ has been discovered, the steep learning curve and difficulty of writing memory safe codes are bariiers to technology diffusion.

Rust is a system language which gives the similar performance and control of hardware to C++ or C and safety of runtime. Unique memory management strategy of Rust, ownership and borrowing, and fearless concurrenry make the language one of the ideal candidate for development of Big Data tools. There are several existing Big Data tools developed with Rust. Inspired by these activity

Write about some development and research for Big Data tools with system language.

Platform depenedance is no longer disadvantage, because of docker and kubernets.

Rust safe language advantage

1.2 Problem Description

Many of popular open source cluster computing frameworks for large scale data analysis, such as Hadoop and Spark, allow programmers to define objects in a host languages, such as Java. The objects are then managed in RAM by the language and its runtime, Java Virtual Machine in the case of Java and Scala. Storing objects in memory enables machine to process iterative computation. One of the fundamental tasks for recent big data analysis is analysis using Machine Learning Algorithms, which require iterative process. As the amount of data increases, memory is required to keep many objects. Therefore, memory management plays a critical role in this task.

Memory management in Java and Scala is performed by garbage collection. The garbage collection brings a significant advantage for programmers by removing responsibility for planning memory management by themselves. Instead, JVM monitors the state of memory and performs garbage collection at certain points. However, these monitoring and auto-execution of garbage collection cost additional computation and might consume computation resources which should be used for data processing. This can significantly decrease performance of the computation.

In contrast, memory management in system language, such as C++, relies on programmers' decision for when to allocate and deallocate memory. The functions, malloc/free consume most of the memory management. Proper implementation of system language for big data processing can be overperform the implementation in host language. Nevertheless, implementing C++ performing proper memory management and guaranteeing security can be unproductive and complicated.

Considering the issue of memory management, we introduce solution based on unique memory management methods implemented in Rust, ownership and borrowing. This unique concepts in Rust secure codes and perform memory management without monitoring memory or calling functions. We introduce implementations of machine learning algorithms in both Java and Rust to assess performances of each memory management system for iterative big data processing tasks.

Chapter 2

Related Work

2.1 Hadoop MapReduce to Spark

In several Big Data mining tools that have been developed, the most notable ones are MapReduce and Spark. MapReduce is a cluster computing framework which supports locality-aware scheduling, fault tolerance, and load balancing. Spark improves operations that MapReduce does not cover well.

MapReduce provides a programming model where the user creates acyclic data flow graphs to pass input data through a set of operations. This data flow programming model is useful for many of query applications. However, MapReduce framework struggles from two of main resent data mining jobs: iterative jobs and interactive analysis.

Iterative job is especially common in Machine Learning Algorithm, such as Gradient Descent. In traditional MapReduce framework, each iteration can be expressed as single MapReduce job so that each job must reload the data from disk. This leads I/O overhead and deteriorates performance of iterative algorithms.

Interactive analysis is also an inevitable task in modern data science. A data scientist want perform exploratory analysis in interactive way. Nevertheless, MapReduce is designed in the way more stable for ad-hoc query, so each analysis can be single MapReduce job. To perform multiple analysis to explore dataset, the data needs to be written to and reload from disk many times.

To overcome these limitations, Spark has been developed as a new cluster computing framework maintaining the innovative characteristics of MapReduce and improving its iterative and interactive jobs with in-memory data structure.

2.2 Resilient Distributed Datasets

The major methods in Spark are Resilient Distributed Datasets (RDDs), a data structure that abstracts distributed memory across different clusters. The immutable coarse-grained transformation, spark-scheduler with lazy-evaluation, and memory management with cacheing achieve computation with fault-tolerance, fast execution, and moderate control on memory efficiency (Zaharia et al., 2012).

A RDD is essentially a multi-layer Java data structure. A top RDD object references Java array, which intern, references a set of tuple objects. The coarse-grained transformations and immutability requires a RDD to be deep-copied to produce a new RDD, but efficiently offers fault tolerance. The lost partitions of a RDD can be recomputed in parallel on different nodes rather than rolling back the whole program.

Spark-pipeline consists of sequence of transformations and actions over RDDs. A transformation produces a new RDD from a set of existing RDD. An action is method that computes statistics from an RDD. Due to lazy-evaluation nature, transformations do not materialize the newly created RDD. Instead, RDD Lineages are created. Lineage is a graph among parent and child RDDs which represents logical execution plan. This enhance fault-tolerance and improve ability to optimize execution plan.

RDDs can be cached in memory for faster access by persist method. Developers can specify a storage level for a persisted RDD, in memory with serialized or deserialized, or on disk. Other than persisted RDD, Spark generates a lot of intermediate RDDs during execution. Since RDD is a Java object, they are managed by Garbage Collection (GC) in the JVM. However, persisted RDDs are never collected by GC. This GC might cause significant deterioration of performance of Spark, because GC shows heavy overhead when there are a number of objects.

2.3 Memory Management in Spark

Spark framework allocates multiple executors, JVMs, that run sequence of transformations and actions. As we describe the previous section, data in Spark is mainly stored as Java

objects in memory, so that they are allocated on JVM heap and managed by JVM Garbage Collection. The data may form three types (Xu et al., 2019): Cached data, Shuffled data, and Operator-generated data.

Spark can cache data in memory to reduce disk I/O. This Cached data usually long-lived Java objects and span multiple stages in Spark-pipeline. Spark allocates a logical storage space to store the cached data as shown in Figure. After aggregation, Spark generates Shuffled data. Shuffled data is usually long-lived, because it need to be kept in memory until the task ends. Spark allocates execution space to store Shuffled data. The storage space and execution space spans 60 % of JVM heap space in default. Operator-generated data is data generated by user-defined operations. Since Operator-generated data may or may not be used, after the operation finish, the data object can be both short-lived or long-lived objects. These are stored in user space allocated on default 40 % of JVM heap.

All data of these types on JVM heap is managed by JVM GC. GC check references graph of objects, mark whether the objects are used and deallocate memory space occupied by unused objects. There are three popular GCs: Parallel, CMS and G1. All of these method track generation of object based on the region of memory. The Java logical heap structure is shown in Figure. It can be separated into three main parts where store objects for each corresponding generation: permanent generation, young generation, and old generation. The region for permanent generation stores metadata required by JVM to describe class and method used in application which will be permanently lived on the region of memory.

The region for young generation mainly consists of two parts: Eden and Survivor space. First, Java objects are created in Eden space and promoted to Survivor space when survive from GC. After objects survive several GCs in Survivor space, they are finally promoted to old generation.

In old generation, JVM lunches multi-thread to perform GC. GC with multi-threading suffers from Stop-The-World (STW) pauses; GC may suspend application threads while performing object marking and deallocation. Different GC algorithms try to solve this problem with trade-off between GC frequency and memory utilization.

Because of the problem of STW and copying objects to different physical memory pages , JVM GC cause huge overhead when number of objects is large. Therefore, GC become severe issue in Big Data processing where might produce significant number of object.

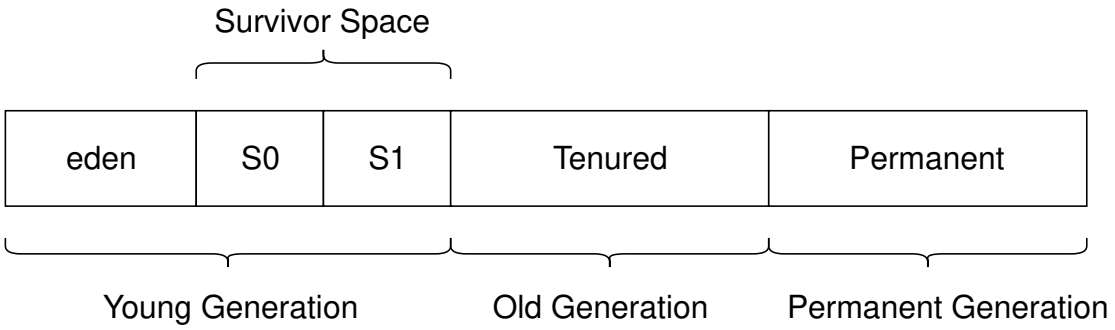


Figure 2·1: Java Heap Structure

2.4 Garbage Collection Tuning

Several solutions are suggested. Main idea is to reduce number of object and execution of GC. One is avoiding pointer-based data structures, such as HashMap and LinkedList. These objects have a "wrapper" object for each entry so that number of object tends to be larger than when when an array is used. Caching serialized object in memory also reduce number of object and memory usage since the set of objects become a byte array. In addition, developers can allocate byte array off-heap of JVM to avoid tracking by GC. Although these memory management solution for GC help developer improve performance of Spark applications, the effort to discover the best GC tuning afflicts developers.

2.5 Application to System Language

Considered overhead produced by Memory Management in JVM, we suggest application of system language to Big Data tool rather than application language, such as Java and Scala. System languages, such as C and C++, are languages that give developer total control over the hardware and de/allocation of memory without GC. These features enable program written in system languages to optimized performance taking full advantage of hardware.

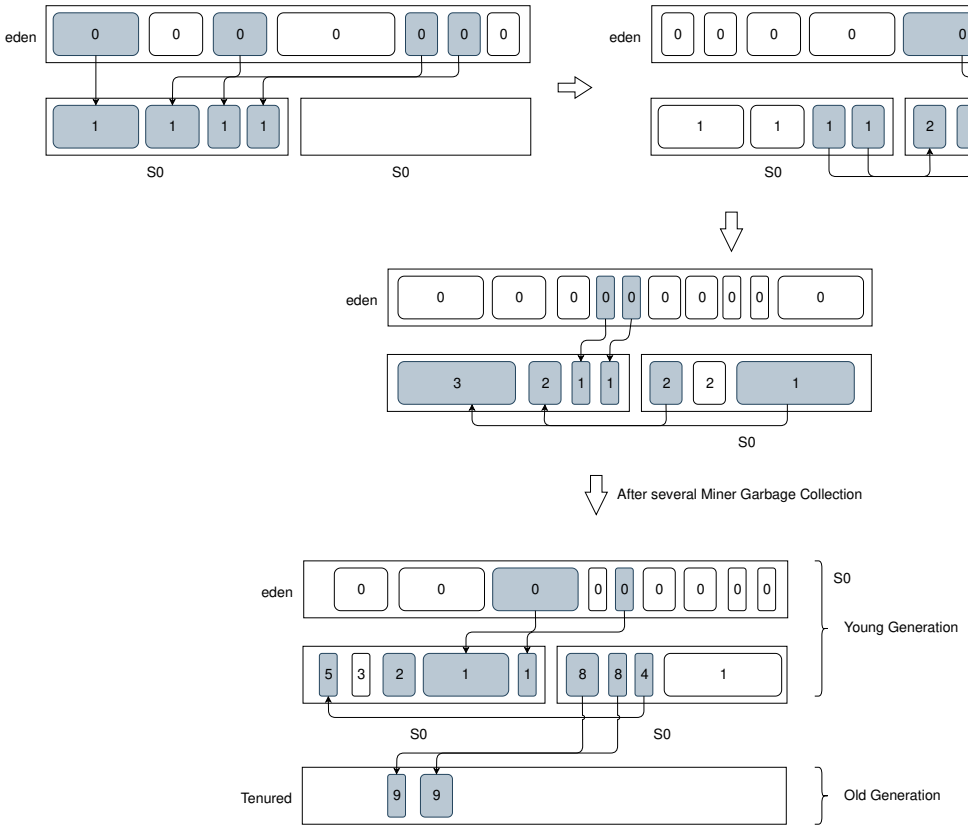


Figure 2·2: Java Garbage Collection

To take example, one can build Big Data tool with C++. C++ is one of the most popular system languages which has Object-oriented features. C++ has functions which provide control over memory to developers. In another word, it is responsibility for one to manage memory properly and safely. The functions, malloc() and free(), take roll for memory allocation and deallocation in respectively. The manual memory de/allocation may cause several problems and require developers attention to the problems with significant effort for debugging and testing. Here, we explain two of the most common problems regarding to memory management in existing system languages.

