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

Contents

List of Tables

List of Figures

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 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 Memory Management in Rust

Each value in Rust has a variable called its 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 life time. When the owner goes out of it's scope, the value will be dropped. When a value already assigned to a variable is assigned to another variable, if the value is allocated on heap its information is copied to the new owner and drop the old owner disabling old variable. Similar thing happens when we pass variable to parameter of function. After passing a variable to a parameter, all the information is copied to new owner through the parameter and old owner is no longer available. The new owner can only live in the function and the object will be dropped. In this case, we have no longer access to the object after the function. To avoid this, Rust has a concept called borrowing. We can set reference for the parameter of function and use the reference for operation within function and drop the reference, but not the ownership.

2.2 Spark and RDD Catching

Spark is one of the most used big data computing framework. Spark uses Resilient Distributed Datasets (RDDs) which implement in-memory data structures used to cache intermediate data across a set of nodes. This enables multiple rounds of computation on the same data, which is required for machine learning and graph analytics iteratively process the data.

In RDD caching, there are different stages of caching, such as MEMOR_YONLY and DISK_ONLY. Currently, for very large data sets, we need to pay attention to garbage collection (GC) and OS page swapping overhead, because these could degrades execution time significantly. Therefore, DISK_ONLY RDD caching can be better configuration in this case. However, writing and reading intermediate data among desk and memory could have bad effects for execution time, due to need of serialization and deserialization.

2.3 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.

2.4 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.

2.5 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 CBLASJ.c
```

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

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

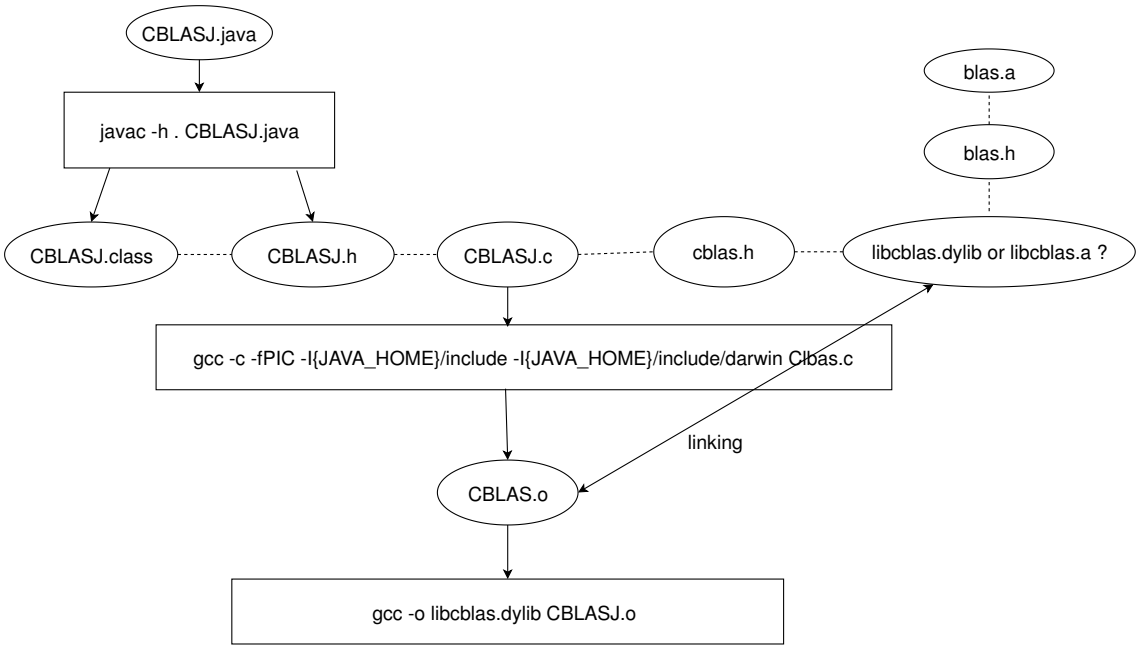

Figure 2·1: Integration of Native Methods

2.6 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.

2.7 Memory Management of Java

Garbage Collection (GC) is a Java memory management method performed by JVM. GC tracks the state of objects on Java heap and triggers removal of unreferenced objects from memory.

The Java 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 young generation of Java heap contains new objects allocated and aged. When the young generation fills up, this triggers minor GC. All of unreferenced objects will be removed from Java heap and remained objects will be aged and eventually promoted to the old generation.

The old generation is used to store long survived objects. Typically, a threshold is set for young generation object and when the age is met, the object will be moved to the old generation. Eventually, the old generation object also needs to be removed when the object is unreferenced. This is called major GC. The process of generational GC is shown in Figure.

