

Bachelor Thesis

Hi-C interaction matrix correction using ICE in Rust

Felix Karg

Examiner: Prof. Dr. Backofen

Advisers: Joachim Wolff, Dr. Mehmet Tekman

Albert-Ludwigs-University Freiburg

Faculty of Engineering

Department of Computer Science

Chair for Bioinformatics

July 10th, 2019

Writing Period

10.04.2019 – 10.07.2019

Examiner

Prof. Dr. Backofen

Advisers

Joachim Wolff, Dr. Mehmet Tekman

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum

Unterschrift

Abstract

foo

(TODO: Write!)

Zusammenfassung

bar

(TODO: Schreiben!)

Contents

1	Introduction	1
1.1	Task Definition	2
2	Background	3
2.1	Chromosome Conformation Technologies	4
2.1.1	Common steps	4
2.1.2	3-C	6
2.1.3	4-C	6
2.1.4	5-C	6
2.1.5	Hi-C	8
2.1.6	Other methods	8
2.2	HiCExplorer	9
2.2.1	Analysis	9
2.2.2	Visualization	12
3	Related Work	13
3.1	Python implementation	13
3.1.1	Advantages	13
3.1.2	Disadvantages	14
3.2	KR-Algorithm	14
3.2.1	Implementation	15
3.2.2	Advantages	15

3.2.3	Disadvantages	15
4	Approach	17
4.1	Problem Description	17
4.2	Iterative Correction and Eigenvector decomposition (Algorithm) . .	17
4.3	Introducing Rust	19
4.3.1	History	19
4.3.2	Categorization	19
4.3.3	Language Features	20
4.3.4	Code Examples	21
4.3.5	Advantages of Rust	24
4.3.6	Disadvantages of Rust	25
4.3.7	Comparing Rust and Python	26
4.3.8	Comparing Rust with C/C++	26
4.3.9	Choosing the right API to call Rust from Python	27
4.3.10	Integration of Rust in Python	30
4.3.11	Using this Implementation	31
4.4	General Approach	31
4.4.1	Beginning	31
4.4.2	Feasability Testing	32
4.4.3	Implementation of the Algorithm	32
4.4.4	Testing and Bugfixing	33
4.4.5	Idiomatic Rust and summing high fractions	33
4.4.6	Packaging	34
4.4.7	Parallelizing	35
5	Results	37
5.1	Server Specification	37
5.2	Data for Testing	38

5.3	Memory Requirements	38
5.3.1	During Correction	39
5.3.2	Maximum required memory	40
5.3.3	Load times	43
5.3.4	Comparing Memory needs	43
5.4	Runtime length	45
5.5	Multicore benefits	45
5.6	Output	49
6	Conclusion	51
	Bibliography	58

List of Figures

1	Major Chromatin Structures	3
2	Comparison between 3-C and its derived methods	5
3	Summarised Hi-C protocol	7
4	Excerpt of HiCExplorer visualizations	10
5	Models related to HiC-Data	11
6	Memory needs iterating 25kb	39
7	Memory needs iterating 50kb	40
8	Maximum needed Memory	41
9	Matrix loading times	44
10	Algorithm Runtimes	46
11	Multicore comparison	47
12	Multicore Memory comparison	48
13	Multicore computation time comparison	49
14	Plotting corrected matrices	50

List of Tables

1	Runtime Comparison Between Rust/C/C++	26
2	Comparison of different Interfaces between Rust and Python	30
3	Server Specification	37
4	Test Data	38
5	Memory needs comparison	45

1 Introduction

(DRAFT: rewrite. Make more fluent for readers.)

Enhancers are usually searched for within a distance of 1 Mb up or downstream of promoters [1]. Structures such as DNA loops (see Figure 5) can bring together an enhancer with a promoter that is far more distant than 1 Mb however [1]. With Hi-C, interactions over the whole genome [2] and even across genomes can be measured [3]. Computational methods using Hi-C data can identify hundreds of thousands of putative enhancers and their target genes [4].

In this work, data obtained from Hi-C (see Chapter 5 for details) will be used. Hi-C is a method for acquiring 3D-information of genomes. This is done by strapping together parts of the genome that are close by, cutting the genome apart with restriction enzymes, combining the ends of strapped-together fragments and using high throughput methods for sequencing them. This is explained in detail in Section 2.1.5.

Such technologies tend to suffer from biological, e.g. different chromatin states [5], and technical factors, e.g. sequencing and mapping [6], making them inherently inaccurate. Biases are unavoidable, in particular, as some regions are more sensible for biotin labeling enrichments (See Section 2.1.5 why this is relevant) they will be measured more often when compared to others. PCR artifacts may be one of the reasons [7]. Mapping locations may be unclear or not unique, introducing even more sources for possible biases. Sequencing methods have certain biases themselves. Some of the measured interactions are questionable, it is unclear if these are actual,

spatially close points, or if it simply is a technical error or a randomly happened interaction.

However, a basic but strong assumption about the structure of the genome can be made, which is that every location has the same amount of interactions with other locations as every other location. The data does not show this due to the several aforementioned inaccuracies. Algorithms such as ICE [8] (Iterative Correction and Eigenvector decomposition, Section 4.2) or KR [9] (Knight-Ruiz, Section 3.2) can be applied to normalize the matrix nonetheless.

1.1 Task Definition

In the three-dimensional space of a cell the DNA forms a structure that is distributed all over the place. Obviously, there exist many points of contact in the DNA, they even form noticable structures, such as DNA loops. Many measured points of contact are random interactions or measurement errors that need to be corrected. For this task, a Python implementation exists but is limited for high resolution data due to high memory usage. This thesis aims to reimplement a more resource efficient method in Rust.

The main goals include testing the integration between Rust, a systems programming language recently gaining in popularity (details in Section 4.3), and Python, how easy it is to let both of these languages interact, and how it compares to the other two current implementations, ICE in Python and KR in C++. More information about them can be found in Section 3.1 and Section 3.2 respectively. The overall goal is to try to implement a more resource efficient version, able to make effective use of parallel computing.

2 Background

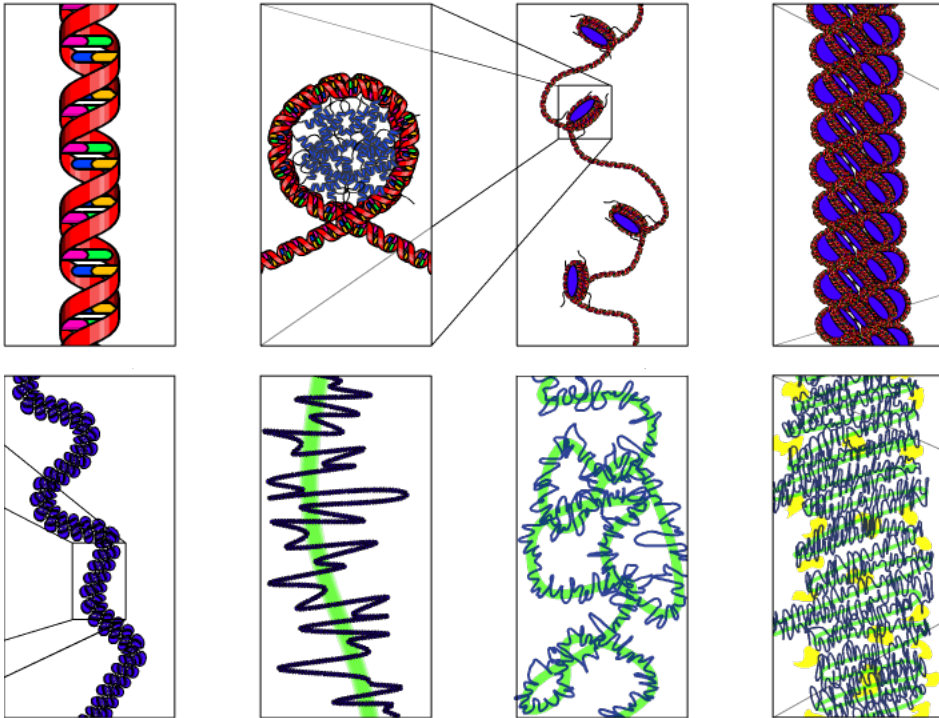


Figure 1: Major Chromatin Structures.

Note that not all are listed, and considerably different structures exist during cell division as well. These structures are representative most of the time.

Image adapted from [10].

Chromatin describes different levels of DNA organization. The well-known double-helix is only the lowest of several structural layers (major chromatin structures are shown in Figure 1). Looking at it from the outside i.e. the highest structural layer,

DNA is organized using different scaffolding proteins. With the help of chromosome conformation technologies the actual three dimensional structures can be confirmed.

2.1 Chromosome Conformation Technologies

2.1.1 Common steps

As can be seen in Figure 2, the first steps, crosslinking, digestion, ligation and the reversal of crosslinking, are the same for all 3-C-based methods.

Cross-linking DNA: The first step is to cross-link DNA strands that are close to each other spatially (see Figure 2 or Figure 3 for reference). This is done by adding formaldehyde, which connects (links) sufficiently close strands together.

A chromatin cross-link is two entirely different parts of the genome held together by a chemical bond with formaldehyde. This process cannot be specifically controlled, so only regions near each other are connected, but not necessarily all regions that are known to be spatially close.

Digestion: The next step is cutting the DNA apart in intervals. For this, restriction-enzymes i.e. restriction endonuclease are used. Commonly used enzymes for this are DpnII, NcoI or HindIII [2]. This will result in a lot of cross-linked fragments, as well as not-cross-linked ones.

Ligation: After reducing the concentration of fragments, DNA ligase is added, to ligate, i.e. weld together, dangling fragment ends. For this a reduction in concentration is done, resulting in favouring the ligation of fragments linked together by formaldehyde. In HiC, Biotin is added in this step to mark ligated fragments. This allows filtering out most fragments that have not been ligated in a later step.

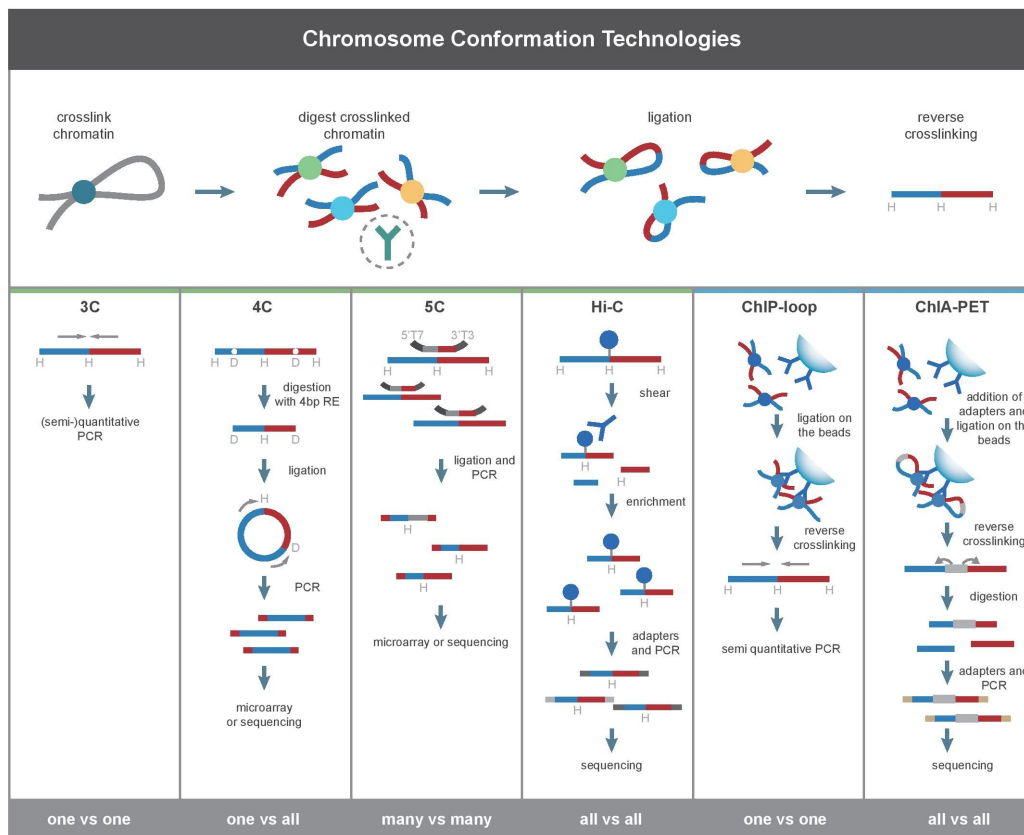


Figure 2: Comparison between 3-C and its derived methods. Clearly seen can be why 3-C, 4-C, 5-C and Hi-C are commonly referred to as 3-C-based techniques.

