

Evaluating Memory Models for Graph-Like Data Structures in the Rust Programming Language: Performance and Usability

Utvärdering av Minnesmodeller för Graf-Liknande Datastrukturer i Programmeringsspråket Rust: Användbarhet och Prestanda

Rasmus Viitanen

Supervisor : Rouhollah Mahfouzi
Examiner : Christoph Kessler

External supervisor : Henrik Sjööh

Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: <http://www.ep.liu.se/>.

Abstract

Representing graphs in Rust is a problematic issue, as ownership forbids typical representations found in e.g. C++. A common approach is to use reference counting to represent graphs, but this can easily lead to memory leaks if cycles are present in the graph. As naïve reference counting is not sufficient, we must search for alternative representations. In this thesis, we explore different memory models that allow safe representations of graph-like data structures in Rust. These memory models are later evaluated in terms of performance and usability. We find that region-based allocation is, in most cases, the best model to use when performance is of importance. In cases where usability is more important, either reference-counting with cycle collection or tracing garbage collection is a solid choice. When it comes to multi-threading, we propose a new implementation of a lock-free transactional graph in Rust. To our knowledge, this is the first lock-free graph representation in Rust. The model demonstrates poor scalability, but for certain graph topologies and sizes, it offers performance that exceeds the other graph models.

Acknowledgments

I would like to thank Rouhollah Mahfouzi for supervising my thesis and clearing up any questions of mine. Furthermore, I would like to thank Christoph Kessler for being the examiner. Finally, the Web R&D team at Configura deserves much praise for their inclusion and helpfulness throughout my thesis. Special thanks to Henrik Sjööh for being my external supervisor.

Contents

Abstract	iii
Acknowledgments	iv
Contents	v
List of Figures	vii
List of Tables	ix
1 Introduction	1
1.1 Motivation	1
1.2 Aim	2
1.3 Research questions	2
1.4 Delimitations	2
1.5 Industry Context	2
2 Background	3
2.1 The Rust Programming Language	3
2.2 Performance	7
2.3 Usability	7
2.4 Graphs	7
2.5 Vertex and Edge Representations	8
2.6 Garbage Collection	9
2.7 Reference Counting	9
2.8 Region-based allocation	10
2.9 Concurrency and Parallelism	10
2.10 Memory Reclamation in Concurrent Systems	12
3 Related Work	14
3.1 Reference Counting	14
3.2 Tracing Garbage Collection	16
3.3 Region-based Memory Management	17
3.4 Memory Management in Concurrent Systems	17
3.5 Graph representations	18
3.6 Benchmarks	18
3.7 Usability	18
3.8 Measurements	19
4 Method	20
4.1 Measuring Performance	20
4.2 Experiment Planning	21
4.3 Execution	31

4.4	Analysis	34
5	Results	39
5.1	GC Bench	39
5.2	The GAP Benchmark Suite	41
5.3	Operations	46
5.4	Usability	47
6	Discussion	48
6.1	Results	48
6.2	Method	51
6.3	Use in Industry	52
6.4	The work in a wider context	52
7	Conclusion	53
	Bibliography	54
A	Glossary	58
A.1	Abbreviations	58

List of Figures

2.1	An impossible representation of a graph in Rust, as both <i>Node 1</i> and <i>Node 3</i> takes ownership of <i>Node 2</i>	6
2.2	A representation of a graph where each edge is stored as a reference. This solves the issue of multiple ownership, but introduces the question as to how the edges can be mutated and which entity owns the actual nodes.	6
2.3	A representation of a graph where a collection takes ownership of the nodes and edges are stored as references.	6
2.4	An example of a cycle leading to unreclaimable memory. When the reference marked with a cross is removed there is no way to reach the cycle by stepping through the references. This means that the cycle can be regarded as garbage and should thus be removed. This is not the case however, as the cycle will keep the reference count from reaching zero.	10
4.1	Layout of the epoch-based graph. The list of edges stores an edge object instead of an index. This means that the edge can hold arbitrary information, such as the weight or the direction of the edge. By storing an atomic pointer to the adjacent vertex instead of an index in the edge object, we avoid traversals in the linked-list when looking up an edge.	24
4.2	Process for measuring performance.	29
5.1	GC Bench for sequential construction. The labels on the horizontal axis represent the depth of the generated trees (Stretch Tree/Long Lived Tree/Min Tree/Max Tree).	40
5.2	GC Bench for parallel and sequential construction. The labels on the horizontal axis represent the depth of the generated trees (Stretch Tree/Long Lived Tree/Min Tree/Max Tree).	41
5.3	Estimated execution times for each kernel in the GAP benchmark suite for the <i>arXiv astro-ph</i> dataset. The estimate is measured as the slope of the regression model. The error bars display the lower and upper bound of the confidence interval.	42
5.4	Estimated execution times for each kernel in the GAP benchmark suite for the <i>Euroroad</i> dataset. The estimate is measured as the slope of the regression model. The error bars display the lower and upper bound of the confidence interval.	43
5.5	Estimated execution times for each kernel in the GAP benchmark suite for the <i>Facebook (NIPS)</i> dataset. The estimate is measured as the slope of the regression model. The error bars display the lower and upper bound of the confidence interval.	44
5.6	Estimated execution times for each kernel in the GAP benchmark suite for the <i>Hamsterster Friendships</i> dataset. The estimate is measured as the slope of the regression model. The error bars display the lower and upper bound of the confidence interval.	45
5.7	Benchmark group: OPS 20/20/25/25/10	46
5.8	Benchmark group: OPS 40/40/10/10/0	47

Listings

2.1	An example of a simple Rust program that summarizes the values in a vector. .	4
2.2	A Violation of Rust's Ownership Rule.	4
2.3	The explicit lifetime parameter ' a ', guarantees that the inner reference is valid for the lifetime of the wrapping Vertex	5
3.1	A reference counted pointer stepping through all the objects in an array. The net result is that only the first and last data objects will have its reference count changed.	15
4.1	Logic for the cursor when inserting a new vertex.	24
4.2	Example of a function that allocates a graph from a file. The input parameter create_node takes a closure that creates a new node from an index number, and connect connects two nodes together.	28
4.3	Example of invoking the load_data function for G::EPOCH	28

List of Tables

4.1	Graph operations of interest.	21
4.2	Memory management model candidates	22
4.3	Graph variations.	26
4.4	Selected datasets	26
4.5	Specification of the machine used for the benchmarks.	27
4.6	Collection Criteria	31
4.7	Sampling Specification	32
4.8	Benchmarking group: arXiv astro-ph	34
4.9	Benchmarking group: Euroroad	35
4.10	Benchmarking group: Facebook (NIPS)	36
4.11	Benchmarking group: Hamsterster Friendships	37
4.12	Benchmarking group: OPS 20/20/25/25/10	38
4.13	Benchmarking group: OPS 40/40/10/10/0	38
5.1	Benchmarking group: OPS 20/20/25/25/10	46
5.2	Benchmarking group: OPS 40/40/10/10/0	46
5.3	Usability for the different graph representations.	47



1 Introduction

This chapter aids in understanding the motivation, aim, specific research questions and delimitations for this thesis. The chapter explains the current state and concerns of the problem, gives a brief explanation as to why this thesis is conducted, and states the expected outcome. Finally, the industry context is presented for this thesis.

1.1 Motivation

Rust is a systems programming language that enables both high-level abstractions and low-level control. The language has a range of use cases, including web development, game development and embedded systems. Rust is sometimes blamed for a high learning curve and a complex design, as it is seemingly different from other traditional programming languages. The design choices that help Rust achieve its safety and performance can oftentimes hinder a programmer from doing things that other languages otherwise allow. For example, Rust's ownership rules that guarantee memory safety, may force a developer into highly complex representations that result in usability hazard. One example of this, is when representing data structures such as graphs. In languages such as C++ or Python, this is most of the time a trivial problem. In C++ we can, for example, use pointers to represent edges in the graph. This is typically not the case for Rust, because Rust forbids multiple ownership of values, while also forbidding mutable references to be aliased. With such constraints it becomes extensively hard to represent arbitrary graphs - especially those that include cycles. Instead, one must search for alternative representations.

Today, several attempts have been made to come up with suitable memory models for graphs and similar structures. One solution, is to introduce raw pointers through `unsafe` Rust and represent the graphs similar to how it would be done in C++. Using `unsafe` Rust however, breaks some of the guarantees that come with using Rust in the first place, which is why `unsafe` Rust most of the time is recommended to be avoided. The alternative is to use non-trivial solutions, such as reference counting, garbage collection or through region-based allocators. The different methods each have their implications. Reference counting, for example, can cause memory leaks and is also prone to run-time overhead – as borrow-checking can not be done during compilation. Other solutions are likely to not offer the same usability.

As Rust continues to grow, it is fair to believe that many will lean towards Rust when developing applications with performance and safety in mind. Meanwhile, graph-like data

structures are extensively used in software solutions. This makes it important to understand which memory models for graph-like data structures can be used and under which circumstances.

1.2 Aim

In this thesis, we explore different memory models that can be used to represent graph-like data structures in the Rust programming language, while still complying with Rust's ownership model. Specifically, we explore the different memory models available for mutable graph-like data representations and quantitatively measure the performance of each model. The representations are expected to differ in terms of usability, hence we aim to evaluate the usability of each model and map it against the measured performance.

1.3 Research questions

1. Which memory models are suitable for arbitrary graph-like representations in Rust?
2. What is the time behavior performance difference for the identified memory models?
3. How do the memory models differ in terms of operability and user error protection?

1.4 Delimitations

While it would be possible to make changes to the Rust language itself, it is not an aim for this thesis. This thesis explores memory models that can be used with currently available Rust features, including those available in the Rust nightly toolchain. The execution setting can either be single-threaded or multi-threaded. Further, the memory models explored are limited to garbage collection, reference counting and region-based allocation. The graph representation itself is not restricted and can use any layout (for example an adjacency list).

1.5 Industry Context

This thesis is conducted on behalf of the company *Configura*. Configura develops 3D software for space planning purposes and has identified Rust as a great fit for developing high performance web applications. The 3D models used by Configura are represented by a graph-like data structure, and loading these into Rust is a troublesome problem. Hence, Configura is interested in finding suitable ways for representing arbitrary graph-like structures in Rust. The representation is preferred to demonstrate high performance, while also being easy to work with in order to minimize development cost.



2 Background

This chapter includes relevant information that is needed in order to understand this thesis. First, a brief introduction is given to the Rust programming language in order to understand why it is inherently hard to represent graphs in Rust. Then, we present general information about graphs and memory reclamation schemes. As this thesis is not limited to single-threaded environments, we also give an introduction to concurrent types and memory reclamation in concurrent systems.

2.1 The Rust Programming Language

Rust is a programming language with many application areas. It is considered to be a systems programming language and is striving for safety and performance. The language is typed, compiled and has properties of both object-oriented and functional programming languages. The Rust ecosystem consists of a multitude of working groups, ranging from game development to embedded and all the way to the web. Even though Rust is similar to other languages, it has a couple of constraints and design choices that can be confusing even for experienced developers. For example, the language is not as object oriented as Java, as it does not allow inheritance, which invalidates many of the popular object-oriented design patterns. Further, the memory management and reference handling is not as relaxed as C++, making it non-trivial to translate C++ code to Rust.

Even though the design choices can be confusing to programmers, the systems in place are there for a good reason, and are the reason as to why Rust can achieve its safety and good performance. The most prominent design choice is the enforcement of the Resource Acquisition Is Initialization (RAII) pattern, which is made possible through what is known as *ownership*. In Rust, all data objects are said to have an owner, where the owner has full control of data that it owns. Multiple ownership of the same data is typically not allowed, but some workarounds exist, see Section 2.1.4. Rust also keeps track of the scope of all entities, so that they are dropped once they are not needed, thus fulfilling the RAII pattern. Once the owner goes out of scope, so does the owned data.¹ This means that data objects typically do not have to define a dedicated constructor, and the programmer does not have to manually call *free/delete*. For entities that need to do some special cleanup when going out of scope, it

¹<https://doc.rust-lang.org/rust-by-example/scope/raii.html>

is possible to implement the *Drop*² trait. It is also possible to escape some of the constraints in Rust by using an `Unsafe` closure, which allows abilities that are considered to be unsafe, such as dereferencing raw pointers.

Listing 2.1: An example of a simple Rust program that summarizes the values in a vector.

```

1 fn summarize(values: std::vec::Vec<u32>) -> u32 {
2     let mut sum = 0;
3     for value in values {
4         sum += value;
5     }
6     sum
7 }
8
9 fn main() {
10    let sum = summarize(vec![1, 2, 3]);
11    println!("Sum is: {}", sum);
12 }
```

2.1.1 Ownership

The concept of ownership is one of the most important parts of Rust, and enables the language to achieve memory-safety and high performance. The ownership model would be very limiting if it was impossible to occasionally lend out and borrow different values from the true owner. This is why Rust allows *borrowing*. What this means, is that entities other than the owner are able to borrow a value. This can be done either mutably or immutably and is done through a reference. A common problem when lending out values, is that values can be changed or even dropped by the owner. This would typically cause dangling pointers or other similar issues in other languages. Rust is able to eliminate these issues by having two simple rules for how references are handled:

R1) A reference cannot outlive its referent.

R2) A mutable reference cannot be aliased.

An example of violating **R2** can be seen in Listing 2.2. A vector is immutably borrowed on line 4 and mutably borrowed on line 5. Because the scope is being tracked by Rust this is not a problem in itself. The line on row 9 however, extends the scope of the immutable borrow to after the mutable borrow. This violates **R2**.³

Listing 2.2: A Violation of Rust's Ownership Rule.

```

1 fn main() {
2     // This will not compile
3     let mut numbers = vec![1, 2, 3];
4     let borrowed_numbers = &numbers;
5     let mutably_borrowed_numbers = &mut numbers;
6
7     // immutable borrow used after a mutable borrow
8     // yields a compilation error
9     println!("{:?}", borrowed_numbers);
10 }
```

²<https://doc.rust-lang.org/std/ops/trait.Drop.html>

³<https://doc.rust-lang.org/nomicon/ownership.html>

2.1.2 Aliasing

Aliasing is a concept dealing with shared memory. When multiple variables or pointers refer to the very same memory region, they are said to alias. Rust’s ownership rules forbid simultaneous aliasing of mutable references as dictated by the second rule of ownership. Immutable references can be aliased freely.⁴

2.1.3 Lifetimes

The rules of ownership says that *a reference cannot outlive its referent*. To be able to detect violations of this rule, Rust must know if the reference is valid in the current scope. To do so, references are marked and bound to a specific *lifetime*. Sometimes, these lifetimes must be explicitly annotated by using a lifetime parameter. An example of explicit lifetimes for a struct can be seen in Listing 2.3.⁵

Listing 2.3: The explicit lifetime parameter `'a`, guarantees that the inner reference is valid for the lifetime of the wrapping `Vertex`.

```
1 struct Vertex<'a> {
2     inner: &'a SomeType
3 }
```

2.1.4 Breaking Single Ownership

The single ownership rule can be broken through reference counting. Reference counting is available in Rust’s standard library through the struct `std::rc::Rc`⁶. This type of reference counting does not work between multiple threads. To enable reference counting between threads, atomicity is required. For this reason, another reference counting entity exists called `std::sync::Arc`⁷. This type of reference counting results in additional overhead compared to non-atomic reference counting.

A more volatile way of breaking single ownership is to use raw pointers. This can be achieved by using the `unsafe` closure. When using raw pointers, one must however be careful with memory allocation and deallocation, as this method does not handle memory automatically.

2.1.5 Interior Mutability

When mutating a Rust value, the *interior mutability* pattern can be used, which is a pattern that essentially breaks Rust’s pointer aliasing rules. Interior mutability can be achieved through a datatype called `UnsafeCell`, which is also the only way to achieve it in safe Rust⁸. Some amenities exist for working with `UnsafeCell`, these are `std::cell::Cell`⁹ and `std::cell::RefCell`¹⁰. Typically, these are wrapped by some reference counting datatype such as `std::rc::Rc` in order to allow multiple ownership. In the case of using atomic reference counting through `std::sync::Arc`, there are other structs that can be used for interior mutability, for example read-write locks or mutexes. When using the interior mutability pattern, the enclosing entity does not need to be marked as mutable.

As with single ownership, `unsafe` Rust can effectively be used to achieve interior mutability. This is achieved by mutating raw pointers. The approach does not impose the same guarantees as `UnsafeCell` and must be handled with special care in order to avoid memory problems.

⁴<https://doc.rust-lang.org/nomicon/aliasing.html>

⁵<https://doc.rust-lang.org/nomicon/lifetimes.html>

⁶<https://doc.rust-lang.org/std/rc/struct.Rc.html>

⁷<https://doc.rust-lang.org/std/sync/struct.Arc.html>

⁸<https://doc.rust-lang.org/std/cell/struct.UnsafeCell.html>

⁹<https://doc.rust-lang.org/std/cell/struct.Cell.html>

¹⁰<https://doc.rust-lang.org/core/cell/struct.RefCell.html>

2.1.6 Graphs in Rust

One attempt at representing a graph in Rust would be to let each vertex in the graph contain a list of its adjacencies. To do this, an initial idea would be to store the adjacent vertex directly in this list. However, this kind of representation is impossible in cases where a vertex is adjacent to more than one node, as it would require the vertex to be owned by multiple instances, which is not allowed in Rust. This problem is illustrated by Figure 2.1.

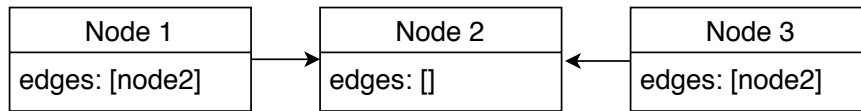


Figure 2.1: An impossible representation of a graph in Rust, as both *Node 1* and *Node 3* takes ownership of *Node 2*.

To circumvent the issue of multiple ownership, which Rust forbids, another attempt would instead be to store references instead of direct objects, which can be seen in Figure 2.2. This solution is also problematic, as it raises questions as to how one would proceed to mutate individual nodes in the graph. Furthermore, the question as to which entity should take ownership of the nodes is also challenging. If we were to store the nodes in some kind of collection, as illustrated by Figure 2.3, we run into problem as to how that graph can be constructed in the first place. First, it would require the nodes to be moved into the node collection. Secondly, in order to form the edge between two nodes, they would have to be accessed via the node collection by a mutable borrow of the first node, and an immutable borrow the second node. Finally, when we want to add the second node to the adjacency list of the first node, we run into problems, as we are currently holding both an immutable and a mutable borrow to the node collection, thus violating the rules of ownership.

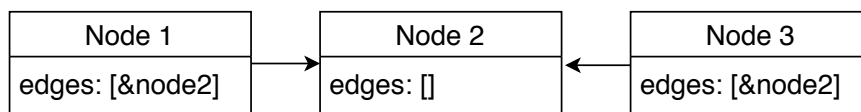


Figure 2.2: A representation of a graph where each edge is stored as a reference. This solves the issue of multiple ownership, but introduces the question as to how the edges can be mutated and which entity owns the actual nodes.

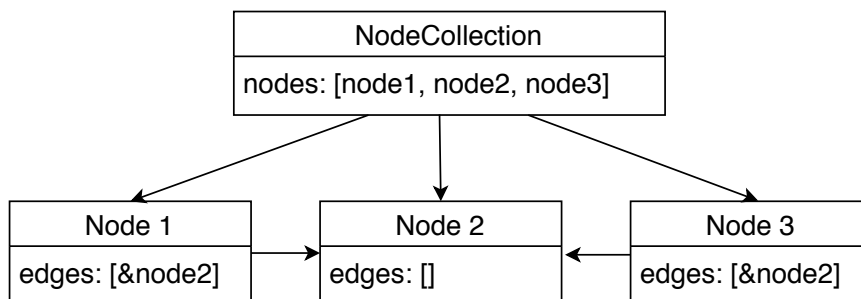


Figure 2.3: A representation of a graph where a collection takes ownership of the nodes and edges are stored as references.

It turns out that representing arbitrary graphs in Rust with traditional references is impossible. Instead, we have to search for alternative approaches that allow e.g. multiple ownership.

2.2 Performance

Performance can mean a lot of different things depending on context. For example, good performance can mean good resource utilization. For this thesis, we use the ISO/IEC 25010:2011 [21] standard to define performance. Specifically, when we talk about performance, we are referring to *time behavior performance*, which is defined by the ISO/IEC 25010:2011 standard as

”Degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements.” [21]

2.3 Usability

When it comes to the usability software quality, the ISO/IEC 25010:2011 [21] standard lists six different parts that are all related to usability. These are: *appropriateness recognizability*, *learnability*, *operability*, *user error protection*, *user interface aesthetics* and *accessibility*. For this thesis, we are particularly interested in the *user error protection* and the *operability* of the graph representations, which is defined as

Operability: ”Degree to which a product or system has attributes that make it easy to operate and control.” [21]

User error protection: ”Degree to which a system protects users against making errors.” [21]

2.4 Graphs

This section explains different properties of graphs and describes the notation used for different properties along with their mathematical definition. For this thesis, we use the same notation as the notation used in the Koblenz Network Collection (KONECT) [23].