Dangling pointer or reference is a pointer or reference pointing to object that no longer exists. The situation of dangling pointer happens because of deallocation of memory without modification of value of the pointer. If the memory region is reallocated for other object and the dangling pointer tries to access the original object, the unpredictable behavior may

result.

Memory leak occurs when memory is allocated and no longer referenced so that the object in the memory location cannot be reached and released. This is result of dereferencing object with deallocation. Memory leak consumes more memory than necessary by making unreachable location.

Some solutions are established to address these problems. Actually GC is a high-level solution that guarantees memory safety. C++ has a different solution called Resource Acquisition is Initialization (RAII). In RAII, objects can live within the scope where they are created. The memory is released when the object goes out of scope. This solution is more predictable and deterministic than GC. However, it is problematic When we need the object out of the scope, returning a value from a function. There are several ways to go around this problems, such as smart pointers, copy constructors, and move semantics. Nevertheless, these non-orthogonal concepts makes code in disorganize and leads to error prone implementation.

2.6 Region-Based Memory Management

2.7 Rust Memory Management

Rust is a system programming language which provides memory safety without runtime checking like GC and necessity of explicit memory de/allocation. In addition to characteristic of memory safety, this language also enables a developer to write concurrent code , fearless concurrency. Developers are able to write code with high performance given fair control of hardware and safety forced to follow restrictive pattern.

Main concepts of Memory Management in Rust are ownership, move, and borrowing. In ownership feature or Rust, each value has a variable called owner. This owner has information about the value, such as location in memory, length and capacity of the value. This owner can live on the scope associated with its lifetime. When the owner goes out of its scope, the value will be dropped. This feature is similar to how RAII in C++ works. However, acquisition of owner out of the scope where it was constructed is available in Rust

with the concept of move.

With move, a value can be transferred from one owner to another, For example, a value is instantiated in a function. When it is returned to a new variable , the ownership of value is moved to the new variable. In Rust, one do not have to explicitly deallocate memory, because ownership of value is passed among variables across scope using move and eventually goes out of scope deallocating memory of the value.

Borrowing lets code use a value temporarily without affecting its ownership so that it reduces unnecessary movement of ownership. One use case is when value is used in function and needed to be passed to the argument. If the argument takes ownership and the function does not return the value, the ownership of value goes out of scope and the memory is deallocated. One can pass reference of the value to the argument instead of owner. The reference goes out of scope, but ownership remains the same.

2.8 Serialization

Paging object solution for faster transportation among clusters Protocol Buffer We cannot serialize pointer (it is difficult to represent complex object in serialized form to interact with).

2.9 Elements Copy and Insertion into Size-initialized Vector in Rust.

In this experiment, four methods are used to insert elements into vector in Rust. One is clone method which performs bitwise deep copy. Another is clone_from which also performs bitwise deep copy, but copies elements of the vector to distination vector rather than creating new one. We initialize the distination vector with the same size to the number of elements we insert into it. Another is copy_nonoverlapping function which copies values from source to distination memory region. The other is pushing elements of source to distination vector one by one. Insertions of elements with 4 size are conducted 1000000, 1500000, 10000000, 15000000, and, their runtimes for each elements type, integer and String are measured.

The figure shows the result of the experiment. Among the runtime performances of integer insertion for every methods, clone, clone_from, and copy_nonoverlapping method shows the similar performance. However, the pushing the copy of elements one by one has much slower runtime performance compared to the rest. This is because integer elements are allocated contiguously in the memory, so that accessing address of memory by pointer reads some next address. This boosts the copy and insertion of elements.

On the other hand, all of methods show the similar runtime performance in experiment for String object insertion. This is because String object is not stored in contiguous memory region. The vector stores pointer to the object and process need to access around different memory region again and again to deeply copy the object.

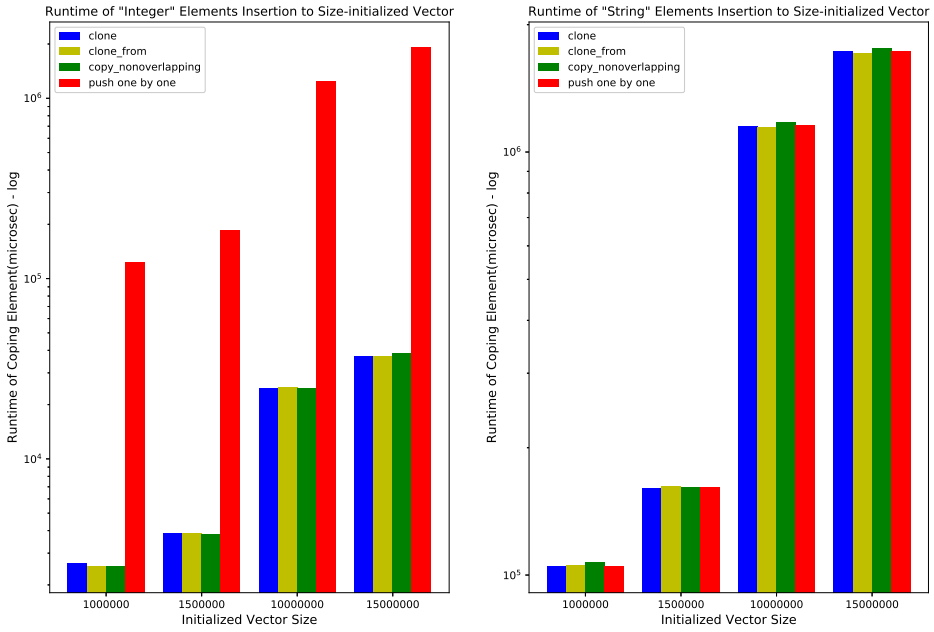


Figure 2·3: Runtime of elements copy from one vector and insertion to the other vector.

2.10 Access time to elements in vector

In this experiment, whether mutability has impacts to operation on the object in terms of runtime performance. According to the experiment, there is no difference on accessing to elements of mutable and immutable vector.

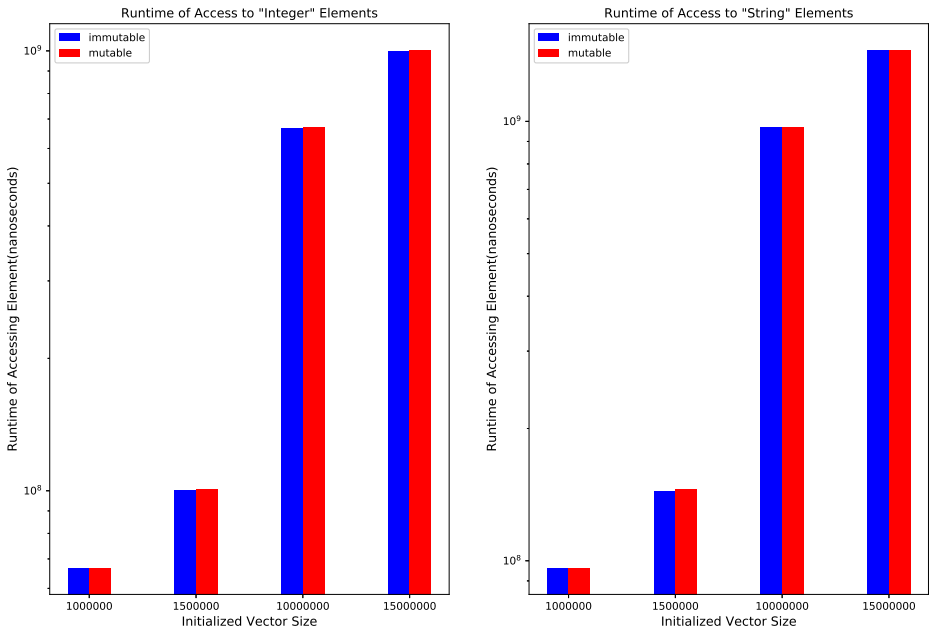


Figure 2·4: Runtime of elements copy from one vector and insertion to the other vector.

2.11 Access time to owned, borrowed, and sliced field of object

In this experiment, differences of access time to among owned, borrowed, and sliced are observed. Each variable has slightly different use of memory.

```
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
}

struct CustomerBorrowed<'a> {
    key: &'a i32,
    age: &'a i32,
    num_purchase: &'a i32,
    total_purchase: &'a f64,
    duration_spent: &'a f64,
    duration_since: &'a f64,
    zip_code: &'a String,
    address: &'a String,
    country: &'a String,
    state: &'a String,
    first_name: &'a String,
    last_name: &'a String,
    province: &'a String,
    comment: &'a String,
    order: &'a OrderBorrowed<'a>
}

struct CustomerSlice<'a> {
    key: &'a i32,
    age: &'a i32,
    num_purchase: &'a i32,
    total_purchase: &'a f64,
    duration_spent: &'a f64,
    duration_since: &'a f64,
    zip_code: &'a str,
    address: &'a str,
    country: &'a str,
    state: &'a str,
    first_name: &'a str,
    last_name: &'a str,
    province: &'a str,
```

2.12 Possible Graph Structure in Rust

In Rust, there are two essential problems to construct graph structure, lifetime and mutability.

The first problem is about what kind of pointer to use to point to other nodes. Since graph can be cyclic, so the ownership consept in Rust is violated if we use Box<Node>.

All of graph structure are immutable at least at creation time. Because graph may have cyclic, we can not create graph at one statement. Although edge of graph has to be mutable to create entire graph structure, we might need multiple references to edges, which violates basic rule of Rust programming.