Image from [11].

Reverse Cross-links: Adding a high concentration of salt for some time will reverse the cross-linking through formaldehyde, leaving the originally spatially close fragments ligated and with a biotin-marker in the case of Hi-C. The reversal of cross-links results in DNA fragments with parts from two regions that may have been far away from each other.

Note that at this point, the fragments are too long for sequencing. Most current sequencing methods can only sequence fragments with a few hundred base pairs length, but the ligated fragments are usually longer than that.

2.1.2 3-C

In 2002 Dekker et al. [12] developed a method to test for interactions between a single pair of genomic loci. Candidates for promoter-enhancer interactions can be tested using this method. In the original method, (semi-) quantitative PCR is used for sequencing, which has been superseded by modern sequencing methods.

2.1.3 4-C

Chromosome conformation capture-on-chip (4C) was developed in 2006 by Simonis et al., Zhao et al. [13] [14], a method to test interactions between one genomic location with all others. This is done by adding a second ligation-step (see Section 2.1.1 or Figure 2 for reference) creating loops from the DNA fragments and applying inverse-PCR. This is a method to specifically amplify unknown sequence parts when both beginning and ending parts are known. For this the loops are cut within the known section, followed by sequencing. Microarray describes a sequencing method no longer used. Since due to inverse-PCR knowledge about both interacting chromosomal regions is not needed, results are highly reproducible for close regions (**TODO: find source for this**).

2.1.4 5-C

Chromosome conformation capture carbon copy (5C) was developed in 2006 by Dostie et al. [15], this method is able to test a region for interactions with itself, such region being no bigger than a megabase. This is done by adding universal primers to all fragments from such a region. 5-C has relatively low coverage, but is useful to analyse complex interactions of specified loci of interest. Genome-wide interaction measuring would require millions of 5C primers, making this method unsuitable. Microarray describes a sequencing method no longer used.

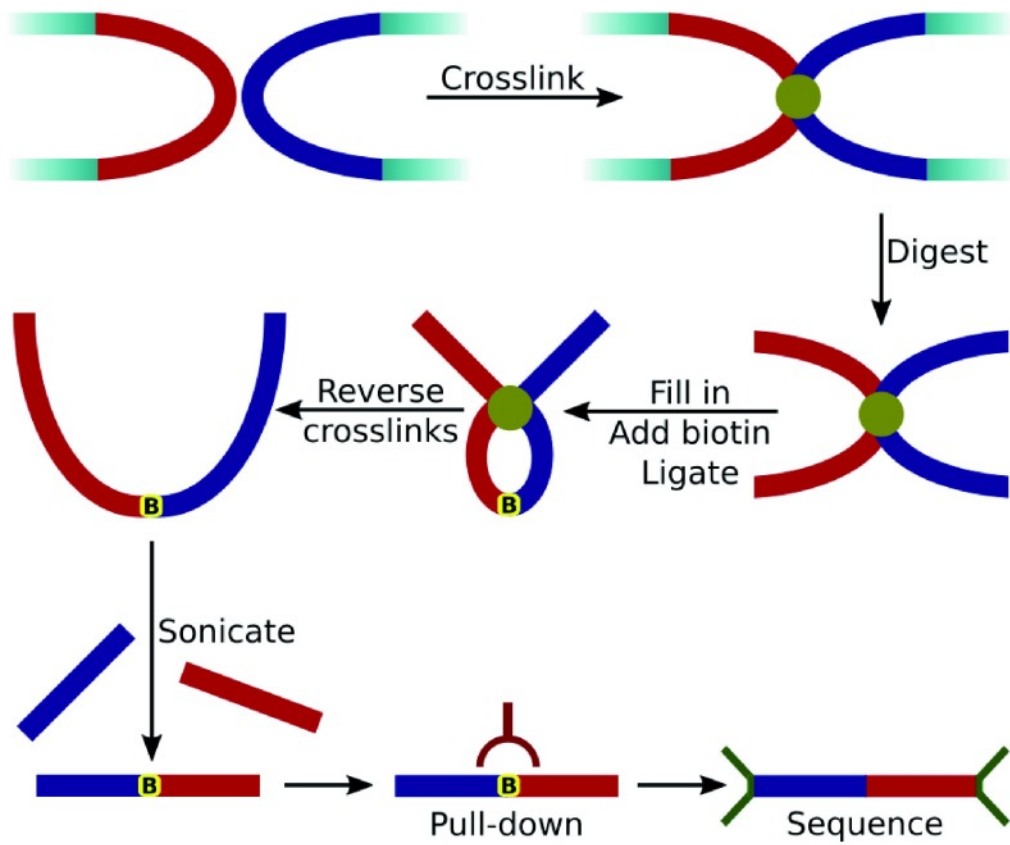


Figure 3: Summarised Hi-C protocol.

Biotin is shown by a yellow marker, while the red and blue parts are different parts of cross-linked fragments. The steps are explained in detail in Section 2.1.1 and Section 2.1.5.

Image adapted from [7].

2.1.5 Hi-C

Hi-C (as shown by Figure 3) was developed in 2009 by Liebermann-Aiden et al. [2]. After the common steps noted earlier, unique to Hi-C is the following sequence of sonication, pulldown (filtering based on biotin markers) and sequencing.

Sonication: Putting the ligated DNA-fragments under the influence of ultrasonic waves is breaking them apart in much shorter fragments (due to long sequences not being able to absorb frequent shocks well), shearing them apart in sequences short enough to enable sequencing.

Filtering and Removal of Biotin: Pulling-down of fragments marked with biotin leaves only those marked with Biotin during earlier ligation (see Section 2.1.1). Subsequently the marker is removed, as it would hinder further sequencing.

Sequencing: Sequencing, short for DNA sequencing, describes processes of measuring a DNA sequence. There are several techniques for doing this, most use PCR (Polymerase Chain Reaction) before or while sequencing, which duplicating fragments several times, allowing them to be sequenced more accurately.

2.1.6 Other methods

As can be seen in Figure 2 other methods, such as ChIP-loop or ChIA-PET exist. They are different from the digestion step onwards, with their subsequent steps using immunoprecipitation (antibodies) for filtering. As they hold no further significance for this thesis, they will not be covered further.

2.2 HiCExplorer

HiCExplorer [16] is a software to process, analyze and visualize Hi-C data. Part of this software are tools to build the interaction matrix, convert between formats, correcting the data (which this is work is part of), analysing it in various ways or extensively plotting it. Facilitated is, among others, the creation of contact matrices, detection of topologically associating domains (TAD) and A/B compartments, merging of matrices, and detection of long-range contacts¹. Those contact matrices may then be visualized, and other data tracks may be added. This **(DRAFT: this is about pyGenomeTracks, add info)** includes annotated genes, compartments, ChIP-seq coverage tracks, viewpoints and long range contacts¹. An excerpt of possible visualizations can be seen in Figure 4. HiCExplorer can be installed on Linux and MacOS.

2.2.1 Analysis

Corrected Hi-C data can then be further analysed, with `hicFindTADs` one can search for TADs (topologically associated domains) [17], for this a TAD-separation score is computed and local minima indicative of TAD boundaries are searched for. A visualized result of such a computation can be seen in Figure 4I (created with `hicPlotTADs`).

DNA is compartmentalized [2] in different domains, a model for this can be seen in Figure 5. They can be found by computing the Eigenvectors as described in [2] or [8] first by using `hicPCA`. With this, a better understanding can be achieved when additionally visualized. Useful metrics include difference, ratio and log2ratio between two matrices. For this, `hicCompareMatrices` can be used. Replications or samples from different conditions can easily be compared when visualized. More information about the analysis capabilities of HiCExplorer can be found in [16].

¹<https://github.com/deeptools/HiCExplorer>, accessed 2019-06-26

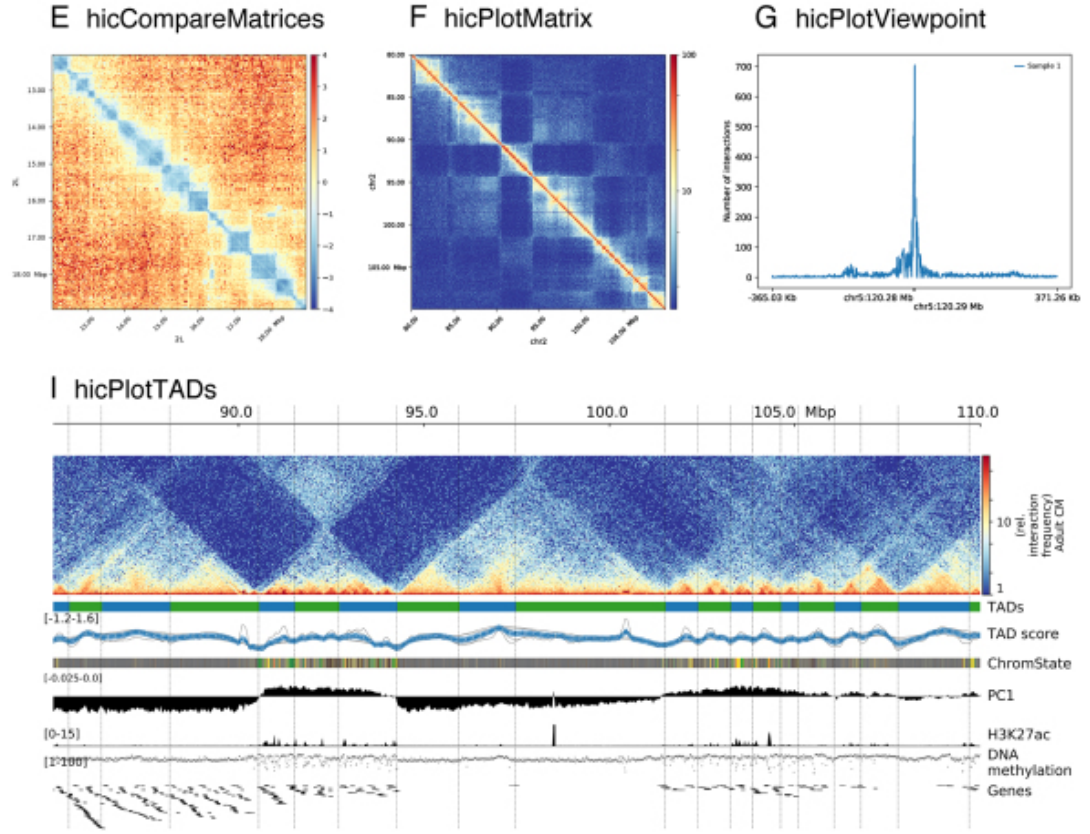


Figure 4: Excerpt of HiCEXplorer visualizations **E)** Pixel difference computed using hicCompareMatrices and visualized using hicPlotMatrix of a Hi-C corrected matrix for wild type condition and knock down. **F)** Plot of a 80 to 105 Mb region contact matrix of chromosome 2 in log scale. **G)** Corrected number of Hi-C contacts shown using hicPlotViewpoint, for a single bin in chromosome 5 (output similar to 4C-seq). **I)** Human chromosome 2 visualization (region 85-110 Mb) using tracks from different tools found in the HiCEXplorer toolbox (primarily TAD-related information).