2.4.1 Properties and Notation

A graph is composed of a set of vertices $V = \{v_1, v_2, \dots, v_n\}$ and a set of edges E , where an edge is formed by a connection between two vertices $e_{(i,j)} = (v_i, v_j)$. We call this graph $G = (V, E)$. A directed graph is created when the edges in the graph have a direction. Cycles are formed when it is possible to start from a vertex v_i , traverse the edges in the directed graph and end up back at vertex v_i . When programmatically creating a graph, a common representation is to store an adjacency list in each vertex. This adjacency list composes of references to every other vertex that is connected to the vertex. To add an edge to a vertex, the edges are specified during the construction of a vertex, or by mutating an already existing vertex. Another approach is to represent the graph using a container that wraps a list of all vertices and edges. Mutating such graph representation will not require its vertices to be mutable, instead the wrapping graph container must be mutable.

The size of a graph is denoted n and is defined as $n = |V|$, which describes the number of vertices. The volume of a graph is denoted m , and is defined as $m = |E|$ and describes the number of edges in the graph¹¹. Individual vertices have a degree, d , which describes the number of edges connected to that vertex. In directed graphs, the degree of a vertex is instead described by both an in-degree and an out-degree. The in-degree describes the number of incoming edges, and the out-degree describes the number of outgoing edges. The maximum

¹¹The size of a graph in other literature is commonly defined as the number of edges. To avoid confusion with KONECT, we have chosen to use this definition.

degree of a graph describes the degree of the vertex with the largest degree within the graph. This is defined as

$$d_{max} = \max_{u \in V} d(u)$$

The average degree, d_{avg} , is a measurement of the density of the graph. The average degree is defined as

$$d_{avg} = \frac{1}{|V|} \sum_{u \in V} d(u) = \frac{2m}{n}$$

The relative edge distribution entropy, H_{er} , of a graph gives information about the degree distribution of the graph. It is a good measurement to understand how uniformly connected the vertices are. If all vertices have the same amount of edges the relative edge distribution entropy is $H_{er} = 1$. In a star-graph it would assume the value of zero. The relative edge distribution entropy is defined as

$$H_{er} = \frac{1}{\ln(n)} \sum_{u \in V} -\frac{d(u)}{2m} \ln\left(\frac{d(u)}{2m}\right)$$

2.4.2 Graph types

Graphs can be either directed or undirected. In an undirected graph the edge relationship is symmetric, meaning that an edge $e_{(u,v)}$ is equal to $e_{(v,u)}$ and represents the same edge. In directed graphs, an edge $e_{(u,v)}$ is not equal to another edge $e_{(v,u)}$, and the edge $e_{(u,v)}$ can exist without the presence of the edge $e_{(v,u)}$.

2.5 Vertex and Edge Representations

There are multiple ways of representing vertices and edges in a graph. For this, a range of data structures can be used. The most common representations use either an adjacency list or an adjacency matrix. In an adjacency list, the edges adjacent to a vertex are stored in a list containing data that describes these edges in some manner. This way of representing a graph makes it easy to extract information about the edges. This can be useful in e.g. a breadth-first search, as getting the edges of a given vertex only requires a look-up in the adjacency list, which is easy and cheap to iterate over. When using an adjacency matrix, the relationship between vertices is instead represented in a $N \times N$ -matrix, where each element e_{ij} in the matrix describes whether the two vertices v_i and v_j are adjacent.

Representing an adjacency list in a computer program requires some form of underlying data structure. In cases where it is acceptable to trade memory for speed, the adjacency list could simply be represented by a HashMap or a HashSet. This would allow for quick access and insertion of individual edges. In the case where it is not suitable to use a HashMap or HashSet, a vector or a linked list could be suitable.

2.5.1 Bloom filters

A Bloom filter is an unsound data structure that can be used to check if an entry has previously been added to the data structure. A Bloom filter is unsound in the manner that a look-up in the filter can yield false positives, however it cannot yield false negatives. It is possible to adjust the probability of a false positive by changing how much space the Bloom filter uses. This is done by changing the number of bits per element in the Bloom filter. The Bloom filter data structure can be of use when inserting a new vertex into a graph that uses a link-based data structure to store its vertices. If the default behavior is to insert the vertex if it is not in the graph, and update the vertex if the vertex is present – a search through the entire list

would be required in order to guarantee that an insertion is unique. By using a Bloom filter, the time complexity is reduced to amortized $O(1)$ for every unique insertion. In the rare case of receiving a false positive by the Bloom filter, the complexity is $O(n)$. In both cases, the complexity for a non-unique insertion is $O(n)$.

2.6 Garbage Collection

Garbage collection is a technique to automatically handle reclamation of memory. There are many techniques for garbage collection such as tracing and reference counting. When we talk about garbage collection in this thesis, we are referring to *tracing garbage collection*. The general case of a tracing garbage collection builds on two phases 1) *Garbage detection* and 2) *Memory reclamation*. During the first phase, the collector traverses all objects of interest and marks reachable objects. This process is also known as *tracing*. During the second phase, all unmarked objects are reclaimed. In a mark-and-sweep collector this is done by *sweeping*, which further examines the memory for garbage [37].

The cost of executing the operations for garbage detection and memory reclamation can be expensive. For this reason, languages that use manual memory management tend to demonstrate better performance than languages with garbage collectors. Not only are garbage collected systems slower, but they are likely to suffer from *stop-times*, which is a huge problem in real-time systems. Stop-times occur when the regular execution of the program has to halt in favor of the garbage collector to be able to perform its operations. To mitigate this issue and minimize the stop-time, collectors oftentimes run their operations in the background as much as possible.

Even though Rust is not a garbage collected language, it is still possible to create and use a custom garbage collector if needed. Rust enables enough freedom to be able to perform both tracing (garbage detection) and memory reclamation. This is demonstrated by the *rust_gc* project¹², where it is possible to attach a garbage collector to specific data objects. Even though languages with manual memory management tend to be faster, a garbage collector could be desired in order to increase the usability and flexibility of the language.

2.7 Reference Counting

Reference counting is a technique to automatically reclaim space when it is not needed. Reference counting works by keeping track of how many referents a data object has. Whenever a new referent is added the reference count is incremented; when a reference is removed it gets decremented. Once the reference count reaches zero, the object will not have any referents and the memory consumed by the object is reclaimed. A problem with reference counting is that cycles can lead to unreclaimable memory. Figure 2.4 illustrates the problem with cycles in reference counted systems [9].

¹²<https://github.com/Manishearth/rust-gc/>

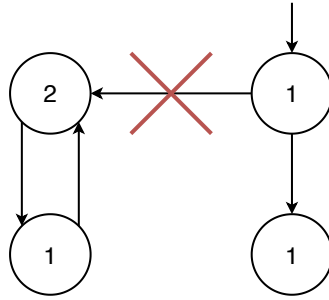


Figure 2.4: An example of a cycle leading to unreclaimable memory. When the reference marked with a cross is removed there is no way to reach the cycle by stepping through the references. This means that the cycle can be regarded as garbage and should thus be removed. This is not the case however, as the cycle will keep the reference count from reaching zero.

2.8 Region-based allocation

Both reference counting and garbage collection causes unwanted overhead and performance concerns. An alternative approach is to handle the memory manually. This can be done by using region-based memory management, which allocates a controllable chunk of memory. This approach is taken in some systems with high real-time requirements [16]. A region is a part of the memory that is allocated and deallocated at once, which stores objects with similar lifetime. By allocating a big region at once instead of single objects, later allocation time and deallocation time of individual objects can be kept short, as adding individual objects to the region does not require new allocations. A big problem with this type of allocation however, is that it can easily lead to memory issues. For example, if we have an active pointer to somewhere in the allocated region and wrongly reclaim that space in the region, we will end up with a dangling pointer. In Rust, we can manually allocate a region in memory by using the allocator provided by the standard library¹³.

2.9 Concurrency and Parallelism

Rust is famous for making it easy to develop concurrent and parallel systems. The language efficiently prevents common issues that arise when running simultaneous tasks through its ownership system. Sometimes, the ownership system is not enough, and it is sometimes desired to use shared ownership. With shared ownership, it can no longer be statically guaranteed that a resource will not be modified by another thread when reading it. To mitigate these issues, there are generally two approaches 1) *Lock-based protection* and 2) *Atomics*.

2.9.1 Lock-based mutual exclusion

One way of preventing issues that arise in concurrent systems is to introduce locks. Locks limit the access to the locked data object such that it can be guaranteed that reads and writes do not interfere in some hazardous way. With the locking mechanism comes the issue of resource contention, where threads will have to wait until they can acquire the lock before access to the data is granted. There are different types of locks that are suitable for different use cases, for example, a read-write lock (available through `std::sync::RwLock`) allows multiple readers or one writer at any given time. A mutex lock guards the data such that it can only be accessed by a singular entity at a time.

¹³<https://doc.rust-lang.org/std/alloc/index.html>

2.9.2 Atomics

When compiling a program, the order of operations can be changed by the compiler in order to achieve a more efficient program. This becomes a problem when running a program on multiple threads, as the order of operations may be important for correctness. For this to behave correctly, a guarantee is needed that asserts that the order of operations on one thread exactly matches the order of operations on another thread. To mitigate this issue it is possible to introduce *Atomics*. Atomics is a concurrency primitive that assures that read and writes of shared data by multiple threads are well-defined behavior, such that each thread is able to finish their operation before the next thread continues.

Atomics make it possible to safely share and edit the same data across multiple threads, as it is possible to guarantee that one instruction takes place at a time. Hence, mitigating the issue of conflicting reads and writes. One interesting property of atomics, other than being safe to share between threads, is that atomic types can be used to implement *lock-free* data structures. A lock-free data structure has the nice property of enabling concurrent read and writes without the possibility of a deadlock, as one thread is always guaranteed to make progress. As lock-free data structures do not need to use locks to synchronize shared resources contention is kept to a minimum. This means that lock-free data structures sometimes demonstrate better performance than systems relying on locks or mutexes.

Several possibilities exist for working with Atomics in Rust. One commonly used datatype is `std::sync::Arc`, which is an atomic reference counter. This type can safely be shared across multiple threads, contrary to the non-atomic reference counter `std::rc::Rc`. While the reference counting for this datatype is lock-free, the wrapped data does not necessarily demonstrate such properties, and it is common to add a lock for the inner type.

For more primitive access to Atomics in Rust, a module named `std::sync::atomic` is available. This module builds on the same atomics used in C++20 and allows different memory orderings to be specified. These memory orderings make it possible to give instructions to the compiler as to how memory is synchronized. For example, one available ordering is `SeqCst`, which stands for sequentially-consistent. This type of ordering ensures that all threads observe the same order of operations. The available memory orderings are (weakest to strongest): `Relaxed`, `Release`, `Acquire`, `AcqRel` and `SeqCst`.

A central operation in the world of atomics is the Compare-And-Swap (CAS) operation. CAS stores a value into the atomic type if the currently stored data matches some data that the performing thread is expecting. The operation takes three arguments, a pointer to the value to be updated, the expected value and the new value.

2.9.3 The ABA Problem

The ABA problem is a problem that occurs in concurrent systems and can lead to devastating issues. The problem is quite easy to understand, but harder to mitigate. Consider the case where two threads are accessing and modifying a value and performs the following sequence of operations:

1. Thread 1 reads the value A
2. Thread 2 modifies the value to B and then back to A
3. Thread 1 reads the value A and continues as normal

The problem here is that thread 1 naïvely assumes that nothing has changed, since the value that was read the first time matches the value that was read the second time. This problem is present in some systems using the CAS operation, but the problem can effectively be eliminated with some extra consideration. One solution is to attach a tag to each value that changes with each modification. With such a solution, the problem is reduced to only

checking that the tag is up to date with the current read value. This check leads to additional overhead, and other solutions are possible that demonstrates better performance [10].

2.10 Memory Reclamation in Concurrent Systems

Memory reclamation works a bit different in concurrent systems, and it is sometimes not feasible to use a traditional memory reclamation mechanism. In a concurrent system using lock-free data structures, additional precautions must be made in order to reassure that memory reclamation does not occur while memory is still in use in the system. The performance loss of using a memory reclamation mechanism that works in a concurrent setting can in some cases outweigh the performance increase of using a lock-free data structure. If the memory reclamation mechanism however is carefully chosen, efficient reclamation can be achieved. Some of the more popular mechanisms for efficient reclamation include: *quiescent-state-based reclamation*, *epoch-based reclamation* and *hazard-pointer-based reclamation* [17].

2.10.1 Lock-free Reference Counting

Lock-free reference counting is similar to traditional reference counting, except that it works in multithreaded and concurrent environments. Every new clone of a reference increments the reference count, and when the reference count reaches zero, the memory is reclaimed. In order to successfully synchronize the reference count between multiple threads, the incrementing and decrementing operations are atomic. In Rust, it is possible to use the `std::sync::Arc` type in order to achieve lock-free reference counting. This type uses a fetch add or subtract instruction in order to atomically increment or decrement the reference count¹⁴.

2.10.2 Quiescent State

Reclamation mechanics that are based on quiescent states reclaim memory by recording which objects were removed in a specific time interval. When the time interval is over, the memory can be reclaimed if it can be proven that no thread is referencing the data. The interval is commonly called a *grace period*.

In order to prove that no thread has active references, the application uses *quiescent states*. This state describes when a thread does not reference any shared data. To enter a quiescent state, one can simply call a function that enters the state when a remove or delete operation is performed. Quiescent-State-Based Reclamation (QSBR) is a blocking reclamation mechanism, meaning that other threads will have to wait for the reclamation process to be completed before execution can continue.

2.10.3 Epoch Reclamation

Epoch-Based Reclamation (EBR) is similar to quiescent state-based reclamation in the sense that the reclamation mechanism also relies on time intervals in order to reclaim memory. This also means that EBR is blocking. The difference between EBR and QSBR is that EBR is more user-friendly, where a developer does not have to manually call a function that enters the quiescent state. Instead, EBR builds on what is known as *critical regions*. A critical region marks that the thread is requesting data from the underlying data structure. Once the requested data is processed and completed the thread exits the critical region.

Threads using EBR operate in what is called *epochs*. The *global epoch* can assume three different epoch states *A*, *B* and *C*. When a thread enters a critical section, it updates its thread-local epoch state such that it matches the global epoch. If a removal or deletion occurs inside a critical section, the object gets stored in a list of garbage for the current epoch. This

¹⁴<https://doc.rust-lang.org/src/alloc/sync.rs.html>

way, the object is only logically deleted and can only be physically deleted once it is safe to do so. Safe physical deletions, and thus memory reclamation, occurs when all threads are in a state that matches the global epoch, which can be checked whenever. If all thread states match with the global epoch, data that was logically deleted two epochs earlier can safely be deleted. Once the physical deletion is done, the global epoch can proceed to the next state. The epoch state is looping, meaning that if the global epoch is in state C , it will proceed to state A .

2.10.4 Hazard Pointers

Hazard pointers is a way to achieve non-blocking memory reclamation and thus eliminate long stop-times. Hazard-Pointer-Based Reclamation (HPBR) works by letting each thread have a set of hazard pointers that marks data that is currently in use by the thread. By marking the data, other threads are instructed that it is not allowed to modify or delete that data until it has been unmarked by the thread owning the hazard pointer. To delete a node, an object is placed into a list of objects to be deleted. This way, the object is logically deleted, but physical deletion is deferred until it is safe to delete the object. Physical deletion occurs when the delete-list reaches a certain size, where the deletion is done by scanning the list and removing the objects that have no hazard pointer currently marking the object. If a hazard pointer is still marking the data, it is once again added to the delete-list. Hazard pointers are multi-reader and single-writer pointers and successfully prevent the ABA problem [28].



3 Related Work

This chapter presents related work regarding different memory models and graph representations. Very little scientific work has been done specifically for Rust, but we expect properties of the models to be independent of language. Further, as Rust is a systems programming language supporting both high- and low-level control, we expect an implementation of the concepts mentioned in this chapter to be possible in Rust. This assumption is strengthened by the different implementations that already exist today.

3.1 Reference Counting

Reference counting was introduced in 1960 by Collins [9] as an idea that multiple data occurrences in list representations do not have to be stored in multiple places in the computer. In its introduction, it is said that two methods have been highlighted in previous literature to solve this issue. One of those methods is the method used in Rust, where data has a single owner and where data can be borrowed by others. The method proposed by Collins [9] for solving this issue is to instead have shared ownership that builds on reference counting. Reference counting, even though proposed in 1960, is still being used today. Two prominent problems of reference counting is the ability to collect cycles and the added run-time overhead. Reference counting is also available in Rust and part of the standard library. Garbage cycles are a known issue in the Rust implementation as well, where references that form a cycle easily lead to memory leaks¹.

3.1.1 Throughput

One problem with reference counting apart from cycle collection is throughput, as creating and destroying references (known as mutating) causes run-time overhead. Several modifications of the reference counting technique presented by Collins [9] back in 1960 have been proposed in order to improve the throughput. These are primarily deferred reference counting, coalescing reference counting, generation based reference counting and ulterior reference counting [3, 14].

Deutsch and Bobrow presented *deferred* reference counting in 1976 [11]. The idea behind deferred reference counting is to avoid some overhead of reference counting by ignoring reference counts for objects that are subject to frequent mutations. This causes higher throughput,

¹<https://doc.rust-lang.org/1.30.0/book/second-edition/ch15-06-reference-cycles.html>

but since the references are deferred, reclamation of memory is not instant. The deferring is done by putting zero-count references in a table, and the memory reclamation occurs once this table is scanned. Using a zero-count table is not the only way of dealing with deferred references [7]. Blackburn and McKinley presented *ulterior* reference counting in 2003 [7]. This mechanic is related to deferred reference counting and works synchronously. It can be seen as a generalization of deferred reference counting and achieves both high throughput and short pause times.

Levanoni and Petrank presented another approach called *coalescing* reference counting in 2007 [32]. This mechanic limits the number of mutations for a reference by only recording the first and last mutations. The idea is that a reference that changes its interior data multiple times in a row, will result in a mutation sequence that is equivalent with only executing the first and last mutation. Suppose that a reference RC refers to an arbitrary data object d_n . We write this $RC(d_n)$. If RC iteratively points to every data object in the set $D = \{d_0, d_1, \dots, d_n\}$, i.e., $RC(d_0) \rightarrow RC(d_1) \rightarrow \dots \rightarrow RC(d_n)$, traditional reference counting executes the algorithm in Listing 3.1. The result is that mutations other than the mutations at the edges have an unchanged reference count. This means that we can ignore the mutations.

Listing 3.1: A reference counted pointer stepping through all the objects in an array. The net result is that only the first and last data objects will have its reference count changed.

```

1 D = [d_0, d_1, ..., d_n]
2 ptr = RC(D[0])
3
4 for new_data in D[1:] {
5     old_data = *ptr
6     *ptr = data[new_data]
7     old_data.rc--
8     new_data.rc++
9 }
```

Shahriyar et al. presented another reference counting system in 2013 that successfully identifies reusable memory and eliminates memory fragmentation [34]. They identify one of the performance issues of previous reference counted systems as poor heap organization and locality.

3.1.2 Cycle reclamation

Apart from throughput limits of reference counting, garbage cycles can cause memory leaks if not properly collected. Bacon and Rajan presented a paper [4] in 2001 that revisits the reference counting method presented by [9]. They propose a way of conducting cycle collection using different garbage collection techniques that collect cycles that would otherwise cause memory leaks. Apart from issues with cycles of garbage, the authors also list two other problems with reference counting, namely: 1) *added storage overhead due to keeping track of the reference count for each object*; 2) *run-time overhead due to changing the reference count for each added or deleted reference*. To solve the issue of cyclic garbage, the authors make a proposal to use a cycle collector that works concurrently and traces locally. They also present a non-concurrent cycle collector that works in a similar fashion. The *concurrent* cycle collector presented in the paper, is the first concurrent cycle collector to ever be presented.

The cycle collector presented by Bacon and Rajan [4] was tested for eight different benchmarks, each consisting of a different number of cycles and size. The largest benchmark allocated 19,2 million objects. The measurements show that the amount of tracing needed by the collector varies heavily based on the layout of the data collected. The evaluation and comparison between the cycle collector and other methods is described in greater detail in another article by Bacon et al. [2]. During the measurements, the cycle collector had a maximum pause time of 6 milliseconds. Comparing this number to a parallel mark-and-sweep

collector, the max pause time was `563 milliseconds`. The implementation of the cycle collector was written in the Java programming language. The short pause times that the cycle collector demonstrates are likely an effect of the loose synchronization between the objects creating and destroying references and the collector, where the normal case does not require any synchronization at all [4]. An open-source Rust implementation of the non-concurrent version of the cycle collector presented by Bacon and Rajan [4] is *bacon_rajan_cc*². Another cycle collector inspired by the collector by Bacon and Rajan [4] is presented by Paz et al. [32]. This cycle collector combines the collector in [4] with what is known as a sliding-view collector. The result of combining these, along with some other design choices, is a cycle collector that achieves very short pause times (less than `2 milliseconds`).