One solution is to use raw pointers. This is the most flexible approach, but also the most dangerous. By taking this way, we are ignoring all the benefits of Rust.

2.13 Experiment for Multithread

Experiment for multithreading in Rust is interesting, because there are many memory allocation and deallocation in each threads. This is because each thread offten need to form independent memory state to each other to perform computation concurrently safely. In addition, the multithreading strategy can be planned in different ways in Rust using different concurrent computation tools, Mutex, Arc, spawn, channel, scope, and so on.

Currently available plan

- spawn, channel and sending data (currently deviding data, but we probably do not need to).
- spawn, forkjoin and shared data.(shared data among child and parent).
- scope, forkjoin and slice shared data.
- Use linkedList instead of Vector (Each node can be sharable data structure). When we have complex object of elements in vector, we want to copy the pointer to the object.

- Compare performace on LinkedList which is not contiguous allocation, but do not need additional memory allocation and Vec.

2.14 Note for next

Complex object of elements copy and insertion among vector is worth to experiment. This can be compared with Java, because memory layouts of struct in Rust and class in Java are different. Rust stores fieds in the contiguous memory region. However, Java stores field elements to different region.

Complex object whose fields has reference elemenets insertion to vector can be evaluated. Operation to the fields can be little more expensive because the pointer to the value of the field is not stored in contiguous memory region.

Generic type and static type function can be compared.

To optimise access to String elements of vector, smallstring can be improve the runtime performance. This is because the smallstring optimization enables short length of string on stack as byte array. This string type sets condition where it makes decition where the string is stored on heap or stack as array at certain length.

Comparing operation on reference and owned variable is also interested to examine.

Comparing operation on various smart pointer type. Rc<T> enables value to have multiple owners, but the value should be immutable. Rc<T> is used in case of single thread. When the situation is multi-thread, Arc<T> is used. Arc<T> performs atomic operation, so it is more expensive than Rc<T>. When we need to mutate value of Rc<T>, RefCell<T> can be useful. RefCell<T> allow us to have mutiple mutable reference and immutable of reference mutable or immutable variable at the same time. When the mutability consistency is voilented, it terminates program during runtime. That is why RefCell<T> is expensive so that the state of mutable consistency should tracked. As Cell<T>, the value is copied in and out of the Cell<T> instead of getting refrence to it. In addition, Mutex<T> provides intetior mutability across multi-thread.

Comparison between mutable and immutable tree or graph structure can be checked.

This is because tree structure requires Rc<T> or Week<T> and in addition RefCell<T> to make it mutable.

Design experiment for Trait object and Generic function. We can have Trait object which is pointer to object which imprements its Trait and use the method of Trait object. This object has additional information other than just reference to tye original object. This is because Rust needs the type information to dynamically calll the right method of Trait object depending on the type of original object. On the other hand, we can have Generic function whose parameter types are Generic types corresponding to Trait. When Rust compiles the code, it emits independent function corresponding to every types that implements the Trains specified in parameter.

Vector of Trait object (need to put element into Box) vs Vector of concrete type.

If we keep first created owner until the last phase, we can use borrowing to operate. If we delete first created owner during the operation, we should use Rc<T>. The question would be, we can use Rc<T> every where?

Multithread in Rust vs Java

Smart pointer

- Box<T>: Box pointer lets value allocated on heap rather than on the stack.
- Rc<T>: Reference counted pointer lets variable take multiple immutable ownership.

Interier mutability

- Cell<T>: For only Copy type, it allows us to mutate variable, even if it is immutable. Howeever, it does not support sharing the variable so that it returns always copy of the value.
- RefCell<T>: This support sharing and interier immutability for all types. This is done by allowing to get mutable reference from immutable variable and push the error detacting time from compile time to runtime.

2.15 LLVM

LLVM (Low Level Virtual Machine) is an umbrella project which contains components of compilation of programming language. Exsiting compilers have tightly coupled functionalities so that it is not possible to embed them into other applications. However, the abstract framework of LLVM decouples the functionalities into peaces and the peaces of functionalities can be reused.

In structure of a compiler, there are three main components; flontend, optimizer, and backend. In flontend, a develper designs the interface of source programming language in way where it can be optimized by optimizer. Then, backend takes optimized code and produce the native machine code. LLVM has a component called LLVM Intermediate Representation (IR), which places itself across frontend to optimizer. IR is designed to host mid-level analyses and transformations that you find in optimizer section of a compiler. High-level language has many common structures and functionalities, so most of all high-level program languages can be represented with IR. Once source code is represented with IR, optimizer can easily find pattern and optimize it in faster time. IR is useful in terms of flontend. This is because developer of language flontend need to know only how the IR works and use the framework to develop a language.

2.16 The existing Big Data tools

The existing Big Data tools are JVM based. JVM abstract hardware so that JVM can rarely achieve near-native speed. A garbage collector is a

2.17 Experiment: Tree aggregate

Three aggregate can be more memory de/allocation intensive experiment, when we load data from disk line by line to aggregate. This is because, we will allocate many small objects and deallocate them many times. We can imitate ShuffledRDDs by writing result to disk and reading it to other thread. We can compare to Java implementation for ration of runtime increase.

2.18 Todos

-
- Redo the experiment with appropriate size
- Document for Memory Management of Java with Garbage Collection and the impact for Big Data tools
- The safety of Region Based Memory Management and advantage of using Rust.
- Serialize problem
- Document for reference counting(measure dropt time using criterion)
- Design experiment for Graph
- Design experiment for Multithread
- Study about LLVM
- Buffer pool

2.19 Done

- Study about OS memory management

2.20 Time Line

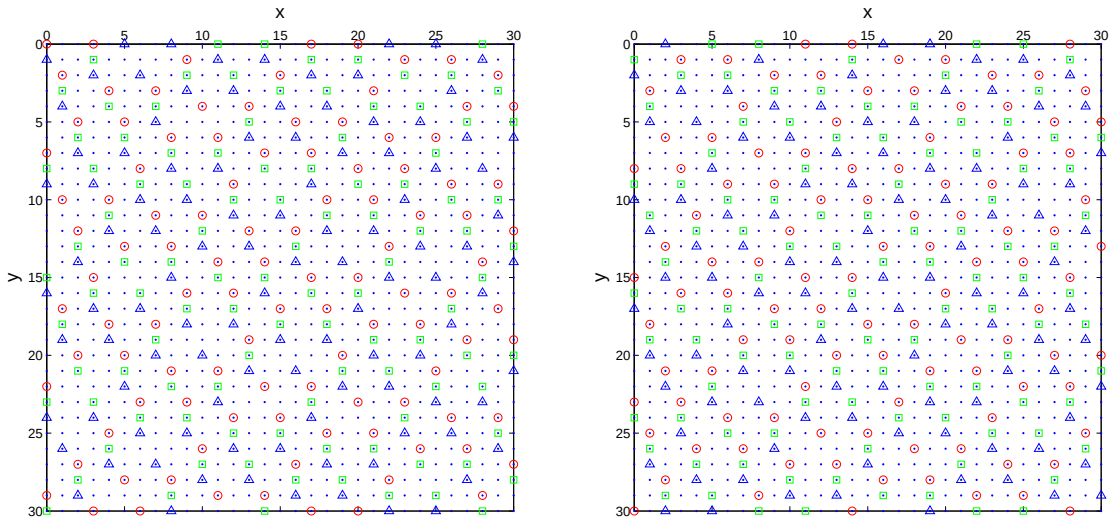
- End Feb: Finish Experiment for Multithread and Graph
- Mid May: ML experiment
- End May: Redo experiment with bench mark dataset.
- Start April: Finish Writing for first revise.

Chapter 3

Body of my thesis

3.1 Some results

Here goes all the important stuff, likely with a lot of graphics like this:



(a) (b)

Figure 3·1: Assignment of single-view intensities to RGB components: (a) view #1; and (b) view #2.

You will also be using a lot of citations. Here is the format required in the dissertation: (Lamport, 1985),(Debreuve et al., 2001).

In all likelihood, you will need to insert tables. See one example on the next page.

Table 3.1: Absolute disparity error per pixel for the test data from Fig. A·1 and different parameter values. In each experiment one parameter is adjusted while other parameters are unchanged.

$\eta = 6000$, $\mu = 2000$

K	u_1	u_2
3	0.52	0.46
7	0.47	0.43
10	0.35	0.36
12	0.37	0.36

$K = 10$, $\mu = 2000$

η	u_1	u_2
1000	0.54	0.45
3000	0.43	0.40
6000	0.35	0.36
9000	0.37	0.37

$K = 10$, $\eta = 6000$

μ	u_1	u_2
100	1.00	1.16
1000	0.53	0.47
2000	0.35	0.36
3000	0.44	0.43

3.2 Experiment of Memory Allocation

This experiment is to test how static and dynamic memory allocation of Java and Rust behave. For the assessment, Element addition to ArrayList in the case of Java and Vector in the case of Rust is emploied here. These data stractures are resizable and we have control to set initial size. There are two parameters; initial size of ArrayList or Vector and thier final size after additions of elements. We are interested in the impact to memory allocation by initial allocation and expantion to runtime.

First, an ArrayList and an Vector are created with specified initial size. Then, in a loop, a element is added for each iteration until the size of the ArrayList or Vector get the specified final size. Each data structure has a different resizing strategy. When an ArrayList hits current limit of its size and expands the limit, it doubles the current size. While Vector does not have specific strategy for its resizing, the expantions of size of both ArrayList and Vector might affect the digradation of runtime performance.

To perform benchmarks, we use Java Microbenchamrk Hardness (JMH) for Java and Criterion for Rust. The benchmark time is calculated mean from several iterarions. We warm up before the execution. The parameters are set in combination of initial size, 10, 100, 1000, and 10000 and final size, 9, 99, 999, 9999. The results are shown in Figure 2-4 and 2-5.

Discussions here are separated to two cases: when initial size is set bigger than final size and when final size is set bigger than initial size. For ArrayList in Java, it shows performance significant degradation when the initial size is bigger than final size. When

we create ArrayList with intial size of 1000 and add 99 elements to the ArrayList, the the average execution time is 1125 ns. However, the average excution time when we create ArrayList with intial size of 100 and add 99 elements is 623 ns. This degration is caused by the cost of initializing the large array. On the other hand, Rust Vector does not have significant cost for the initialization of size compared to the cost of addition of elements. In the case of Vector with initial size of 1000 and add 99 elements to the Vector, the average excution is 320 ns. In the case of the initial size of 100 and 99 elements additions, the average excution is 279 ns. This is such small degradation compared to element addition in Vector.

