Undergraduate 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

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Abstract

foo bar

Zusammenfassung

German version is only needed for an undergraduate thesis.

(TODO: Schreiben!)

Contents

1 Introduction **1**

1.1 Core setup 2

1.2 Algorithm 2

1.3 Motivation 2

1.3.1 Motivation for Hi-C 2

1.3.2 Motivation for Rust 3

2 Background **5**

2.1 Chromosome Conformation Capture and Hi-C 5

2.1.1 Overview 5

2.1.2 Cross-linking DNA 6

2.1.3 Digestion 6

2.1.4 Ligation 7

2.1.5 Reverse Cross-links 7

2.1.6 Sonication 7

2.1.7 Filtering and Removal of Biotin 8

2.1.8 Sequencing 8

2.2 deepTools 9

2.3 HiCExplorer 9

3 Related Work **13**

3.1 Python implementation 13

3.2 KR-algorithm 14

4 Approach **15**

4.1 Problem 15

4.2 Iterative Correction and Eigenvector decomposition (Algorithm) . . 15

4.3 Operation 18

4.3.1 Installation 18

4.3.2 Build 18

4.4 Differences between Rust and Python 20

4.5 Using Rust 22

4.5.1 Testing the integration of Rust 22

4.5.2 Introducing Rust 22

4.5.3 Advantages of Rust 23

4.5.4 Disadvantages of Rust 24

4.6 Choosing the right API to call Rust from Python 24

4.7 General Approach 27

4.7.1 Beginning 27

4.7.2 Feasability Testing 27

4.7.3 Implementation of the algorithm 28

4.7.4 Testing and Bugfixing 28

4.7.5 Idiomatic Rust 28

4.7.6 Packaging 29

4.7.7 Parallelizing 30

4.8 Testing 30

5 Experiments **33**

6 Conclusion **35**

7 Acknowledgments **37**

Bibliography **42**

List of Figures

1 Summarised Hi-C protocol 11

2 Pairplot 34

1 Introduction

Hi-C is a method commonly used for getting 3D-information of genomes. Such technologies tend to suffer from technical (e.g. sequencing, mapping) [1] and biological factors (e.g. distinct chromatin states) [2], making them inherently inaccurate.

However, a basic 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 (Iterative Correction and Eigenvector decomposition, Section 4.2) or KR (Knight-Ruiz, Section 3.2) can be applied to "normalize" the matrix nonetheless.

ICE, the algorithm implemented in this work, "normalizes" the data we have iteratively. Eigenvector decomposition of the normalized data can then give us new insights on local chromatin states or global patterns of chromosomal interaction [3].

We will not do the Eigenvector decomposition, but the iterative correction ("normalization") before that.

1.1 Core setup

This work is part of the HiCExplorer (Section 2.3) tool from the Deeptools (Section 2.2) framework. HiCExplorer is mainly written in Python, being limited by the high resource requirements of the iteration process.

As we will see, the implementation in rust is faster even when using only one core.

(TODO: run experiments!!)

(EXTEND: Add more details!!)

1.2 Algorithm

Fundamentally, every part in our genome has the same amount of summed up interactions with other parts of the genome. The algorithm takes severe advantage of this property, downright enforcing it. A full description can be found in Section 4.2.

1.3 Motivation

1.3.1 Motivation for Hi-C

Within cells, the three-dimensional structure of chromatin can now be analyzed using techniques based on chromatin conformation capture (C3), including Hi-C. With Hi-C **(DRAFT: what is the long name?)**, we can construct spatial proximity maps over the whole genome, giving us clues as to which regions are close to each

other. With C3 it can be tested if two genomic locations are close to each other, but with Hi-C we get (biased but still) all spatial connections of the genome at once.

1.3.2 Motivation for Rust

There is several reasons for using Rust, including faster development than C++ and better tooling, while still having the same if not slightly better performance. A massive advantage is the easiness of adding parallelism, and the modularity of Rust code. Libraries in Rust can easily be stacked, and the compiler will complain if the memory is not handled the way it should be (in C++ it is hard to correctly manage all memory, especially from libraries, easily resulting in segfaults and similar issues hard to debug). The benefits to using Rust instead of Python are only apparent if performance or safety (of execution, at runtime) is needed - which is the case here.