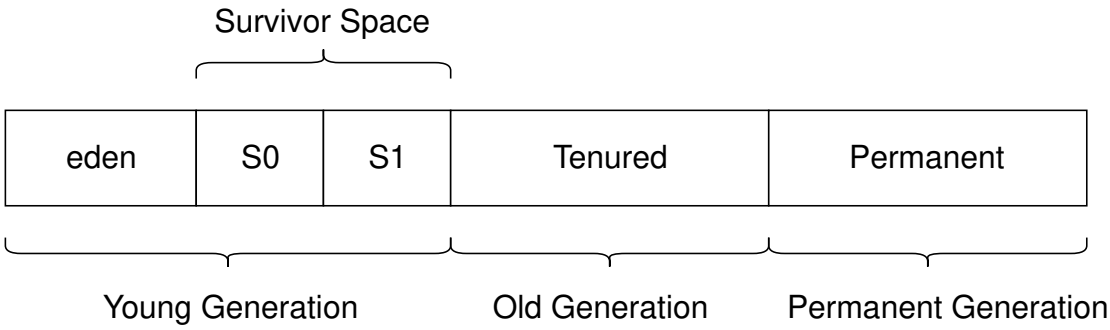

Figure 2·2: Java Heap Structure

2.8 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.

2.9 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

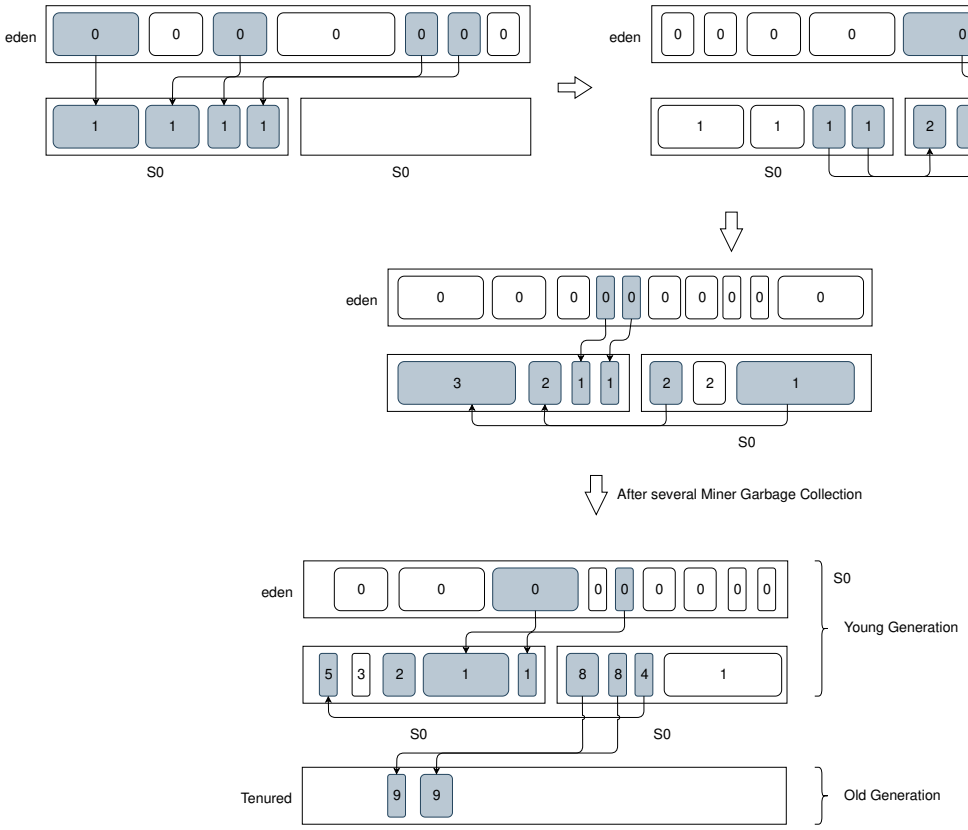

Figure 2·3: Java Garbage Collection

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.

2.10 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.

2.11 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$_from which al so per forms bitwise deep copy, but copies elements of th$

The figure shows the result of the experiment. Among the runtime performances of integer insertion for every methods, clone, clone$_from, and copy_n o n overlapping methods shows the similar per formance. However, th$

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.

2.12 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.

2.13 Access time to onwned, borrowed, and sliced field of object

In this experiment, differences of access time among to owned, borrowed, and sliced are observed. The figure is representation of a Customer object. Among the experiment, we focus on $\mathrm{zip}_code, address, country fields$ $ownner, reference, slice$.