Image adapted from [16].

Hi-C Matrices and Models

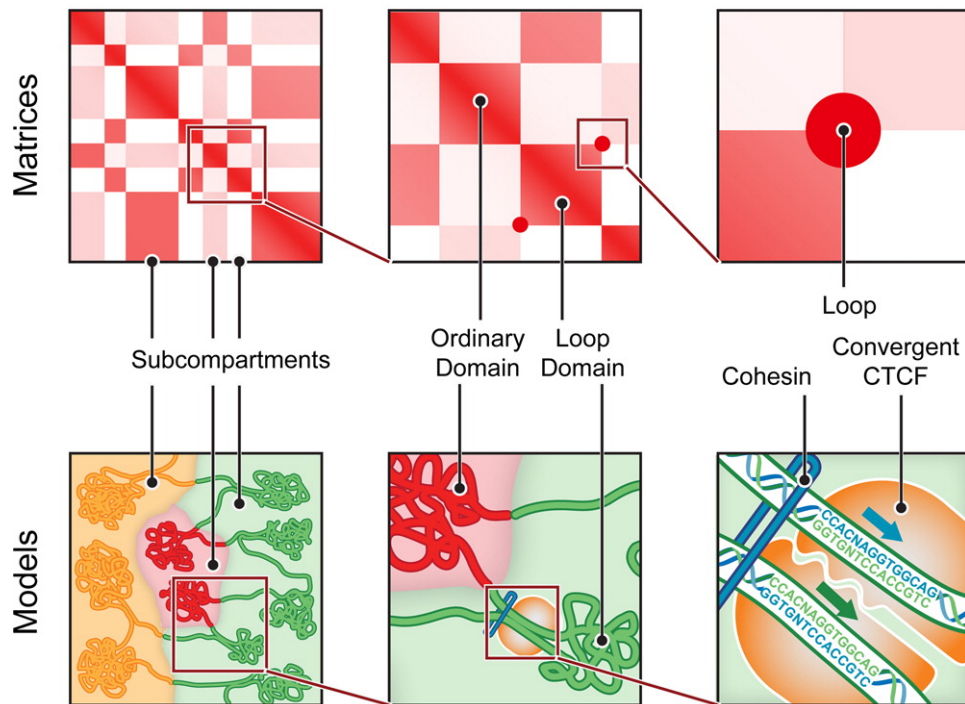


Figure 5: Models related to HiC-Data.
Image from [3].

2.2.2 Visualization

Visualizations are necessary when the goal is to understand complex structures fast or even at all. It is impossible look at rows of numbers and notice that one significantly higher value, which immediately catches the eye as a heatmap.

Basic plotting of contact matrices (as seen in Figure 4F) can be done using `hicPlotMatrix`. Options include region, color and value ranges, as well as plotting A/B compartments or other additional data when added. `hicPlotViewpoint` can visualize the number of interactions around a specific reference region or point in the genome (see Figure 4G).

Computed TADs (as described in Section 2.2.1, see Figure 4I) can be plotted using `hicPlotTADs`. This tool can plot multiple matrices and additional data. Contact matrices are rotated by 45°, and TADs are marked with triangles. The colormap, different ways for visualization, and several configurations to plot coverage tracks can be selected.

More information about plotting with HiCEXplorer can be found in [16].

3 Related Work

The main work is the implementation in Rust as well as the testing of the integration with Python. Related work includes the original implementation in Python as well as the recent implementation of the KR-algorithm in C++. Disadvantages and advantages of the respective implementations will be evaluated in the following.

The description of the implemented algorithm can be found in Section 4.2.

3.1 Python implementation

The original implementation was written in Python, since HiCExplorer is written in Python. This implementation is using common python dependencies extensively, including the compressed sparse row matrix (CSRMatrix) implementation from `scipy`, as well as the scientific number manipulation library `numpy`.

3.1.1 Advantages

The implementation itself is considerably short, the file having only 86 Lines, including imports and frequent comments. The advantage of using Python here is showing, as most lines are not for functionality itself, but for timing, logging and ease of

debugging. With the iteration itself starting no earlier than Line 40, most are High-Level `numpy` / `scipy` commands, some being themselves implemented in C/C++ to be sufficiently fast.

Another advantage of Python in general are fast implementation times, which are possible through the concise syntax making the spotting of mistakes easier.

3.1.2 Disadvantages

The downsides of this implementation being that datastructures in Python are extensively objectified, meaning they require more working memory, and that even though Python has existing parallelism, a global interpreter lock (python is only interpreted usually, but this still holds for compiling with cpython) prevents multiple threads to use the same parts of code and memory without duplication. Since Python already has comparatively high memory requirements (**TODO: link high memory needs**), it is not practicable to add the same amount for every further core.

For reference, the Python-implementation can be found here¹.

3.2 KR-Algorithm

What follows is a short description to the Algorithm known as Knight-Ruiz from [9]. Essentially, the algorithm is computing the same steps, but it is taking advantage of conjugate gradients, thus needing potentially less iterations.

¹<https://github.com/deeptools/HiCExplorer/blob/master/hicexplorer/iterativeCorrection.py>, accessed 2019-06-26

3.2.1 Implementation

The KR-algorithm was originally implemented in Matlab, here we compare with a version in C++. Calls from Python to C++ can be done over the C-API with considerable help through Python-header files.

3.2.2 Advantages

A commonly mentioned advantage of C++ is the speed of execution, and fine-grained control over Memory available. However, implementations in C++ can be several orders of magnitude faster than their respective implementation in Python. An advantage of the Algorithm itself is that as long as the matrix itself has total support (meaning that at least one diagonal has only positive nonzero values, this can be artificially done setting zeros to some small positive value), it will converge. Thus, it will converge for way more matrices than the ICE-algorithm.

3.2.3 Disadvantages

Even though the execution is fast, the development process tends to be slow. This is due to the free memory control, which is hard to get right as this requires upholding of implicit assumptions at several places. As these assumptions are implicit only, it is easy to forget them or 'cut corners' when not possible. Those bugs leading to Segmentation-faults (accessing invalid memory) are notoriously hard to find, as they do not follow determinism. Parallelism is even harder to add, since data races (also nondeterministic) and other sources for hard-to-get right problems are added. Additionally, the syntax is considerably complex, making it rather hard to understand. For reference, the implementation of the KR algorithm can be found here².

²<https://github.com/deeptools/Knight-Ruiz-Matrix-balancing-algorithm>,
accessed 2019-06-26

4 Approach

4.1 Problem Description

As noted in Section 1.1, the main goals include testing the integration of Rust within Python, by implementing a counter-version to the original Python implementation of the iterative part of the ICE algorithm, and then comparing it with the original Python-implementation as well as the recent implementation of the KR-algorithm in C++. It will be tested if the memory efficiency can be improved, also how well the parallelization using Rust works really, and how the integration from Python to Rust works. For C/C++ there exist the Python headers, and extensive support from Pythons package manager `pip` and common packaging tools like `setuptools`. For Rust, as it turns out, the support using Python headers is not as easy, and the support for building packages from Pythons side is early at best. Both of this will be covered in Section 4.3.10 and Section 4.3.9 respectively.

4.2 Iterative Correction and Eigenvector decomposition (Algorithm)

The ICE algorithm was proposed by Imakaev et al. 2012 [8], described in detail in their supplementary material, and defines an iterative correction approach as follows.

The goal is to obtain the the vector of biases B_i and the true contact map T_{ij} with their relative contact probabilities. This is done by explicitly solving the system of the following two equations:

$$O_{ij} = B_i B_j T_{ij} \quad (1)$$

$$\sum_{i=1, |i-j|>1}^N T_{ij} = 1 \quad (2)$$

Equation (1) is stating, that when applying B back again on our corrected matrix T_{ij} , it will be the same as the original matrix O_{ij} again. Equation (2) states, that the sum over the corrected matrix, over arbitrary elements in the upper left triangle, but only one from each column, sums up to one. T_{ij} is doubly stochastic ($\forall_j \sum_{i=1}^N T_{ij} = 1$ and $\forall_i \sum_{j=1}^N T_{ij} = 1$), **(EXTEND: does it say that it is the same? why is this actually valid?)**

In the algorithm, this is achieved in the following way. First, W_{ij} , a copy of O_{ij} is created. This matrix will converge to T_{ij} during the iterative process. The elements of B are initialized with 1.

$$S_i = \sum_j W_{ij} \quad (3)$$

$$\Delta B_i = S_i / \text{mean}(S) \quad (4)$$

Each iteration starts by first calculating the coverage by summing up each row (or column, matrix is symmetric so this does not matter) (Equation (3)) and additional biases based on this by dividing them through their own mean (Equation (4)).

$$W_{ij} = W_{ij} / \Delta B_i \Delta B_j \quad (5)$$

$$B_i = B_i \cdot \Delta B_i \quad (6)$$

Then W_{ij} is iterated by dividing by $\Delta B_i \cdot \Delta B_j$ (Equation (5)), after which B_i is iterated by multiplying with the current biases (Equation (6)). W_{ij} accumulates divisions by ΔB_i , just as B_i accumulates the products of ΔB_i . This is repeated until the variance of ΔB becomes negligible, at which point W_{ij} has converged to T_{ij} .

4.3 Introducing Rust

4.3.1 History

Rust started out 2006 as a personal project of Graydon Hoare, a Mozilla employee [18]. The Mozilla foundation started sponsoring in 2009 [18]. The first compiler was written in OCaml, but 2011 `rustc` was able to compile itself with the `llvm` backend [19]. Starting with Rust 1.0, which itself got released on May 15, 2015 [20], there was a new stable point version every six weeks [20]. Early on, Rust had frequent breaking changes [21], recently barely anyone had breakage when updating [22].

4.3.2 Categorization

Rust is classified as a high-level language, even though fine low-level control is possible. This is due the high amount of high-level zero-cost abstractions. Rust has a type system with strong guarantees, promising e.g. that all references (pointers) are valid, or thread safety (memory access from other threads does not result in data races / undeterminism). This is possible through concepts such as ownership and lifetimes. Even though one can program in an object oriented way, Rust is primarily not object-oriented. Additionally it is imperative, procedural, generic and functional.

4.3.3 Language Features

Syntax: The concrete syntax seems similar to C/C++ (curly braces, function signatures), however it is more similar to that of ML or Haskell. A particular example for this case are type classes called traits here, similar to C++ templates but inspired from Haskell, supporting polymorphism and generic types. Generic parameters can be constraints, by requiring that generic type to implement a certain Trait.

Memory safety: Rust is designed to be memory safe, and does not permit dangling pointers, null pointers, data races in safe code, or usage of uninitialized variables. In case a Null is needed, the Option-type is provided. Thus, the compiler can guarantee the validity of all references at compile time using its borrow-checker.

Memory management: Rust does not have a garbage collector, instead, the resource acquisition is initialization (RAII) convention is used, with optional reference counting. Resource management is deterministic with very little overhead, favoring stack allocation without implicit boxing. References are not run time counted, as their usage is verified at compile time. with this, memory safety can be guaranteed, limiting possible undefined behaviour tremendously.

Ownership: In Rust, all values have a unique owner, and the scope of the value is the same as the owners. Immutable references can be passed using `&T`, mutable references by `&mut T`. Pass by value works by passing `T`. Only **one** mutable reference can exist at any point, or any number of immutable ones. This is enforced at compile-time.

Borrowing: Borrowing results directly from the concept of ownership. As mentioned, only one mutable borrow (reference) can happen at a time, however that borrowing variable can further borrow it to other variables or functions. The number of immutable borrows is unlimited, meaning there can be multiple references reading but not modifying part of the memory. This is necessary to guarantee memory safety, as only one mutable reference can write to it at any point in time, wherever that is (in the code).