2 Background

2.1 Chromosome Conformation Capture and Hi-C

2.1.1 Overview

Chromatin usually describes different levels of how DNA organizes itself. The well-known double-helix is only the lowest of several structural layers.

Looking at it from the outside (highest structural layer), DNA looks similar to a big ball of wool. With the help of Hi-C (or other methods) we can visualize the spatial proximity.

Chromosome Conformation Technologies describe several similar methods to compare genomic loci. They all start by:

- creating chromatin cross-links (Section 2.1.2),
- digesting the cross-linked chromatin (Section 2.1.3),
- ligating the ends and marking with Biotin (Section 2.1.4) and
- reversing the cross-link (Section 2.1.5)

to get a sequence with two parts, those parts have been spatially close to each other and were marked with Biotin during ligation.

Other Chromosome Conformation Technologies proceed differently, Hi-C, our focus, continues with the following steps:

- shortening the cross-links by sonication (Section 2.1.6),
- filtering leaving only those with biotin (Section 2.1.7) and
- sequencing (Section 2.1.8).

The full process can be seen in Figure 1.

2.1.2 Cross-linking DNA

The first step is to cross-link DNA strands that are close to each other spatially (see Figure 1 for reference). This is done by adding formaldehyde, which bonds (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, not necessarily two 'known to be close' regions.

2.1.3 Digestion

The next step is cutting the ball of wool apart in intervals. For this, restriction-enzymes are used (specifically restriction endonuclease). Commonly used enzymes for this are EcoR1 or HindIII, cutting the genome every 4000 base-pairs **(DRAFT: info taken from Wikipedia, add some other source)**.

This will result in a lot of cross-linked fragments, as well as not-cross-linked ones.

2.1.4 Ligation

After reducing the concentration of fragments, DNA ligase is added, to ligate (weld together) dangling fragment ends. The reduction in concentration is done since mostly fragments close together are ligated, and we intend to ligate fragments held together by formaldehyde. Also, Biotin is added to mark the points of ligation. This will let us filter out a lot of fragments that have not been ligated later on.

2.1.5 Reverse Cross-links

Adding a high concentration of salt for some time will reverse the cross-linking through formaldehyde, leaving us with our two originally spatially close fragments ligated and with a biotin-marker.

Our fragments, however, are too long to sequence them effectively (remember that we have now ligated fragments of around 8000 base-pairs, most sequencing methods can only deal with sequence lengths of a few hundred base-pairs).

2.1.6 Sonication

The next step is to put them under influence of ultrasonic waves, breaking them apart in much shorter fragments (due to long sequences not being able to absorb frequent shocks well), short enough to actually enable sequencing.

2.1.7 Filtering and Removal of Biotin

Here we 'pull-down' fragments marked with biotin, filtering all the fragments and leaving only those having been ligated earlier (see Section 2.1.4). Subsequently we remove the marker, since it would get in the way of sequencing.

2.1.8 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 duplicates the fragments several times, to sequence them more accurately.

2.2 deepTools

DNA sequencing is generating more data than ever before, for which deepTools, being a framework, has useful programs to "process the mapped reads data for multiple quality checks, creating normalized coverage files in standard bedGraph and bigWig file formats"[1] and more. Using these normalized, standardized files, many visualizations showing the actual connections or functional annotations, can be made.

2.3 HiCExplorer

HiCExplorar, being part of the deepTools-framework, is a collection of tools, that are used to process, analyze and visualize Hi-C data. Part of this collection are tools to convert between formats, correcting the data (which this is work is part of), normalizing it, analysing it in various ways, and extensively plotting it. Facilitated is, among othters "the creation of contact matrices, correction of contacts, topologically associating domains (TAD) detection, A/B compartments, merging, reordering or chromosomes, conversion from different formats including cooler and detection of long-range contacts."[2]Those contact matrices may then be visualized, also showing other types of data, including "genes, compartments, ChIP-seq coverage tracks, long range contacts and the visualization of viewpoints"[2].