A more recent proposal to handling cyclic garbage in reference counted systems was presented by Frampton et al. [14] in 2009. This collector is based on mark-and-sweep collection and has similarities to the cycle collector proposed by Bacon and Rajan [4]. The collection consists of three phases *1) Root*; *2) Mark*; *3) Sweep*. In addition to the phases, a queue is held in memory by the collector that tracks all unvisited objects. First off, the algorithm executes the *root* phase, where all objects that live outside the collected space *and* are referenced by rooted objects in the collected spaces are added to the unvisited queue. Secondly, the collector enters the *mark* phase, in which it processes the unvisited queue and checks if there are any marked objects. In the case of an unmarked object, it will be marked and all the objects it is referring to will also be added to the queue. After all objects in the queue have been processed; the algorithm checks that objects that potentially cause so-called collector-mutator races are indeed added to the queue. If they are not, they will be added in this step and the *mark* phase is restarted. Finally, objects that remain unmarked are freed. A notable mention for this mark-and-sweep-collector, is that it makes optimizations during the *mark* phase. This is done by checking if the objects are acyclic, which in their case consists of checking if any of the member fields is a pointer or if it points to another acyclic object. A lot of other optimizations are also done in order to speed up the collector, such as optimizing the *sweep* phase by only sweeping cyclic objects and objects they point to.

3.2 Tracing Garbage Collection

McCarthy [27] introduced garbage collection the same year as reference counting was introduced [9]. The general case of garbage collection is known to suffer from poor performance, which has led to many attempts in finding new techniques for high-performance garbage collection. Wilson [37] presented many techniques for synchronous garbage collection and explained the differences between the different mechanics. In this article, it is mentioned that systems requiring high performance most commonly benefit from tracing garbage collectors over reference counting collectors. Hybrids of mark-and-sweep collectors and copy collectors are often used to achieve maximum performance.

Bacon et al. [3] showed that high performance collectors are hybrids of both reference counting and garbage collection. This was found when they developed two collectors, one based on reference counting and one based on garbage collection. When optimizing both of them for performance, they started to see some similarities between the two. This led them to formulate a unified theory of garbage collection that claims that garbage collection and reference counting are duals. They also strengthen the claim that tracing garbage collectors achieve higher throughput than collectors based on reference counting, however tracing garbage collectors are prone to higher pause times. It should also be mentioned that pause times in tracing garbage collection is rather unpredictable.

Blackburn and McKinley [6] proposed a mark-region garbage collector called *Immix* that promises good performance and space efficiency. A mark-region collector is a type of non-moving collector that allocates and reclaims larger regions in memory instead of single fields.

²<https://github.com/fitzgen/bacon-rajan-cc>

The Immix collector is able to perform better than other available collectors, such as mark-and-sweep collectors.

Lin et al. [25] explored the possibility of implementing an Immix based garbage collector in the Rust programming language. They found that it was possible to implement a collector with very little fuzz by using already existing features and implementations of e.g. task queues. They identify that Rust is a good fit for implementing high-performance garbage collectors. They compared the performance of the implementation to a similar garbage collector in C, where their proof-of-concept demonstrated similar performance. The resulting collector composes of 220 LOC, excluding external code. To measure performance, they relied on micro benchmarking of three different operations: 1) *allocation*, 2) *object marking* and 3) *object tracing*. They measured the time taken by each operation for 50 million objects of 24 bytes each, and repeated this process twenty times. The comparison used the *gcbench* micro benchmark. *gcbench* is briefly mentioned by [18] as well.

3.3 Region-based Memory Management

An alternative to reference counting and tracing garbage collection is to handle the memory manually. One way of handling this, is by allocating large controllable regions in memory that we can later store our data in. Hamza and Counsell [16] explored the state-of-the-art of region-based memory management in Java. More specifically, the Real-Time Specification for Java (RTSJ)³ based allocation strategy called *scoped allocation*, which is a form of region-based memory allocation. They identify region-based memory management as a mechanic that demonstrates higher performance than both garbage collection and reference counting. One highlighted problem that exists in Java regarding region-based memory management, is the limitation of not knowing the lifetime of the objects that are to be allocated. This is an issue, because we can allocate a long-living region for short-lived objects. This will lead to occupied memory that is unused. In Rust, we expect this to be a non-issue, as lifetimes are explicit and the allocated region is expected to have the same lifetime as our graph. Hamza and Counsell also highlight that the region-based memory model can lead to usability troubles for developers, as it often requires more logic for handling the memory as well as making sure that the references are safe. Scoped memory as specified by RTSJ is reclaimed when all allocated objects are dropped. Hence, deallocation time is cheap, as the deallocation only occurs once every object is dropped. Allocation time is proportional to the size of the allocated object as long as there is enough space left in the allocated region. In the rare case of the region having too little space left, allocation time is increased, as the region must be extended [19].

3.4 Memory Management in Concurrent Systems

Memory reclamation in concurrent systems requires additional care to make sure that memory is not reclaimed when the memory is still referenced by another thread. Hart et al. [17] compared the performance of three popular reclamation mechanisms, namely: QSBR, EBR and HPBR. One property of lock-free data structures is that they can improve performance, but with languages with manual memory management, the mechanism for memory reclamation can in some cases outweigh the performance gains of the concurrent datatype. For this reason, it is important to pick the best suited mechanism in order to minimize the performance penalty.

Each mechanism has different properties that makes it suitable for different use cases. A common mechanism is Lock-Free Reference Counting (LFRC), which is available through `std::sync::Arc` in Rust. LFRC often has the issue of poor performance, and adds significant overhead that oftentimes outweighs the performance gain of the lock-free data structure [17]. The performance issues of LFRC is a side effect of relying on per-element atomic instructions [17]. Another mechanism mentioned by Hart et al. was presented by Michael [29] called

³<https://www.rtsj.org/>

Hazard Pointers. Hazard pointers generally demonstrate better performance than LFRC [17, 29]. The memory reclamation approach also demonstrates comparable performance to lock-based reclamation strategies even in single-threaded environments. EBR is another mechanism that demonstrates competitive performance to both LFRC and HPBR [17, 29]. A description of the reclamation strategies can be found in Section 2.10.

3.5 Graph representations

Graphs can be represented in memory in a range of different ways. One example is to use an adjacency list. When using an adjacency list in a multi-threaded context, it is required to use some mechanic that locks the underlying data when it is accessed or modified. This is commonly done by using a *mutex* or a *read-write* lock. Some attempts have been made to reduce the overhead of the locking mechanism. Linked lists can, for example, be implemented in a lock-free manner, but implementing a lock-free adjacency list is a problem that is much more complex. Painter, Peterson and Dechev [30] proposed a way of representing lock-free transactional adjacency lists. This representation is believed to be the first correct lock-free transactional adjacency list to be presented. The adjacency list builds on two important concepts: a multi-dimensional list and lock-free transactional transformation theory. The multi-dimensional list is used to represent the adjacent objects, and the lock-free transactional theory enables transactional operations to be performed on the list. The result is an adjacency list that does not need to acquire locks on vertices when performing operations, thus yielding better performance than previous lock-based implementations. This claim should however be carefully considered, as the representation requires vertices to be stored in a link-based data structure. This limitation can be problematic for operations requiring fast look-up.

3.6 Benchmarks

The previous work when it comes to benchmarks for graph-like data structures is limited. There are, however, established ways to benchmark e.g., garbage collectors. For example, Lin et al. [25] used a benchmark called *GC Bench* to measure the performance of their garbage collector. GC bench has been used by other researchers as well, e.g. Boehm [8]. The benchmark works by allocating and deallocating binary trees of various sizes.

Another benchmark that focuses on graph processing operations is called *the GAP benchmark suite* [5]. The suite consists of six different kernels that include functions commonly used in graph processing. Namely, the kernels are Betweenness Centrality (BC), Breadth-First Search (BFS), Connected Components (CC), Page Rank (PR), Single-Source Shortest Path (SSSP) and Triangle Counting (TC). In addition to the kernels, the suite provides a set of large input graphs that are selected to be diverse, as well as a reference implementation of the suite written in C++.

3.7 Usability

Tamir et al. [36] mentions that usability testing is a tedious task, where it is not always applicable to use objective metrics collected through software inspections. Instead, it is necessary to use human test subjects. For example, operability could be measured by specifying a goal, and allow human test subjects to achieve that goal by operating the system under test. The effort to achieve that goal would then be a good measure as to how easy or difficult the system is to operate. The required effort could be measured in a number of ways. For example, it is possible to measure the number of keystrokes and mouse clicks or the level of eye movement it took to achieve the goal.

User error protection is seemingly easier to measure than operability. For example, the probability of user errors are highly correlated with the coupling between objects and the lines

of code in a system, among many other software quality metrics [31]. When it comes to API usability, Zibran et al. [38] explored common factors for usability related bugs. They found that many of the issues is partly related to the quality of the documentation, the number of methods exposed and incorrect memory management. Rama and Kak [33] explored metrics for API usability, where they identified nine different pain-points in API design that can be quantified. Some of these include methods with long parameter lists, multiple methods with similar names, not specifying thread-safe methods and poor documentation.

3.8 Measurements

Kitchenham et al. [22] explain guidelines for empirical research in software engineering. One important guideline that is given in the article, is that experiments should have a well-defined data collection procedure. Another issue that is raised, is the issue of bias. In the experiment, it is possible that the analysis will be unfair due to bias. Hence, the experiment should be conducted such that it is unknown which graph model and representation resulted in which measurement until they have been properly analyzed. It is also important to collect the raw data that are produced during the experiment, such that it can be reviewed by the readers. Finally, the case of determining statistical significance and practical significance is important, where data should be collected such that it allows for significance to be determined. For example, the collected dataset must be of sufficient size in order to prove statistical significance, and practical significance claims can be supported by measuring confidence limits.

While not much work has been done regarding performance measurements in Rust specifically, guidelines for performance measurements in Java is stated by Horký et al. [20]. In the paper, it is mentioned that warmup is an important part of performance measurements. Warmup is the idea of letting the program run for a while, in order to warmup the processor caches and other parts of the program that might differ based on how long the program has run for. It is also stated that one should be aware of other optimizations made by the compiler. The paper gives an example of not using the results of a computation, which therefore might be optimized away by the compiler. This issue can be mitigated in Rust by using the *test*-crate available in Rust nightly⁴.

⁴<https://doc.rust-lang.org/nightly/unstable-book/library-features/test.html>



4 Method

This chapter explains the used method for this thesis. First, we present a motivation of the chosen general method. Then, we describe in detail the different phases of this thesis, namely: *Experiment Planning*, *Execution*, and *Analysis*. These descriptions include the work process, motivation and intent at finer granularity.

4.1 Measuring Performance

Benchmarking is one of the most important parts of this thesis, and it is important that we understand how measurements should be done in order to achieve reliable results. When representing graphs, the underlying memory model and the graph representation model is what is expected to have an effect on the performance and usability. When dealing with different memory models, we have identified reference counting, garbage collection and region-based allocation as suitable memory management models. For lock-free datatypes, HPBR, QSBR, and EBR are identified as suitable memory reclamation schemes.

Several studies have compared performance of reference counting and tracing garbage collectors [2, 7, 14]. Identifying the performance is typically done by measuring both pause time duration and throughput. When running benchmarks, it is important to consider the topology of the graph. For example, graphs with many or long cycles are not expected to show the same results as a tree-graph. Bacon et al. [2] ran benchmarks for a range of different graph topologies. For example, they created one graph that only produced cyclic garbage using a gaussian distribution in order to fully stress their cycle collector. For a fair result, the size, number of cycles and number of references per vertex should thus be varied, as these factors are expected to affect the performance [4].

Lin et al. [25] relied on micro benchmarking when measuring performance. They measured the time taken for three different operations: 1) *allocation*, 2) *object marking*, and 3) *object tracing* for their garbage collector implementation. They also measured their collector by running the benchmark **GC Bench**. When conducting such benchmarks, it is important to consider side effects that occur due to the compiler doing optimizations [20]. To circumvent this issue, it is possible to use a tool for benchmarking that makes sure that the compiler does not produce a program that does less work than we intend.

Comparing performance of garbage collectors and manual memory management such as region-based allocation is generally not a problem in languages that are designed for manual

memory management. In this case, it is possible to just replace the memory management with a garbage collector, and measure the performance of each method without making any other precautions [18]. This is the case in Rust, where it is possible to attach a garbage collector to specific data objects.

In this thesis, we take inspiration by the work of Lin et al. [25] and use the micro benchmarking approach to measure individual operations, and **GC Bench** in order to measure allocation and deallocation. Additionally, we take inspiration by Bacon et al. [2], and run several benchmarks with different graph topologies via the **GAP benchmark suite**.

4.2 Experiment Planning

This section describes the planning phase of the experiment. It includes the overall design of the experiment; selection of graphs; selection of memory models; instrument for measurements; data collection; and analysis procedure.

4.2.1 Design

Micro benchmarking has shown to be effective for measuring performance [25]. Using micro benchmarking, different operations can easily be measured independently [25]. For this thesis, the main interests for measuring performance include the graph operations listed in Table 4.1. The operations are all suitable for micro benchmarking, where the same algorithm can easily be implemented and used for each graph representation, meaning that the underlying memory model will be the only entity affecting the measured performance. Regarding the operations, the graph allocation performance for the graphs is important to determine upfront time, and the other operations determine during regular execution once the graph is loaded into memory.

As there are a multitude of ways for traversing graphs, the GAP benchmark suite has been chosen in order to measure traversals. The suite includes common graph processing kernels and is diverse enough to enable a comprehensive analysis. Allocation is not included in the GAP benchmark suite, which is why we must search for alternative benchmarks. We deemed GC bench a good candidate for benchmarking allocation and reclamation, as it has been used in previous research by e.g., Lin et al. [25].

Table 4.1: Graph operations of interest.

Operation
Allocation
Node insertion
Node deletion
Edge insertion
Edge deletion
Find
Traversal

The graph operations are not the only interest for this thesis. Scalability and other characteristics are important in order to enable a complete evaluation. This is because the graph representations should work with different sized graphs, and it is important to consider other cases than the time to perform a single operation. The maximum stop-time is also important to measure, as it can lead to the whole program halting for a duration that is unacceptable. To motivate the intent behind measuring stop-time, we consider the case where a user is interacting with a 3D object, if the maximum pause-time is a whole second and it occurs as the user is rotating the object, it will lead to the application freezing for the duration of the stop-time, leading to poor user experience.

4.2.2 Memory Models and Memory Model Selection

The subjects for this experiment are the different memory models and graph representations that are suitable for use in the Rust programming language. The selection for what is suitable in Rust is based on a pre-study that exhaustively covers the current state-of-the-art, both in academia and already existing Rust implementations. To cover the state-of-the-art, a search for academic research was conducted. The search included queries for region-based allocation, reference counting, garbage collection, and concurrent memory reclamation. This resulted in the work presented in Chapter 3. Further, a search on the popular code hosting sites *GitHub* and *GitLab* was conducted in order to find already existing Rust implementations. Using this approach, several implementations were found, however not all of them were production ready and not always up to par with the academic research that was found during the state-of-the-art search. Due to the immaturity of the Rust ecosystem, the number of available implementations were few in numbers, and it is believed that all significant open-source implementations were found during our search. Additionally, we reached out to members of the Rust language team in order to get further insights to current memory models and related language features for tackling the issue of representing graphs in Rust. They provided no additional information that had not already been found during our search.

In order to decide whether to proceed with the implementations found online, a set of selection criteria was formulated. The first selection criterion considers the graph operations, where it must be possible to add, delete and update individual objects in the graph. This criterion ruled out some of the region-based allocators, such as the popular *rust-type-arena*, as it would not be able to deallocate individual objects¹. The second criterion considers the maturity of the implementations and ruled out some individual projects that were not mature enough to be used for a graph implementation. Finally, any implementation that were far off the state-of-the-art were not considered. Some projects such as `rust_gc` were still selected – even though it might be possible to implement a faster garbage collector. The reason for selecting this project, is that it was close enough to the state-of-the-art presented in academia.

After the search for implementations were completed, it was noted that some concepts found in the academic research had not been implemented at all in Rust. For this reason, a lock-free transactional adjacency list was implemented that used EBR as the memory reclamation scheme. By implementing this, at least one model for each memory management type were selected, i.e., reference counting, garbage collection, concurrent reclamation and region-based allocation. An overview of the memory management models selected for further evaluation can be seen in Table 4.2.

Table 4.2: Memory management model candidates

Name	Memory Management	Multi-threading support
<i>G :: RC</i>	Reference Counting	NO
<i>G :: ARC</i>	Atomic Reference Counting	YES
<i>G :: CC</i>	Reference Counting with Cycle Collection	NO
<i>G :: MSGC</i>	Mark and Sweep-based Garbage Collection	NO
<i>G :: IMMIXGC</i>	Immix-based Garbage Collection	YES
<i>G :: EPOCH</i>	Epoch-based Reclamation	YES
<i>G :: ARENA</i>	Region-based Memory Management	NO

¹<https://github.com/SimonSapin/rust-typed-arena>

4.2.2.1 G::RC

Reference counting is a common reclamation scheme, and is extensively used in the Rust programming language. For this reason, a memory model based on reference counting was chosen for further evaluation. For our graph representation, the reference-counting pointers consists of `std::rc::Rc`, where interior mutability of nodes is enabled via `std::cell::RefCell`. To represent edges in the graph, the implementation uses a BTreeMap and strong reference counts.

4.2.2.2 G::ARC

For a multi-threaded approach to reference counting, a memory model based on atomic reference counting using `std::sync::Arc` was chosen for further evaluation. While being similar to $G :: RC$, the performance in a single threaded environment is expected to be worse for this reference-counting scheme, as the atomic operations yield additional overhead. To represent edges in the graph, the implementation uses a BTreeMap and strong reference counts. To achieve interior mutability, a read-write lock (`std::sync::RwLock`) is used.

4.2.2.3 G::CYCLE

As reference counted systems have trouble handling cycles, a third model based on reference counting was chosen for further evaluation. This memory model handles cycles by performing cycle collection. The cycle collection is based on the work by Bacon and Rajan [4]. The used project is *bacon_rajan_cc*².

4.2.2.4 G::MSGC

A garbage collected memory model based on *mark-and-sweep* memory collection was chosen, as garbage collection is expected to be more user-friendly and able to scale better than reference counting. This garbage collector is based on the project *rust_gc*³, which is a mark-and-sweep collector.

4.2.2.5 G::IMMIXGC

A second garbage collected memory model was chosen, as *mark-and-sweep* is not the only way to perform garbage collection. Immix collectors have shown to demonstrate better performance than some *mark-and-sweep* collectors, which renders immix collection an interesting candidate. This collector is based on the work by Lin et al. [25]. Unfortunately, we observed false reclamation of active objects. For this reason, the collector is not part of the final results.

4.2.2.6 G::EPOCH

Research suggests that it is possible to implement lock-free transactional adjacency lists[30]. Such lists can effectively be used for a graph representation that should perform better than its lock-based relative. To successfully implement a lock-free transactional graph in Rust, a memory reclamation scheme that works in concurrent settings is needed. EBR, HPBR, and QSBR are seen as good candidates for memory reclamation in concurrent systems. As the performance difference is quite small between the schemes and EBR being more user-friendly, EBR was chosen as the underlying memory reclamation scheme for our lock-free transactional graph. The epoch based reclamation scheme is based on the project *crossbeam-epoch*⁴, and the graph representation is based on the research by Painter et al. [30].

²<https://github.com/fitzgen/bacon-rajan-cc>

³<https://github.com/Manishearth/rust-gc/>

⁴<https://github.com/crossbeam-rs/crossbeam>

Some modifications were necessary for our graph representation in order to achieve a true graph-like data structure with as good performance as possible. The work presented by Painter et al. [30] suggests that operations for inserting, deleting and finding edges or vertices in the adjacency list primarily builds on indices. As the primary data structure for the adjacency list is a linked list, with entries ordered by their id, it becomes expensive to locate a specific entry in the list and get e.g. linking information that is necessary for inserting new entries. This is especially true when the list grows, as the worst-case complexity is $O(n)$. In the case where we are executing a breadth-first search of the graph, we would need to first lookup some start node in the graph, get all its adjacencies and then use their id to locate their corresponding vertex in the linked list. This would of course be very expensive. In order to circumvent lookup of adjacencies in the linked list, we instead chose to store a data object for each of our vertices' adjacencies. This data object includes a shared atomic pointer to the entry in the linked list. We also made it possible to store arbitrary data in this object, which enables things such as weighted or parallel edges. An illustration of the layout can be seen in Figure 4.1.