In ArrayList when the initial size is smaller than the final size, there can be degradation in performance. When its initial size and final size are set 100 and 999 respectively, the average execution time is 10884 ns. However when its initial size and final size are set 1000 and 999 respectively, the average execution time is 4250 ns. This result shows the degradation of in performance caused by the copy of existing elements into newly allocated array whose size is double of the last array.

This characteristic can be seen in the case of Vector in Rust. Vector with initial size 100 and final size 999 performs the average excution time 3514 ns, but one with initial size 1000 and final size 999 performs 2723 ns. When the Vector reachs its capacity, it allocates a larger buffer and copies the present elements to it. This cost results in the degradation of the average excution time.

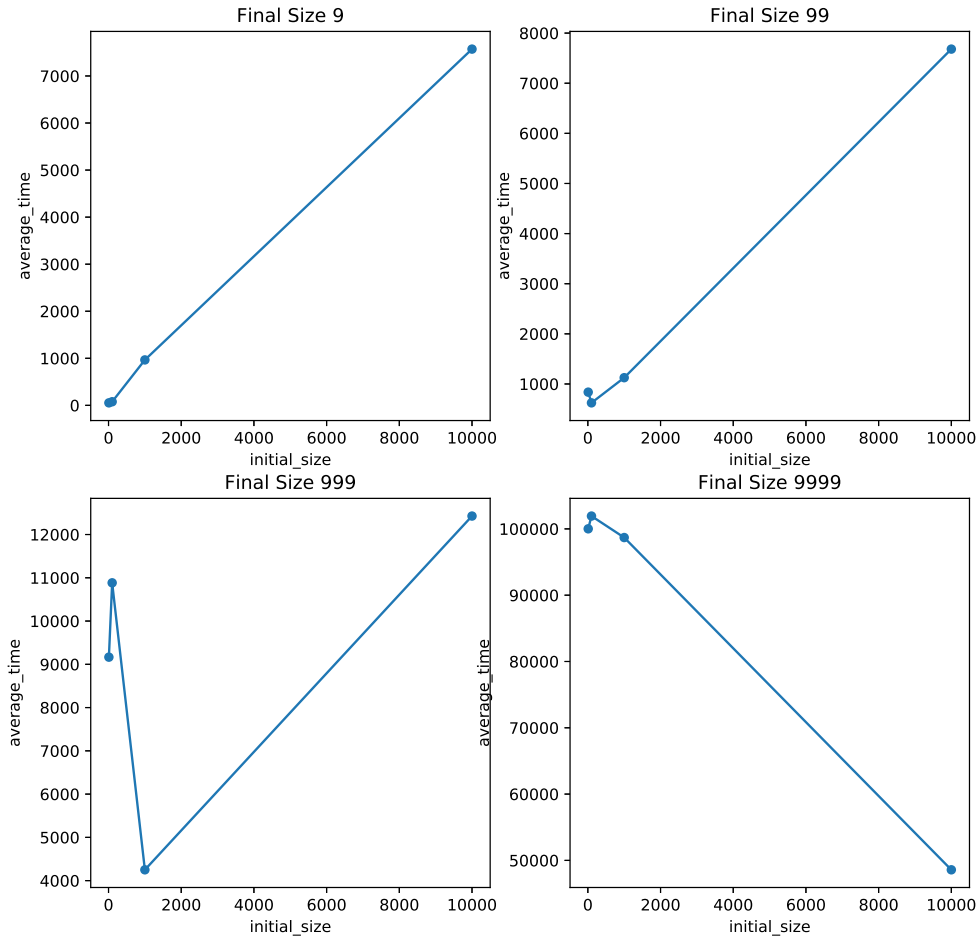


Figure 3·2: Memory allocation of ArrayList in Java

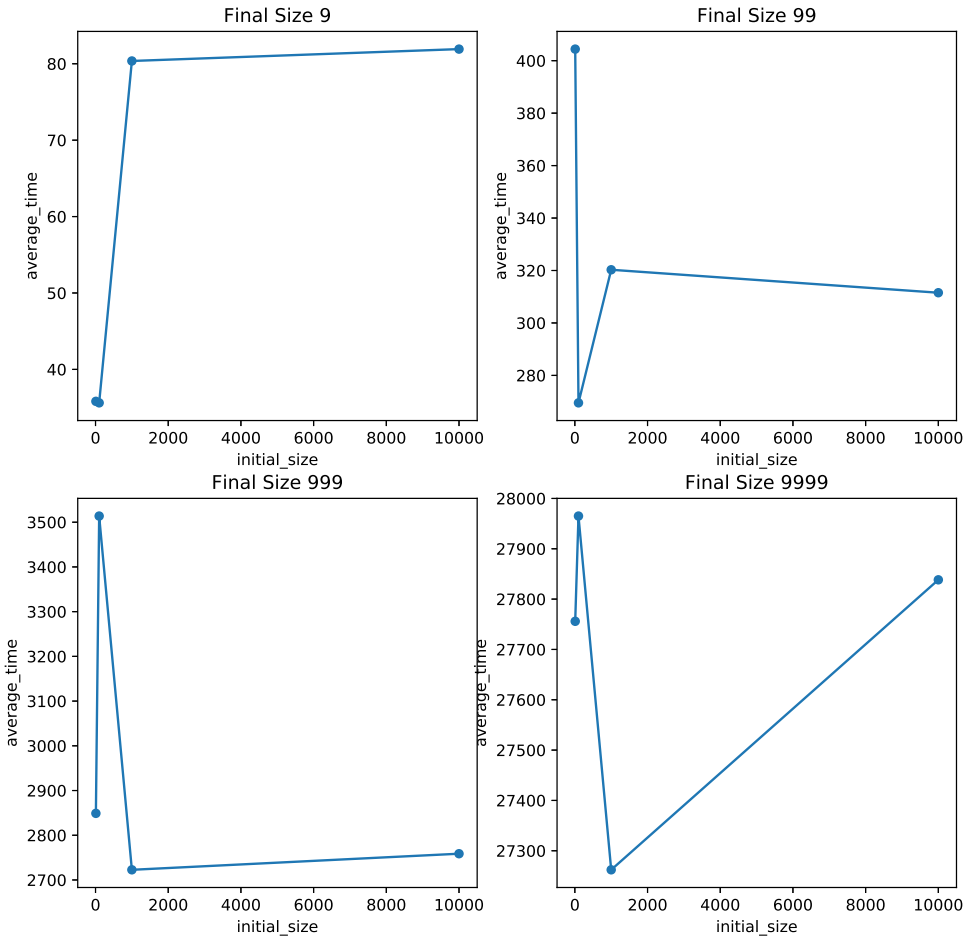


Figure 3·3: Memory allocation of Vector in Rust

3.3 Experiment of Memory Allocation for Different Element Type.

This experiment is to test how static and dynamic memory allocation of Java and Rust behave. For the assessment, Element addition to ArrayList in the case of Java and Vector in the case of Rust is emploied here. These data stractures are resizable and we have control to set initial size. There are two parameters; initial size of ArrayList or Vector and thier final size after additions of elements. We are interested in the impact to runtime performance by initializing memory allocation and dynamicaly allocating memory space.

First, an ArrayList and an Vector are created with specified initial size. Then, in a loop, a element is added for each iteration until the size of the ArrayList or Vector get the specified final size. Each data structure has a different resizing strategy. When an ArrayList hits current limit of its size and expands the limit, it doubles the current size. While Vector does not have specific strategy for its resizing, the expantions of size of both ArrayList and Vector might affect the digradation of runtime performance.

Second, four types of element are used for elements addition to each data structure: integer, array of charactors, string, and Customer object. Assamption is that there would be different behavior between element additions of dynamically resizable and static size objects. Customer object has three fields. These fields are total order, weight of order, and zip code whose types are integer (i32 in rust), double (f32 in rust), and string respectively. Figure 2-6 and 2-7 are representations of customer objects in Java and Rust.

Figure 2-10, 2-11, 2-12, and 2-13 represent the result of the experiments. For both data structures, integer elements addition shows the fastest runtime among all object types. This is because the compilers know each integer need 4 bytes to be stored in memory so that the space for memory that should be allocated is easily inspected. For the same reason, the initialization of data structures whose elements are integers alaways improves runtim performance.

The elements addition of strings and array of character behave similally among each languages. These two types of elements addition perform the similar speed and significantly slower than integer addition. Customer object addition is the slowest in Java. However, in

Rust the addition of Customer object is the second fastest among all element typs. The impacts of initialization of Java ArrayList vary among elemet types. On the other hand, the initailization Rust Vector always improves runtime performance for any of 4 types of elements addition.

```
class Customer {
    int totalOrder;
    double weightOrder;
    String zipCode;
}
```

Figure 3·4: Representation of Customer object in Java.

```
struct Customer {
    total_order: i32,
    weight_order: f32,
    zip_code: String,
}
```

Figure 3·5: Representation of Customer object in Rust.