The result shows that access times to onwer and reference are almost the same, but the one to slice is relatively faster compared to the other two.

2.14 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.

For

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

Design experiment for comparing among the performance of smart pointers.

Box¡T¿ - Box pointer lets value is allocated on heap rather than on the stack Rc¡T¿ - Reference counted pointer lets variable to take multiple immutable ownership. RefCell¡T¿

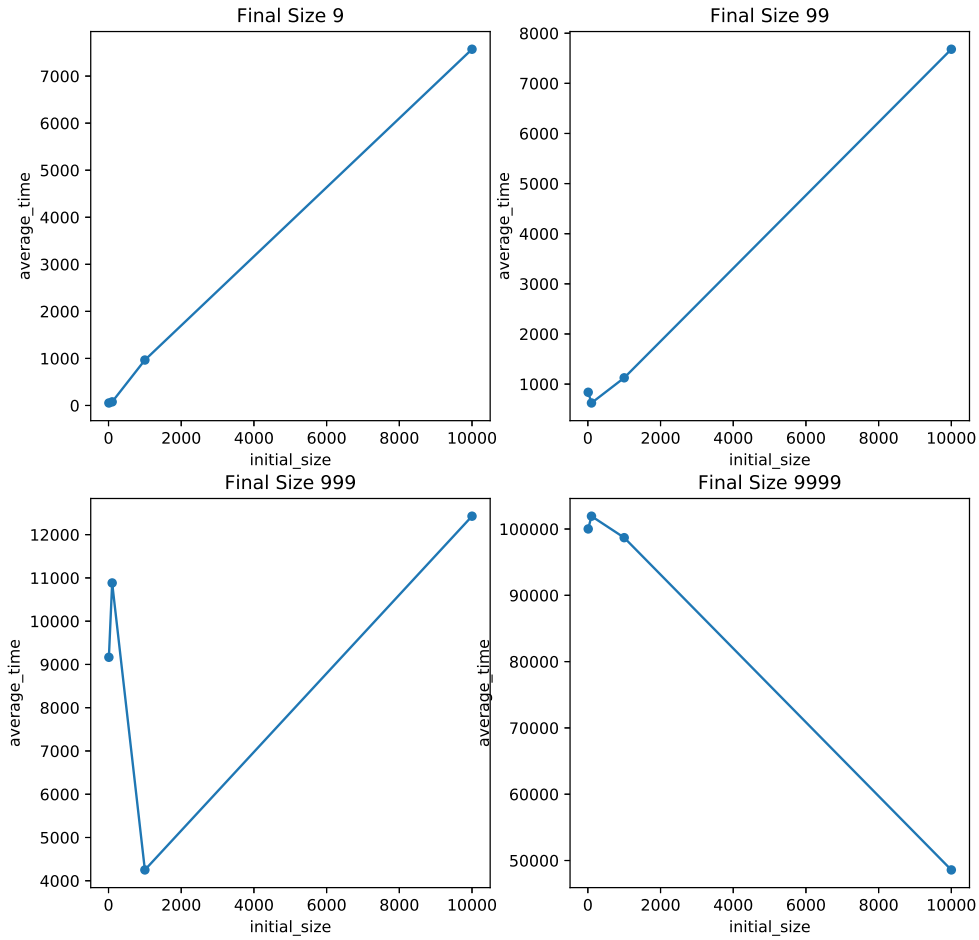

Figure 2·4: Memory allocation of ArrayList in Java