Lifetimes: Lifetimes are the simple concept of keeping track how long each variable and each reference is alive, this is preventing the simple case of variables going out of scope but returning a pointer to it. The compiler can keep track of this in even much more complex environments. Non-Lexical-Lifetimes¹ also work together with borrowing, resulting in variables returning their borrow before the end of the scope, as can be see in the first example of Section 4.3.4.

Tooling: The reason Rust is loved [23] this much is at least partly due to tooling. This includes the dedicated package manager `cargo`, the linter `rustfmt` or `cargo-fix` (a subcommand that can be added later), that format Rust code after predefined guidelines, or fix most compiler lint warnings automatically and upgrade to newer conventions, respectively.

More information about Rust can be found here².

4.3.4 Code Examples

Demonstrating Ownership and Borrowing: By executing Code Example 1 the output from Output 1 will be returned.

¹Introduced in version 1.31 for the 2018-edition, and 1.36 for the 2015-edition. Before that, Code Example 2 would not compile.

²<https://www.rust-lang.org/>, accessed 2019-06-26

Code Example 1

```
1 fn main() {
2     let mut v = vec![];    // ---| v owns the (empty) vector
3     v.push("Hello");       // <--| vector gets first element
4                             //      |
5     let x = &v[0];         // -| | x borrows the first element from v
6     v.push("world");       // <X-| v cannot mutably borrow the vector
7                             //      | | while x has immutably borrowed it
8     println!("{}", x);     // -| | x needed at least until here
9 }
```

Output Nr. 1

```
error[E0502]: cannot borrow `v` as mutable, it is also borrowed as immutable
--> src/main.rs:5:5
|
5 |     let x = &v[0];
|               - immutable borrow occurs here
6 |     v.push("world");
|     ~~~~~ mutable borrow occurs here
7 |
8 |     println!("{}", x);
|               - immutable borrow later used here
```

As the compiler is complaining, `v` needs a *mutable* borrow to modify `v`, however `x` still has an *immutable* borrow! The borrow from `x` cannot be ended yet, because it should be printed later. As the mutable borrow from `v` could modify it in a way such that the reference `x` would be invalid (e.g. delete `v`), this is a potential memory safety problem. However it is fine to print `x` first, and modify `v` afterwards. In Output 2 can be seen what happens when printing the second element of `v` instead of `x` in the

last line, and printing `x` before adding the second element of `v` (as shown in Code Example 2).

Code Example 2

```
1 fn main() {  
2     let mut v = vec![];    // ---| v owns the (empty) vector  
3     v.push("Hello");       // <--| vector gets first element  
4                             //    |  
5     let x = &v[0];         // -| | x borrows the first element from v  
6     println!("{}", x);     // -| | x needed only until here  
7                             //    | x returns the borrow here  
8     v.push("world");       // <--| v can now modify the vector  
9                             //    | (mutable borrow needed)  
10    println!("{}", v[1]);   // <--| v can be printed without trouble  
11 }                          // ---| x, v going out of scope
```

Output Nr. 2

```
Hello  
world
```

Demonstrating Ease of Parallelization: Due to the strong guarantees from the compiler, Memory Safety can be extended to thread safety. In Code Example 3 the function `heavy_operation` is applied to every element in the list. For this, over the list `somelist` is being iterated, and `heavy_operation` mapped over by taking the values from the map-closure - Closures are comparable with lambda-functions from Python, in that they can take arguments and are unnamed functions.

Code Example 3 and Code Example 4 demonstrate how easy it is to turn non-parallel code (Code Example 3) in parallelized code (Code Example 4). The difference here

being the imported `rayon::prelude::*` and instead of `iter` now applying `par_iter` to the original list.

Code Example 3

```
1 let otherlist = somelist.iter()
2             .map(|&v| heavy_operation(v))
3             .collect();
```

Code Example 4

```
1 use rayon::prelude::*;
2
3 let otherlist = somelist.par_iter()
4             .map(|&v| heavy_operation(v))
5             .collect();
```

4.3.5 Advantages of Rust

In General: As seen in Section 4.3.3 and Section 4.3.4, Rust has several high-level features ready to use, supporting the developer tremendously. Strong compiler guarantees allow easy parallelization and using Rust libraries without worry, since the compiler will complain if it used wrong. In combination with semantic versioning³, this allows adding and using dependencies without worry. The result are many small libraries which depend upon each other, instead of few big ones. Dependencies upon one hundred Rust libraries are not uncommon, and the strong guarantees from the compiler enforce correct usage.

³<https://semver.org/>, accessed 2019-06-26

For this project: Even though high modularity exists, the Rust ecosystem is comparatively young. meaning even though many libraries exist, they are not as complete as their Python/C/C++ counterparts. Since no implementation of CSRMatrix (compressed sparse row matrix) having the required features, even though several existed, it was implemented again, adding the one necessary feature, and not implementing any other. This has the advantage of being a very specific solution, possibly faster and smaller than the general ones available.

Also, as seen in [\(TODO: Link section results\)](#), Parallelization can decrease the elapsed computation time.

4.3.6 Disadvantages of Rust

In General: General disadvantages of Rust include the young ecosystem with slightly less diversity, or too much feature-incomplete diversity. The steep learning curve in the beginning, needs to be mentioned, since Rusts features cannot be selectively activated. Compared to languages like GO, Rust has considerably higher initial compile times. Even though breakage rarely happens [22], a new point-release happens every six weeks, frequently introducing new features requiring time to fully understand. The userbase is still growing, and many features frequently used in other languages, such as `async` or specializations, are not yet available for users of the stable compiler. [\(DRAFT: steep early learning curve of Rust mentioned in the disadvantages?\)](#)

For this project: In particular, the unavailability of a CSRMatrix implementation with the needed features is concerning. Thus, the current implementation does not have any more features than are needed for the current algorithm, being only a tiny subset of the features provided by the `scipy` implementation.

4.3.7 Comparing Rust and Python

Rust and Python are two quite different programming languages, a direct translation is not possible. Both implementations are the same semantically, however details differ. Since Rust has a much finer control of memory and the applying of functions to data structures, some operations have been explicitly separated while others have been combined.

Depending on the questions asked, either language may prevail. While Python allows (seemingly) faster development cycles, it is more prone to runtime errors and library misuse. For Rust, the compiler provides strong guarantees, requiring more development time up front but less to fix bugs.

4.3.8 Comparing Rust with C/C++

Speed comparison	C	Rust	C++
n-body	7.49	5.72	8.18
binary-trees	3.48	3.15	3.79
pidigits	1.75	1.75	1.89
reverse-complement	1.78	1.61	1.55
spectral-norm	1.98	1.97	1.98
fannkuch-redux	8.61	10.23	10.08
k-nucleotide	5.01	5.25	3.76
fasta	1.36	1.47	1.33
mandelbrot	1.65	1.96	1.5
regex-redux	1.46	2.43	1.82
Fastest in:	3/10	4/10	4/10

Table 1: Comparing Runtime speeds. Runtime measured in seconds. Numbers from the benchmarksgame⁴.

⁴ Rust comparison with C: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html>, and with C++: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-gpp.html>, both accessed 2019-06-26

As can be seen in Table 1, the runtime of Rust is very close to that of C and C++ (as is memory, not compared here). Both C and C++ are currently much more widely used, but there are already voices calling to replace C++ with Rust⁵.

Resource needs may be close to the same, but from a developer standpoint Rust has consistently been the ‘most loved Language’ [23] for the last four years, whereas both C and C++ both rank considerably high in the category ‘dreaded’ [23]. Reasons why Rust may be such a loved language are listed in Section 4.3.5. Due to the barely integrated external static analysis done for C/C++-code, both C and C++ are more prone to memory bugs [24], thus having higher developer time requirements.

4.3.9 Choosing the right API to call Rust from Python

There are three main ways to execute Rust code from Python. In the following, the feasibility of them for this thesis is evaluated. And compared in Table ??.

rust-cpython: One common way is rust-cpython. This library requires Rust 1.25 or higher, the current stable version at the time of writing is 1.33. Rust-cpython grants access to the Python gil (global interpreter lock) with which Python code can be evaluated and Python objects modified. The resulting library can easily be

⁵<https://hub.packtpub.com/will-rust-replace-c/>, accessed 2019-06-26

Code Example 5

```
1  #[macro_use]
2  extern crate cpython;
3
4  use cpython::{PyResult, Python};
5  // add bindings to the generated python module
6  // N.B: names: "librust2py" must be the name of
7  // the `.so` or `.pyd` file
8  py_module_initializer!(librust2py,
9      initlibrust2py, PyInit_librust2py, |py, m| {
10     m.add(py, "__doc__",
11         "This module is implemented in Rust.")?;
12     m.add(py, "sum_as_string",
13         py_fn!(py, sum_as_string_py(a: i64, b:i64)))?;
14     Ok(())
15 });
16 // logic implemented as a normal rust function
17 fn sum_as_string(a:i64, b:i64) -> String {
18     format!("{}", a + b).to_string()
19 }
20 // rust-cpython aware function. All of our python
21 // interface could be declared in a separate module.
22 // Note that the py_fn!() macro automatically converts
23 // the arguments from Python objects to Rust values;
24 // and the Rust return value back into a Python object.
25 fn sum_as_string_py(_: Python, a:i64, b:i64)
26     -> PyResult<String>
27 {
28     let out = sum_as_string(a, b);
29     Ok(out)
30 }
```

imported into Python after renaming the compiled library. Native Rust code requires some wrapping first, as can be seen in Code Example 5. The code is taken from the `rust-cpython` example which can be found here⁶. **(TODO: correctly align!!)**

This wrapping, though quite common and based on the Python C-API makes it hard to write purely idiomatic Code in Rust. Since Python is directly affected, the interactions with Python need to be considered while writing Rust-Code, including the affecting of memory Python is managing. This was deemed too much complexity overhead.

pyO3: Another common approach is using the `pyO3`-library, which started off as a fork of `rust-cpython`, but has since seen drastic changes. However, as `pyO3` continues to use unstable Rust features, it is only possible to compile this library with the nightly version of the compiler. Even though most unstable features have been stabilized by now, Specialization⁷ is still missing several steps, as it is not sound regarding the type system yet, and not all questions are answered. Nightly features are subject to tremendous change, making this option a questionable at best for this implementation, as the goal is to have a stable implementation.

Generate dylib: The described way in the official rust docs is to create a `dylib` and import the resulting library in the respective language dynamically⁸. Here is no renaming necessary, but the communication between Rust and Python is more low-level. The main wrapper is on the side of Python, transforming Arguments to pointers and C-Representations. Rust needs to export an interface usable from C, which can be done by a simple `extern`. Rust has the `\#[no_mangle]` and `\#[repr(C)]` (procedural) macros, preventing the compiler to mangle (renaming of functions) and

⁶<https://github.com/dgrunwald/rust-cpython>, accessed 2019-06-26

⁷<https://github.com/rust-lang/rust/issues/31844>, accessed 2019-06-26

⁸<https://doc.rust-lang.org/1.2.0/book/rust-inside-other-languages.html>, accessed 2019-06-26

guaranteeing the representation in the memory layout to be as it would be in C, when they need to be exported.

API Comparison	rust-cpython	pyO3	dylib
any renaming needed	Yes	Yes	No
stable Rust	Yes	No	Yes
platform-specific compilation flags	Yes	Yes	No
using Memory managed by Python	Yes	Yes	Optional
additional implementation effort	Medium	Medium	Low
difficulty of creating python packages	Easy ⁹	Easy ⁹	Normal
Good in:	2/6	1/6	5/6

Table 2: Comparing different API interfaces.