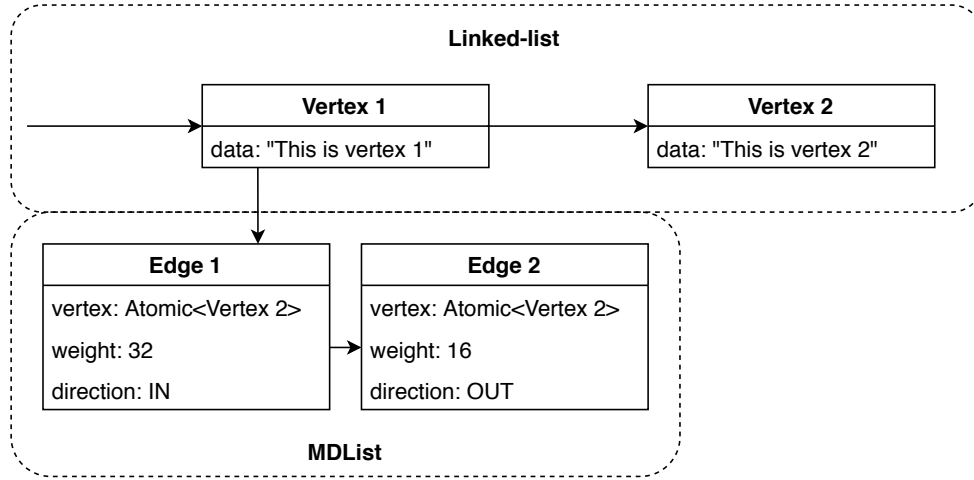


Figure 4.1: Layout of the epoch-based graph. The list of edges stores an edge object instead of an index. This means that the edge can hold arbitrary information, such as the weight or the direction of the edge. By storing an atomic pointer to the adjacent vertex instead of an index in the edge object, we avoid traversals in the linked-list when looking up an edge.

In addition to our changes to the edge representations, a new operation was added called `G::EPOCH::connect`. The reason for adding this operation to the graph is that the operation `InsertEdge` presented by Painter et al. required two look-ups in the linked list. One look-up for finding the node we are adding an edge to; one look-up for finding the edge node in the linked list. To avoid these two look-ups, which both have the worst-case time complexity of $O(n)$, `G::EPOCH::connect` takes an entry from the adjacency list as an input parameter, along with one argument for an edge representation. This way, the complexity of adding an edge, where we have direct access to both the parent and the child, is reduced to $O(\log(n))$ complexity, which is required for finding the entry in the multi-dimensional array used to store the edge.

Our third alteration was to introduce a cursor in the linked-list, in order to be able to achieve constant complexity for vertex insertion. The cursor is simply a pointer to the end of the linked list. As the linked list is ordered by id, any insertion of a node with a higher id than any of the entries in the list will be reduced to only linking it to whatever the cursor is pointing to and then updating the cursor. An issue arises when a node is inserted with an id that is less than the node that the cursor is pointing to. If this is the case, a search is done from the root of the list until the correct position is found.

Listing 4.1: Logic for the cursor when inserting a new vertex.

```

1 fn insert(vertex, root, cursor) {
2     if vertex.id >= cursor.id {
3         cursor.next = vertex;
4         cursor = cursor.next;
5     } else {
6         let current = root;
7         while vertex.id < current.id {
8             if current.next == NULL {
9                 break;
10            }
11            current = current.next;
12        }
13        current.next = vertex;
14    }
15 }

```

In the case where the distribution of key id's is not sorted, e.g. 0, 5, 3, 7, 2, a way to achieve good performance would be to create a mapping function that maps each node id to the n_{th} insertion number, i.e. $0 \rightarrow 0, 5 \rightarrow 1, 3 \rightarrow 2, 7 \rightarrow 3, 2 \rightarrow 4$. This would guarantee that any insertion with any arbitrary key would have constant insertion time. However, as the mapping function needs to store the insertions in e.g. a B-Tree, the insertion complexity is in reality $O(\log(n))$. To keep track of the n_{th} insertion, an atomic counter could be used to be able to sync the count between threads with minimal overhead.

As keeping track of the inserted id's is memory expensive, an alternative approach using a lock-free Bloom filter was also implemented. This implementation discarded the sorted linked-list and instead used an unsorted linked-list, but still required unique entries. Any unique insertion where the Bloom filter successfully returns that a vertex is not present in the list is then reduced to $O(1)$, as it only requires a look-up in the Bloom filter, and a push of the vertex to the end of the adjacency list. However, when the Bloom filter returns a false positive, the list has to be searched from the start, resulting in a time complexity of $O(n)$. This also means that an operation to check if a new vertex is already in the graph is also reduced to amortized $O(1)$, as it follows the same behaviour. This is the version of **G::EPOCH** that was selected for evaluation.

4.2.2.7 G::ARENA

Region-based allocators are able to demonstrate high performance, for this reason a region-based allocator was chosen for further evaluation. For this study, we chose an allocator called *generational-arena*⁵ that builds on generations in order to mitigate the ABA problem, see Section 2.9.3. What this means, is that a generation number is stored along with the index to individual objects in the arena, and every mutation increments this generation number. Hence, it is possible to detect changes by simply checking that the generation number stored in the index is the same as the generation number in the arena.

4.2.3 Graphs and Graph Selection

The different memory models are likely to have different performance characteristics depending on which graph is allocated, traversed or modified. For example, in a reference counted system the level of overhead increases with the reference count, hence the memory model should demonstrate quite poor scalability and should thus be properly tested. To properly exhaust the different properties of the memory models, it is important to choose graphs such that we

⁵<https://github.com/fitzgen/generational-arena>

can observe these symptoms. In order to do this, four important properties were identified, see Table 4.3. The number of edges is commonly referred to as the size of a graph and should thus be a good indicator for how scalable the graph representation is. It is however possible to have a graph with a very large number of edges, but few nodes. In this case, only the operations that affect the edges would give meaningful scalability symptoms. For this reason, the average degree of the graph was also an important factor to consider, as well as the degree distribution. Finally, the GAP benchmark suite mentions that some sequential algorithms can experience poor performance if the diameter of the graph is big. Hence, we select graphs with different sized diameters.

Table 4.3: Graph variations.

Property
Number of nodes
Number of edges
Degree and Degree distribution
Diameter

For suitable real-life graphs, different graphs provided by KONECT were selected. In order to observe different characteristics of the graphs and identify for which topologies they are suitable, different graphs were selected such that the set of graphs were as diverse as possible. We were not able to use the suggested graphs provided in the GAP benchmark suite [5], as they were too big for our hardware and models to handle. Instead, we chose smaller graphs with the same characteristics as those presented in GAP. The selected datasets are listed in Table 4.4, along with their properties mentioned in Table 4.3. The triangle count is also listed, in order to give a hint of how much work is done when running the the GAP benchmark suite’s TC-kernel.

Table 4.4: Selected datasets

Dataset	V	E	Avg. Deg	H_{er}	D	TC
Hamsterster friendships [15]	1 858	12 534	13.492	90.8%	14	16 750
Euroroad [12, 35]	1 174	1417	2.4140	98.5%	62	32
Facebook (NIPS) [13, 26]	2 888	2 981	2.0644	70.9%	9	91
arXiv astro-ph [1, 24]	18 771	198 500	21.102	93.1%	14	1 351 441

The `Hamsterster friendships` dataset was chosen for its high average degree, along with a fair diameter. `Euroroad` was chosen due to the big diameter of 62 along with a uniform edge distribution. The `Facebook (NIPS)` dataset was chosen for its degree distribution of about 70%. Finally, the `arXiv astro-ph` dataset was chosen for its big size.

4.2.4 Instrumentation and Setup

To conduct the measurements for this thesis, the micro-benchmarking tool *Criterion*⁶ was used together with the performance profiling tool *Intel® VTune™ Profiler*⁷. The Rust toolchain was *nightly-x86_64-pc-windows-msvc* with *rustc 1.42.0-nightly*. The machine used for the benchmarks had the properties listed in Table 4.5.

⁶<https://github.com/bheisler/criterion.rs>

⁷<https://software.intel.com/en-us/vtune>

Table 4.5: Specification of the machine used for the benchmarks.

CPU	Memory	Disk Drive
Intel® Core™ i7-10510U	16 GB	M.2 PCIe NVMe SSD
1.8 GHz (4.9 GHz turbo)	2666MHz	512 GB
4 Cores (8 Threads)	-	2,2 MB/sec (read)
8 MB Intel® Smart Cache	-	1,4 MB/sec (write)

4.2.4.1 Criterion

Criterion was chosen as the benchmarking library for this experiment. With **Criterion** being a statistics-driven library for micro-benchmarking, we were able to capture a great variety of statistical data. **Criterion** also prevents issues with compiler optimizations. Most of the time, the benchmarking library resolves these issues on its own, but manual use of the function `criterion:black_box` helps us assure that the compiler does not replace our benchmarking function with some pre-evaluated constant.

4.2.4.2 Intel® VTune™ Profiler

As the graph representations are not always trivially implemented, Intel® VTune™ Profiler⁸ was used for performance bottleneck identification and microarchitecture exploration. The tool helps to identify how well the cache is utilized in the processor, as well as measuring how long each function call in the program takes to execute. For example, in our Rust program, Intel® VTune™ Profiler will mark the lines in our source code with how long they took to execute. By using Intel® VTune™ Profiler, it was possible to reason about the specifics of each implementation, and explore the possibility that some result is the effect of the implementation itself and not the underlying memory model. For example, a graph representation that uses a multi-dimensional array could demonstrate very poor performance if the implementation of finding the n_{th} -root of a given dimension is not properly optimized. With a system to specifically identify such bottlenecks, it becomes considerably easier to reason about possible improvements to the implementation itself.

In addition to poorly designed algorithms and choice of data structures, it is possible that the microarchitecture utilization is poor. For example, the application could suffer from memory stalls, branch misprediction or use of floating-point operations. This is especially true for `G::EPOCH`, as it relies on atomic operations, and depending on the specified memory ordering for each of these instructions, the performance of the application can be drastically improved. It is also possible that the implementation of `G::EPOCH` would be subject to false sharing, as updating an atomic value invalidates its corresponding cache line. The presence of this issue could have a considerable impact on the demonstrated performance.

4.2.4.3 Code Uniformity

In order to allow each graph representation to execute a similar set of operations, a set of helper functions were created that take closures as input arguments. These closures include logic for executing operations that are not equal for all representations. For example, when loading a graph into memory from a file, all operations are equal except for the constructor for creating a node and the connect function used to connect two nodes together. Instead, it is possible to create a helper function that injects these functions. An example of this is given in Listing 4.2.

⁸<https://software.intel.com/en-us/vtune>

Listing 4.2: Example of a function that allocates a graph from a file. The input parameter `create_node` takes a closure that creates a new node from an index number, and `connect` connects two nodes together.

```

1 pub fn load_data<A, C, T: Clone>(
2     file: &str,
3     mut create_node: A,
4     mut connect: C,
5 ) -> HashMap<u64, T>
6 where
7     A: FnMut(u64) -> T,
8     C: FnMut(&T, &T),
9 {
10     /* Implementation Elided */
11 }

```

Invoking this function, it can be asserted that `load_data` will execute the same code for each graph representation, with the exception for node construction and the connection of any two nodes. This means that any differences in performance should be the product of either the node construction function or the connect function. An example of invoking `load_data` can be seen in Listing 4.3.

Listing 4.3: Example of invoking the `load_data` function for `G::EPOCH`.

```

1 let epoch = epochgraph::EpochGraph::new(16, 1 << 16);
2 load_data(
3     dataset,
4     |id| epoch.add_vertex(id, Some(id)),
5     |node1, node2| epoch.connect(*node1, *node2),
6 );

```

4.2.5 Data Collection Procedure

To capture performance data from the different graph representations, a benchmarking system was set up. The system consists of a composer that generates benchmarks for each of the operations listed in Table 4.1. In order to create these benchmarks, the composer requires necessary helper functions in order to generate benchmarks that fulfill code uniformity for each of the graph representations. After creating a new benchmark, the benchmark composer adds each benchmark to a benchmark group. A benchmark group includes one benchmark for one operation for each of the graph representations.

When the benchmark group is created, the benchmark executor runs it in a single session. Some benchmarks, such as the allocation of a graph, requires external data for the graphs, which is provided as an input argument to the executor. With the help of `Criterion`, the executor makes sure that there are no significant outliers during the measurements, and produces a report that includes information such as throughput, execution time, and confidence interval.

An overview of the data collection procedure can be seen in Figure 4.2.

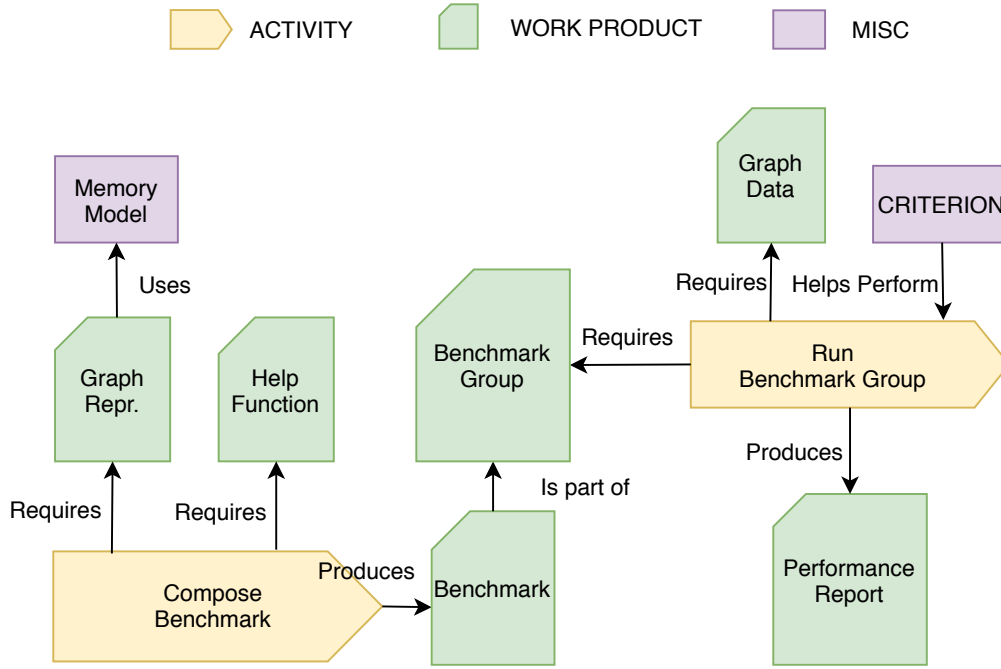


Figure 4.2: Process for measuring performance.

The activity that actually performs the benchmarks is called *Run benchmark group* and is composed of four different activities, namely:

1. Warmup
2. Measurement
3. Analysis
4. Comparison

Of these four activities, only warmup and the measurement activity are regarded as part of the data collection procedure. Analysis and comparison is part of the analysis procedure, and is mentioned under the analysis procedure subsection, see Section 4.2.6. During warmup, the system executes the benchmark without capturing any data about the system under test. This step is important, as starting the measurements immediately could result in false performance indications. After the warmup is completed, the system continues to perform the measurements on the system under test. The measurements are performed in samples and each sample runs the benchmark for a set number of iterations. The number of iterations per benchmark is dynamic and is determined by how long each sample takes to execute. The goal is to perform as many samples as possible in a given time interval. For example, if the target number of samples is ten with a target time of ten seconds, the number of iterations per sample would be 10, given that each iteration takes 0.1 seconds.

It is reasonable to believe that some benchmarks can falsely demonstrate poor performance due to other work being scheduled by the operating system and must thus be detected. In order to mitigate this issue, one solution would be to run the benchmarks in some virtual machine and closely monitor the resources used by the VM. Another solution is to simply store the result of each run, run the benchmarking process multiple times in a random order and then use the information from previous runs in order to detect eventual performance regressions or improvements. If all runs show no indication on performance differences, it is reasonable to conclude that eventual noise is negligible. However, with this approach it is only possible

to reason about how the different graph representation relate to each other. This means that we cannot draw any conclusion of confidence regarding e.g., how many operations per second each graph representation is able to perform, but it is possible to confidently deduce that some graph representation performs better than some other representation.

For GC Bench, the size of the graphs became too big and the executions too long for the benchmarking system to handle. To be able to measure GC Bench, we instead executed the benchmark one sample at a time, with one iteration per sample, until the observed performance fulfilled the noise threshold of $\pm 2\%$. For example, if the first execution took $100ms$, and the second execution $103ms$, the performance increase would be $+3\%$, which required the benchmark to be restarted.

4.2.6 Analysis Procedure

Performance measurements on a computer are likely to demonstrate different results depending on a large set of uncontrollable factors. Some examples of these factors include the temperature of the processor and resource contention as a side effect of the operating system scheduling other tasks in parallel to our measurements. In order to mitigate these issues and be able to analyze the results even though there is some variation in our measured data, we can rely on different mathematical approaches that makes it possible to assert e.g., confidence. In order to properly be able to analyze and interpret the results, plots, and graphs is often a good starting point. In order to reason about the difference in our measurements, we can use what is known as a *violin plot*. *Violin plots* are excellent for visualizing statistical dispersion and probability denseness of multiple measurements. This way, it becomes easier to see trends in the data as to how confident our measurements are. To be able to catch outlier samples, other plots can be used such as the *scatter plot*. *Scatter plots* are suitable when representing individual samples, and can easily visualize outlier samples.

Other measurements other than plots are suitable for giving a concrete value of the expected performance for each graph representation. For this, there are a multitude of possible statistic measures. One common statistic is arithmetic *mean*, denoted \bar{x} .

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i \quad (4.1)$$

This statistic is suitable for finding a value that indicates how long an execution would take in the average case. However, it is not an ideal measurement when outlier values are present. In order to detect when the mean is not reliable, it was necessary to detect eventual outliers in the collected data. This was done by matching the data to a normal distribution. Any values that lied outside of the 25th or the 75th percentile were classified as severe outliers.

In addition to outliers, eventual noise in the benchmark environment can lead to unreliable data. In order to mitigate this issue, it is suitable to use a linear regression model. For our experiments, we used bootstrap resampling to generate a confidence interval from the collected samples and Ordinary least squares (OLS) as the regression model. This way, it was possible to use the regression model to calculate a good estimate by using the slope of the model, hence reducing the effects of noise and outliers.

It is possible that the OLS regression model was poorly fitted. Hence, the coefficient of determination R^2 was also calculated. Finally, in cases where we run out of memory and have to extend the size of our data structure, such as arena allocation, we expect to observe large outliers. This means that we cannot rely on standard deviation. Instead, we used a measure of *median absolute deviation* to reason about dispersion. Other graph representations than region-based allocation were also likely to yield severe outliers, which is why this measurement was taken for all graph representations.

4.2.7 Evaluation of Validity

The environment can heavily impact the results of our measurements. This could be very problematic, as the validity of the measurements can not be guaranteed. In order to detect problems with the environment, the benchmarking system was setup to run each benchmark for more than one iteration and in a random order for each iteration. This enabled the opportunity to formulate a hypothesis stating that the performance difference between each run is not statistically significant and should not change. With this hypothesis in place, ensuring that the environment is not tampering with the collected data is reduced to simply testing this hypothesis. In order to do so, we relied on common hypothesis testing methods, such as calculating *p-values* and performing *T tests*. In order for the result of a benchmark to be accepted, it was required to fulfill the hypothesis, or be within an accepted noise threshold. The set of fulfillment criteria is listed in Table 4.6. If any of the criteria were not fulfilled, the benchmarking process was restarted.

Table 4.6: Collection Criteria

Collection criteria
Severe outliers $\leq 20\%$
Noise tolerance $\leq 2\%$
$p \geq 0.05$

In addition of making sure that the environment does not affect the results, it was also necessary to investigate that each graph representation was correctly implemented. This was done by setting up a test suite that assured that each graph representation performed the same amount of work. One of the performed tests was to let each representation load a graph from a file, then traverse the graph using a breadth-first search, record the order of traversal and finally check that the order was correct. For the reclamation process in epoch-based reclamation and garbage collection to be triggered, we made sure that the loaded graphs were of sufficient size.

4.3 Execution

This section describes how the graphs were measured in the benchmark system created during the experiment planning phase.

4.3.1 Data Collection and Sampling

Data collection must be of sufficient size to be able to confidently prove performance differences. Meanwhile, the data generation must be carried out such that the integrity of the data is assured. In order to maximize the integrity of the collected data, the collection procedure and data generation procedure must be well defined.

To be able to determine differences between runs and assert that the memory models were benchmarked under the same circumstances, all graph variations were measured during the same benchmark execution. This execution ran each individual benchmark three times to detect any differences in the environment.

4.3.1.1 Criterion

For criterion, a warmup period of three seconds was decided in order to fill up caches. Target sampling time was set to five seconds or 55 iterations, depending on which is fastest. The confidence level was set to 95% and the confidence interval was determined through bootstrap resampling of 100 000 samples. Further, the number of target samples were set to 10 samples per benchmark.

Table 4.7: Sampling Specification

Warmup Time	Measurement Time	Sample Size
3s	5s or 55 iterations	10

4.3.1.2 Intel® VTune™ Profiler

In order to detect performance bottlenecks, Intel® VTune™ Profiler was setup with hardware event-based sampling with a CPU sampling interval of 0.01 ms. Each operation was executed 1 000 000 times for each single run in order to enable sufficient microarchitecture exploration and hotspot detection. The short CPU sampling interval causes high overhead for the measurements, but is needed for achieving a detailed description of the performance and microarchitecture utilization.