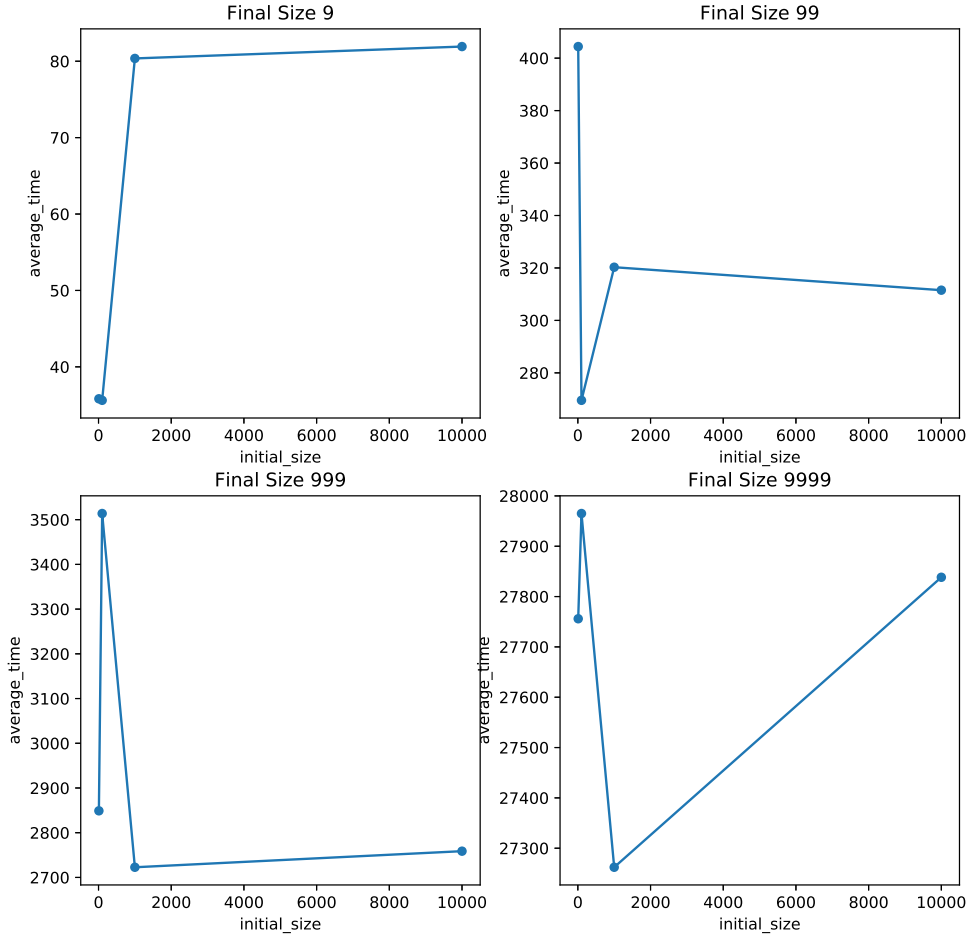

Figure 2·5: Memory allocation of Vector in Rust

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

Figure 2·6: Representation of Customer object in Java.

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

Figure 2·7: Representation of Customer object in Rust.