When comparing the available options as can be seen in Table 2, dylib appears to be the best option. See Section 4.4 for details regarding the communication between Python and Rust. **(TODO: link exact paragraph)**

4.3.10 Integration of Rust in Python

The integration of both C and C++ in Python are considerably straightforward and well-supported. Even though Rust is considerably new, a multitude of options exists, especially for both `rust-cpython` and `pyO3`. However, none fully offered what was needed for packaging Python with a dylib, and building it. `setuptools-rust`¹⁰ sounded promising, but requires `rust-cpython` or `pyO3` bindings to work correctly. A different `setuptools` extension called `milkshake`¹¹, a package specifically for the distribution of dynamically linked libraries with Python, promised to be capable of including both our library and the python-wrapper for it. This promise however turned out to be undocumented.

⁹This was not tested, but `pyO3-pack` is a zero-configuration package builder, <https://github.com/PyO3/pyO3-pack>, accessed 2019-06-26.

¹⁰<https://github.com/PyO3/setuptools-rust>, accessed 2019-06-26

¹¹<https://github.com/getsentry/milkshake>, accessed 2019-06-26

The implementation now uses `milksnake` to resolve dependencies like the not-yet compiled library, but the `setuptools` mechanisms for including the library itself and the Python wrapper.

4.3.11 Using this Implementation

Installation

(TODO: for installing using conda add dependencies)

`smb` can be run on any Unix-based operating system (tested using ubuntu-18.04) with Conda, Python and common development packages, e.g. `libopenssl-dev`, `python3-dev`, `build-essential` and the like installed. For the installation itself just enter `conda install -c kargf smb`.

For using, not building, installation of Rust is not needed.

Build

Detailed build instructions can be found in the corresponding GitHub repository: <https://github.com/fkarg/HiC-rs>. For this, an installation of Rust, as well as `conda` and `pip` are needed.

4.4 General Approach

4.4.1 Beginning

In the beginning, after consulting related material ([8], [2] and [7]), the Python-implementation was studied for details. Then, the feasibility of communicating

between Rust and Python was tested first, see Section 4.3.9 for the selection of the API. For this, small examples passing a simple list back and forth were implemented.

4.4.2 Feasability Testing

Next, the general feasibility of the project was tested, this includes looking for usable libraries. The Python-implementation showed a strong dependency for `numpy` and `scipy`, but nothing similar was available for Rust. There are several good linear algebra libraries, and ports of `numpy`, however no CSRMatrix implementation providing usable iteration over its rows. There existed at least three different CSRMatrix implementations at the time of research, however as all of them were considerably lacking in features compared to the `scipy` implementation and not many features were needed, it was implemented separately with only the needed features, including being constructed through a C-API interface, and no effort was made to add the needed feature to any of the other implementations. During the implementation, tests for the CSRMatrix written in Rust were written in Python, testing the integration as well. This caught bugs like interpreting data as a different data type.

4.4.3 Implementation of the Algorithm

Initially, the writing of the algorithm happened by trying to translate the algorithm line by line from Python. As this is not possible due to the considerably different programming languages, it happened paragraph-wise, rather. This first transcription was considerably naive, and not idiomatic for Rust. Even though a `numpy` port existed, it was not used as the actually needed operations were considerably little, and probably faster without porting to `numpy` first. Most operations themselves were on `scipy` CSRMatrix implementation, and no port of `scipy` was available. This way, direct control over all critical parts of execution was assumed.

4.4.4 Testing and Bugfixing

Just as it does in Haskell (and probably any other strongly typed language), the concept of compiler driven development exist. This means that e.g. for refactoring, some part is modified, and then compiler errors and warnings are fixed one after another. As soon as the compiler does not complain any more, the new functionality might not yet be implemented, but the new field or renamed function is now being referenced correctly everywhere. A test suite in Rust as well as in Python was set up. In rust, variables are immutable by default, testing led to changing a borrowed reference to a borrowed mutable reference at two points, and looking into slices again in detail.

4.4.5 Idiomatic Rust and summing high fractions

Code Example 6

```
1 // let list1 = vec![0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0];
2 // let list2 = vec![0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0];
3 println!("sum: {}", list1.iter().sum()); // sum: 1
4 println!("sum: {}", list2.iter().sum()); // sum: 0
```

Code Example 7

```
1 // let list1 = vec![0.0, 0.0, 0.0, 0.16000000000000003, 0.0, 0.0, 0.0, 0.0];
2 // let list2 = vec![0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0];
3 println!("sum: {}", list1.iter().sum()); // sum: 0.4
4 println!("sum: {}", list2.iter().sum()); // sum: inf
```

While gradually transforming the naive Python-translation to idiomatic Rust, at some point results ended up being NaN pretty fast. Two step sbefore results ended

up being `NaN` the situation was as described in Code Example 6. Here everything worked fine, summing up over a considerably short list produced the expected results. Only one iteration later these elements have been multiplied with several factors, their product being `0.16`. This in and by itself should not have changed anything big, however looking at Code Example 7 their sums were now not exactly what one would expect at first, with one non-zero number among zeros being equal to another one and a sum of zeros being `inf` for some reason. As it turns out, the factor they have been subject to was indeed `0.16`, however it was `0.16` with high fraction values. This means that summation of `0.16` and `0.0` (the zero also having a high fraction) is being sufficiently inaccurate to not be accurately represented by floating point values, resulting in `0.4` instead of the expected `0.16` after adding only four times. The same happened with the summation of the innocuous-looking `0.0`. They had high fractions from the original multiplication by `0.16000000000000003`, their continued summation probably resulting in an overflow. This new number just happens to be one of the many representations of `inf`.

4.4.6 Packaging

Part of the goal is to package the code for easy use from the HiCEXplorer as a `conda` package. The only Python dependency after `setuptools` and `pip` ended up being `milksnake`, an extension for `setuptools` for packaging dynamically linked libraries, particularly ones written in Rust. This is described in Section 4.3.10 in more detail. As `milksnake` was not available as a `conda` package, it ended up getting ported. Initially, `conda skeleton pypi milksnake` created a package that even `conda` could not build, with the issue being that `milksnake` was only provided as a `*.zip` file and `conda` had hardcoded the format `*.tar.gz`. After doing this, a `travis` build server (with tests) was set up, to continuously monitor further progress.

4.4.7 Parallelizing

As is demonstrated in Section 2, the parallelization of Rust code can be really easy. This is why this was done last, as nothing really needed to change. by any tremendous amount. There is another variant for parallelizing code in Rust (with explicit threads) which was also tested, but it provided no further benefit and proved to be too much overhead. This might not be the case for truly big matrices however.

(TODO: Add summary regarding Rust) (DRAFT: put note somewhere that this server is different from the earlier mentioned travis test server, make distinction clear)

5 Results

In this chapter, primarily in the graphics, the implemented version is denoted as ‘RUST’, and the original Python implementation as ‘ICE’. This is done consistently as a compromise to avoid confusion regarding two differently colored ‘ICE’ columns and too long identifiers. It should be clear that ‘RUST’ is not representative for the language, but only for this particular implementation, also implementing ICE, not to confused with ‘ICE’, the Python implementation.

5.1 Server Specification

See Table 3 for the specification of the test server.

Virtual Server Specification

Available Cores / Threads	16 / 32
Working Memory (RAM)	120 GByte

Processor Specification

Processor	Intel® Xeon® E5-2630V4 ¹
Number of Cores/Threads	10 / 20
Base/Turbo frequency	2.2 GHz / 3.1 GHz

Table 3: Server Specification.

For reproducibility, here are technical details about the hardware the following tests were run on.

5.2 Data for Testing

Name	25kb_raw.h5	50kb_raw.h5
Filesize	1.1 GByte	732 MByte
Size	123'841	61'928
Bin length	25'000	50'000
Non-zero elements	1'530'533'003	1'053'216'825

Table 4: Test Data overview.

A third matrix, with a bin length of 10'000 was available, for which the existing working memory was not sufficient.

Details about the different available matrices for further testing can be seen in Table 4. These are matrices that have been shrunk in bin size with `hicMergeMatrixBins`, but the original data is from [3].

5.3 Memory Requirements

The memory requirements for these three algorithms are, or at least should be, equal during loading, as the exact same code is executed. During loading, memory was seen to fluctuate considerably, and was measured going up over 85 GByte at times. Initial loading time is proportional to matrix size, and includes pre-processing. Prior to correction, zero values, NaNs and outliers are removed from the data. This needs between five and eight minutes respectively (see Figure 9 for reference).

¹<https://ark.intel.com/content/www/us/en/ark/products/92981/intel-xeon-processor-e5-2630-v4-25m-cache-2-20-ghz.html>, accessed 2019-06-26

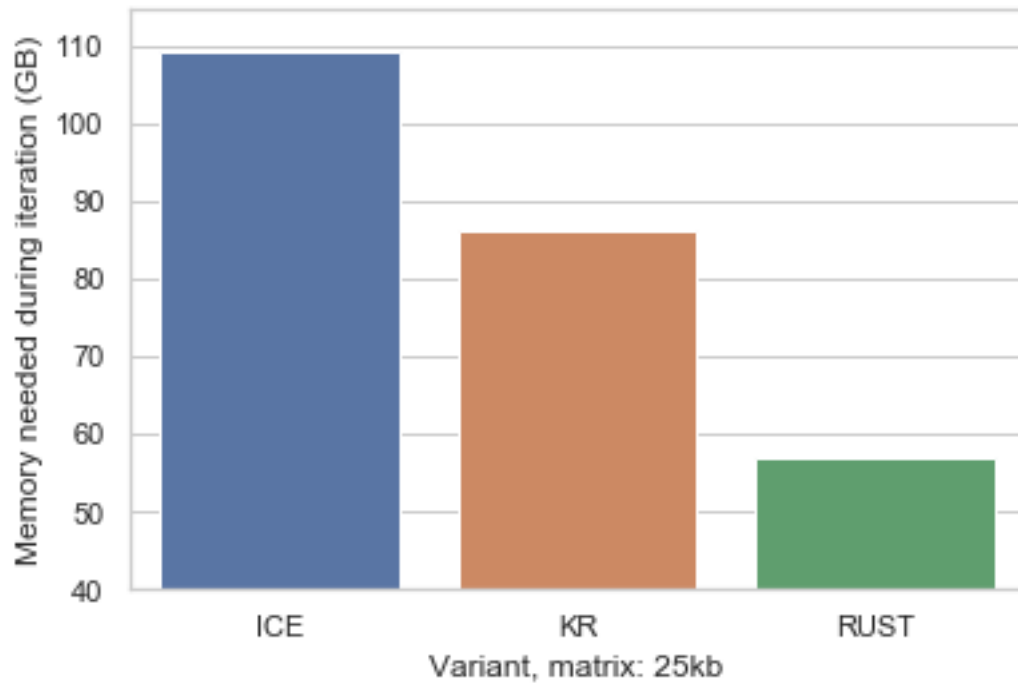


Figure 6: Memory needed during iteration for correcting the 25kb matrix. Smaller is better.

5.3.1 During Correction

Memory needs, at least for the iterative correction, have cycles. These cycles correspond to the in Section 4.2 described steps, as the additional bias matrices are not needed at any point other than for the correction itself. This might be different for the implementation itself however, as e.g. in the Rust implementation the memory needed for temporary biases is initialized before the first iteration and is freed only after the last iteration finished.