4.3.2 Data Collection Performed

For our experiment, we gathered data from two different benchmarks that have been used in previous research, namely GC Bench and the GAP benchmark suite. Additionally, we measured two different compositions of operations, in order to be able to reason about single operations. The source code for our benchmarks and graph representations can be found on GitHub⁹.

The GAP Benchmark Suite The GAP benchmark suite [5] includes six different kernels and aims to measure operations that are commonly used in graph processing. As the suite’s reference implementation is implemented in C++, it was necessary to first port it to Rust. We were able to create sequential implementations for every kernel, but due to language constraints only CC, PR and TC have both sequential and parallel implementations. Details about the kernels can be seen below, for further information, see the benchmark specification [5].

- **Betweenness Centrality**
Using Brandes Algorithm
- **Breadth-first Search**
Using Direction optimizing
- **Connected Components**
Using the Afforest and Shiloach-Vishkin method
- **PR**
Using an iterative method in pull direction
- **SSSP**
Delta Stepping
- **TC**
Using an order invariant method with possible relabeling

GC Bench The GAP benchmark suite does not include benchmarks for building new graphs. In order to fill this gap, GC Bench was chosen, as it measures the allocation and deallocation time of binary trees. First, a large tree is allocated in order to stretch the memory space, then a long lived tree is constructed that lives for the duration of the benchmark. After the long lived tree, the benchmark starts to recursively construct trees with increasing depths. The minimum and maximum depth can be specified before starting the benchmark. For our execution, we decided on a minimum depth of four and a maximum depth ranging from eight to fourteen with a step size of two.

⁹<https://github.com/rasviitanen/graphs-in-rust>

OPS Benchmark The OPS benchmark consist of two different distributions of operations that aims to benchmark write-heavy workloads. *OPS/40/40/10/10/0* executes 40% insert vertex operations, 40% delete vertex operations, 10% insert edge operations and 10% delete edge operations. *OPS/20/20/25/25/10* executes 20% insert vertex operations, 20% delete vertex operations, 25% insert edge operations, 25% delete edge operations and 10% find operations. The different compositions of operations are selected to reflect the benchmarks used by Painter et al. [30].

4.4 Analysis

The collected results were analyzed and tested for changes in the benchmark environment. Below are the performance changes output by the last run of our benchmarking system.

Table 4.8: Benchmarking group: arXiv astro-ph

Name	Change	p - value	Verdict
<i>BC/ARC</i>	-7.8211% - 1.6185% + 5.0639%	$p = 0.66 > 0.05$	No change
<i>BC/RC</i>	-14.799% - 2.3639% + 6.8209%	$p = 0.73 > 0.05$	No change
<i>BC/CC</i>	-0.6285% + 7.4832% + 21.061%	$p = 0.19 > 0.05$	No change
<i>BC/GC</i>	-7.0517% + 0.2008% + 7.5359%	$p = 0.96 > 0.05$	No change
<i>BC/ARENA</i>	-3.9848% + 2.3596% + 9.3906%	$p = 0.50 > 0.05$	No change
<i>BC/EPOCH</i>	-5.6092% + 1.2805% + 10.260%	$p = 0.77 > 0.05$	No change
<i>BFS/ARC</i>	-2.5493% + 9.6515% + 28.845%	$p = 0.23 > 0.05$	No change
<i>BFS/RC</i>	-6.1146% - 0.8646% + 4.0420%	$p = 0.77 > 0.05$	No change
<i>BFS/CC</i>	-18.020% - 8.9333% - 0.9701%	$p = 0.08 > 0.05$	No change
<i>BFS/GC</i>	+0.0265% + 4.3413% + 9.5048%	$p = 0.12 > 0.05$	No change
<i>BFS/ARENA</i>	-2.8230% + 1.5261% + 5.4385%	$p = 0.51 > 0.05$	No change
<i>BFS/EPOCH</i>	-4.3985% + 2.8184% + 9.8920%	$p = 0.46 > 0.05$	No change
<i>CC/ARC</i>	-1.2456% + 0.5803% + 2.3926%	$p = 0.59 > 0.05$	No change
<i>CC/ARC_{mt}</i>	-4.8783% + 0.4671% + 5.9473%	$p = 0.87 > 0.05$	No change
<i>CC/RC</i>	-0.1189% + 0.4613% + 1.1406%	$p = 0.18 > 0.05$	No change
<i>CC/CC</i>	-1.0900% - 0.2433% + 0.4759%	$p = 0.59 > 0.05$	No change
<i>CC/GC</i>	-0.9688% - 0.2744% + 0.4308%	$p = 0.47 > 0.05$	No change
<i>CC/ARENA</i>	+0.1113% + 1.0956% + 2.5047%	$p = 0.09 > 0.05$	No change
<i>CC/EPOCH</i>	-1.8111% + 0.2483% + 2.1724%	$p = 0.83 > 0.05$	No change
<i>CC/EPOCH_{mt}</i>	-7.3797% - 2.5574% + 2.6822%	$p = 0.37 > 0.05$	No change
<i>PR/ARC</i>	-2.1918% - 0.8174% + 0.3271%	$p = 0.28 > 0.05$	No change
<i>PR/ARC_{mt}</i>	-3.8365% - 0.1324% + 3.5638%	$p = 0.95 > 0.05$	No change
<i>PR/RC</i>	-1.9799% - 1.1894% - 0.4838%	$p = 0.01 < 0.05$	Within threshold
<i>PR/CC</i>	-1.9752% - 1.0643% - 0.0403%	$p = 0.05 > 0.05$	No change
<i>PR/GC</i>	-0.3309% + 0.1453% + 0.6337%	$p = 0.58 > 0.05$	No change
<i>PR/ARENA</i>	+0.2573% + 0.7796% + 1.2908%	$p = 0.01 < 0.05$	Within threshold
<i>PR/EPOCH</i>	-3.8273% - 1.7636% + 0.1189%	$p = 0.13 > 0.05$	No change
<i>PR/EPOCH_{mt}</i>	-5.6315% - 1.5771% + 2.6071%	$p = 0.50 > 0.05$	No change
<i>SSSP/ARC</i>	-1.9688% + 2.2557% + 6.1101%	$p = 0.31 > 0.05$	No change
<i>SSSP/RC</i>	-1.4045% + 2.4491% + 7.4410%	$p = 0.32 > 0.05$	No change
<i>SSSP/CC</i>	+0.9453% + 4.1085% + 7.2647%	$p = 0.02 < 0.05$	Within threshold
<i>SSSP/GC</i>	-2.3784% + 1.7388% + 5.8610%	$p = 0.47 > 0.05$	No change
<i>SSSP/ARENA</i>	-6.8498% - 1.6118% + 3.4080%	$p = 0.57 > 0.05$	No change
<i>SSSP/EPOCH</i>	-13.096% - 6.1596% + 0.7650%	$p = 0.14 > 0.05$	No change
<i>TC/ARC</i>	-0.4654% - 0.0580% + 0.3558%	$p = 0.79 > 0.05$	No change
<i>TC/ARC_{mt}</i>	-1.7277% - 0.1127% + 1.3936%	$p = 0.90 > 0.05$	No change
<i>TC/RC</i>	-1.4055% - 1.1368% - 0.8287%	$p = 0.00 < 0.05$	Within threshold
<i>TC/CC</i>	-2.0522% - 1.2022% - 0.4472%	$p = 0.01 < 0.05$	Within threshold
<i>TC/GC</i>	-0.0438% + 0.7293% + 2.0937%	$p = 0.26 > 0.05$	No change
<i>TC/ARENA</i>	-0.7763% - 0.1597% + 0.4091%	$p = 0.62 > 0.05$	No change
<i>TC/EPOCH</i>	-0.7732% - 0.3744% - 0.0250%	$p = 0.07 > 0.05$	No change
<i>TC/EPOCH_{mt}</i>	-4.7742% - 0.1823% + 4.8792%	$p = 0.94 > 0.05$	No change

For the benchmarking group *arXiv astro-ph*, the change in performance of the benchmark can be seen in Table 4.8. The first column describes the name for the kernel, and the memory management model that was used. The second column displays the performance change, where the middle value is the difference estimate; the left value is the lower bound of the confidence

interval; the right value is the upper bound of the confidence interval. The last two columns display the p-value, and a change verdict. In case of the p-value being greater or equal to 0.05, the verdict is *No change*, as it fulfills our hypothesis. Otherwise, as in the case for *PR/RC*, the change is within the noise tolerance, thus receiving the *Within threshold* verdict.

Table 4.9: Benchmarking group: Euroroad

Name	Change	p -value	Verdict
<i>BC/ARC</i>	-3.2027% - 1.5173% + 0.0865%	$p = 0.10 > 0.05$	No change
<i>BC/RC</i>	+0.5683% + 1.5993% + 2.7041%	$p = 0.01 < 0.05$	Within threshold
<i>BC/CC</i>	-1.1165% - 0.0552% + 1.1165%	$p = 0.93 > 0.05$	No change
<i>BC/GC</i>	-2.7510% - 0.9775% + 0.6994%	$p = 0.31 > 0.05$	No change
<i>BC/ARENA</i>	-1.4142% + 0.0848% + 1.6952%	$p = 0.93 > 0.05$	No change
<i>BC/EPOCH</i>	-3.0272% - 1.7560% - 0.4496%	$p = 0.02 < 0.05$	Within threshold
<i>BFS/ARC</i>	-9.4859% + 0.5610% + 12.342%	$p = 0.92 > 0.05$	No change
<i>BFS/RC</i>	-4.6463% + 1.2432% + 8.1824%	$p = 0.72 > 0.05$	No change
<i>BFS/CC</i>	-4.9417% + 3.2749% + 11.824%	$p = 0.47 > 0.05$	No change
<i>BFS/GC</i>	-7.6616% - 1.4194% + 6.8492%	$p = 0.73 > 0.05$	No change
<i>BFS/ARENA</i>	-6.6288% + 2.4324% + 11.867%	$p = 0.63 > 0.05$	No change
<i>BFS/EPOCH</i>	-4.5765% + 3.1361% + 11.927%	$p = 0.48 > 0.05$	No change
<i>CC/ARC</i>	-0.0798% + 0.7548% + 1.4296%	$p = 0.08 > 0.05$	No change
<i>CC/ARC_{mt}</i>	-1.3427% - 0.6472% + 0.1013%	$p = 0.11 > 0.05$	No change
<i>CC/RC</i>	-1.5294% - 0.6678% + 0.2607%	$p = 0.18 > 0.05$	No change
<i>CC/CC</i>	-0.6111% + 0.2790% + 1.1154%	$p = 0.56 > 0.05$	No change
<i>CC/GC</i>	-0.5858% - 0.2656% + 0.0192%	$p = 0.13 > 0.05$	No change
<i>CC/ARENA</i>	-0.7900% - 0.1519% + 0.4987%	$p = 0.66 > 0.05$	No change
<i>CC/EPOCH</i>	-0.4746% + 0.3654% + 1.2690%	$p = 0.44 > 0.05$	No change
<i>CC/EPOCH_{mt}</i>	-1.7481% - 0.2944% + 1.2574%	$p = 0.72 > 0.05$	No change
<i>PR/ARC</i>	-0.4865% + 0.7962% + 1.9351%	$p = 0.24 > 0.05$	No change
<i>PR/ARC_{mt}</i>	-0.3829% + 1.1654% + 2.5578%	$p = 0.15 > 0.05$	No change
<i>PR/RC</i>	-0.1857% + 0.3791% + 1.0091%	$p = 0.25 > 0.05$	No change
<i>PR/CC</i>	-1.9451% - 1.4016% - 0.9325%	$p = 0.00 < 0.05$	Within threshold
<i>PR/GC</i>	-0.3352% + 0.1794% + 0.7580%	$p = 0.54 > 0.05$	No change
<i>PR/ARENA</i>	-1.3291% - 0.4121% + 0.4921%	$p = 0.41 > 0.05$	No change
<i>PR/EPOCH</i>	+0.6831% + 1.7368% + 2.8635%	$p = 0.01 < 0.05$	Within threshold
<i>PR/EPOCH_{mt}</i>	+0.0824% + 1.3839% + 2.6793%	$p = 0.07 > 0.05$	No change
<i>SSSP/ARC</i>	-1.4861% - 0.3304% + 0.7548%	$p = 0.59 > 0.05$	No change
<i>SSSP/RC</i>	-0.9478% + 0.3629% + 1.8396%	$p = 0.66 > 0.05$	No change
<i>SSSP/CC</i>	-0.3680% + 0.7219% + 1.8251%	$p = 0.24 > 0.05$	No change
<i>SSSP/GC</i>	-2.6596% - 0.6162% + 1.0309%	$p = 0.60 > 0.05$	No change
<i>SSSP/ARENA</i>	-2.1842% - 0.9852% + 0.1796%	$p = 0.14 > 0.05$	No change
<i>SSSP/EPOCH</i>	-0.5842% + 0.9840% + 2.6449%	$p = 0.27 > 0.05$	No change
<i>TC/ARC</i>	-0.1430% + 0.6659% + 1.3292%	$p = 0.11 > 0.05$	No change
<i>TC/ARC_{mt}</i>	-1.3734% - 0.1793% + 0.9958%	$p = 0.78 > 0.05$	No change
<i>TC/RC</i>	+0.0974% + 0.4316% + 0.8299%	$p = 0.04 < 0.05$	Within threshold
<i>TC/CC</i>	+0.2020% + 0.5224% + 0.8224%	$p = 0.01 < 0.05$	Within threshold
<i>TC/GC</i>	-0.0523% + 0.2977% + 0.6943%	$p = 0.17 > 0.05$	No change
<i>TC/ARENA</i>	-1.5051% + 0.2672% + 1.9186%	$p = 0.78 > 0.05$	No change
<i>TC/EPOCH</i>	-3.6591% + 3.6398% + 15.687%	$p = 0.68 > 0.05$	No change
<i>TC/EPOCH_{mt}</i>	-0.2481% + 1.1954% + 2.8073%	$p = 0.16 > 0.05$	No change

For the benchmarking group *Euroroad*, the change in performance of the benchmark can be seen in Table 4.9. In cases where the change estimate is greater than the noise threshold, such as *TC/EPOCH* and *BFS/CC*, the p-value is much bigger than 0.05, which indicates that the change was insignificant.

Table 4.10: Benchmarking group: Facebook (NIPS)

Name	Change	p -value	Verdict
<i>BC/ARC</i>	-1.3643% - 0.7195% - 0.0268%	$p = 0.07 > 0.05$	No change
<i>BC/RC</i>	-1.7415% - 1.2031% - 0.5190%	$p = 0.00 < 0.05$	Within threshold
<i>BC/CC</i>	+0.0484% + 0.5529% + 1.0052%	$p = 0.05 < 0.05$	Within threshold
<i>BC/GC</i>	-0.1757% + 0.2004% + 0.5673%	$p = 0.32 > 0.05$	No change
<i>BC/ARENA</i>	+0.7431% + 3.8196% + 7.7448%	$p = 0.03 < 0.05$	Within threshold
<i>BC/EPOCH</i>	-2.4029% - 0.9809% + 0.0794%	$p = 0.17 > 0.05$	No change
<i>BFS/ARC</i>	-2.0379% + 1.8144% + 6.6472%	$p = 0.44 > 0.05$	No change
<i>BFS/RC</i>	-1.8713% + 1.6730% + 5.1996%	$p = 0.43 > 0.05$	No change
<i>BFS/CC</i>	-6.1069% - 1.8932% + 2.3787%	$p = 0.42 > 0.05$	No change
<i>BFS/GC</i>	-4.4773% - 0.8324% + 2.4796%	$p = 0.67 > 0.05$	No change
<i>BFS/ARENA</i>	-7.2876% - 2.6826% + 2.0575%	$p = 0.31 > 0.05$	No change
<i>BFS/EPOCH</i>	-3.3179% - 0.3427% + 2.7397%	$p = 0.84 > 0.05$	No change
<i>CC/ARC</i>	-1.0513% - 0.7003% - 0.3097%	$p = 0.00 < 0.05$	Within threshold
<i>CC/ARC_{mt}</i>	-1.0465% + 0.0091% + 1.0497%	$p = 0.98 > 0.05$	No change
<i>CC/RC</i>	-1.2774% - 0.8447% - 0.4946%	$p = 0.00 < 0.05$	Within threshold
<i>CC/CC</i>	-0.3908% + 0.0179% + 0.4146%	$p = 0.94 > 0.05$	No change
<i>CC/GC</i>	-0.2308% + 0.1433% + 0.6128%	$p = 0.54 > 0.05$	No change
<i>CC/ARENA</i>	-1.2677% - 0.8675% - 0.2527%	$p = 0.00 < 0.05$	Within threshold
<i>CC/EPOCH</i>	-1.9917% - 1.2523% - 0.5431%	$p = 0.01 < 0.05$	Within threshold
<i>CC/EPOCH_{mt}</i>	-1.4109% - 0.3823% + 0.8002%	$p = 0.53 > 0.05$	No change
<i>PR/ARC</i>	+0.1270% + 1.3913% + 2.8005%	$p = 0.06 > 0.05$	No change
<i>PR/ARC_{mt}</i>	-1.4284% - 0.4187% + 0.5973%	$p = 0.44 > 0.05$	No change
<i>PR/RC</i>	-1.2205% - 0.4661% + 0.2205%	$p = 0.26 > 0.05$	No change
<i>PR/CC</i>	-0.6776% - 0.1837% + 0.3816%	$p = 0.55 > 0.05$	No change
<i>PR/GC</i>	-1.3423% - 0.9152% - 0.4289%	$p = 0.00 < 0.05$	Within threshold
<i>PR/ARENA</i>	-0.7646% - 0.1940% + 0.2908%	$p = 0.53 > 0.05$	No change
<i>PR/EPOCH</i>	+0.1539% + 0.8259% + 1.4983%	$p = 0.03 < 0.05$	Within threshold
<i>PR/EPOCH_{mt}</i>	-0.9142% + 0.4433% + 1.7199%	$p = 0.53 > 0.05$	No change
<i>SSSP/ARC</i>	-1.3725% - 0.4428% + 0.3975%	$p = 0.39 > 0.05$	No change
<i>SSSP/RC</i>	+0.6325% + 1.6227% + 2.6263%	$p = 0.01 < 0.05$	Within threshold
<i>SSSP/CC</i>	-0.6050% + 0.6526% + 1.8235%	$p = 0.32 > 0.05$	No change
<i>SSSP/GC</i>	-1.6109% - 0.9018% - 0.1832%	$p = 0.03 < 0.05$	Within threshold
<i>SSSP/ARENA</i>	+0.3860% + 2.0042% + 3.6482%	$p = 0.03 < 0.05$	Within threshold
<i>SSSP/EPOCH</i>	-0.7858% + 0.1207% + 1.0935%	$p = 0.81 > 0.05$	No change
<i>TC/ARC</i>	-0.5312% + 0.8163% + 2.3594%	$p = 0.32 > 0.05$	No change
<i>TC/ARC_{mt}</i>	-0.6663% - 0.0298% + 0.5951%	$p = 0.94 > 0.05$	No change
<i>TC/RC</i>	-12.738% - 4.0848% + 1.1888%	$p = 0.63 > 0.05$	No change
<i>TC/CC</i>	-0.2538% + 0.3748% + 0.9825%	$p = 0.28 > 0.05$	No change
<i>TC/GC</i>	+0.3750% + 1.2131% + 2.0651%	$p = 0.02 < 0.05$	Within threshold
<i>TC/ARENA</i>	-0.0487% + 0.5349% + 1.1198%	$p = 0.11 > 0.05$	No change
<i>TC/EPOCH</i>	-1.0881% - 0.1708% + 0.6827%	$p = 0.73 > 0.05$	No change
<i>TC/EPOCH_{mt}</i>	-2.3554% - 1.0929% + 0.2088%	$p = 0.13 > 0.05$	No change

For the benchmarking group *Facebook (NIPS)*, the change in performance of the benchmark can be seen in Table 4.10. The benchmarks often demonstrate a low p-value, but in these cases, the measured change is very low, and falls well within the noise threshold.