[1] `https://github.com/deeptools/deepTools`, accessed 2019-06-26
[2] `https://github.com/deeptools/HiCExplorer`, accessed 2019-06-26

(TODO: was kann man tun mit den visualisierungen, und auf welche wege visualisiert man das etc? Auf welchem Wege: Z.b. mit Software des HiCExplorer: hicPlotMatrix, hicPlotTADs, hicAggregate. Die Visualisierungen braucht man um komplexere Inhalte leichter (und vor allem schnell) verstaendlich zu machen. Niemand sieht z.B. in einer Matrizen sehr schnell hoehere Zahlenwerte, als Heatmap dagegen dargestellt sieht man sofort dass da ein anderes Muster in einem Bereich ist.)

(TODO: einführung in den HiCExplorer auf die ich verlinken könnte? Siehe eine der letzten Mails da hab ich ein Paper verlinkt.)

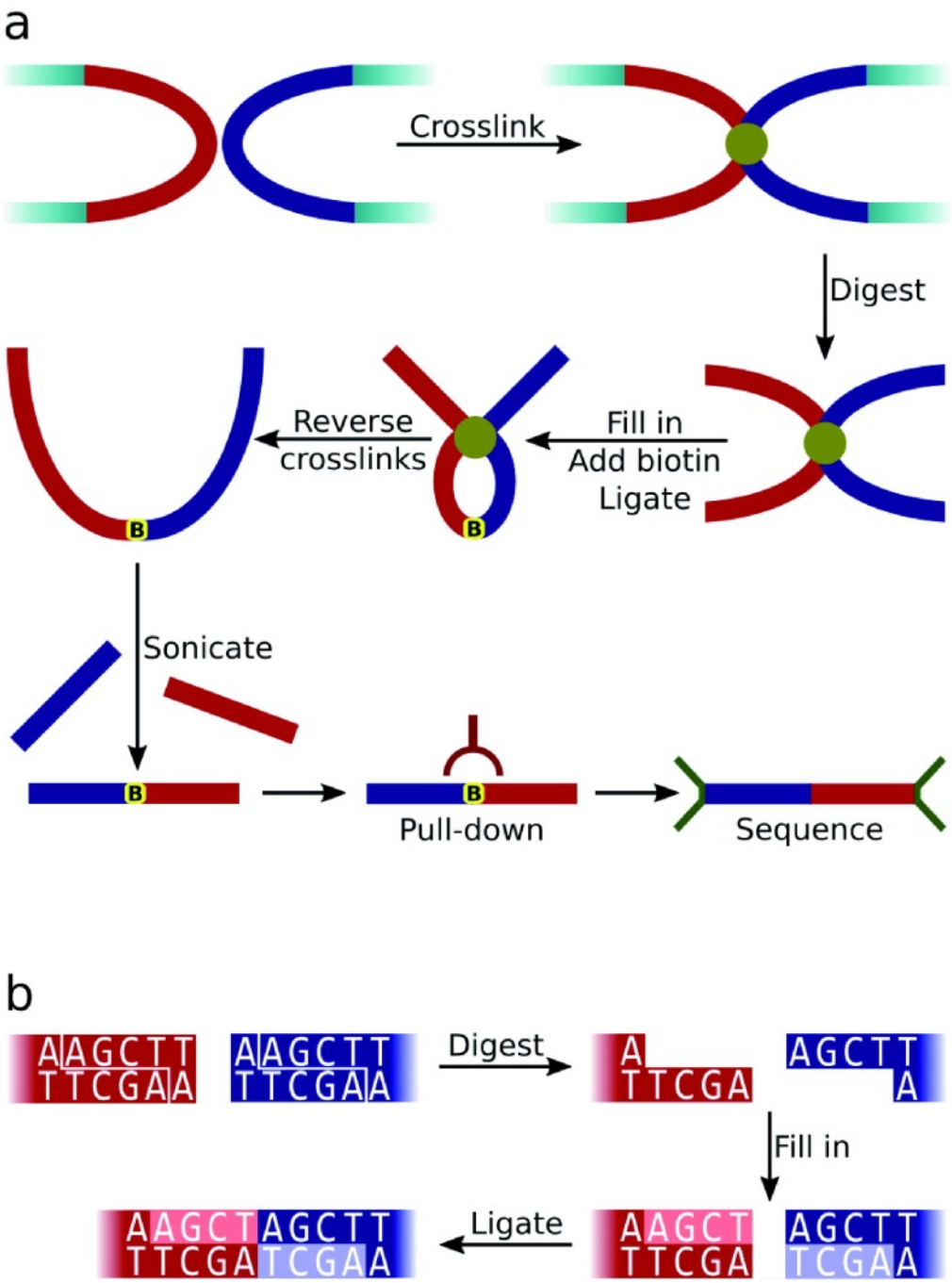


Figure 1: **a)** Diagram summarising the Hi-C experimental protocol. The red and blue rectangles represent cross-linked restriction fragments while the yellow marker shows the position of biotin incorporation. **b)** Generation of the Hi-C ligation junction sequence by successive digestion (with HindIII in this example), fill in and blunt-ended ligation steps. The modified restriction site sequence is not found in the original genomic sequence.

Image and description taken from [4].

3 Related Work

Since the main work is the implementation in Rust as well as the testing of an integrating in Python, related work includes the original implementation in Python as well as the recent implementation of the KR-algorithm in C++. We will take a look at how they have been implemented, advantages and disadvantages.

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 the deepTools-framework is mainly written in Python. This implementation is using common python dependencies extensively, including the Compressed Sparse Row Matrix (CSR Matrix) implementation from `scipy` (documentation can be found here[1]), as well as the scientific number manipulation library `numpy`[2].

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 imports are not for functionality, but rather for timing and logging. With the

[1] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html`, accessed 2019-06-26