A comparison of the three implementations regarding memory requirements during correction of the 25kb connection matrix can be seen in Figure 6. As can be clearly seen, even though the recent KR implementation needs less memory than the original ICE implementation, the new ICE implementation in Rust needs even less, about half the amount the pure Python implementation needs. Note that these are only

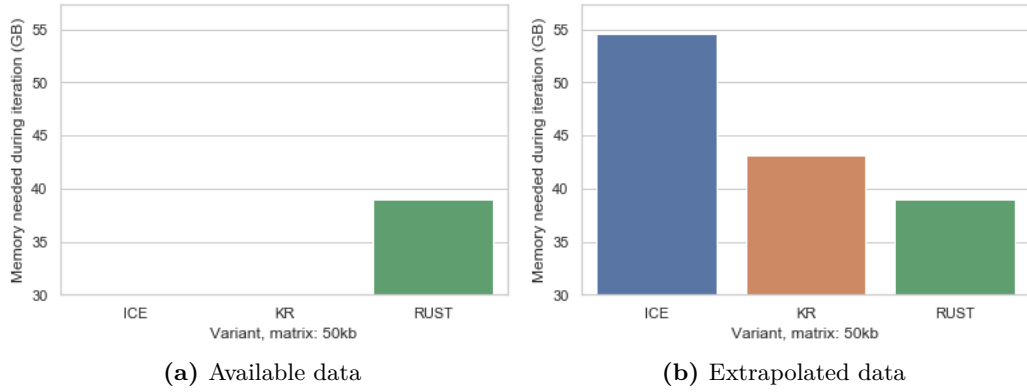


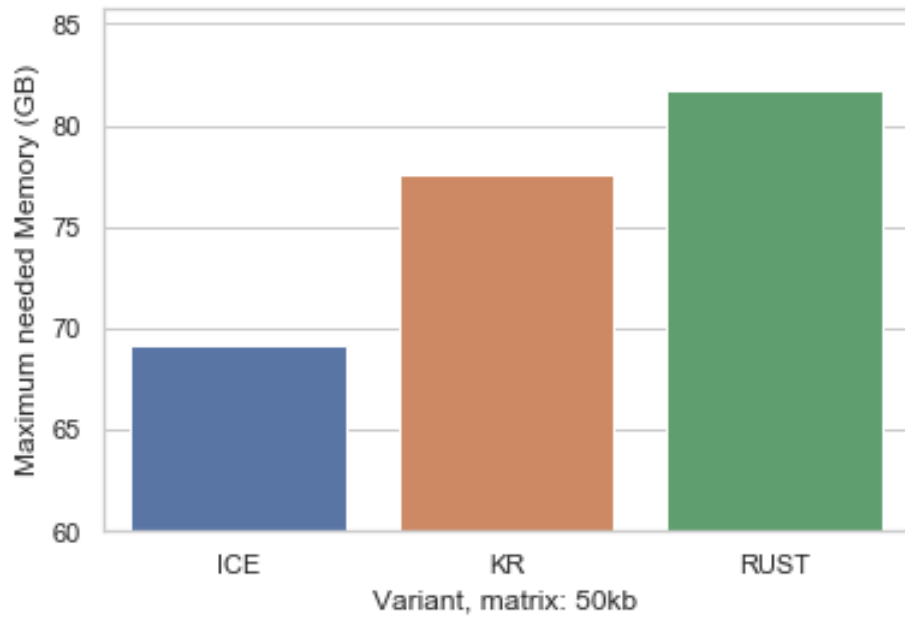
Figure 7: Available and Extrapolated Data about Memory needed during iteration of the 50kb matrix. Rust value is accurate, values for ICE and KR are not accurately available, but have been observed to be in the extrapolated areas. Smaller is better.

comparisons of memory needs during iteration, there is a different amount of memory needed for pre-processing and post-processing.

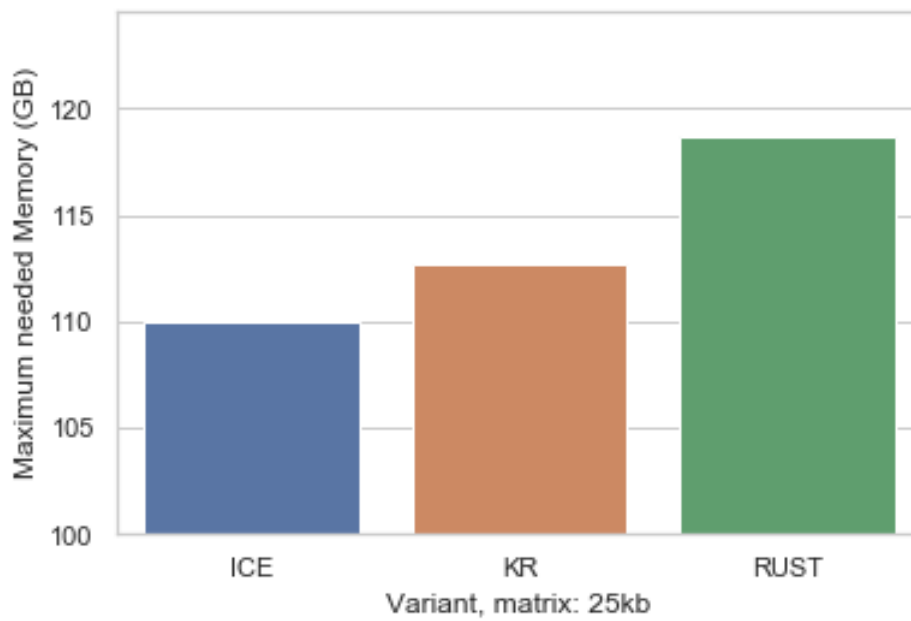
As can be seen in Figure 7a, no reliable data except for the Rust implementation exists for memory needs during computing of the 50kb matrix. In Figure 7b there is an extrapolation based on the data for the 25kb matrix and observed data, These values should not be taken at face value, but they should not be too far off the truth. Here, the advantage of Rust diminishes. Parts of the memory are still used from the initial pre-processing, and the provided matrix itself (to the regarding algorithm) was only a copy. Directly after loading (but before starting iteration), the needed memory when correcting the 50kb matrix was 12GByte.

5.3.2 Maximum required memory

Maximum required memory, or maximum resident set size, describes the highest amount of memory needed at any point during execution. Maximum resident set size is compared in Figure 8.



(a) Maximum resident set size when correcting 50kb matrix



(b) Maximum resident set size when correcting 25kb matrix

Figure 8: Maximum amount of needed Memory when correcting 50kb and 25kb matrix. Smaller is better.

After correction, the post-processing procedures differ. Both the Python and C++ implementation return the corrected matrix as well as the correction factors, while the Rust implementation is supposed to only return the correction factors, as it does. This means that after returning the correction factors, the corrected matrix from the rust part is not passed back, and a copy of the original matrix is then corrected in Python again before saving the corrected matrix.

When loading the matrix in the beginning, a considerable amount of working memory is needed. However, the highest needs of memory are, at least for KR and the RUST version, happening after the correction, when the corrected values are set, the correction factors added and then saved. This might be due to duplicate artifacts in memory, needing to be in memory in both Python and the other language, while in the python version memory needs are considerably lower during this process.

In Rust, indices are of type `usize`, which size is platform-dependent. In the case of the testserver (more details in Section 5.1 and Table 3) this is the size of a `int64`, whereas coming from Python (and thus most likely in C++) it is only a `int32`. Indices are a considerable amount of memory in the case of a `CSRMatrix`, as there are more indices than there are actual elements in the matrix, and the elements are only of type `float32`.

It is unclear if the RUST implementation (or the post-processing, since it is different) would use more memory if available, as the needed memory was supremely close to the actually available one. It is also unclear, if this implementation would use less if less was available. The required memory could most likely be reduced by returning the corrected matrix as well, or directly correcting the original matrix within the memory from Python. This has been deemed possible, as the `CSRMatrix` structure is build on top of `numpy` `ndarrays` implementation, with `numpy` having a well documented C-API.

5.3.3 Load times

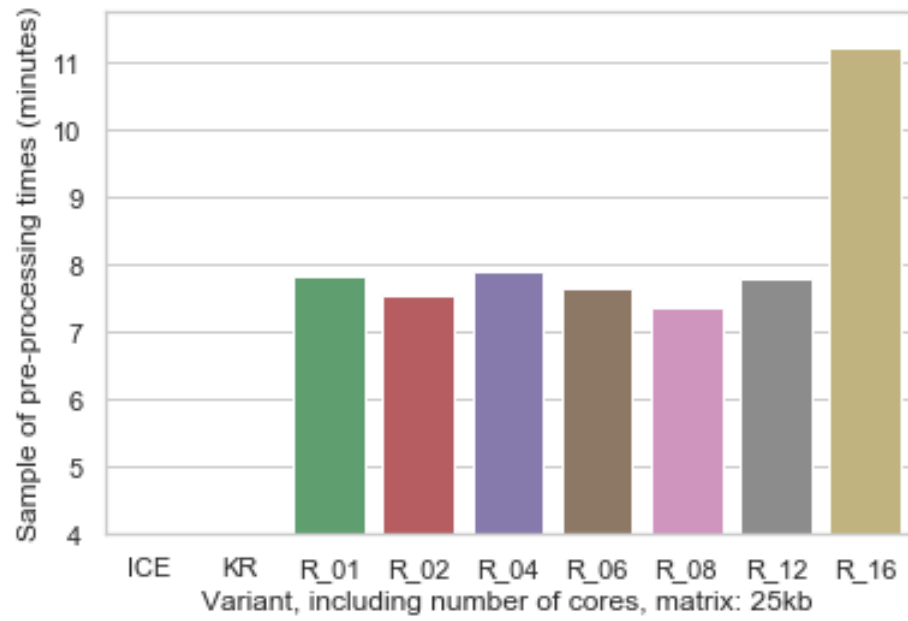
Loading includes pre-processing the to-be corrected matrices. Pre-processing steps include removal of zero-values, NaNs from the dataset and heavy outliers, as they would affect correction. However, even though the same steps were performed in all cases, load times had significant outliers, as can be seen in Figure 9. Here, loading times from separate runs are shown. The runs differ in that they were run in different configurations, which start to differ after loading, as explained in more detail in Section 5.5. However as there is no difference up to this point, they can very well be compared. Considering that there should be no difference, it is shocking to see that significant differences between loading times exist.

In the case of Figure 9a, most loading times are within half a minute of each other, which could still be called deterministic considering the size of the matrix and the amount of data written back and forth. The big jump as seen in variant R_16 however, should not be possible. Considering an average time of around 7.5 minutes, 11.2 minutes are close to 50% more than that (49.4%) ! This can only be described by the conditions of the virtual server varying based on usage from other users, for which there is no data available, except for deviations in this data, without knowledge what the actual values would have been.

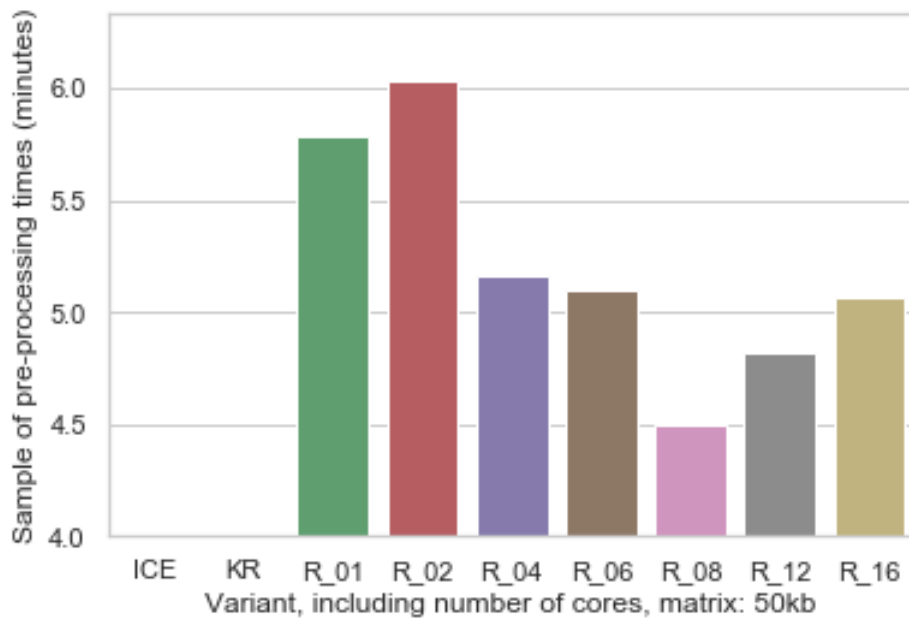
Loading times in Figure 9b do not differ by proportions as high, but the clearly visible disparity is not something favourable.