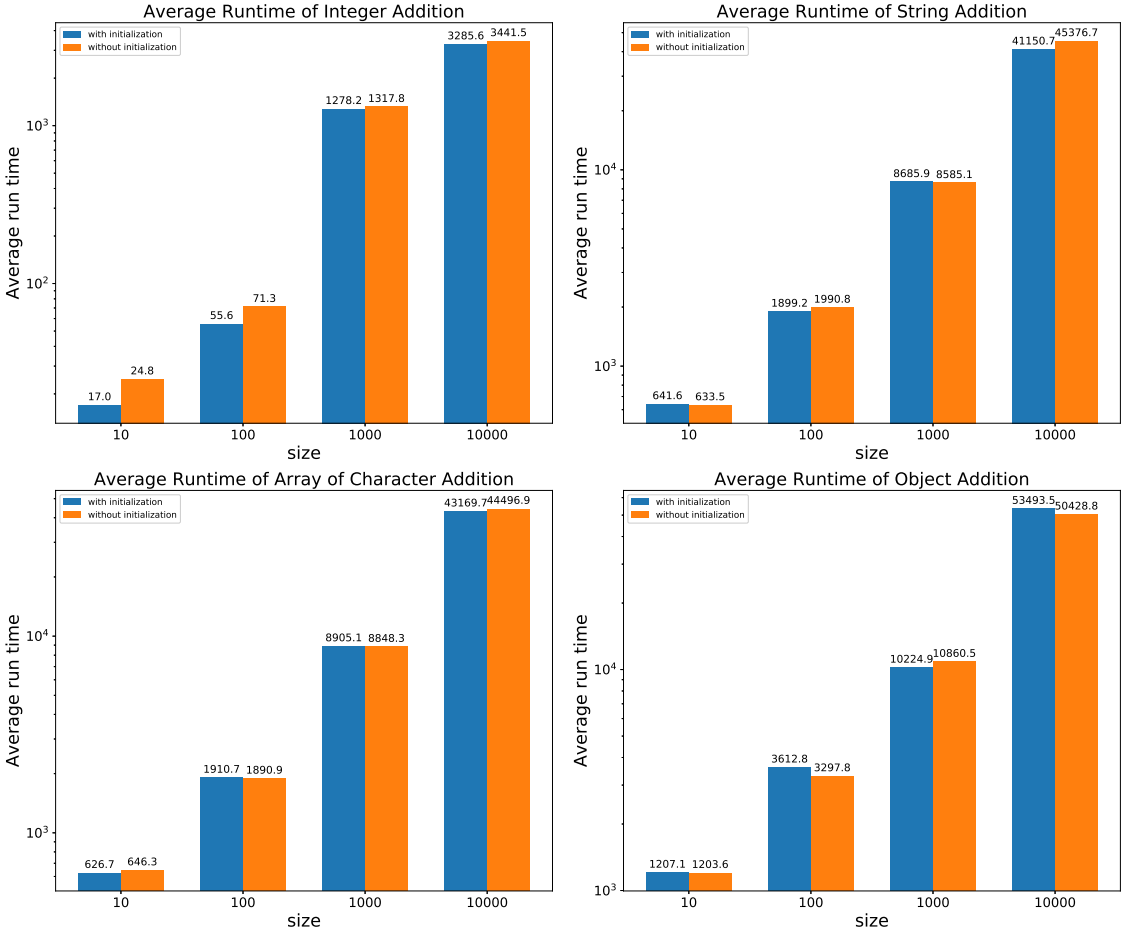

Figure 2·8: Memory allocation of Java ArrayList

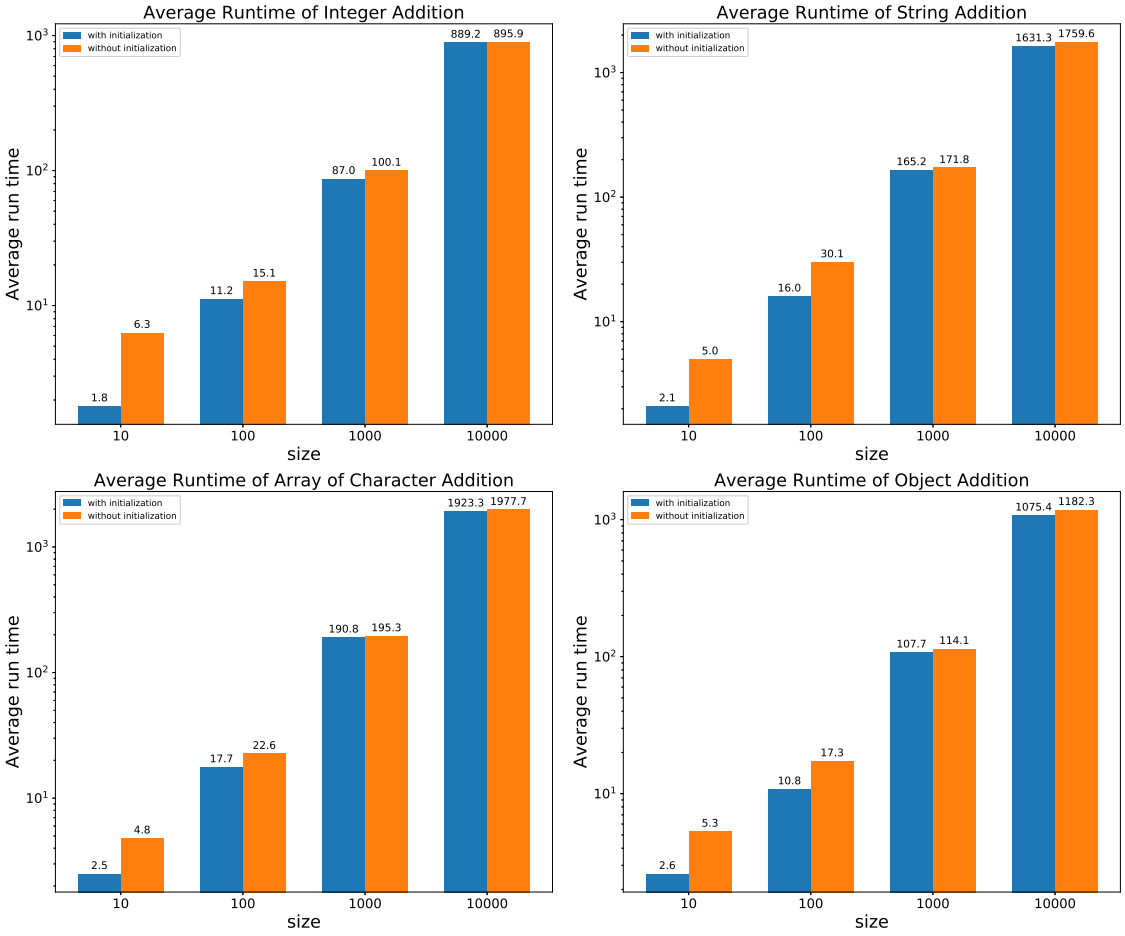

Figure 2·9: Memory allocation of Rust Vector

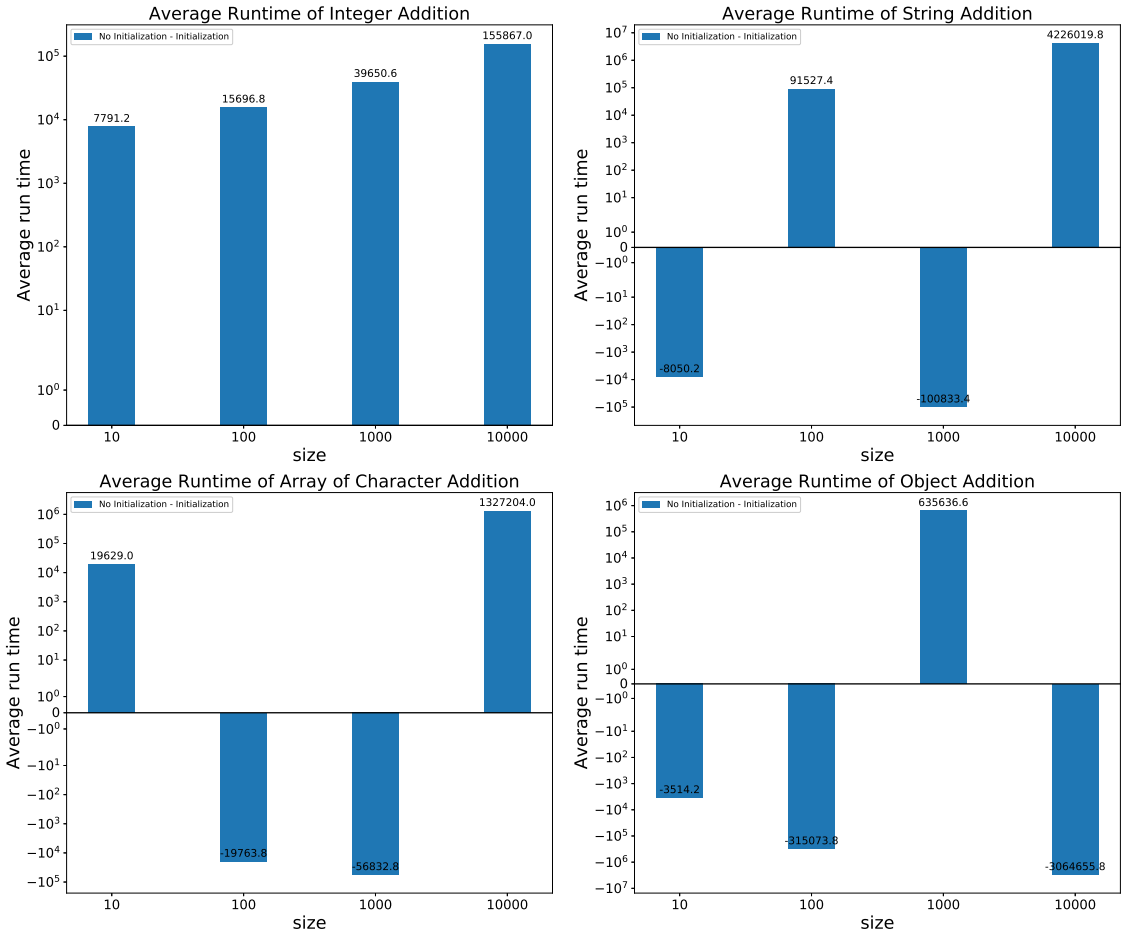

Figure 2·10: Difference of Memory allocation of Java ArrayList between Non-initialization and Initialization