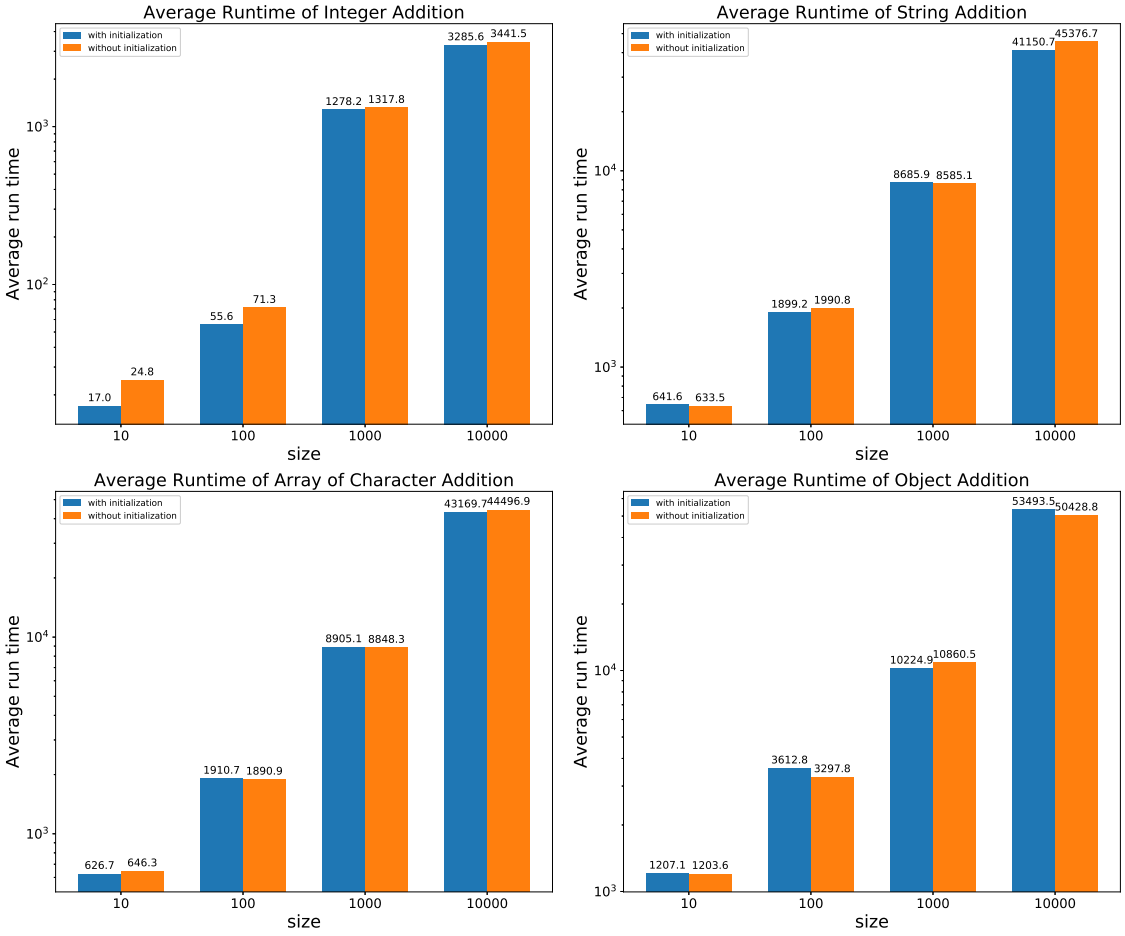


Figure 3·6: Memory allocation of Java ArrayList

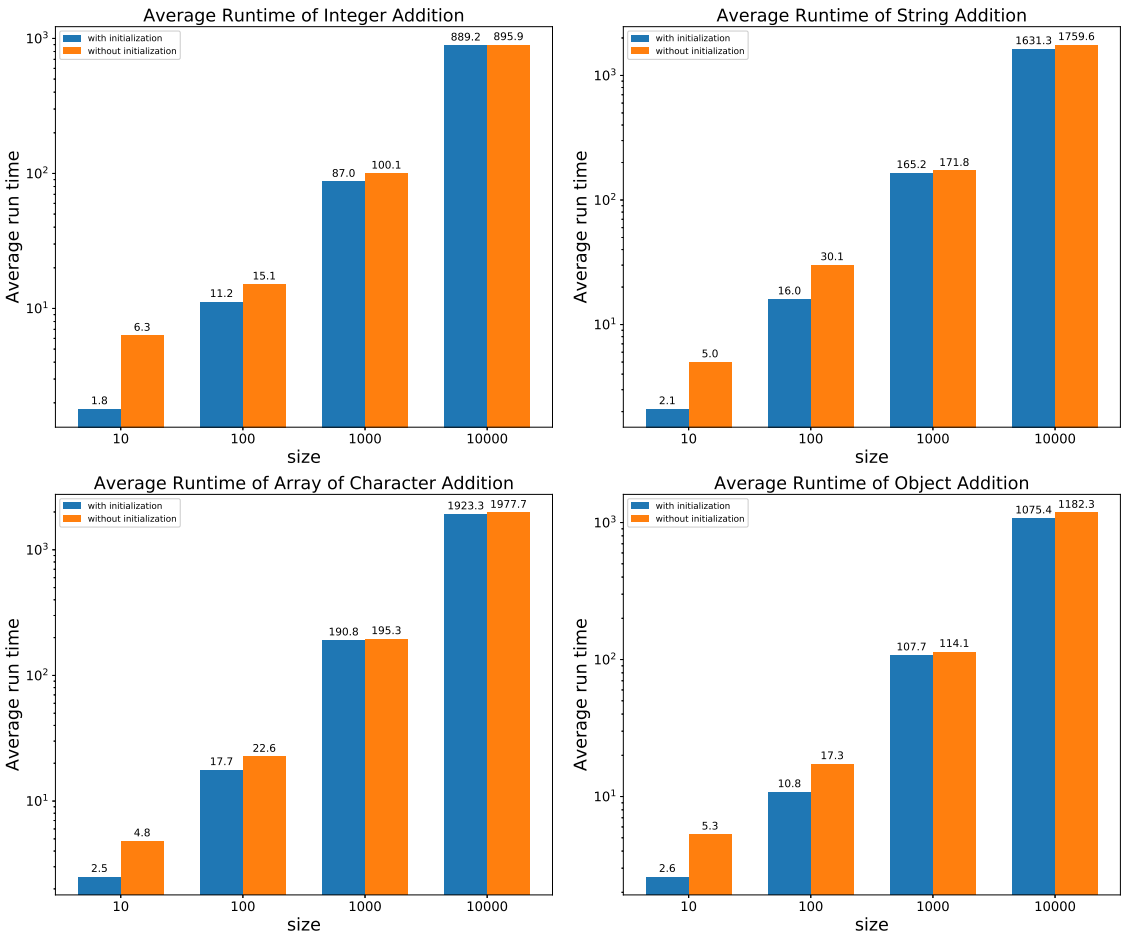


Figure 3·7: Memory allocation of Rust Vector

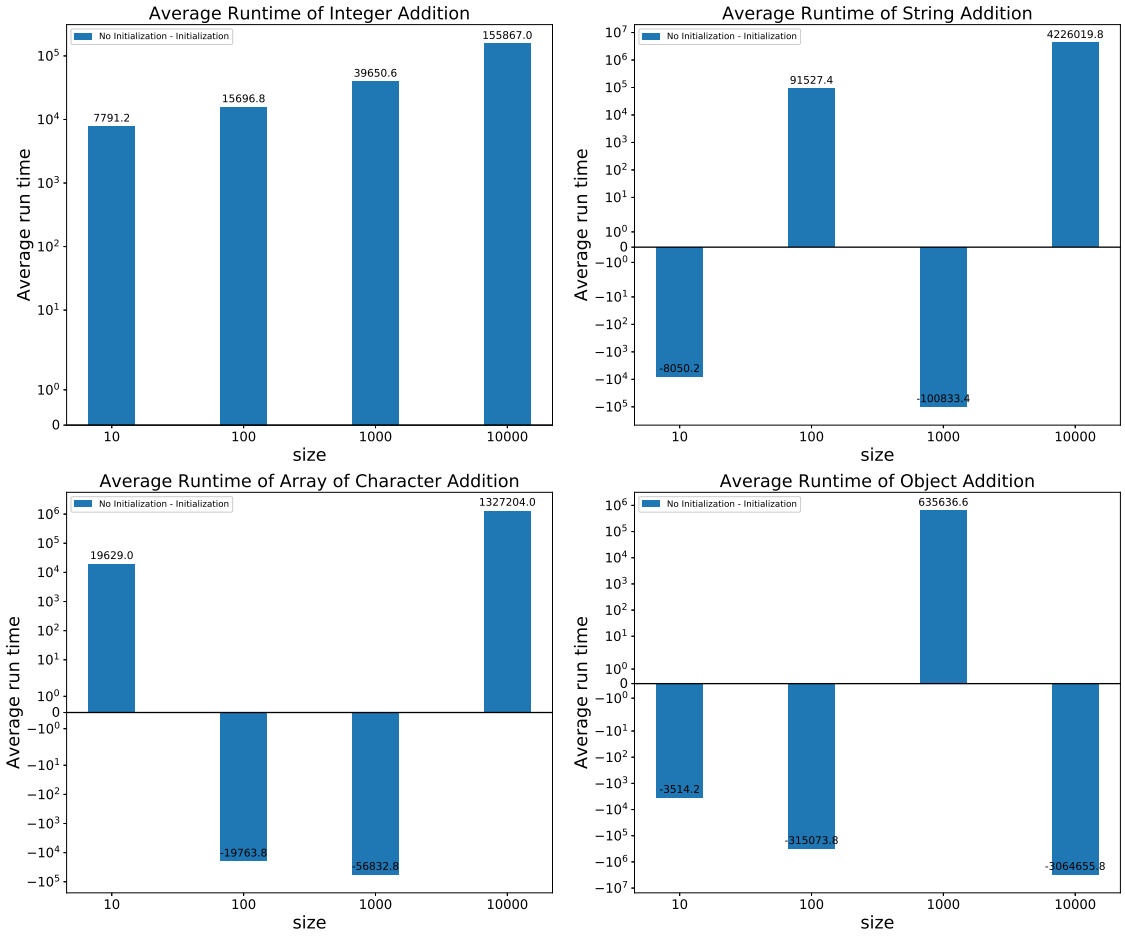


Figure 3·8: Difference of Memory allocation of Java ArrayList between Non-initialization and Initialization

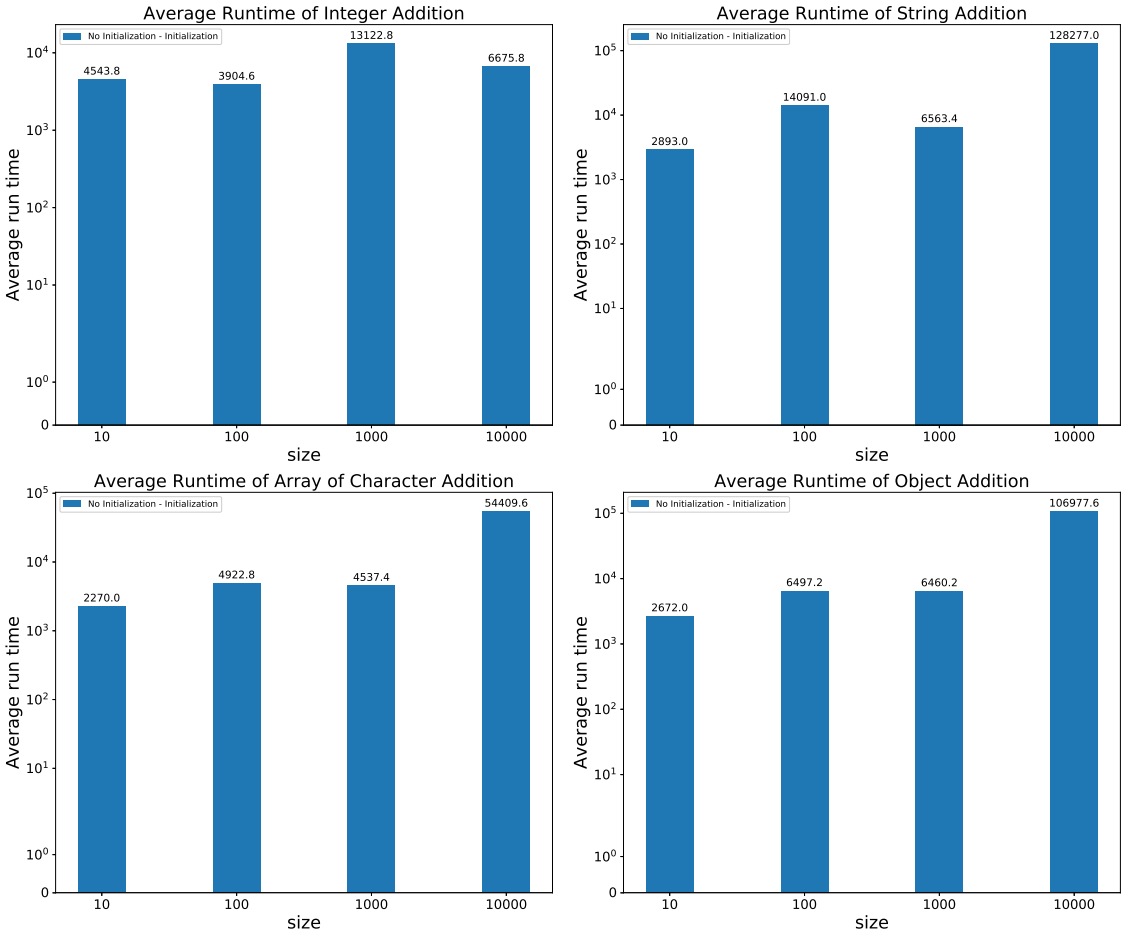


Figure 3·9: Difference of Memory allocation of Rust Vector between Non-initialization and Initialization