5.3.4 Comparing Memory needs

As can be seen in Table 5, regarding memory usage each of the algorithms has some advantage and another disadvantage. If working memory is a critical resource, the question as to which is more critical, long-term lower usage, or lower spike usage, becomes decisive for the selection of the algorithm.



(a) Loading time in minutes for 25kb matrix



(b) Loading time in minutes for 50kb matrix

Figure 9: Matrix loading times for the different matrix sizes. Even though the operations are the same during pre-processing, actual times fluctuated. No data is available for ICE and KR variants, however as the pre-processing is the same, their values can be thought to be in the same range as the others. Equal is better.

Overall memory comparison	ICE	KR	RUST
During iteration (50kb)	54.6	43.1	39
Maximum (50kb)	69.2	77.6	81.7
During iteration (25kb)	110	86	57
Maximum (25kb)	110	112.7	118.6

Table 5: Overall comparison between required memory for different situations. The ICE values for Maximum resident values are not considered, as they are probably wrong for some reason. See Section 5.3.2 for details.

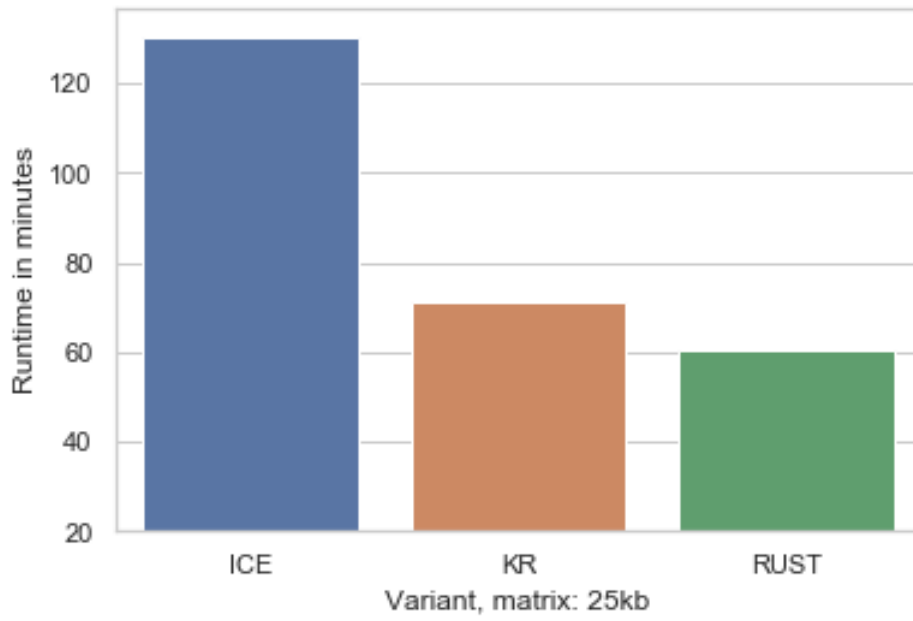
5.4 Runtime length

In Figure 10 a comparison of runtimes can be seen. The original implementation in Python is taking the longest by a considerable amount, with the implementation in Rust needing about half the time in both situations, less than KR which is at only a third for 50kb. The difference itself, which is about ten minutes, is roughly the same for both matrices though. In both cases the difference is around ten minutes, whereas it is proportionally more when the total runtime is only four times that amount versus ten minutes being only a seventh of the total runtime, Figure 10b and Figure 10a respectively.

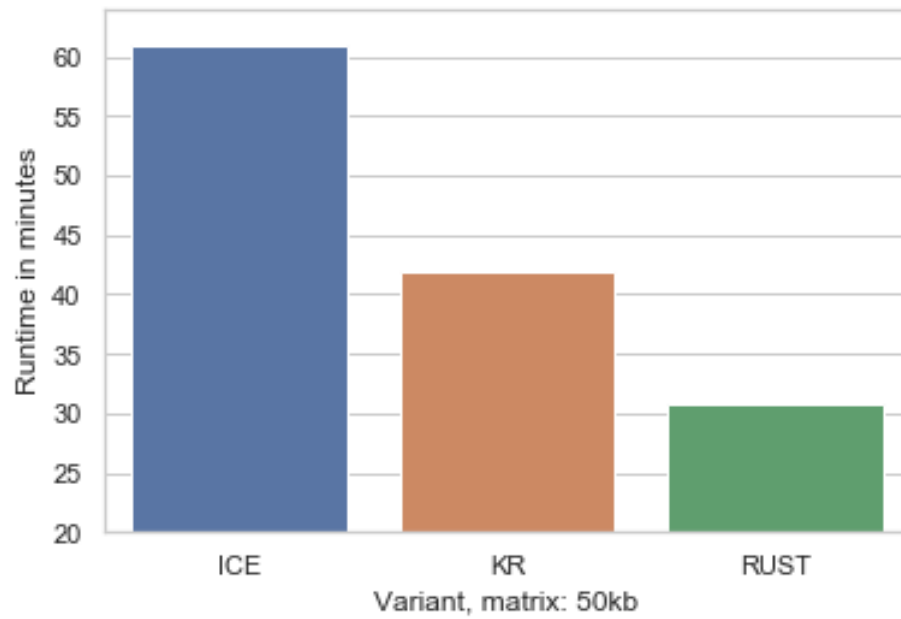
An open question remains as to why this difference is roughly the same, even though the python implementation needs about twice as much time, as does the Rust one. It might be that the KR implementation is faster for considerably bigger matrices. Testing this was not possible as the memory of the provided virtual machine (see Section 5.1 and Section 5.3.2 for details) did not allow loading of bigger matrices, and repeatedly failed trying to do so.

5.5 Multicore benefits

As can be seen in Section 2, turning a single-core Rust program into a multi-core program is not particularly difficult. Taking a look at Figure 11, the advantage of



(a) Runtime in minutes for correcting 25kb Matrix



(b) Runtime in minutes for correcting 50kb Matrix

Figure 10: Algorithm Runtimes for correcting the different matrices. It remains an open question why the difference between KR and RUST stays the same, even though both ICE and RUST double their computation time. Smaller is better.

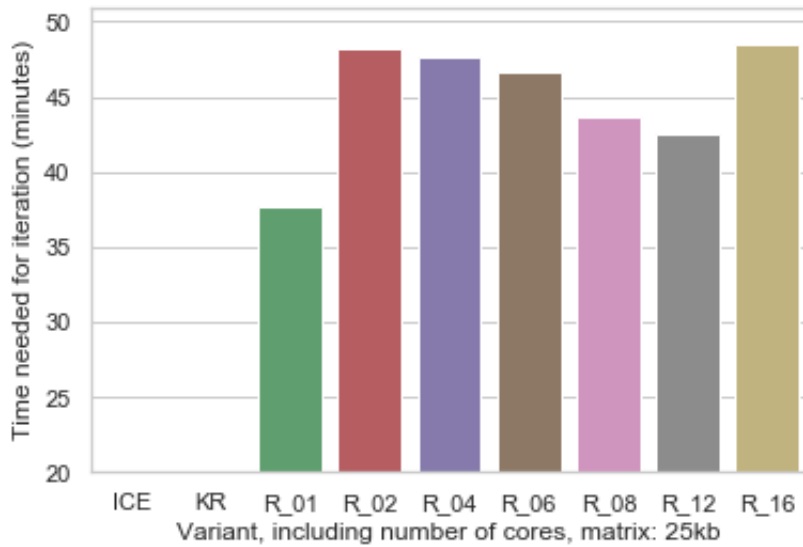
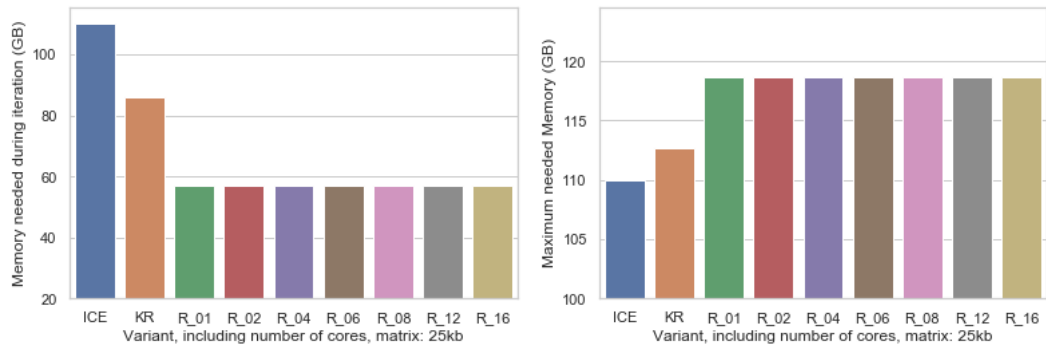


Figure 11: Multicore runtimes for correcting the 25kb matrix. These are pure computation times, excluding both pre-processing and post-processing. The multicore-variant at one core is slightly worse than the two-core version. Thus the non-multicore single-core version was included instead. No numbers for ICE and KR are available. Smaller is better.

using multiple cores is everything but apparent, in fact it is even slower than not using it. There are multiple reasons for this. First, most of the parallelized operations are considerably small, sometimes just slightly more complex than a list summation. In the case of parallelizing list summations as well, total runtime becomes similar to the original ICE implementation, as there are several summations for each iteration. Even though the operations themselves are rather small, the overhead for how to distribute them is still added, even in the case of using only one core. This overhead is spreaded across several cores as well, but even using 12 cores is not sufficient for amortizing the additional overhead. The scheduler of `rayon`, the used library for this, is a work-stealing scheduler, meaning that initially every thread in the threadpool gets a roughly equal amount, and as soon as one core is done earlier than the others, its stealing work from other cores. Initially all work is distributed over the available threads in chunks. This initial distribution makes up a significant amount of overhead. In the case of 16 Threads (and above in cases tested), the main overhead is coming

from too many threads having finished earlier and stealing work from each other.

Even though the current version does support parallelization, it does not provide benefits. They might only visualize for even bigger matrices, where the amount of needed work amortizes the initial overhead better. Additionally, there is more potential for combining operations for parallelization. For some, this is clearly not possible, but for several it would be possible to combine them, reducing the overhead for parallelization at one point, and parallelizing heavier operations instead.



(a) Memory needed during iteration when correcting 25kb matrix (b) Maximum resident set size when correcting 25kb matrix

Figure 12: Multicore Memory needs for correcting the 25kb matrix. Memory needs stay the same in all versions. Based on the numbers, there is no difference between the single-core version and any multi-core version regarding memory usage.

However, as can be seen in Figure 12, memory is not negatively affected in any way when parallelizing. Regarding memory, the same holds true for the 50kb matrix. Pure computation times for the 50kb matrix as seen in Figure 13 do not seem to offer a clear trend regarding the usage of multiple cores.

Conclusion: Using multiple cores for this implementation does not provide any benefit, however it does have significant downsides other than longer runtime. The algorithm may be rewritten to allow better multicore usage.