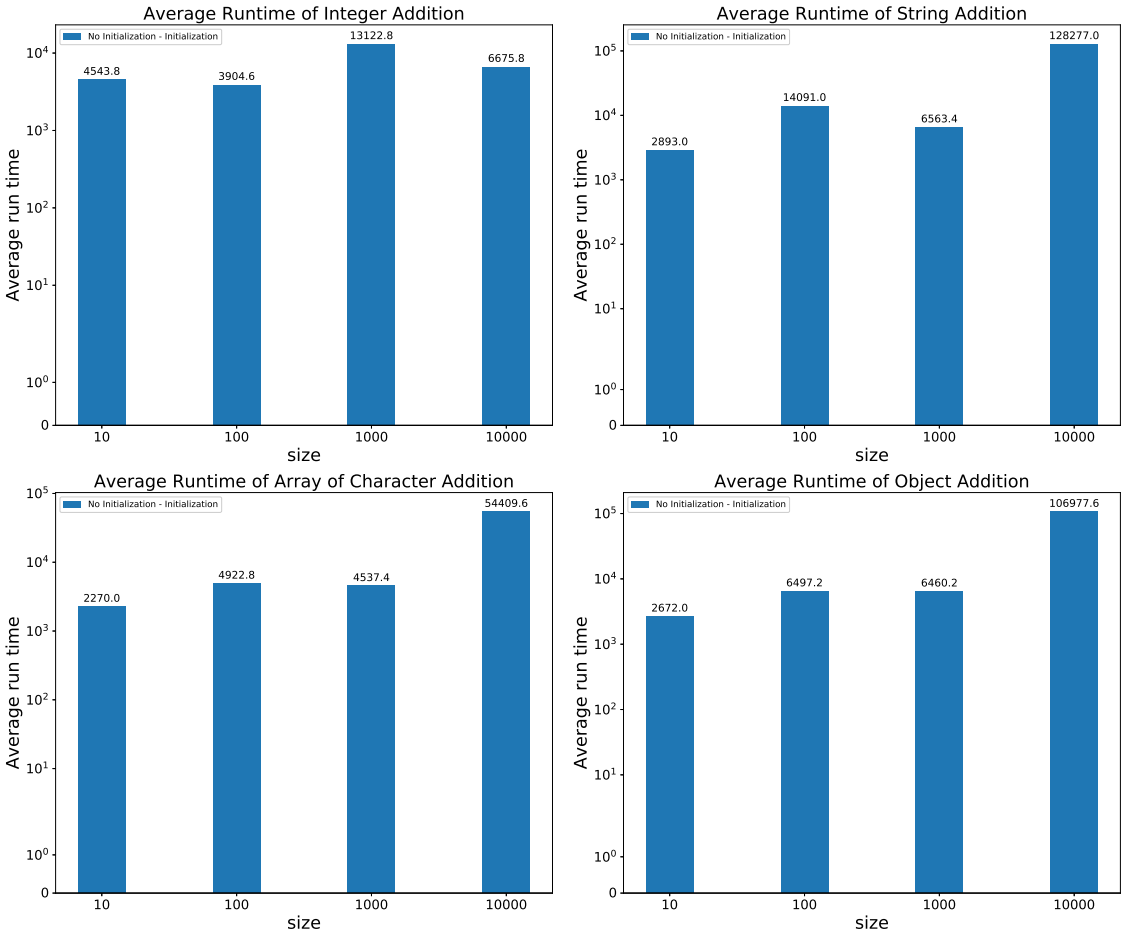

Figure 2·11: Difference of Memory allocation of Rust Vector between Non-initialization and Initialization

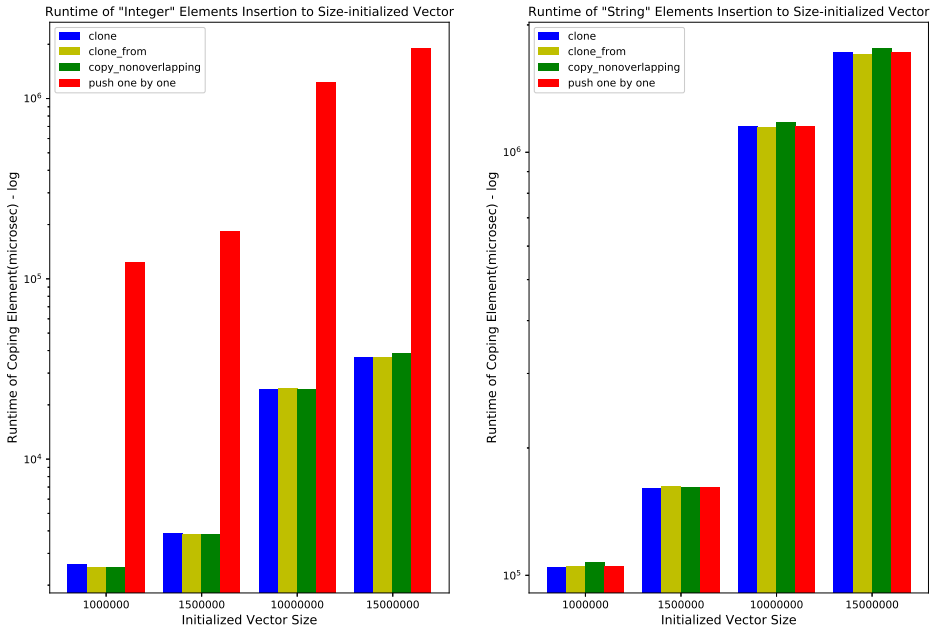

Figure 2·12: Runtime of elements copy from one vector and insertion to the other vector.