Table 4.11: Benchmarking group: Hamsterster Friendships

Name	Change	p - value	Verdict
<i>BC/ARC</i>	-3.3131% - 1.6202% + 0.0072%	$p = 0.09 > 0.05$	No change
<i>BC/RC</i>	-2.3531% - 1.0078% + 0.2705%	$p = 0.18 > 0.05$	No change
<i>BC/CC</i>	-3.1017% - 1.2975% + 0.6490%	$p = 0.22 > 0.05$	No change
<i>BC/GC</i>	-0.6968% + 1.1147% + 3.0516%	$p = 0.27 > 0.05$	No change
<i>BC/ARENA</i>	-0.5530% + 1.2564% + 2.7792%	$p = 0.16 > 0.05$	No change
<i>BC/EPOCH</i>	-1.0814% + 0.9981% + 3.1935%	$p = 0.41 > 0.05$	No change
<i>BFS/ARC</i>	-11.432% - 4.8710% + 1.4799%	$p = 0.19 > 0.05$	No change
<i>BFS/RC</i>	-2.8923% + 2.3614% + 8.8754%	$p = 0.47 > 0.05$	No change
<i>BFS/CC</i>	-11.374% - 6.1295% - 0.2775%	$p = 0.07 > 0.05$	No change
<i>BFS/GC</i>	-6.4481% + 1.2935% + 8.4995%	$p = 0.74 > 0.05$	No change
<i>BFS/ARENA</i>	-4.2669% + 3.7151% + 13.823%	$p = 0.46 > 0.05$	No change
<i>BFS/EPOCH</i>	-5.7366% - 0.2802% + 4.9466%	$p = 0.93 > 0.05$	No change
<i>CC/ARC</i>	-0.2429% + 0.5161% + 1.3409%	$p = 0.24 > 0.05$	No change
<i>CC/ARC_{mt}</i>	-1.5207% - 0.0659% + 1.1746%	$p = 0.93 > 0.05$	No change
<i>CC/RC</i>	-3.5825% + 2.4090% + 8.6675%	$p = 0.48 > 0.05$	No change
<i>CC/CC</i>	-1.0973% - 0.5061% + 0.1184%	$p = 0.13 > 0.05$	No change
<i>CC/GC</i>	-1.7010% - 0.8512% + 0.2122%	$p = 0.17 > 0.05$	No change
<i>CC/ARENA</i>	+0.0828% + 0.9007% + 1.7685%	$p = 0.05 < 0.05$	Within threshold
<i>CC/EPOCH</i>	-0.8415% - 0.2521% + 0.3375%	$p = 0.44 > 0.05$	No change
<i>CC/EPOCH_{mt}</i>	-1.4783% + 0.2193% + 1.9398%	$p = 0.81 > 0.05$	No change
<i>PR/ARC</i>	-0.8045% - 0.2782% + 0.2062%	$p = 0.31 > 0.05$	No change
<i>PR/ARC_{mt}</i>	-0.7634% + 0.6361% + 1.8461%	$p = 0.39 > 0.05$	No change
<i>PR/RC</i>	-0.0032% + 0.5368% + 1.0868%	$p = 0.09 > 0.05$	No change
<i>PR/CC</i>	-0.8847% - 0.4298% + 0.1180%	$p = 0.15 > 0.05$	No change
<i>PR/GC</i>	+0.6293% + 1.0883% + 1.5384%	$p = 0.00 < 0.05$	Within threshold
<i>PR/ARENA</i>	+0.0771% + 0.3211% + 0.5281%	$p = 0.02 < 0.05$	Within threshold
<i>PR/EPOCH</i>	-0.7194% + 0.0692% + 0.7870%	$p = 0.87 > 0.05$	No change
<i>PR/EPOCH_{mt}</i>	-0.7607% + 0.2695% + 1.3200%	$p = 0.63 > 0.05$	No change
<i>SSSP/ARC</i>	-0.3073% + 1.0720% + 2.4431%	$p = 0.17 > 0.05$	No change
<i>SSSP/RC</i>	-1.1838% + 0.1249% + 1.3019%	$p = 0.87 > 0.05$	No change
<i>SSSP/CC</i>	-1.0566% + 0.1699% + 1.4592%	$p = 0.81 > 0.05$	No change
<i>SSSP/GC</i>	-1.5820% + 0.6159% + 2.4570%	$p = 0.56 > 0.05$	No change
<i>SSSP/ARENA</i>	-3.3533% - 2.1100% - 0.8077%	$p = 0.01 < 0.05$	Within threshold
<i>SSSP/EPOCH</i>	-1.4532% + 0.1831% + 1.8017%	$p = 0.82 > 0.05$	No change
<i>TC/ARC</i>	-0.9707% - 0.3258% + 0.2473%	$p = 0.35 > 0.05$	No change
<i>TC/ARC_{mt}</i>	-0.7945% + 0.2777% + 1.4069%	$p = 0.65 > 0.05$	No change
<i>TC/RC</i>	-0.0789% + 0.3795% + 0.8301%	$p = 0.12 > 0.05$	No change
<i>TC/CC</i>	+0.1069% + 0.5770% + 1.0675%	$p = 0.04 < 0.05$	Within threshold
<i>TC/GC</i>	-0.5531% - 0.1146% + 0.2993%	$p = 0.63 > 0.05$	No change
<i>TC/ARENA</i>	-0.7150% + 0.3249% + 1.4043%	$p = 0.57 > 0.05$	No change
<i>TC/EPOCH</i>	-1.8848% - 1.0558% - 0.4279%	$p = 0.01 < 0.05$	Within threshold
<i>TC/EPOCH_{mt}</i>	-0.8873% + 0.2762% + 1.4754%	$p = 0.67 > 0.05$	No change

The change in performance for the *Hamsterster Friendships* benchmark group can be seen in Table 4.11. The most troublesome measurement is *BFS/CC*, which demonstrates a change of more than 6%, along with a p-value of 0.07. However, the measurement still falls within the criteria.

Table 4.12: Benchmarking group: OPS 20/20/25/25/10

Name	Change	p - value	Verdict
<i>OPS/EPOCH_{mt}</i>	-13.263% - 6.6609% + 0.0367%	$p = 0.10 > 0.05$	No change
<i>OPS/EPOCH</i>	-5.1610% - 1.0887% + 3.5009%	$p = 0.66 > 0.05$	No change
<i>OPS/ARC</i>	-2.0022% + 2.1388% + 6.8230%	$p = 0.37 > 0.05$	No change
<i>OPS/ARC_{mt}</i>	-4.1750% + 2.6347% + 9.9514%	$p = 0.50 > 0.05$	No change
<i>OPS/RC</i>	-0.0306% + 9.5001% + 20.141%	$p = 0.07 > 0.05$	No change
<i>OPS/CC</i>	-6.2243% + 4.1997% + 15.152%	$p = 0.45 > 0.05$	No change
<i>OPS/GC</i>	-39.459% - 23.058% - 1.5049%	$p = 0.06 > 0.05$	No change
<i>OPS/ARENA</i>	-4.8206% + 1.3914% + 7.3909%	$p = 0.66 > 0.05$	No change

The change in performance for the *OPS 20/20/25/25/10* benchmark group can be seen in Table 4.12. Here, the most notable change is for *OPS/GC*, which demonstrates a change of more than 20%, with a fairly low p-value of 0.06.

Table 4.13: Benchmarking group: OPS 40/40/10/10/0

Name	Change	p - value	Verdict
<i>OPS/EPOCH_{mt}</i>	-5.8080% - 0.3195% + 4.6192%	$p = 0.91 > 0.05$	No change
<i>OPS/EPOCH</i>	-1.9879% + 0.9409% + 4.1149%	$p = 0.57 > 0.05$	No change
<i>OPS/ARC</i>	-8.0630% + 2.0602% + 14.449%	$p = 0.75 > 0.05$	No change
<i>OPS/ARC_{mt}</i>	-6.7197% - 1.1225% + 4.7703%	$p = 0.71 > 0.05$	No change
<i>OPS/RC</i>	-12.227% - 1.2963% + 11.142%	$p = 0.84 > 0.05$	No change
<i>OPS/CC</i>	-17.630% + 40.143% + 170.53%	$p = 0.63 > 0.05$	No change
<i>OPS/GC</i>	-24.473% - 2.5861% + 22.752%	$p = 0.85 > 0.05$	No change
<i>OPS/ARENA</i>	-11.054% - 3.5571% + 4.5223%	$p = 0.42 > 0.05$	No change

The change in performance for the *OPS 40/40/10/10/0* benchmark group can be seen in Table 4.13. Here, the most notable change is for *OPS/CC*, which demonstrates a change of more than 40%. The p-value however, is 0.63, which indicates that the change is insignificant even though the performance change is large.

A decorative element consisting of several thin, vertical black lines of varying heights, creating a stylized '5' shape.

5 Results

This chapter presents the results. First, the result of GC Bench is presented, then the results of the GAP Benchmark suite. Finally, some benchmarks including different distributions of operations is presented. The results are later discussed in Chapter 6.

5.1 GC Bench

This section presents the result for both sequential and parallel versions of GC Bench. The run was made with the stretch tree depth set to ten, long lived tree depth and maximum tree depth set from eight to fourteen with a step depth of two, and a minimum tree depth of four.

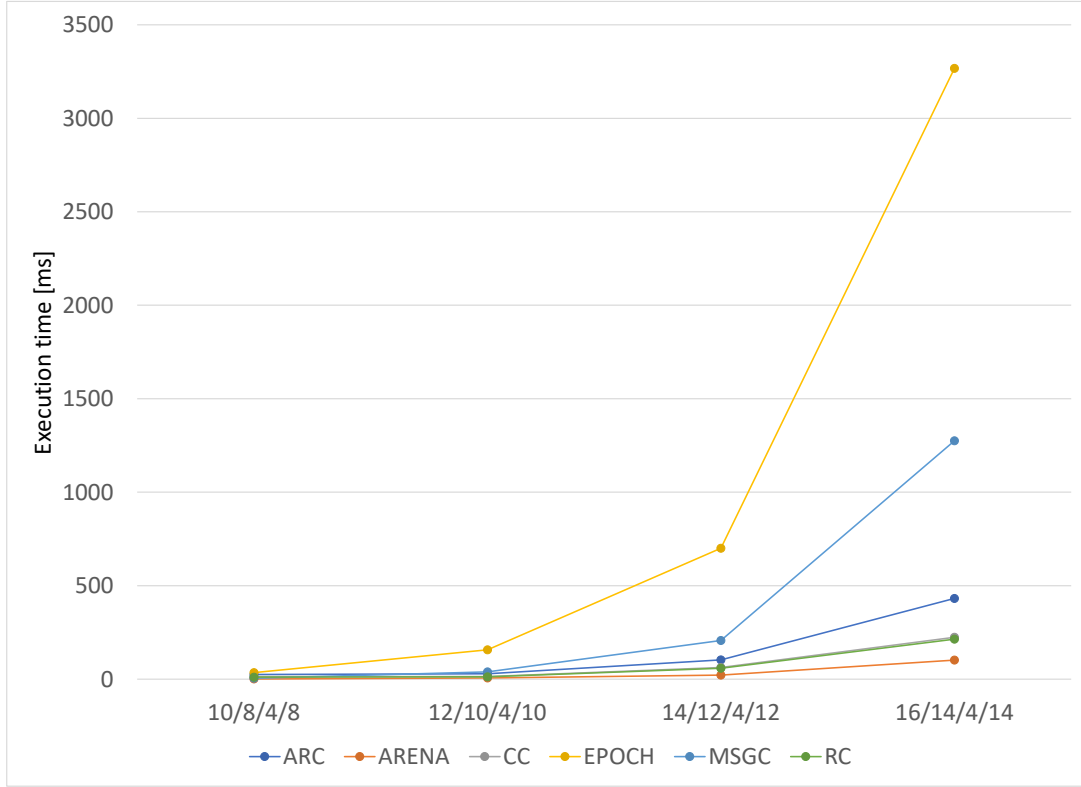


Figure 5.1: GC Bench for sequential construction. The labels on the horizontal axis represent the depth of the generated trees (Stretch Tree/Long Lived Tree/Min Tree/Max Tree).

As seen in Figure 5.1, for sequential construction, $G :: EPOCH$ performs worst, which is particularly eminent as the depth of the allocated tree increases. Slightly faster is $G :: MSGC$, which performs close to the reference counted versions $G :: RC$ and $G :: CC$ when the scale is small. However, as the tree increases, the performance of $G :: MSGC$ falls behind. When it comes to $G :: CC$ and $G :: RC$, they are similar in performance for all scales. Finally, $G :: ARENA$ demonstrates the best performance of them all.

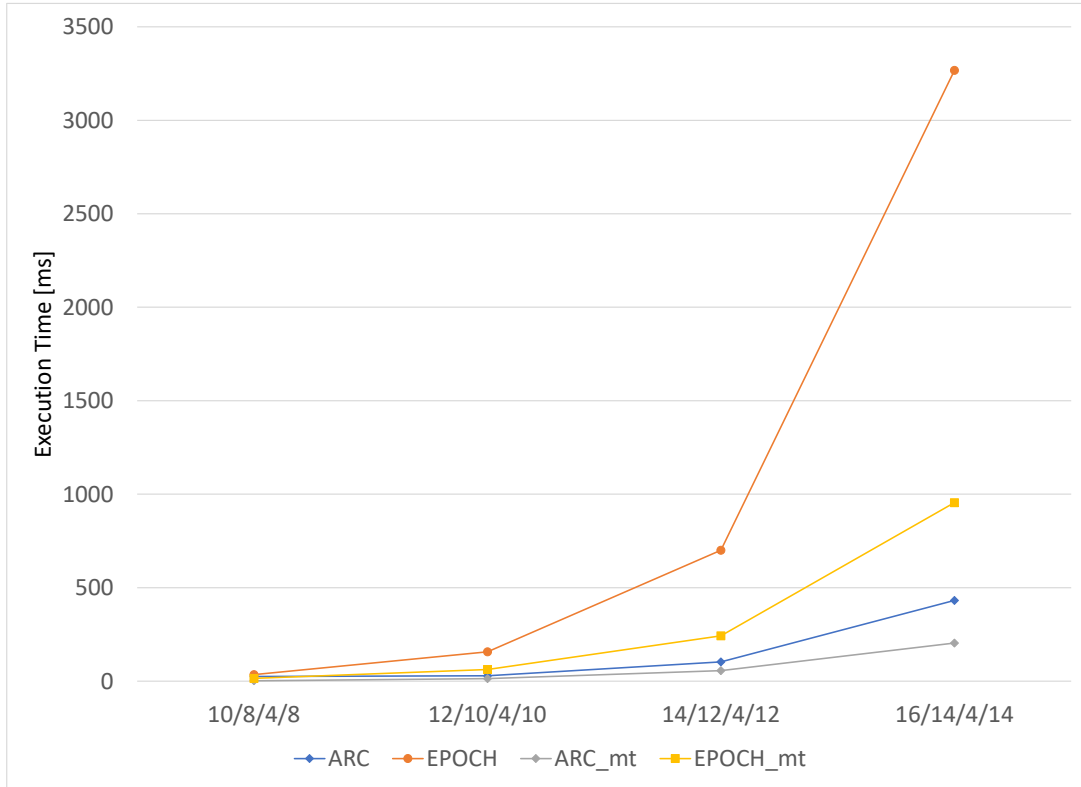


Figure 5.2: GC Bench for parallel and sequential construction. The labels on the horizontal axis represent the depth of the generated trees (Stretch Tree/Long Lived Tree/Min Tree/Max Tree).

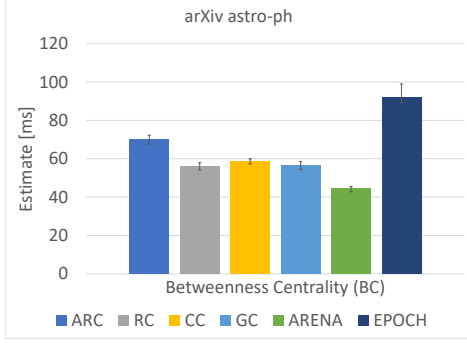
For GC Bench with parallel construction, the performance rankings are the same as for sequential construction, however, in Figure 5.2 we see that $G :: EPOCH$ gains a more significant increase from parallel execution, as opposed to $G :: ARC$, which only demonstrates slightly increased performance when run in parallel.

5.2 The GAP Benchmark Suite

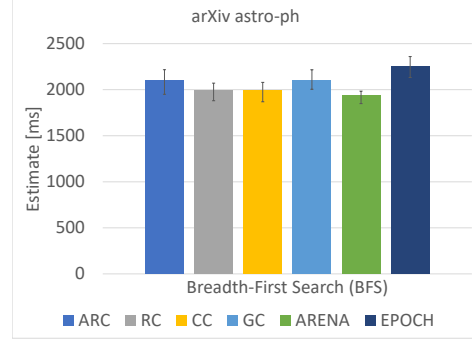
This section presents the results for the GAP Benchmark suite. The tables show an estimate of the runs that is made by measuring the slope of the regression model. The lower bound and upper bound show the upper and lower bound of the confidence interval.

5.2.1 arXiv astro-ph

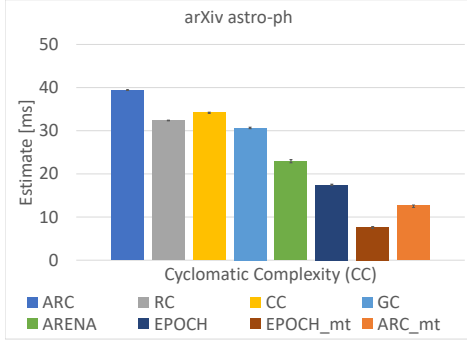
The results of the *arXiv astro-ph* benchmark group can be seen in Figure 5.3.



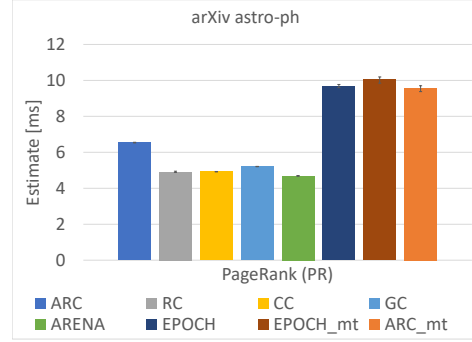
(a) Betweenness Centrality



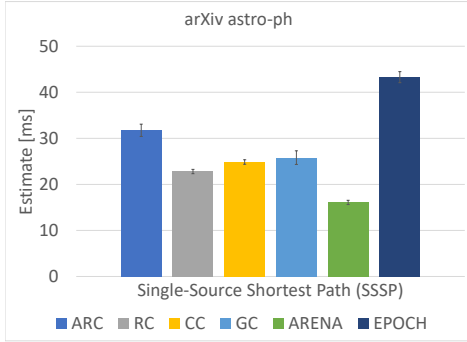
(b) Breadth-First Search



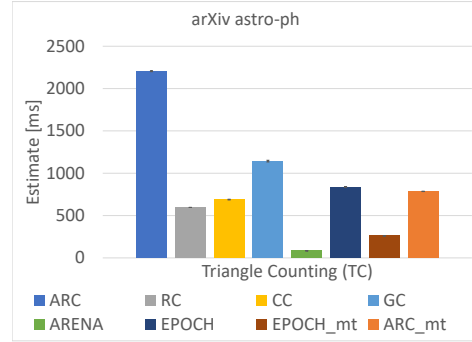
(c) Cyclomatic Complexity



(d) PageRank



(e) Single-Source Shortest Path

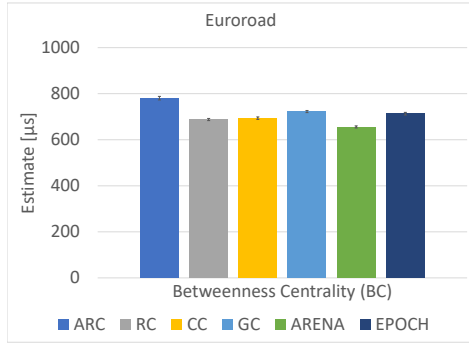


(f) Triangle Counting

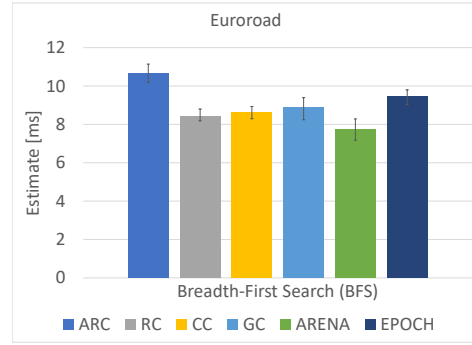
Figure 5.3: Estimated execution times for each kernel in the GAP benchmark suite for the *arXiv astro-ph* dataset. The estimate is measured as the slope of the regression model. The error bars display the lower and upper bound of the confidence interval.

5.2.2 Euroroad

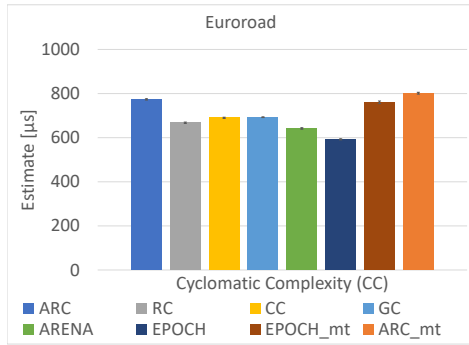
The results of the *Euroroad* benchmark group can be seen in Figure 5.4.