3.4 Assessment of different reference methods in Rust

3.4.1 Concept

The goal of this experiment is to assess efficiency of different reference strategies in Rust, borrowing and reference counting.

By levaraging reference counting, a value can be shared like what borrowing plays the role in Rust programming. The difference is that reference counting checks number of reference pointing to the actual data and makes sure the data is not deleted until all the references are dereferenced. Using reference counting is sometimes preferable approach for developers, we do not have care about lifetime which is usually troublesome when we use borrowing approach many times in our code. However, the possible problem regarding to reference counting is the cost for tracking the number of references. Having this assumption, this experiment will show difference of behavior among reference countinig and borrowing.

In this experiment, CustomerBorrowed and CustomerRc in figure are used to see difference of dropping time among borrowing and reference counting. In the CustomerRc and OrderRc struct, all fields take reference counting (Rc<T>). Similaly to the experiment in the last section, vectors of CustomerRc and CustomerBorrowed are created and droped the elements one by one.

The result shows significant difference of dropping time among the two objects; deletion of CustomerRc is much slower than CustomerBorrowed. This is because reference counters which are feilds of CustomerRc have to check the number of reference pointing to the actual content and decide deallocate the memory or not. However, memory management and lifetime strategy of borrowing is already determined at compile time.

In this experiment, an assessment is conducted to valify whether there is difference between behavior of borrowing and reference counting. These two methods can be used in similar situation. Especially, when we want reference to an original variable and keep the original variable until all of references are dereferenced, we can have reference with both storategies, borrowing and reference counting. One advantage of using reference counting is that we do not have to pay attention to lifetime. However, the assumption here is that

use of reference counting might be computationally expenesive than borrowing, because reference counting has to track the number of reference pointing to the value. Considered this assumption, we assess difference of time for dropping reference using borrowing and reference counting strategies.

Similally to the last section, we implement struct whose fields are borrowing reference and reference counting and measure the time to drop the struct.

3.4.2 Result

The result shows the time to drop borrowing reference is significatly faster than reference counting. This says that we should use borrwoing storategies whenever high performance computation is critical.

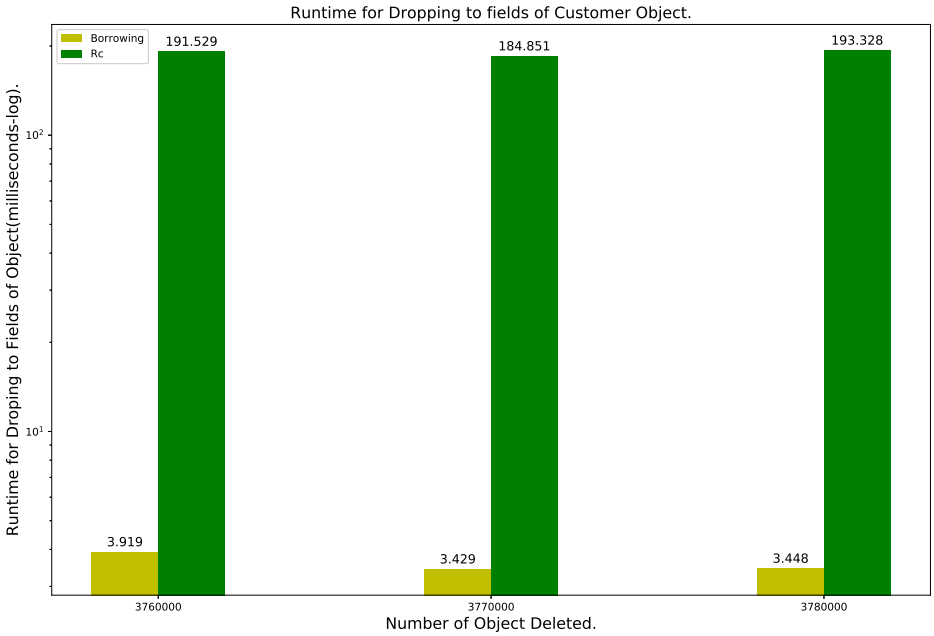


Figure 3·10: Runtime for Droping to fields of Customer Object

3.4.3 Discussion

3.5 Experiment for Merge-sort

3.5.1 Concept

This experiment is to assess importance of careful memory management in multithreading of Rust. In Rust programming, we usually need Atomic Reference Counting (Arc) to share data among threads. Arc is a simple interface that allows threads to share data, but it may cause significant overhead in Big Data processing. Arc allows multiple variable to have ownerships of a particular value similarly to Rc, but also supports atomic feature enabling the ownerships exist in different threads.

As explained in last experiment, deletion of Rc has significant overhead than normal reference. By learning this result, our assumption is that deletion of Arc has also overhead when we compare to normal reference. In many situation where developer write a multi-threading code, Arc is used and the deletion happens multiple times. To assess runtime overhead of algorithm with Arc, We implement merge-sort algorithm in two different ways. One is sharing source vector with Arc. The other is passing slice of source vector to child thread.

Our merge-sort algorithms are implemented with recursion. For each call of recursive function, Arc or slice of the source vector needs to be passed and deleted when the function returns value. These merge-sort algorithms trigger large number of Arc or reference deletion proportional to the number of call recursive function. Merge-sort algorithm can be separated in three phases: splitting phase, copying phase, and merging phase.

The splitting phase is merely acquiring index of range. At this phase, multiple threads are generated and Arc or reference of source vector are passed by calling recursive function. Copying phase occurs in the base case of the recursive call. The element in the source vector in a certain index is deep-copied into newly allocated vector. At merge phase, merge function receives two sorted independent vector and merge them into single new vector.

We use scope method from crossbeam crate to perform multithread programming. Scoped thread can have reference to value from its parent thread by ensuring children threads are joined before their parent thread returns value. By using scoped thread, we can

implement two merge-sort algorithms in identical way except whether the function receives Arc or reference of source vector. The representations of source vector for each algorithm are shown in figure.

```
// Source vector for algorithm with Arc.
arr: Arc<VecDeque<T>>

// Source vector for algorithm with reference (slice).
arr: &[T]
```

Figure 3·11: Representation of Source Vector

3.5.2 Result

The elements of source vector are CustomerOwned objects. We generates source vectors in size of 1 million, 2 million, 3 million and 4 million. Finally, merge-sort is performed based on Customer key value. The figure shows the result for runtime performance of merge-sort algorithms on difference size of vectors.

3.5.3 Discussion

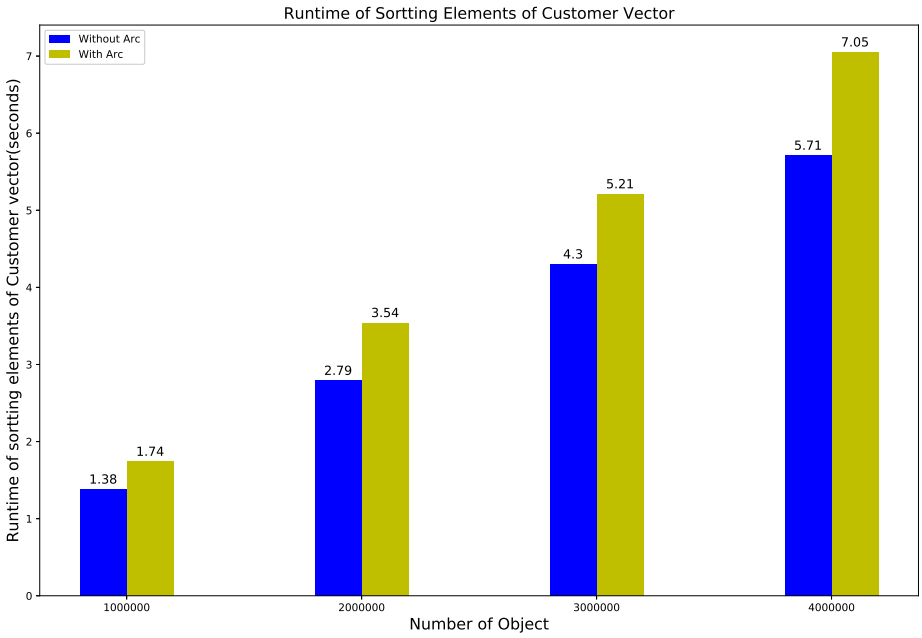


Figure 3·12: Runtime of Sortting Elements of Customer Vector

3.6 Experiment for Tree-aggregation

3.6.1 Concept

Tree-aggregate is a communication patten heavily used for Machine Learning algorithm in Spark (MLlib). In the traditional aggregation function in Spark, results of aggregation in all executor clusters are sent to the driver. That is why this operation suffers from the CPU cost in merging partial results and the network bandwidth limit. Tree-aggregate is a communication pattern which overcomes these problems by breaking aggregate operation in multi-level represented like tree structure.

In our experiment, tree-aggregation algorithms are examined in multi-threading. This experiment is to evaluate the impact of having Arc (Atomic Reference Counting) as elements of vector. In Big Data mining tool, such as Spark, it generates intermediate objects from original source vector. In tree-aggregation, aggregated HashMap like data structure is created in each step or node. Acquisition of elements in source vector is required to perform this aggregation. There are several ways.