[2] `https://www.numpy.org/`, accessed 2019-06-26

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.

There is practically no downside to this implementation, except for the high requirements for resources, especially working memory (RAM), and Pythons not really existing paralellisation. **(EXTEND: update after results, compare requirements)**

For reference, the Python-implementation can be found here[3].

3.2 KR-algorithm

(DRAFT: describe C++ implementation (the algorithm only superficially), advantages, disadvantages, specific implementation details (C-API, calls from Python to C++, ...))

What follows is a short description to the Algorithm known as Knight-Ruiz from [5], as this algorithm is inherently different from ICE being described later and implemented in both other implementations (the ICE algorithm is described in Section 4.2).

(TODO: read paper / algorithm + introduce here)

For reference, the implementation of the KR algorithm can be found here[4].

[3] `https://github.com/deeptools/HiCExplorer/blob/master/hicexplorer/iterativeCorrection.py`, accessed 2019-06-26

[4] `https://github.com/deeptools/Knight-Ruiz-Matrix-balancing-algorithm`, accessed 2019-06-26

4 Approach

4.1 Problem

In the three-dimensional space of a cell the DNA forms a structure that looks close to that of a ball of wool. Obviously, many points of contacts of the DNA wire with itself, called DNA interactions, exist in this "ball of wool" and form structures including DNA loops. However, many of these contacts are random contacts or measurement errors that need to be corrected. A Python implementation exists but is limited for high resolution data due to high memory usage. This [..] project aims to reimplement a more memory efficient method in C++ (which ended up being Rust).

4.2 Iterative Correction and Eigenvector decomposition (Algorithm)

The Algorithm as defined in [3] (Supplementary Note):
"We perform iterative correction on the resulting contact maps to obtain biases

B_i and 'true' T_{ij} relative contact probabilities by explicitly solving the system of equations:

$$O_{ij} = B_i B_j T_{ij}$$

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

[...]