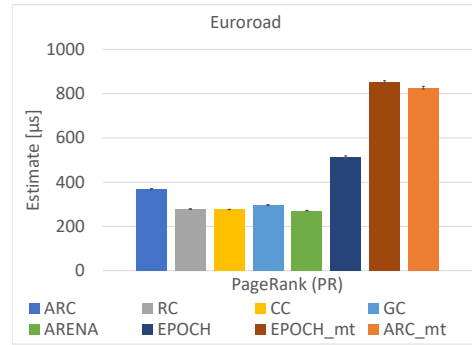
(a) Betweenness Centrality



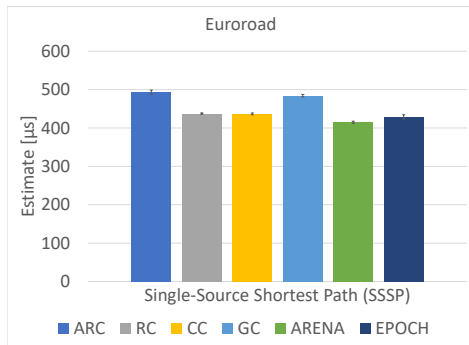
(b) Breadth-First Search



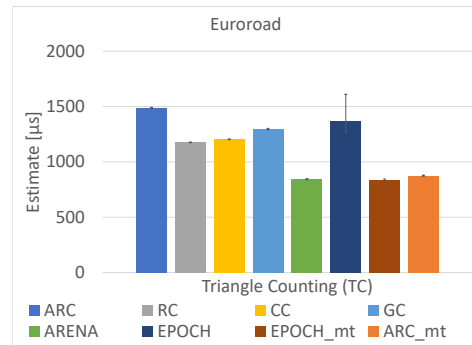
(c) Cyclomatic Complexity



(d) PageRank



(e) Single-Source Shortest Path

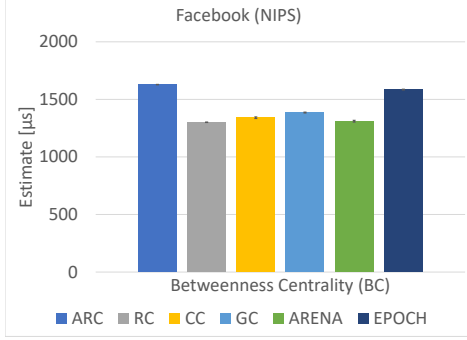


(f) Triangle Counting

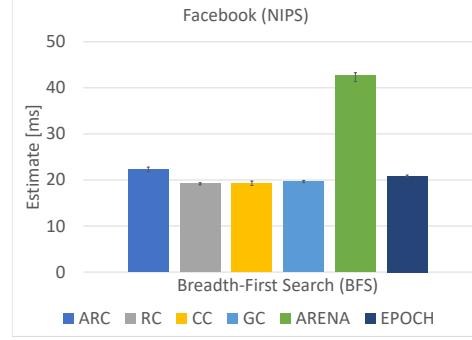
Figure 5.4: Estimated execution times for each kernel in the GAP benchmark suite for the *Euroroad* dataset. The estimate is measured as the slope of the regression model. The error bars display the lower and upper bound of the confidence interval.

5.2.3 Facebook (NIPS)

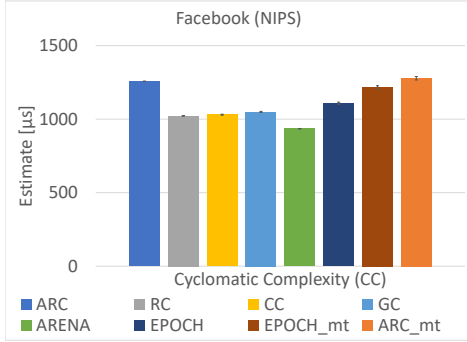
The results of the *Facebook (NIPS)* benchmark group can be seen in Figure 5.5.



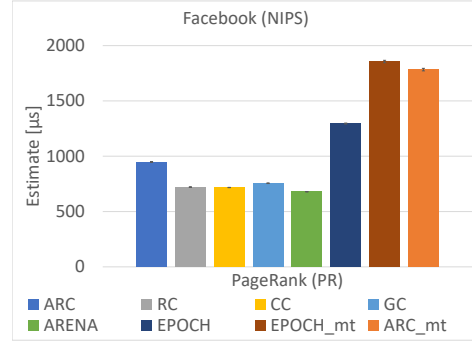
(a) Betweenness Centrality



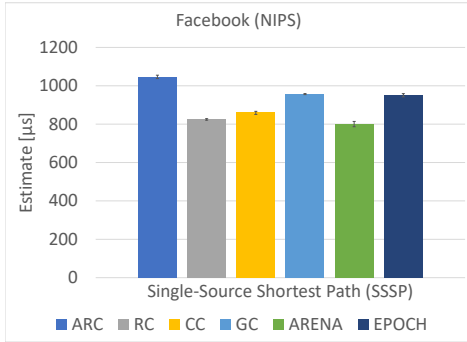
(b) Breadth-First Search



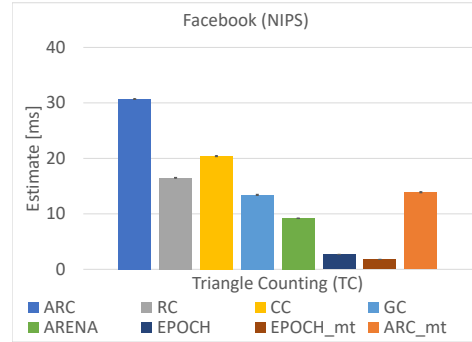
(c) Cyclomatic Complexity



(d) PageRank



(e) Single-Source Shortest Path

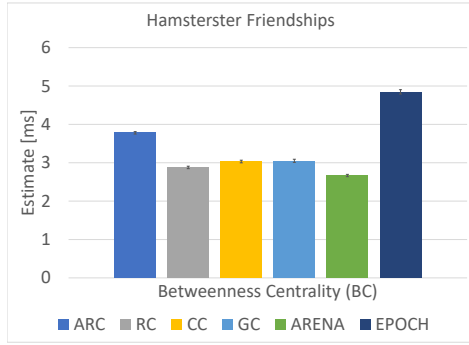


(f) Triangle Counting

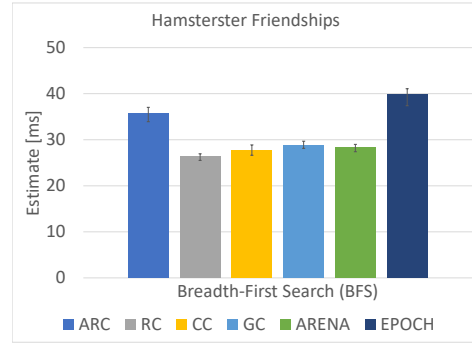
Figure 5.5: Estimated execution times for each kernel in the GAP benchmark suite for the *Facebook (NIPS)* dataset. The estimate is measured as the slope of the regression model. The error bars display the lower and upper bound of the confidence interval.

5.2.4 Hamsterster Friendships

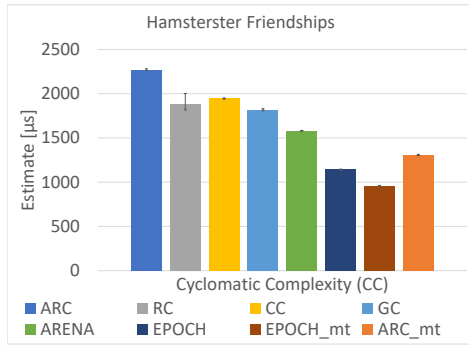
The results of the *Hamsterster Friendships* benchmark group can be seen in Figure 5.6.



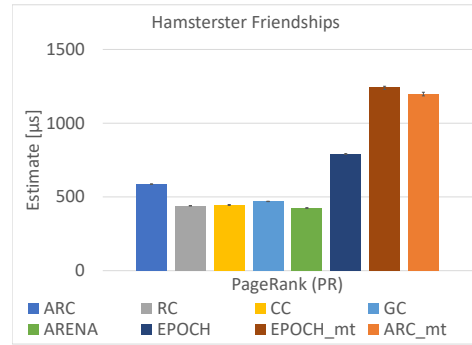
(a) Betweenness Centrality



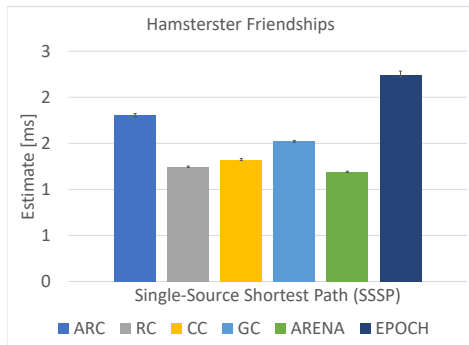
(b) Breadth-First Search



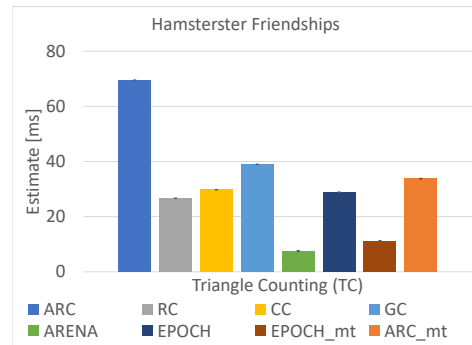
(c) Cyclomatic Complexity



(d) PageRank



(e) Single-Source Shortest Path



(f) Triangle Counting

Figure 5.6: Estimated execution times for each kernel in the GAP benchmark suite for the *Hamsterster Friendships* dataset. The estimate is measured as the slope of the regression model. The error bars display the lower and upper bound of the confidence interval.

5.3 Operations

This section presents simple operations made on a graph with 256 vertices and an average degree of ten. The number of operations for each benchmark was set to 1000.

5.3.1 Operations 20/20/25/25/10

This operation distribution targets edge operations, while also including expensive operations such as inserting and deleting vertices. It performs 20% vertex insertion, 20% vertex deletion, 25% edge insertion, 25% edge deletion and 10% find vertex operations. The results can be seen in Table 5.1, along with a violin plot in Figure 5.7.

Table 5.1: Benchmarking group: OPS 20/20/25/25/10

Name	Lower bound	Estimate	Upper bound
<i>OPS/ARC</i>	218.75us	222.68us	225.54us
<i>OPS/ARC_{mt}</i>	453.78us	500.51us	527.23us
<i>OPS/RC</i>	212.63us	223.73us	239.40us
<i>OPS/CC</i>	193.02us	203.35us	224.18us
<i>OPS/GC</i>	261.08us	306.73us	380.79us
<i>OPS/ARENA</i>	200.41us	205.04us	207.69us
<i>OPS/EPOCH</i>	580.75us	599.24us	611.89us
<i>OPS/EPOCH_{mt}</i>	242.61us	246.26us	252.39us

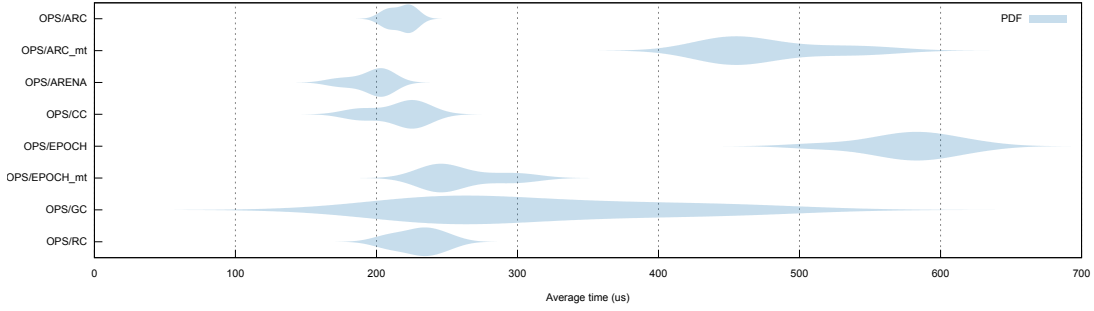


Figure 5.7: Benchmark group: OPS 20/20/25/25/10

5.3.2 Operations 40/40/10/10/0

This operation distribution targets vertex operations. It performs 40% vertex insertion, 40% vertex deletion, 10% edge insertion, 10% edge deletion and 0% find vertex operations. The results can be seen in Table 5.2, along with a violin plot in Figure 5.8.

Table 5.2: Benchmarking group: OPS 40/40/10/10/0

Name	Lower bound	Estimate	Upper bound
<i>OPS/ARC</i>	195.26us	214.42us	239.36us
<i>OPS/ARC_{mt}</i>	439.30us	461.25us	477.20us
<i>OPS/RC</i>	158.51us	171.74us	186.20us
<i>OPS/CC</i>	167.51us	303.98us	605.38us
<i>OPS/GC</i>	293.00us	315.89us	359.42us
<i>OPS/ARENA</i>	164.51us	169.95us	173.14us
<i>OPS/EPOCH_{mt}</i>	293.14us	296.61us	301.61us
<i>OPS/EPOCH</i>	738.23us	754.19us	767.05us

The negative average times observed in Figure 5.8 are an effect from resampling. Due to the severe outlier of over $1000us$ for `G::CC`, the resampling suggests that if a sample can be extreme in the positive direction, it should also be possible in the negative direction.

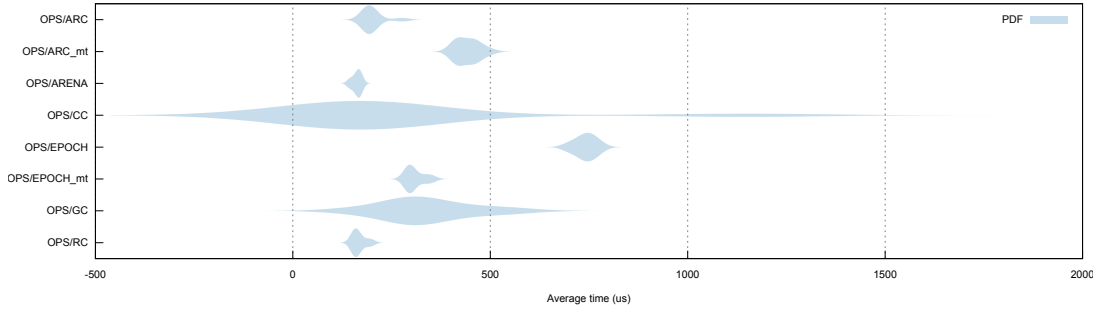


Figure 5.8: Benchmark group: OPS 40/40/10/10/0

5.4 Usability

Rust is flexible enough to allow the graph representations to expose a similar interface. This means that for a graph representation G_a , the effort of operating the graph is the same as for any other representation G_b . Hence, as suggested by Section 3.7, the operability of the graph representations is identical. There is however one exception of this, where both `G::CC` and `G::GC` require the nodes and any data type that is stored within a node to be valid for the static lifetime and implement a trait called `Trace`. The effort required to implement this trait depends entirely on the complexity of the contained type.

In terms of user error protection, the graph representations are all very similar. As they all expose the same interface, there is no need for quantifying e.g. the number of exposed methods or the length of the parameter lists as mentioned in Section 3.7. However, there are some differences between the models, where it is possible to encounter user errors. One possible user error is for `G::ARENA`, where all graph models except for `G::ARENA` builds on pointers that are guaranteed to point to valid objects. This means that it is safe to dereference any node except for `G::ARENA`, where it might be possible to end up with an invalid node. When it comes to memory safety, `G::RC` and `G::ARC` will both cause memory leaks if they contain any cycle. Finally, the reference-count based solutions can lead to borrow hazards, and careless use of `G::ARC` in multi-threaded environments can lead to deadlocks. A summary of the usability can be seen in Table 5.3.

Table 5.3: Usability for the different graph representations.

Name	Operability	User error protection
<code>G::ARC</code>	+	-
<code>G::ARENA</code>	-	-
<code>G::CC</code>	-	+
<code>G::EPOCH</code>	+	+
<code>G::GC</code>	-	+
<code>G::RC</code>	+	-



6 Discussion

This chapter includes an analysis and discussion of the method and the captured results. First, a discussion on the different benchmarks is presented. Second, a discussion on the performance differences is held, followed by a section on usability. Then the chosen method is discussed, along with eventual improvement suggestions. Finally, the work is placed in the industry context, as well as a wider context.

6.1 Results

This section includes a discussion of the results for the different benchmarks.

6.1.1 GC Bench

The demonstrated performance for GC Bench offers no surprises. Region-based allocation is the fastest, followed by the reference counting, garbage collection and EBR placing last, as shown by Figure 5.1. The poor performance of EBR is likely imposed by the link-based structure that is used to store vertices in the graph, as each insertion operation requires location of its predecessor, which computationally takes linear time. The epoch-based model requires a large amount of atomic operations to function, which results in it being even slower. Adding to the poor performance is also the need for executing each operation in a transaction. For GC Bench, each insertion was made by running a single operation in a single transaction. This means that offering a lock-free transactional graph is unnecessary in this case, and the model would demonstrate much better performance if the operations were instead executed directly on the graph, or if all operations were executed in a single transaction.

In our implementation of `G:EPOCH`, we offer no raw execution of operations, as we deemed the transactions necessary to achieve soundness and correctness. However, if a developer is certain that no other entities will operate on the graph in parallel, it should be possible to execute operations without the need for encapsulating them in a transaction. In Rust, this could be done by offering unsafe bindings, which would indicate that developers using the functions must do so with extra care.

As for the arena based allocator, we experienced some pause times as an effect of memory running out of the arena and doubling in size during the early stages of our experiment. In cases where the size of the graph is known preconstruction, it is possible to specify the size of

the arena in order to avoid this issue. If only a size hint is available, that can also avoid the pause times for the most part, with the worst-case scenario being that the size of the arena has to be extended if the size exceeds the size hint. With graphs that heavily vary in size through its lifetime, the issue of arena allocation can be that the initial allocation is either too small, yielding more frequent pause times when the graph grows; or the initial allocation is too big, resulting in an unnecessary amount of memory being occupied by the graph.

As for multithreading, `G::ARC` and `G::EPOCH` are both `Send` and `Sync`, meaning that they can safely be sent and shared between threads. We noticed that the core utilization of the epoch based variant is much higher than for `G::ARC`, which is shown in Figure 5.2. This is most likely due to the underlying data structure. In `G::EPOCH`, we use a concurrent link-based structure to store vertices, and a concurrent multi-dimensional array to store edges. With these structures, every mutation works by performing a CAS-operation, in cases where the CAS fails, the operation is put into a spinlock until it is successful. The operation is thus scoped to at most one or two nodes, which means that CAS failure occurs rather seldom. `G::ARC` instead uses a lock-based solution via a read-and-write lock. Due to possible deadlocks, the granularity is kept fairly low, in the sense that eventual mutations must lock the entire data structure that is holding the vertices. This means that core utilization will not be very good for write-heavy workloads. In GC bench, we avoided storing the nodes in a separate data structure to avoid this issue. Instead, each node stores two children. This way, the core utilization is much higher, as we are able to mutate multiple nodes in parallel. A final note is that GC Bench allocates many nodes with few edges. For `G::EPOCH`, operations on the list of vertices takes linear-time, but operations on the edge list takes logarithmic time. It is possible that the performance of `G::EPOCH` would exceed `G::ARC` if the vertex/edge-ratio was dominated by edges.

6.1.2 GAP Benchmark Suite

The results for the GAP Benchmark Suite is quite peculiar, as the performance heavily depends on the kind of graph that is operated on.

For sequential execution, the results indicate that either `G::ARENA` or `G::RC` offers best performance. For example, `G::ARENA` excels at executing the CC-kernel, where it is the only memory model performing under one millisecond for the Facebook (NIPS) dataset, shown in Figure 5.5. There are two reasons for this. First, the CC-kernel requires several clones or copies of the nodes. This is a cheap operation for `G::ARENA`, as it is only a matter of copying indices, while for the reference counting graph models, cloning is far more expensive. Secondly, the kernel includes few dereferences of the actual nodes, meaning that we do not have to pay the price of looking up the index in the actual arena. However, in cases where the graph has a high diameter such as the Euroroad dataset, `G::EPOCH` is even faster than `G::ARENA`, as seen in Figure 5.4. The reason for this is that dereferencing is mainly done on edges, which is much cheaper for `G::EPOCH` than it is for `G::ARENA`. With a high diameter, a larger amount of edges need to be dereferenced.

An interesting observation is that the BC-kernel demonstrates the same performance ranking for the different models as the CC-kernel, with the exception of `G::EPOCH` always being slower than `G::ARENA`, as seen in Figure 5.3, 5.4, 5.5, 5.6. The reason for this seems to be that the kernel frequently calls for the edges of a particular vertex. While it is cheap to get the edges of an already located vertex for `G::EPOCH`, locating the vertex is actually quite expensive. Whereas for `G::ARENA`, finding the vertex in the first place is just a matter of looking up the index.

Surprisingly, the BFS-kernel showed a significant difference in performance for `G::ARENA` for the Facebook (NIPS) network presented in Figure 5.5. The reason is not as clear as in previous cases. Most likely is that a few nodes have a very high degree and cloning all the edges of such a node is much more expensive than cloning a single reference to the container, as is done by other implementations.