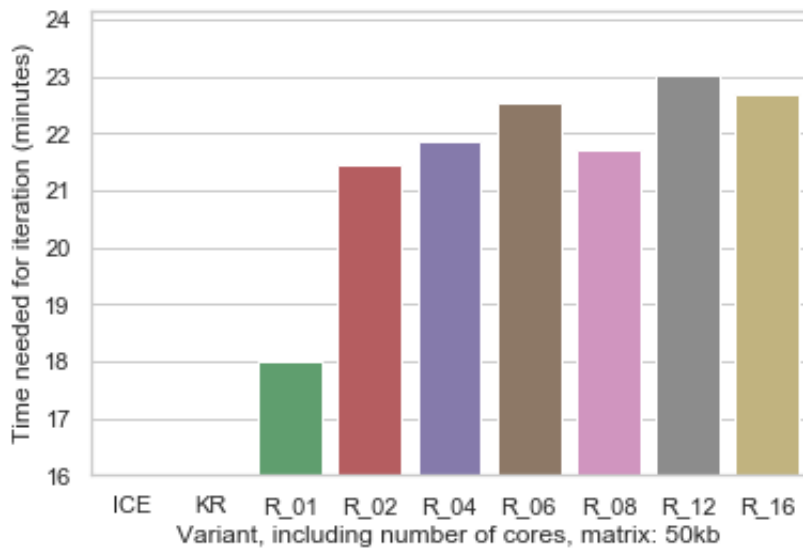
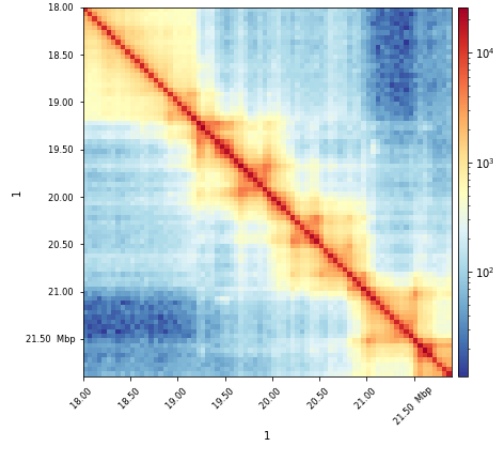


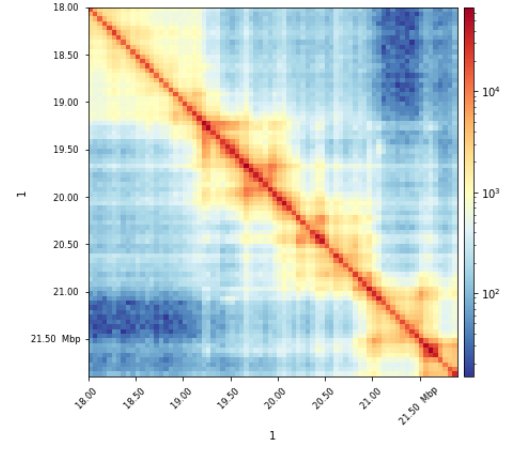
Figure 13: Multicore computation time comparison for 50kb matrix. These are pure computing times, excluding both pre- and post-processing. No numbers for ICE and KR are available. Smaller is better.

5.6 Output

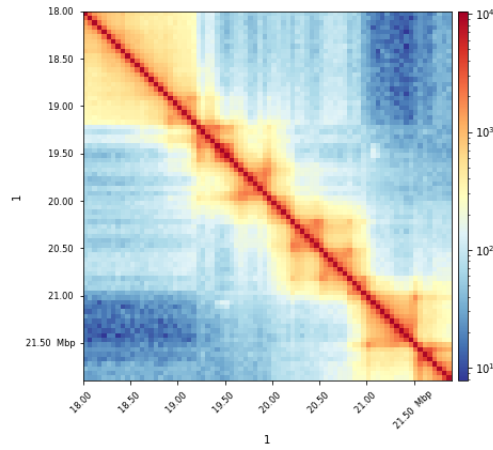
One essential point of a reimplementaion is the correctness of of the output. As can be seen in Figure 14, the output from both the original ICE implementation and the KR version are pretty much the same. The output of the RUST implementation is closer to the original values however. Due to time constraints the differences between the output matrix in the Rust implementation could not be further explored, but are likely due to different thresholds in the implementation of the algorithm.



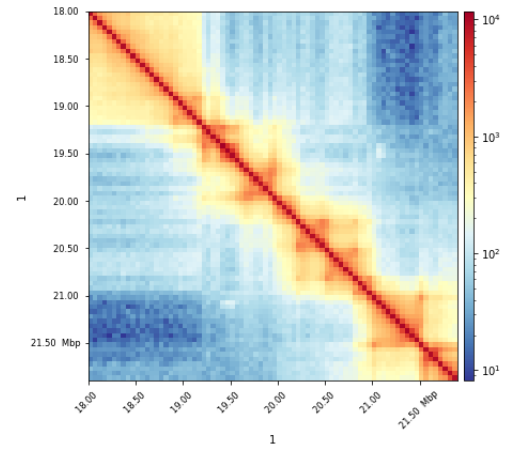
(a) original values



(b) RUST



(c) ICE



(d) KR

Figure 14: Original and corrected matrices. In (b) there appear to be higher maximum values, setting the cutoff lower creates pictures undistinguishable to those from ICE or KR. Images were created using hicPlotMatrix.

6 Conclusion

(TODO: run aspell at least once)

(TODO: general evaluation; worked better or worse, why, what could be tried / changed, evaluation of rust in this context, ...)

(TODO: clean up repository)

(TODO: remove deepTools entirely)

(TODO: Add Glossary?)

(DRAFT: use more formal language)

(TODO: biggest points: Python to rust and how it worked, give code examples but not too much)

(TODO: clean up github and add documentation)

(TODO: make sure everything is cited appropriately!)

(TODO: make sure everything is cited appropriately!)

(TODO: make sure everything is cited appropriately!)

(TODO: for presentation: 1/3 for everyone, 1/3 for supervisor + experts, 1/3 only I am expert)

ToDo Counters

To Dos: 20; 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

Parts to extend: 1; 1

Draft parts: 5; 1, 2, 3, 4, 5

Bibliography

- [1] L. A. Pennacchio, W. Bickmore, A. Dean, M. A. Nobrega, and G. Bejerano, “Enhancers: five essential questions,” *Nature Reviews Genetics*, vol. 14, no. 4, p. 288, 2013.
- [2] E. Lieberman-Aiden, N. L. Van Berkum, L. Williams, M. Imakaev, T. Ragoczy, A. Telling, I. Amit, B. R. Lajoie, P. J. Sabo, M. O. Dorschner, *et al.*, “Comprehensive mapping of long-range interactions reveals folding principles of the human genome,” *science*, vol. 326, no. 5950, pp. 289–293, 2009.
- [3] S. S. Rao, M. H. Huntley, N. C. Durand, E. K. Stamenova, I. D. Bochkov, J. T. Robinson, A. L. Sanborn, I. Machol, A. D. Omer, E. S. Lander, *et al.*, “A 3d map of the human genome at kilobase resolution reveals principles of chromatin looping,” *Cell*, vol. 159, no. 7, pp. 1665–1680, 2014.
- [4] G. Ron, Y. Globerson, D. Moran, and T. Kaplan, “Promoter-enhancer interactions identified from hi-c data using probabilistic models and hierarchical topological domains,” *Nature communications*, vol. 8, no. 1, p. 2237, 2017.
- [5] L. Teytelman, B. Ozaydin, O. Zill, P. Lefrancois, M. Snyder, J. Rine, and M. B. Eisen, “Impact of chromatin structures on DNA processing for genomic analyses,” *PLoS ONE*, vol. 4, p. e6700, Aug 2009. [PubMed Central:PMC2725323] [DOI:10.1371/journal.pone.0006700] [PubMed:12067662].

- [6] M. S. Cheung, T. A. Down, I. Latorre, and J. Ahringer, “Systematic bias in high-throughput sequencing data and its correction by BEADS,” *Nucleic Acids Res.*, vol. 39, p. e103, Aug 2011. [PubMed Central:PMC3159482] [DOI:10.1093/nar/gkr425] [PubMed:20824077].
- [7] S. Wingett, P. Ewels, M. Furlan-Magaril, T. Nagano, S. Schoenfelder, P. Fraser, and S. Andrews, “Hicup: pipeline for mapping and processing hi-c data,” *F1000Research*, vol. 4, 2015.
- [8] M. Imakaev, G. Fudenberg, R. P. McCord, N. Naumova, A. Goloborodko, B. R. Lajoie, J. Dekker, and L. A. Mirny, “Iterative correction of hi-c data reveals hallmarks of chromosome organization,” *Nature methods*, vol. 9, no. 10, p. 999, 2012.
- [9] P. A. Knight and D. Ruiz, “A fast algorithm for matrix balancing,” *IMA Journal of Numerical Analysis*, vol. 33, no. 3, pp. 1029–1047, 2013.
- [10] R. Wheeler, “Chromatin structures.” https://commons.wikimedia.org/wiki/File:Chromatin_Structures.png, 2005. Licensed under CC BY-SA 3.0; Image has been cropped; accessed 2019-06-26.
- [11] G. Li, L. Cai, H. Chang, P. Hong, Q. Zhou, E. V. Kulakova, N. A. Kolchanov, and Y. Ruan, “Chromatin interaction analysis with paired-end tag (chia-pet) sequencing technology and application,” *BMC Genomics*, vol. 15, p. S11, Dec 2014.
- [12] J. Dekker, K. Rippe, M. Dekker, and N. Kleckner, “Capturing chromosome conformation,” *science*, vol. 295, no. 5558, pp. 1306–1311, 2002.
- [13] M. Simonis, P. Klous, E. Splinter, Y. Moshkin, R. Willemsen, E. De Wit, B. Van Steensel, and W. De Laat, “Nuclear organization of active and inactive chromatin domains uncovered by chromosome conformation capture-on-chip (4c),” *Nature genetics*, vol. 38, no. 11, p. 1348, 2006.

- [14] Z. Zhao, G. Tavoosidana, M. Sjölander, A. Göndör, P. Mariano, S. Wang, C. Kanduri, M. Lezcano, K. S. Sandhu, U. Singh, *et al.*, “Circular chromosome conformation capture (4c) uncovers extensive networks of epigenetically regulated intra-and interchromosomal interactions,” *Nature genetics*, vol. 38, no. 11, p. 1341, 2006.
- [15] J. Dostie, T. A. Richmond, R. A. Arnaout, R. R. Selzer, W. L. Lee, T. A. Honan, E. D. Rubio, A. Krumm, J. Lamb, C. Nusbaum, *et al.*, “Chromosome conformation capture carbon copy (5c): a massively parallel solution for mapping interactions between genomic elements,” *Genome research*, vol. 16, no. 10, pp. 1299–1309, 2006.
- [16] J. Wolff, V. Bhardwaj, S. Nothjunge, G. Richard, G. Renschler, R. Gilsbach, T. Manke, R. Backofen, F. Ramírez, and B. A. Grüning, “Galaxy hicexplorer: a web server for reproducible hi-c data analysis, quality control and visualization,” *Nucleic acids research*, vol. 46, no. W1, pp. W11–W16, 2018.
- [17] F. Ramírez, V. Bhardwaj, L. Arrigoni, K. C. Lam, B. A. Grüning, J. Villaveces, B. Habermann, A. Akhtar, and T. Manke, “High-resolution tads reveal dna sequences underlying genome organization in flies,” *Nature communications*, vol. 9, no. 1, p. 189, 2018.
- [18] G. Hoare, “Rust faq.” <https://prev.rust-lang.org/en-US/faq.html>, 2019. accessed 2019-06-26.
- [19] G. Hoare, “stage1/rustc builds.” <https://mail.mozilla.org/pipermail/rust-dev/2011-April/000330.html>, 2011. accessed 2019-06-26.
- [20] “Rust version history.” <https://github.com/rust-lang/rust/blob/master/RELEASES.md>, 2019. accessed 2019-06-26.

- [21] “The rise and fall of languages in 2013.” <http://www.drdobbs.com/jvm/the-rise-and-fall-of-languages-in-2013/240165192>, 2013. accessed 2019-06-26.
- [22] “Rust survey 2018 results.” <https://blog.rust-lang.org/2018/11/27/Rust-survey-2018.html>, 2018. accessed 2019-06-26.
- [23] “Stack overflow survey.” https://insights.stackoverflow.com/survey/2019#technology_-_most-loved-dreaded-and-wanted-languages, 2019. accessed 2019-06-26.
- [24] “Memory management bugs: An introduction.” <https://backtrace.io/blog/backtrace/introduction-to-memory-management-errors/>, 2016. accessed 2019-06-26.