After the vector of biases is computed, the corrected map of relative contact probabilities is obtained by $T_{ij} = O_{ij}/(B_i B_j)$. Algorithmically, the iterative correction is implemented as follows. We start by creating a working copy of the matrix O_{ij}, denoted W_{ij} as the iterative process gradually changes this matrix to T_{ij}. We initialize the iterative procedure by setting each element of the vector of total biases B to 1. We begin each iteration by calculating the coverage $S_i = \sum_j W_{ij}$. Next, additional biases ΔB_i are calculated by renormalizing S_i to have the unit mean $\Delta B_i = S_i/$ mean (S_i). We then divide W_{ij} by $\Delta B_i \Delta B_j$ for all (i, j) and update the total vector of biases by multiplying by the additional biases. Iterations are repeated until the variance of the additional biases becomes negligible; at this point W_{ij} has converged to T_{ij}."

In between, a couple of other corrections are described. Later, also in the supplementary note from [3], about Eigenvectors:

"**Eigenvector analysis of interchromosomal contact map.**
Eigenvector analysis of a corrected interchromosomal contact map T involves expanding the matrix as a sum of outer products between eigenvectors, E_i^k, weighted by their eigenvalues:

$$T_{ij} = \sum_k \lambda_k E_i^k E_j^k + \langle T \rangle$$

where $\langle T \rangle$ denotes the mean value of the matrix, and the magnitude of the eigenvalue λ_k describes the amount of information captured by the corresponding eigenvector E^k, where k runs from 1 to N. Eigenvectors are then sorted by the absolute value of their eigenvalues, and eigenvectors corresponding to the three largest eigenvalues, E^1, E^2 and E^3, are used for further analysis [..]. Iterative correction is a key prerequisite for eigenvector expansion; performing eigenvector expansion (or principal-component analysis, PCA) on the raw data entangles biases and eigenvectors, making the result nontransparent and bias dependent. Moreover, E^1 is clearly interpretable as the solution to a linear model of chromatin interaction preferences."

4.3 Operation

4.3.1 Installation

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

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

4.3.2 Build

To build the package, assuming you have conda installed, execute the following commands:

```
# first, install rust:
curl https://sh.rustup.rs -sSf | sh -s -- -y

# alternatively install rust with conda:
conda install -c conda-forge rust

# confirm install:
cargo --version
rustc --version
```

```
# download repository and navigate in it
git clone https://github.com/fkarg/HiC-rs
cd HiC-rs

# navigate to the rust code and compile (optional)
cd smb
cargo build
cd ..

# install missing python dependencies
pip install -r requirements.txt

# execute the setup.py (will also compile rust if not done yet)
python setup.py build
```

(TODO: update packages!!)

4.4 Differences between Rust and Python

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

The biggest difference, however, is that in Rust certain tasks can easily be parallelized; even after originally writing it for only one core. An example would be:

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

In this code, some `heavy_operation` is being applied iteratively for every element in `somelist` and later assigned to `otherlist`.

This can easily be parallelized by changing it to the following:

```
use rayon::prelude::*;

let otherlist = somelist.par_iter().map(|&v|
                heavy_operation(v)).collect();
```

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

4.5 Using Rust

4.5.1 Testing the integration of Rust

(TODO: Answer this question later!!)

One of the main questions for this work was to find out if it is possible to integrate Rust in Python for the HiCExplorer, and if it makes sense to do so.

For Rust and Python to interact, there are of course several ways. Those integrating a library written in Rust to allow them to be called from Python will be described in depth in Section 4.6.

4.5.2 Introducing Rust

"Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety" - this was, up until recently, the motto of Rust. It recently got changed into "A language empowering everyone to build reliable and efficient software." At the core of Rust is a compiler applying linear typing, rsulting in the concepts of Ownership and Lifetimes. These concepts, even though notoriously hard to learn, are the main reason the rust compiler can guarantee thread safety, can prevent segfaults, and can run blazingly fast.

Rust does not have a garbage collector, but frees memory the moment it is not needed anymore, which it knows through the Lifetime every variable and reference (pointer) has. Ownership prevents you from modifying data structures in unintended

ways, and combined with lifetimes, preventing almost all segfaults. Also, it runs blazingly fast, comparable to C[1], and C++[2].

This is but an introduction to Rust, but the tooling must be mentioned. Rust has a dedicated package manager called `cargo`. Packages are available through `https://crates.io`. Also there are tools like `rustfmt` or `cargo-fix` (a subcommand that can be added later), that format Rust code after predefined guidelines, or fixes most compiler warnings automatically, respectively.