The TC-kernel is by far the most computationally heavy kernel. For this kernel, `G::ARENA` demonstrates very good performance. As the triangle count is quite small for both Euroroad, and Facebook (NIPS), it is the most interesting to look at the results of Hamsterster friendships and the arXiv astro-ph. As this kernel is much bigger, and takes a longer time to execute, the multi-threaded versions actually gain a lot of performance as indicated by Figure 5.3 and 5.6. On our machine, the multi-threaded version of `G::EPOCH` performs nearly as good as `G::ARENA`. It is possible that it would be even better if the benchmarks were run on a machine with additional CPU cores. For the PR-kernel however, the multithreaded variants performed worse for every graph model. The reason seems to be that the overhead of spawning the tasks on separate threads exceeds the performance gain of parallel execution.

6.1.3 Operations Benchmark

When looking at executing operations in the graph, such as inserting or deleting vertices, the epoch based graph is by far the slowest, as shown by Tables 5.1 and 5.2. The operations benchmark was done using a relatively small graph of 256 vertices, so if we were to scale the graph even further, it would be even slower as indicated by GC bench in Figure 5.1. An important note here, is that the operations benchmark did not use an access map to avoid looking up indices in the graph that have already been visited. This means that operations that insert or delete edges have to first locate the parent vertex by traversing the link-based structure that is holding the vertices, and then append the edge to the list of edges. Other than this, an interesting observation in the operations benchmark, is that Tables 5.7 and 5.8 both suggest that the multithreaded version of `G::ARC` performs worse than its sequential version. The reason seems to be that the work done in each iteration is too small to justify spawning the task on a separate thread, thus yielding overhead that exceeds the gained performance of parallel execution. This issue can be seen in the results of the GAP benchmark suite as well, especially for the PR-kernel. Additionally, almost all operations (90% or 100%) were write operations, meaning that every thread required to access the graph exclusively – leading to extensive resource contention.

6.1.4 Performance

It is clear that the most performance gains come from the type of data structure that we are able to store the nodes in. For the reference counting-based graph models, it is possible to use a BTreeMap. A BTreeMap offers the best possible performance in terms of look-ups, meanwhile it is also sorted, which many of the kernels in the GAP benchmark suite require. For `G::EPOCH`, the vertices must be stored in a linked-list in order for the lock-free transactional theory to work properly, so any graph with a large set of indices, and where we are not able to store a fast access map of some kind will suffer from poor performance. This issue does not appear in the GAP benchmark suite, as we have used a fast access map for the indices, but in implementations such as GC Bench, the poor scalability is quite noticeable. It would be very interesting if it was possible to implement a lock-free transactional adjacency list that does not require a link-based data structure. To the best of our knowledge, there is currently no work that presents a way of achieving this. One solution to speed up vertex location in the `G::EPOCH` graph could perhaps be to store the vertices in a multi-dimensional array as well, or perhaps even a skiplist.

6.1.5 Usability

The graph models are quite similar in terms of operability, see Section 5.4, except for some cases. For example, in `G::CC` and `G::GC`, each node must implement the `Trace` trait. This is often a non-issue, as Rust offers procedural macros that makes it very easy to automatically implement the trait. This can, however, be a problem in the case where we want to use

generic types, as each type that is stored within the node also has to implement the `Trace` trait. Additionally, as the collection is global, the objects must be alive for the static lifetime. In the future, it should be possible to store the collector inside the graph itself, thus eliminating the need for the static lifetime.

When it comes to user error protection, `G::ARENA` operates on the graph via indices. This means that the underlying object may be deleted from the arena while an index is held by some other entity. A developer must therefore be extra careful when deleting entries in the arena.

For the reference counting-based implementations, interior mutability is achieved via `std::cell::RefCell`, which means that borrow checking is done during runtime rather than statically. This means that breaking eventual ownership rules would cause a panic during execution. For the most part, this is a non-issue, as typical operations that operate on the actual graph oftentimes are a matter of a few lines of code.

As one has to be careful when it comes to borrowing, it is extremely important to be careful when working with `G::ARC`. This graph uses read-and-write locks for synchronization. For example, the lock can be poisoned as a result of a thread panic, or we can end up with a deadlock if a write-lock is acquired when holding a read-lock. This issue is mitigated in the lock-free transactional graph `G::EPOCH`, where it is not possible to run into any concurrency issues. In our experiment, we actually observed some deadlocks in the early stages of the `G::ARC` graph model and some borrow hazards in the `G::RC` solution.

With the reference counting-based graphs, handling cycles is also a problematic issue. With `G::CC` it is necessary to manually call the cycle collector. The most problematic part of this, is to decide when and where to perform the collection. The most obvious choice is to trigger collection once the graph is constructed, or when a mutation occurs on the graph. However, with more frequent cycle collection comes larger performance penalties. For short lived graphs, it would be feasible to allow leakage of memory while the graph is alive, and then invoke the cycle collector in the destructor of the graph in order to clean up the leaks once it goes out of scope. For long lived graphs, it might not be feasible to allow memory leaks, as they might grow arbitrarily big. In this case, it would be more fair to invoke the collector after some predefined set of operations, similar to the collector in `G::EPOCH`. `G::ARC` and `G::RC` on the other hand require extra caution not to end up with memory leaks. A possible solution is to store strong references to every node in some data structure, and then use weak reference counting within the graph itself. This way, any cycle in the graph would not lead to memory leaks. However, a new problem arises, where it is possible that the weak references point to some invalid object. Additionally, it is possible that no reference at all exists in the graph for some node, meaning that it is stored to no avail. The issue here becomes to manage the data structure holding the true references and making sure that they match whatever is stored as weak references within the graph.

6.2 Method

The lack of formalized evaluation methodologies for graphs and graph operations has been a troublesome problem for this thesis. The GAP benchmark suite is one of few attempts that tackles this shortcoming. The suite mainly targets evaluations for large-scale graph processing, which means that its suggested graphs are larger than what a typical computer can handle and the kernels do not include graph mutations. Given the circumstances and lack of work in the area, compromises had to be made, and the GAP benchmark suite was chosen after all. The suite enabled a good foundation for benchmarking different types of read operations on the graphs, and by adding GC bench, we also enabled measurements for allocation and deallocation. Finally, by adding the operations benchmark, we were also able to measure the impact of mutations made in a graph.

A big problem that we see in our measurements is the need for evaluating for more graph topologies. The chosen graphs are diverse, but it would be favorable to run the benchmarks on other types of graphs as well. Ideally, one would vary one factor at a time, to be able to isolate that the result is actually an effect of that very factor. For example, it would have been good to measure graphs with the same amount of vertices and edges and only vary the diameter of the graph. This would be a time consuming task, but it would make the results much easier to interpret and draw conclusions from.

Another problem that we ran into is the immaturity of the Rust ecosystem. For example, when porting the GAP benchmark suite to Rust, we ran into problems regarding parallel libraries and had to build some dependencies from source in order to get the functionality we were after. In some cases, there was not an available solution and we ended up doing a lot of hacks to make it work properly.

When it comes to the evaluation of usability, it would be possible to conduct a quantitative analysis and measure e.g. the number of entry points that allow for user errors. We decided to not use a quantitative approach, because of the very few differences there are between the different models. For example, the reference-count based graphs are open to the possibility of borrow hazards. The likelihood of these could probably be measured, but it would be a very tedious task, which is not justified by the small contribution it would mean to the rest of the thesis. Instead, we have focused on identifying what types of user errors the graphs are susceptible for at large. This means that we rather discuss things such as what types of user errors it is possible to run into, rather than evaluating the probability of a user error. The same goes for measuring the *operability*, where the differences between the graph representations are very few due to Rust allowing us to expose a similar interface for all graph representations. This makes us believe that simply identifying the differences is a sufficient evaluation.

6.3 Use in Industry

The use-case for Configura mainly includes traversals of nodes in long-lived graphs. This means that the graphs do not necessarily have to be fast to construct, but rather fast to traverse. For small graphs, the performance is not that important, as the difference in performance between the graph models is not that big. For large graphs however, it is important to have a representation that offers good performance. For this reason, we suggest that Configura should use `G::ARENA` for their application, as it offers best performance. However, in cases where it is difficult to make sure the indices for `G::ARENA` are valid, we instead recommend to use `G::CC`, as it does not leak memory and the performance is very similar to `G::RC`. Once multi-threading is widely available on the web, we also suggest `G::EPOCH` to be a suitable candidate.

6.4 The work in a wider context

By providing an evaluation and approaches to graph representations in Rust, we have provided guidance to developers who want to represent graph-like data structures. By looking at the usability, we are able to provide guidance on which graph model to use when usability is the main focus. This allows developers to pick graph models that enable use without fear of runtime errors, which sometimes can be the sole reason for choosing Rust as the programming language in the first place. This feature can be critical for some applications, where a deadlock or a borrow hazard during runtime would lead to devastating outcomes.

As for performance, we have provided an evaluation that allows developers to pick the right memory model for their use-case, thus enabling them to minimize the amount of work made in graph-related operations. This can have large impacts on the ability to develop programs in Rust that are fast enough to be suitable for critical applications.

7

Conclusion

There are multiple ways of representing graphs in Rust. As for memory models, we have identified reference counting, tracing garbage collection and region-based allocation to be suitable candidates. For concurrent settings, EBR is a good fit for Rust.

When it comes to performance, there is no silver bullet covering every use case, but the memory models rather exhibit properties that work well for some graph topologies and bad for others. Reference counting for example, demonstrates very good performance in the general case, but when many references are cloned outside of the graph, the overhead is rather large. Reference counting with possible cycle collection demonstrates slightly slower performance than plain reference counting and is probably the best alternative in terms of performance to handling garbage cycles and mitigating eventual memory leaks. Tracing garbage collection is slightly slower than reference counting, but has the benefit of being able to handle cycles.

A contribution of this thesis is that previous work within the Rust community has mainly discussed reference counting, tracing garbage collection and arena allocation for representing graphs. In this thesis, we present an alternative way via a lock-free graph data structure using EBR, that for some operations and graph topologies demonstrates better performance than any other representation. Meanwhile, it also offers fearless concurrency, which the `G::ARC` representation does not. However, the `G::EPOCH` representation demonstrates poor scalability, which is an effect of using a link-based data structure to store its vertices.

In terms of operability, `G::GC` and `G::CC` require some trait implementations for their nodes in order to be traceable, which can be a hindrance for complex types. When looking at the user error protection software quality, `G::ARENA` can easily lead to invalid indices, which makes it harder to work with. With the reference count based solutions, it is possible to run into runtime errors in the form of borrow hazards or deadlocks.

7.0.1 Future work

We were not able to create a working implementation of `G::IMMIX`, but the initial experiments we did on the collector showed promising results. It would thus be interesting to see how another garbage collector other than the one used in `G::MSGC` performs compared to the other models. Another interesting idea for future work is to get rid of the linked-list in the lock free transactional graph and instead use some data structure that offers less than linear complexity for look-up.




Bibliography

- [1] *arXiv astro-ph network dataset – KONECT*. Apr. 2017. URL: <http://konect.uni-koblenz.de/networks/ca-AstroPh>.
- [2] David Bacon, C. Attanasio, Han Lee, Vadakkedathu Rajan, and Stephen Smith. “Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector”. In: *ACM SIGPLAN Notices* 36 (Aug. 2002). DOI: 10.1145/378795.378819.
- [3] David F. Bacon, Perry Cheng, and V. T. Rajan. “A Unified Theory of Garbage Collection”. In: *SIGPLAN Not.* 39.10 (Oct. 2004), pp. 50–68. ISSN: 0362-1340. DOI: 10.1145/1035292.1028982. URL: <https://doi.org/10.1145/1035292.1028982>.
- [4] David F. Bacon and V. T. Rajan. “Concurrent Cycle Collection in Reference Counted Systems”. In: *Proceedings of the 15th European Conference on Object-Oriented Programming*. ECOOP ’01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 207–235. ISBN: 3540422064.
- [5] Scott Beamer, Krste Asanović, and David Patterson. *The GAP Benchmark Suite*. 2015. arXiv: 1508.03619 [cs.DC].
- [6] Stephen M Blackburn and Kathryn S McKinley. “Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance”. In: *ACM SIGPLAN Notices* 43.6 (2008), pp. 22–32.
- [7] Stephen M. Blackburn and Kathryn S. McKinley. “Ulterior Reference Counting: Fast Garbage Collection without a Long Wait”. In: *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’03. Anaheim, California, USA: Association for Computing Machinery, 2003, pp. 344–358. ISBN: 1581137125. DOI: 10.1145/949305.949336. URL: <https://doi.org/10.1145/949305.949336>.
- [8] Hans-J. Boehm. “Reducing Garbage Collector Cache Misses”. In: *SIGPLAN Not.* 36.1 (Oct. 2000), pp. 59–64. ISSN: 0362-1340. DOI: 10.1145/362426.362438. URL: <https://doi.org/10.1145/362426.362438>.
- [9] George E. Collins. “A Method for Overlapping and Erasure of Lists”. In: *Commun. ACM* 3.12 (Dec. 1960), pp. 655–657. ISSN: 0001-0782. DOI: 10.1145/367487.367501. URL: <https://doi-org.e.bibl.liu.se/10.1145/367487.367501>.

- [10] D. Dechev, P. Pirkelbauer, and B. Stroustrup. “Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs”. In: *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. May 2010, pp. 185–192. DOI: 10.1109/ISORC.2010.10.
- [11] L. Deutsch and Daniel Bobrow. “An Efficient, Incremental, Automatic Garbage Collector”. In: *Communications of the ACM* 19 (Sept. 1976), pp. 522–. DOI: 10.1145/360336.360345.
- [12] *Euroroad network dataset – KONECT*. Apr. 2017. URL: http://konect.uni-koblenz.de/networks/subelj_euroroad.
- [13] *Facebook (NIPS) network dataset – KONECT*. Apr. 2017. URL: <http://konect.uni-koblenz.de/networks/ego-facebook>.
- [14] Daniel Frampton, Stephen Blackburn, Luke Quinane, and John Zigman. “Efficient Concurrent Mark-Sweep Cycle Collection”. In: (Jan. 2009).
- [15] *Hamsterster friendships network dataset – KONECT*. Apr. 2017. URL: <http://konect.uni-koblenz.de/networks/petster-friendships-hamster>.
- [16] Hadadji Hamza and S. Counsell. “Region-Based RTSJ Memory Management: State of the art”. In: *Science of Computer Programming* 77 (May 2012), pp. 644–659. DOI: 10.1016/j.scico.2012.01.002.
- [17] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. “Performance of memory reclamation for lockless synchronization”. In: *Journal of Parallel and Distributed Computing* 67.12 (2007), pp. 1270–1285.
- [18] Matthew Hertz and Emery D. Berger. “Quantifying the Performance of Garbage Collection vs. Explicit Memory Management”. In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’05. San Diego, CA, USA: Association for Computing Machinery, 2005, pp. 313–326. ISBN: 1595930310. DOI: 10.1145/1094811.1094836. URL: <https://doi.org/10.1145/1094811.1094836>.
- [19] T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J.-P. Lesot, and F. Parain. “Region-based memory management for real-time Java”. In: *Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISORC 2001*. May 2001, pp. 387–394. DOI: 10.1109/ISORC.2001.922863.
- [20] Vojtěch Horký, Peter Libič, Antonin Steinhauser, and Petr Tůma. “DOs and DON’Ts of Conducting Performance Measurements in Java (Tutorial Paper)”. In: *Proc. 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*. New York, NY, USA: ACM, 2015, pp. 337–340. ISBN: 978-1-4503-3248-4. DOI: 10.1145/2668930.2688820. URL: <http://doi.acm.org/10.1145/2668930.2688820>.
- [21] Geneva International Organization for Standardization. and Geneva International Electrotechnical Commission. “Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.” In: (2011).
- [22] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. “Preliminary guidelines for empirical research in software engineering”. In: *IEEE Transactions on Software Engineering* 28.8 (Aug. 2002), pp. 721–734. ISSN: 2326-3881. DOI: 10.1109/TSE.2002.1027796.
- [23] Jérôme Kunegis. “KONECT: The Koblenz Network Collection”. In: *Proceedings of the 22nd International Conference on World Wide Web. WWW ’13 Companion*. Rio de Janeiro, Brazil: Association for Computing Machinery, 2013, pp. 1343–1350. ISBN: 9781450320382. DOI: 10.1145/2487788.2488173. URL: <https://doi-org.e.bibl.liu.se/10.1145/2487788.2488173>.

- [24] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. “Graph Evolution: Densification and Shrinking Diameters”. In: *ACM Trans. Knowledge Discovery from Data* 1.1 (2007), pp. 1–40.
- [25] Yi Lin, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. “Rust as a Language for High Performance GC Implementation”. In: *SIGPLAN Not.* 51.11 (June 2016), pp. 89–98. ISSN: 0362-1340. DOI: 10.1145/3241624.2926707. URL: <https://doi.org/10.1145/3241624.2926707>.
- [26] Julian McAuley and Jure Leskovec. “Learning to Discover Social Circles in Ego Networks”. In: *Advances in Neural Information Processing Systems*. 2012, pp. 548–556.
- [27] John McCarthy. “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”. In: *Commun. ACM* 3.4 (Apr. 1960), pp. 184–195. ISSN: 0001-0782. DOI: 10.1145/367177.367199. URL: <https://doi.org/10.1145/367177.367199>.
- [28] M. M. Michael. “Hazard pointers: safe memory reclamation for lock-free objects”. In: *IEEE Transactions on Parallel and Distributed Systems* 15.6 (June 2004), pp. 491–504. ISSN: 2161-9883. DOI: 10.1109/TPDS.2004.8.
- [29] M. M. Michael. “Hazard pointers: safe memory reclamation for lock-free objects”. In: *IEEE Transactions on Parallel and Distributed Systems* 15.6 (June 2004), pp. 491–504. ISSN: 2161-9883. DOI: 10.1109/TPDS.2004.8.
- [30] Zachary Painter, Christina Peterson, and Damian Dechev. “Lock-Free Transactional Adjacency List”. In: *Languages and Compilers for Parallel Computing*. Ed. by Lawrence Rauchwerger. Cham: Springer International Publishing, 2019, pp. 203–219. ISBN: 978-3-030-35225-7.
- [31] M. B. R. Pandit and N. Varma. “A Deep Introduction to AI Based Software Defect Prediction (SDP) and its Current Challenges”. In: *TENCON 2019 - 2019 IEEE Region 10 Conference (TENCON)*. 2019, pp. 284–290.
- [32] Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V. T. Rajan. “An Efficient On-the-Fly Cycle Collection”. In: *ACM Trans. Program. Lang. Syst.* 29.4 (Aug. 2007), 20–es. ISSN: 0164-0925. DOI: 10.1145/1255450.1255453. URL: <https://doi.org/10.1145/1255450.1255453>.
- [33] Girish Maskeri Rama and Avinash Kak. “Some Structural Measures of API Usability”. In: *Softw. Pract. Exper.* 45.1 (Jan. 2015), pp. 75–110. ISSN: 0038-0644. DOI: 10.1002/spe.2215. URL: <https://doi.org/10.1002/spe.2215>.
- [34] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S. McKinley. “Taking off the Gloves with Reference Counting Immix”. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’13. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 93–110. ISBN: 9781450323741. DOI: 10.1145/2509136.2509527. URL: <https://doi.org/10.1145/2509136.2509527>.
- [35] Lovro Šubelj and Marko Bajec. “Robust Network Community Detection Using Balanced Propagation”. In: *Eur. Phys. J. B* 81.3 (2011), pp. 353–362.
- [36] Dan Tamir, Oleg V. Komogortsev, and Carl J. Mueller. “An Effort and Time Based Measure of Usability”. In: *Proceedings of the 6th International Workshop on Software Quality*. WoSQ ’08. Leipzig, Germany: Association for Computing Machinery, 2008, pp. 47–52. ISBN: 9781605580234. DOI: 10.1145/1370099.1370111. URL: <https://doi.org/10.1145/1370099.1370111>.
- [37] Paul R. Wilson. “Uniprocessor Garbage Collection Techniques”. In: *Proceedings of the International Workshop on Memory Management*. IWMM ’92. Berlin, Heidelberg: Springer-Verlag, 1992, pp. 1–42. ISBN: 354055940X.

- [38] M. F. Zibran, F. Z. Eishita, and C. K. Roy. “Useful, But Usable? Factors Affecting the Usability of APIs”. In: *2011 18th Working Conference on Reverse Engineering*. 2011, pp. 151–155.



A Glossary

A.1 Abbreviations

BC Betweenness Centrality.

BFS Breadth-First Search.

CAS Compare-And-Swap.

CC Connected Components.

EBR Epoch-Based Reclamation.

HPBR Hazard-Pointer-Based Reclamation.

KONECT the Koblenz Network Collection.

LFRC Lock-Free Reference Counting.

OLS Ordinary least squares.

PR Page Rank.

QSBR Quiescent-State-Based Reclamation.

RAII Resource Acquisition Is Initialization.

RTSJ Real-Time Specification for Java.

SSSP Single-Source Shortest Path.

TC Triangle Counting.