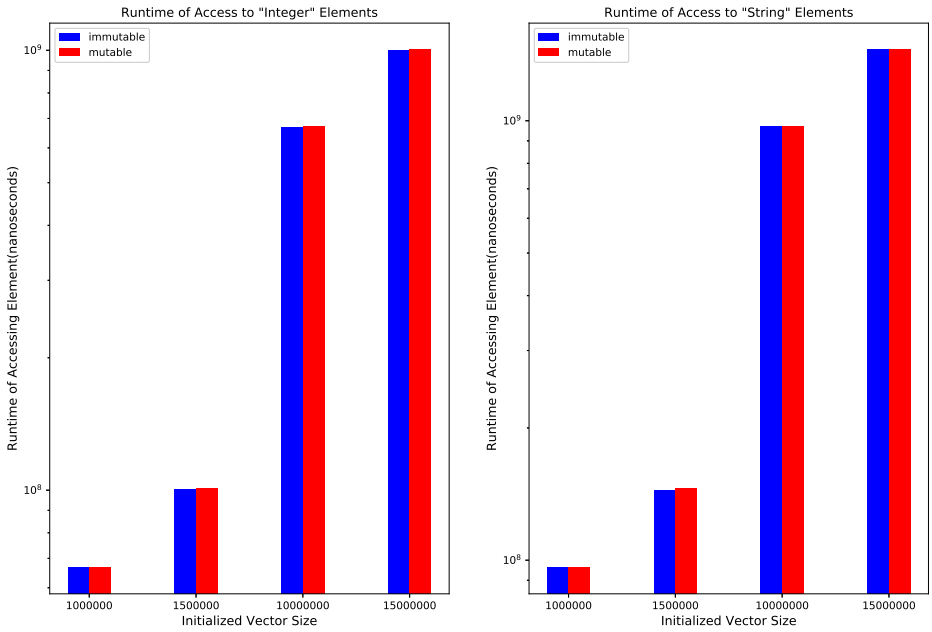

Figure 2·13: Runtime of elements copy from one vector and insertion to the other vector.

```
struct Customer<'a> {
    total_order: i32,
    weight_order: f32,
    zip_code: String,
    address: &'a String,
    country: &'a str
}
```

Figure 2·14: Representation of Customer object in Rust.

Chapter 3

Body of my thesis

3.1 Some results

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

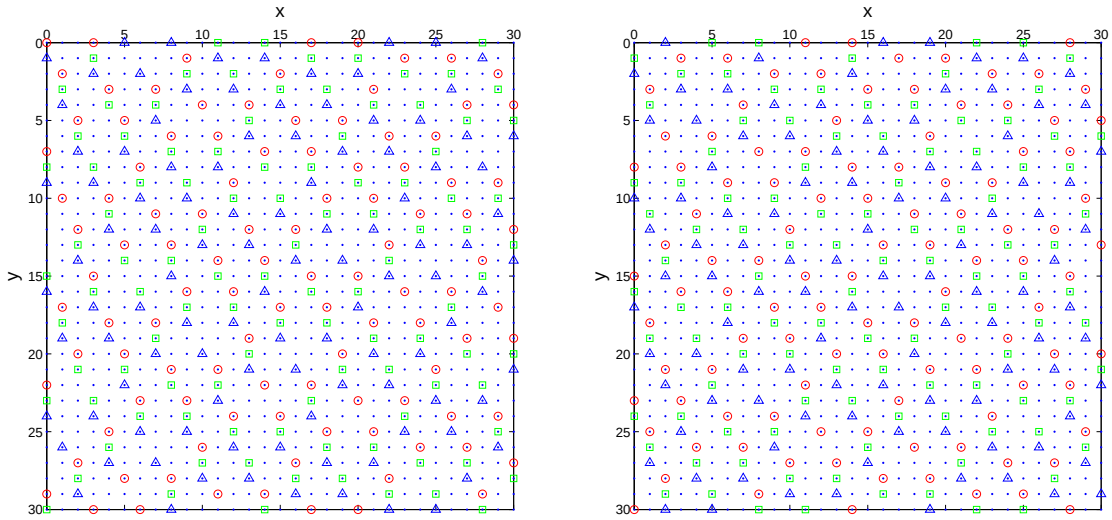

Figure 3·1: Assignment of single-view intensities to RGB components: (a) view #1; and (b) view #2.

You will also be using a lot of citations. Here is the format required in the dissertation: (**?**),(**?**).

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. **??** 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

Of course, there must be a Table of Contents at the beginning of the thesis.

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

This is the appendix.

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.