Rust has continuosly claimed StackOverflows position of 'most Loved Language'[3] for the last few years, while both C and C++ rank comparatavely high in the category 'dreaded'.

More information about Rust can be found here[4].

4.5.3 Advantages of Rust

The advantage of using Rust over using `numpy` / `scipy` from Python for this work might not be immediately obvious, since the CSRMatrix had to be implemented. We considered using the `numpy` C-API for a while, but its too big to be actually useful for our use-case.

The advantage here is actually much simpler: since we barely need any of the features provided, implementing them ourselves is not much work and gives us way more fine grained control as to what is actually happening.

[1] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html`, accessed 2019-06-26
[2] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-gpp.html`, accessed 2019-06-26
[3] `https://insights.stackoverflow.com/survey/2019#technology-_-most-loved-dreaded-and-wanted-languages`, accessed 2019-06-26
[4] `https://www.rust-lang.org/`, accessed 2019-06-26

This includes the paralellization of some parts of the code, which might not have been possible if ther were some other library doing things in its own way (numpy using the C-API would be an example here).

A future advantage is also the modularity of Rust code, meaning in the future additional external libraries (and with them, features) could easily be integrated.

4.5.4 Disadvantages of Rust

The disadvantages follow pretty much directly from looking at the advantages, we do not need much, but we had to implement it ourselves, there is not too much functionality in the case of the CSRMatrix, only the utterly necessary parts. This obviously limits the applicability of this code, no effort has been made to create a generalized solution - several CSRMatrix implementations already exist in Rust, none coming remotely close to the one in `scipy`, but ours is falling short of all the others by a wide margin (at least in most categories).

4.6 Choosing the right API to call Rust from Python

There are three main ways to execute Rust code from Python. In the following, common techniques are investigated.

One common way is rust-cpython. This library requires Rust 1.25 or higher (current versions are 1.33/34/35 for stable/beta/nightly respectively). 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 (directly from compiled rust) can easily be imported into Python (but needs to be renamed). Native Rust code requires some wrapping first, as shown here:

```rust
#[macro_use] extern crate cpython;
use cpython::{PyResult, Python};
// add bindings to the generated python module
// N.B: names: "librust2py" must be the name of
// the `.so` or `.pyd` file
py_module_initializer!(librust2py,
        initlibrust2py, PyInit_librust2py, |py, m| {
    m.add(py, "__doc__",
        "This module is implemented in Rust.")?;
    m.add(py, "sum_as_string",
        py_fn!(py, sum_as_string_py(a: i64, b:i64)))?;
    Ok(())
});
// logic implemented as a normal rust function
fn sum_as_string(a:i64, b:i64) -> String {
    format!("{}", a + b).to_string()
}
// rust-cpython aware function. All of our python
// interface could be declared in a separate module.
// Note that the py_fn!() macro automatically converts
// the arguments from Python objects to Rust values;
// and the Rust return value back into a Python object.
fn sum_as_string_py(_: Python, a:i64, b:i64)
        -> PyResult<String>
{
    let out = sum_as_string(a, b);
    Ok(out)
}
```

This kind of wrapping, though quite common and based on the Python C-API makes it hard to write idiomatic Code in Rust. Also, since Python is directly affected, the interactions with Python need to be considered while writing Rust-Code. In computer science one does usually not intentionally strive for complexity.

Another common approach is using the pyO3-library, which started off as a fork of rust-cpython, but has since seen quite drastic changes. For example, its using requires at least Rust version '1.30.0-nightly 2018-08-18' (or, in the newest version, '1.34.0-nightly 2019-02-06'). This is due to the usage of unstable features, most of which have recently been able to be promoted to stable. Unstable features are only available in the nightly toolchain. Still missing is Specialisation, which has at the time of writing still a long way to go. The library would also result in an easily importable (needs to be renamed first, still) cdylib (same as rust-cpython). The still intermingled way of writing the interface (certainly better but not by much compared to rust-cpython) as well as the dependency on unstable nightly rust versions led to the decision of not using it either.