One way is deep-copy elements of vector. This solution allocated newly created objects by deep-copy. Aggregation is performed on copied objects, stores them in the data structure and sends it to next node. Deep-copy generates duplicates of objects in vector and aggregated data structure. This can lead to memory intensive moment when we need memory space for the duplicated objects in addition.

The other way is to get reference to the elements. Since an original source vector is deallocated after a local aggregation, Simple reference to elements does not live long enough and allow the aggregation result to be sent to next node. Instead of simple borrowing, we need owner in the aggregation result. Reference Counting (Rc) in Rust is a way to have multiple owners to a value. Since our experiment is implemented in multithreading, Atomic Reference Counting (Arc) is used instead of Rc. With Arc, multiple ownership pointer can be possessed by different variables across multiple threads. Therefore a value is not deallocated until all of owners to it are dropped. This does not require extra memory allocation, because only acquisition of new ownership to value is needed. However, deletion

of Arc type checks whether the value is still owned by other variables. This checking may be an overhead in algorithms where generate a lot of intermediate data structures, because deletion of the data structures occurs in frequent.

Two algorithms are implemented using the above two different methods and evaluated their runtime performance. The both algorithms perform tree-aggregation where runs on seven partitions of source vector. Each steps load Customer vector from disk and aggregate it by Customer last name. Once a node finishes aggregation, it sends result to parent node. After parent nodes receive aggregation results from all of its children nodes, it joins all aggregation results including its and sends next parent. One algorithm performs aggregation by deep-copying elements from source vector loaded from disk. In the other algorithm, each element of source vector is wrapped in Arc, and its reference is acquired while aggregation.

3.6.2 Result

Figure shows runtime performance of two tree-aggregate algorithms. The runtime of algorithm with deep-copy is slower than algorithm with Arc for every vector size. This is result of overhead of deep-copy is larger than deletion of Arc.

3.6.3 Discussion

As we explained, Arc has overhead to be deleted because it has to check if the value is still referred. Even though the deletion of Arc is slow, deep copy of complex objects has more impact in deterioration of runtime performance. In the algorithm with deep copy, the elements of each partition in each node are deep-copied once during aggregation.

If total number of elements are 1000000, the all of 1000000 elements are deep-copied once during execution. On the other hand, acquisition and deletion of Arc occurs several time for each element object. First, acquisition of Arc of all elements in loaded vector happens during aggregation in each node and the loaded vector is deleted after the aggregation. Second, deletion of all elements in aggregated structures from children nodes and the node is occurs after joining these aggregated structures to single one.

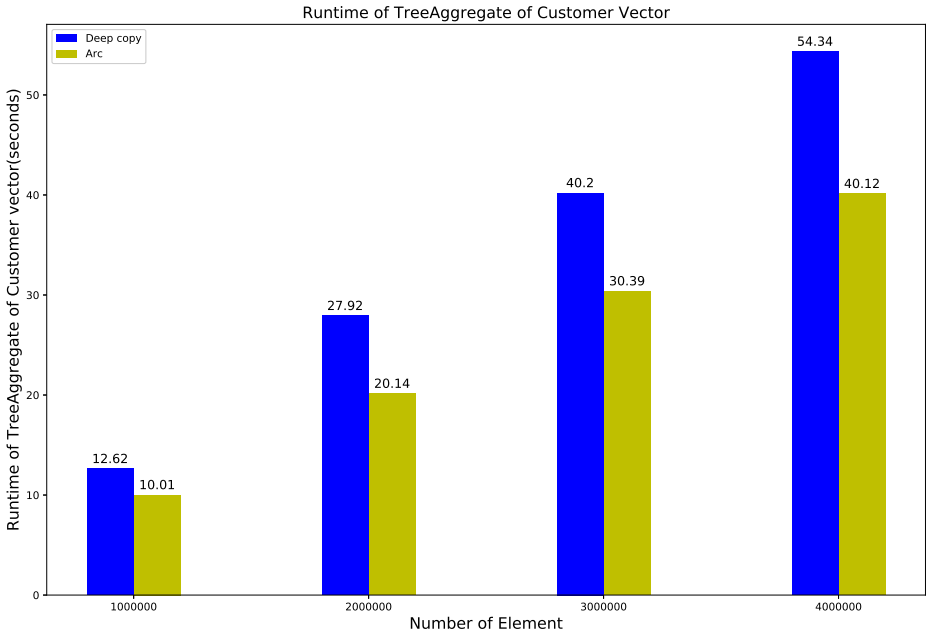


Figure 3·13: Runtime of Tree-aggregate algorithm

The result shows a deep-copy of Customer vector is computationally much more expensive than acquisition and deletion of Arc. This result suggests that having element objects in Arc is efficient than deep-copying element from original source vector in tree-aggregation algorithm where generates intermediate data structure.

```
fn aggregate_local(arr :&[Arc<CustomerOwned>])
{
    let mut agg = HashMap::new();
    let n = arr.len();
    for i in 0..n {
        let customer = Arc::clone(&arr[i]);
        let last_name = customer.last_name.clone();
        let vector = agg.entry(last_name).or_insert_with(Vec::new);
        vector.push(customer);
    }
    return agg;
}
```

Figure 3·14: Aggregation function with Arc

```
fn aggregate_local_copy(arr :&[CustomerOwned])
{
    let mut agg = HashMap::new();
    let n = arr.len();
    for i in 0..n {
        let customer = arr[i].clone();
        let last_name = customer.last_name.clone();
        let vector = agg.entry(last_name).or_insert_with(Vec::new);
        vector.push(customer);
    }
    return agg;
}
```

Figure 3·15: Aggregation function with deep-copy

Chapter 4

Conclusions

4.1 Summary of the thesis

Time to get philosophical and wordy.

IMPORTANT: In the references at the end of thesis, all journal names must be spelled out in full, except for standard abbreviations like IEEE, ACM, SPIE, INFOCOM, ...

Appendix A

Proof of xyz

A.1 Memory and Process in Operation Systems

A process is a section of computation job. A process can work on a CPU core. We can divide process as well. Basically, each process does not share their memory. However, for multiprocessing, we could avoid this restriction. Processes can be represented as tree structure, because a process may create other child processes. Process has 4 states, new, running, waiting, and ready. Process is represented in process control block (PCB) with state type, process ID, registers, and so on. The scheduling for process assigning to CPU core is implemented in queues containing PCB. There are two main queues in this scheduler: ready queue and wait queue. The head of process in ready queue is selected for execution and once the process requested I/O request or production of child process, the running process will be stored n wait queue. Once the request that the process waiting for end, the waiting process will be pushed tail of ready queue.

Processes executing concurrently in the operating system may be either independent processes or cooperating processes executing in the system. A process is independent if it does not share data with any other processes. A process cooperating if it can affect or be affected by the other processed executing in the system. In cooperating process, there are two kinds, shared memory and message passing. In shared memory, it removes restriction of not interfering memory region. Message passing can be useful for distribution systems as well.

For a pair of processes to communicate through message, a socket is needed to be established. A socket is identified by an IP address concatenated with a port number. When two process communicate, each process will have socket. If another process of the

same machine wants to communicate, we need new socket to be established. The protocol used in the socket connection can be TPC and UDP.

A.2 Multi-threading and Parallelism

A thread is a basic unit of CPU utilization, so that a process can have multiple thread. Threads share mainly code and data. Multi-threading is increasingly popular as the multicore programming becomes in common, because we can run multiple thread on different core. Creating thread is much cheaper than creating process and it shares resources so that we do not need additional methods to allow threads to communicate each other, such as sharing memory and message passing.

A.3 Memory management in Operation System

In computer storage hierarchy, the closest storage to CPU is register. It is built into each CPU core and accessible within one cycle of the CPU clock. However, the same cannot be said of main memory, which is accessed via a transaction on the memory bus. This takes many cycles of the CPU clock. The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for faster access. Such a cache plays a role for this.

For the layout of main memory, it must be ensured that each process has a separate memory space, including operation system. The base register and limit register, whose roles are lower bound of memory region and specific size of range respectively, can achieve that goal.

Usually a program resides on a disk as binary executable file. To run, the program must be brought into memory and placed within the context of a process. The process is bound to corresponding parts of the physical memory. Binding program to memory address is staging process. There three stages: compile time, load time execution time. The source program is compiled by compiler producing object file. After the compilation, the object file is linked with other object file by linker creating executable file . Finally, the executable

file will be loaded to run execute. At this run time dynamic library link can be done.

If where the process will reside in memory at compile time, absolute code is generated. If this is unknown at the time, the binding will be done at load time. At this time, the compiler must generate relocatable code. Otherwise, the binding will be done at execution time.

A process does not interact with addresses of physical memory, instead virtual memory. The memory-management unit (MMU) takes roles to map logical address to physical address. OS needs to ensure that any of physical memory spaces of processes do not overlap. Since one process can be created and deleted and the corresponding memory space should be de/allocated, optimization for use of physical memory space is important; we need to allocate memory contiguously avoiding fragmentation.

There are several approaches to deal with this problems. However, we will focus on paging here, which is the most used method OS use to manage memory. A frame and page are a unit of Separated physical and virtual memory space in fixed size (4KB or 8KB) respectively. A process can use as many as pages and corresponding frames obtained by page table matching. This strategy does improve external fragmentation, but not for internal fragmentation. The smaller size of page has smaller fragmentation, but mapping from page to frame has overhead and also disk I/O is more efficient when the amount of data transfered is larger.

A.4 Demand Paging

A process can have multiple pages. However, loading entire executable code from secondary storage to memory is not necessarily needed to get jobs done. A strategy used in several operating systems is loading only the portion of programs that are needed, demand page.

In the storage, some pages currently used are in memory and the others are in secondary storage. The page table specifies whether pages are valid or invalid, which means are in memory or not. Access to a page marked invalid causes page fault and some steps to resolve the error will be required.

The first part of process of demand paging would be that we check an internal table to check whether the reference is valid or invalid. If the reference is valid, the process reads the content from the memory. Otherwise, we terminates the process and find the free frame in physical memory. Then, we schedule a secondary storage operation to read the desired page into newly allocated frame. When the storage complete reading the page, we modify the internal table to indicate the page is now in memory. Finally, we restart the instruction that was interrupted.

However, there would be a case where the memory does not have any free frame. In this case, a victim frame that will be replaced with new coming frame should be selected. To perform this selection efficiently, a modify bit is tracked for each frame or page. The modify bit represents whether the page is modified since it is loaded from secondary storage. If the page or frame is modified, when we swap page we need to update the content in the secondary storage. However, it is not modified, we can simply delete the frame and replace with new frame.

A.5 Copy on Page

When a parent process creates child process and if these process shares contents on particular page and modify it, only the page which has the contenst will be copied.

A.6 Threads and Concurrency

A.7 BLAS LAPACK

Basic Linear Algebra Subprograms (BLAS) are standard building blocks for basic vector and matrix operations. There are 3 levels of operation. The level 1 BLAS performs scalar, vector and vector-vector operations, the level 2 BLAS performs matrix-vector operation, and the level 3 BLAS performs matrix-matrix operation.

LAPACK is developed on BLAS and has advanced functionalities such as LU decomposition and Singular Value Decomposition (SVD). Dense and banded matrices are handled, but not general sparse matrices. The initial motivation of development of LAPACK was to

make the widely used EISPACK and LINPACK libraries run efficiently on shared-memory vector and parallel processors. LINPACK and EISPACK are inefficient because their memory access patterns disregard the multi-layered memory hierarchies of the machines, thereby spending too much time moving data. LAPACK addresses this problem by recognizing the algorithms to use block matrix operations, such as matrix multiplication, in the innermost loops. These block operations can be optimized for each architecture to account for the memory hierarchy, and so provide a portable way to achieve high efficiency on diverse modern machines. However, LAPACK requires that highly optimized block matrix operations be already implemented on each machine.

ARPACK is also a collection of linear algebra subroutines which is designed to compute a few eigenvalues and corresponding eigenvectors from large scale matrix. ARPACK is based upon an algorithmic variant of the Arnoldi process called the Implicitly Restarted Arnoldi Method (IRAM). When the matrix is symmetric it reduces to a variant of the Lanczos process called the Implicitly Restarted Lanczos Method(IRLM). The Arnoldi process only interacts with the matrix via matrix-vector multiplies. Therefore, this method can be applied to distributed matrix operations required in big data analysis.

The original BLAS and LAPACK are written in Fortran90. Linear algebra library used for Spark is netlib-java, which is a Java wrapper library for Netlib, C API of BLAS and LAPACK. The reason why the developers addressed to use this package is that the BLAS and LAPACK are already bug free and implementing linear algebra library from scratch can usually buggy.

However, the main advantage of use of BLAS and LAPACK is system optimized implementation. So if we implement original Fortran linear algebra library, it cannot perform as well as BLAS and LAPACK. And the performance would not be such different from one of implementation in Java or Rust. If we want to test only memory management between Rust and Java, it can be enough implementation of linear algebra operation from pure Java and Rust sacrificing the best performance taking advantage of system optimization.

A.8 Netlib-Java

Netlib-java is a Java wrapper of BLAS, LAPACK, and ARPACK. Netlib-java choose implementation of linear algebra depending on installation of the libraries. First, if we have installed machine optimised system libraries, such as Intel MKL and OpenBLAS, netlib-java will use these as the implementation to use. Next, it try to load netlib references which netlib-java use CBLAS and LAPCKE interface to perform BLAS and LAPACK native call. The last option is to use f2j which is intended to translate the BLAS and LAPACK libraries from their Fortran77 reference source code to Java class files, instead of calling native libraries by using Java Native Interface (JNI).

We can use JNI to call native libraries from Java. The JNI is a native programming interface which allows Java code that runs inside a Java Virtual Machine (VM) to interoperate with applications and libraries written in other programming languages.

A.9 Create Java interface of CBLAS with JNI

1. Download BLAS and build using make file. In the figure2.1, the built file is libblas.a and the header file is blas.h.

2. Download CBLAS and build using make file (I am not sure whether we should build archive file or shared library). In figure, the build file would be libcblas.a or libcblas.dylib and header file is cblas.h.

3. Create java file which will be the Java interface of CBLAS.

4. Compile java file with -h header flag to create class file (CBLASJ.class) and header file (CBLASJ.h).

```
$ javac -h . CBLASJ.java
```

5. Create C file (CBLASJ.c) which will bind Java interface and CBLAS library. And compile it with JNI to create object file (CBLASJ.o).

```
$ gcc -c -fPIC -I${JAVA_HOME}/include -I${JAVA_HOME}/include/darwin CE
```

6. Compile shared library linking library (libcblas.a or libcblas.dylib) object file (CBLASJ.o).

```
$ gcc -o libcblas.dylib (or libcblas.a) CBLASJ.o
```

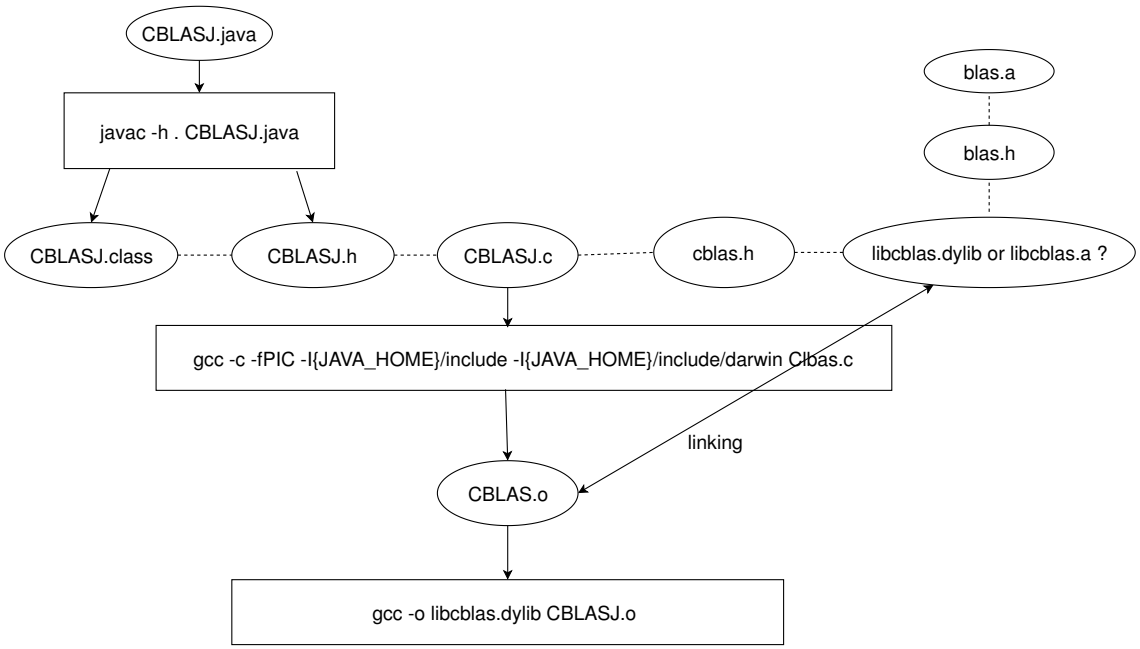


Figure A·1: Integration of Native Methods

A.10 Matrix Computation and Optimization in Apache Spark

Matrix operation is a fundamental part of machine learning. Apache Spark provides implementation for distributed and local matrix operation. To translate single-node algorithms to run on a distributed cluster, Spark addresses separating matrix operations from vector operations and run matrix operations on the cluster, while keeping vector operations local to the driver.

Spark changes its behavior for matrix operations depending on the type of operations and shape of matrices. For example, Singular Value Decomposition (SVD) for a square matrix is performed in distributed cluster, but SVD for a tall and skinny matrix is on a driver node. This is because the matrix derived among the computation of SVD for tall and skinny matrix is usually small so that it can fit to single node.

Spark uses ARPACK to solve square SVD. ARPACK is a collection of Fortran77 designed to solve eigenvalue problems. ARPACK is based upon an algorithmic variant of the Arnoldi process called the Implicitly Restarted Arnoldi Method (IRAM). When the matrix A is symmetric it reduces to a variant of the Lanczos process called the Implicitly Restarted Lanczos Method (IRLM). ARPACK calculate matrix multiplication by performing matrix-vector multiplication. So we can distribute matrix-vector multiplies, and exploit the computational resources available in the entire cluster. The other method to distribute matrix operations is Spark TFOCS. Spark TFOCS supports several optimization methods.

To allow full use of hardware-specific linear algebraic operations on single node, Spark uses the BLAS (Basic Linear Algebra Systems) interface with relevant libraries for CPU and GPU acceleration. Native libraries can be used in Scala are ones with C BLAS interface or wrapper and called through the Java native interface implemented in Netlib-java library and wrapped by the Scala library called Breeze. Following is some of the implementation of BLAS.

- f2jblas - Java implementation of Fortran BLAS
- OpenBLAS - open source CPU-optimized C implementation of BLAS
- MKL - CPU-optimized C and Fortran implementation of BLAS by Intel

These have different implementation and they perform differently for the type of operation and matrices shape. In Sark, OpenBlas is the default method of choice. BLAS interface is made specifically for dense linear algebra. Then, there are few libraries that efficiently handle sparse matrix operations.

A.11 Memory Management of each Linear Algebra Library

The pure Java linear algebra library, such as La4j, EJML, and Apache Common Math, use normal GC performed by JVM to manage memory. This is because the implementation of these libraries are in purely Java.

Netlib-java, Jblas or other simple Java wrapper of BLAS, LAPACK, and ARPACK with Java Native Interface (JNI) use normal GC as well. This is because the native code deals with Java array by obtaining a reference to it. After the operation, the native method releases the reference to the Java array with or without returning new Java array or Java primitive type object.

ND4J has two types of its own memory management methods, GC to pointer of off-heap NDArray, and MemoryWorkspaces. ND4J used off-heap memory to store NDArrats, to provide better performance while working with NDArrays from vative code such as BLAS and CUDA libraries. Off-heap means that the memory is allocated outside of the Java heap so that it is not managed by the JVM's GC. NDArray itself is not tracked by JVM, but its pointer is. The Java heap stores pointer to NDArray on off-heap. When a pointer is dereferenced, this pointer can be a target of JVM's GC and when it is collected, the corresponding NDArray will be deallocated. When using MemoryWorkspaces, NDArray lives only within specific workspace scope. When NDArray leaves the workspace scope, the memory is deallocated unless explicitly calling method to copy the NDArray out of the scope.

References

Debreuve, E., Barlaud, M., Aubert, G., Laurette, I., and Darcourt, J. (2001). Space-time segmentation using level set active contours applied to myocardial gated SPECT. *IEEE Trans. Med. Imag.*, 20(7):643–659.

Lamport, L. (1985). *LaTeX—A Document Preparation System—User's Guide and Reference Manual*. Addison-Wesley.

Xu, L., Guo, T., Dou, W., Wang, W., and Wei, J. (2019). An experimental evaluation of garbage collectors on big data applications. *PVLDB*, 12(5):570–583.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Gribble, S. D. and Katabi, D., editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association.

CURRICULUM VITAE

Joe Graduate

Basically, this needs to be worked out by each individual, however the same format, margins, typeface, and type size must be used as in the rest of the dissertation.