The third way, that is actually been promoted in the official Rust docs, is to generate a dylib and import that in python. No renaming necessary, but the communication between Rust and Python is a bit more low-level. The main wrapper is on the side of Python, transforming Arguments to Pointers and C-Representations, whilst the Rust part needs to conform to C-practices, which includes receiving a list by getting a pointer to the first element and the length of it. Other than that, the Rust code has additional `\#[no_mangle]` and `\#[repr(C)]` (procedural) macros, preventing the compiler to mangle (renaming of functions) and guaranteeing the memory layout to be as it would be in C. Since like this neither language depends on something only internal (or combinatorial), and both just depend upon the 'common, unchanging' C-interface, this seems to be the preferred way.

4.7 General Approach

(TODO: Introduce main approach)

(DRAFT: Steps: read papers, test communication between Python and Rust (API), implementing CSRMatrix, implementing algorithm, fixing bugs (in-memory), making buildable (milksnake, conda), further bug fixing, some testing, building test-suite and a lot of bugfixing, reading papers again, starting to write stuff down.)

4.7.1 Beginning

Having read the provided papers ([3], [6] and [4]) I started looking in the Python-implementation. First things first I started testing the feasability of communicating between Rust and Python. The only available way for this is the raw C-API both adhere to.

4.7.2 Feasability Testing

Having succeded in calling functions in Rust, and passing the arguments correctly, I started to look in the Python-implementation again. Since in python `numpy` and `scipy` were used quite extensively (especially the Compressed Sparse Row Matrix from `scipy` and available operations through `numpy`) and there was no library available providing functionality similar enough, I implemented the minimal version of a CSRMatrix that would be needed, and tested its functionality.

4.7.3 Implementation of the algorithm

The initial translation from Python to rust happened more or less on a line-by-line basis, as much as this was possible. Seeing the Python-implementation section-wise as a comment I started out with a comparably naive translation from Python to Rust. As I did not have `numpy/scipy` available, specific operations had to be done differently, and I needed to care a lot more about the memory (of the variables, also their availability) than the Python-implementation did.

4.7.4 Testing and Bugfixing

Having succeded at convincing the compiler, I wanted to test my implementation. The compiler in Rust is quite capable, reducing common bugs tremendously. My knowledge about Rust not being on the expert-level, I made the error of not writing back the changes made to the matrix in the matrix (more specifically, the part that should have done that was being handed a immutable matrix). This and some smaller bugs got solved easily, so I set up a small testing environment, even calling from python.

4.7.5 Idiomatic Rust

While gradually transforming the naive Python-translation to idiomatic Rust, at some point results ended up being `NaN` pretty fast. A while of debugging later, I reduced it to the following situation:

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

```
println!("{}, sum: {}", list2, list2.iter().sum());
```

Here the output was still “sum: 1” and “sum: 0”. One iteration later however, all the elements have only been multiplied with some factors, their product being `0.16`.

```
// let list1 = vec![0.0, 0.0, 0.0, 0.160000000000003, 0.0, 0.0, 0.0, 0.0];
// let list2 = vec![0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0];
println!("{}, sum: {}", list1, list1.iter().sum());
println!("{}, sum: {}", list2, list2.iter().sum());
```

The expected output here would be “sum: 0.16” and “sum: 0”, or something around that. However, the actual results were “sum: 0.4” and “sum: inf”. 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 high fraction) is being sufficiently inaccurate to not be accurately represented by floating point values. With this happening multiple times, it was unavoidable.

The same happened with the summation of the innocious-looking `0.0`. They had high fractions from the original multiplication by `0.160000000000003`, their continued summation resulting in an overflow. This new number just happens to be one of the representations of `inf`.

4.7.6 Packaging

Next was the Packaging of my code. As my work should be used from within the HiCExplorer, My part is supposed to be available as a python-package. The only real

python dependcy (apart from those required for packages) ended up being milksnake, itself a helper for compiling the rust part of my package.

The conda-part was not as easy though, as milksnake was not resolvable there. I ended up porting `milksnake` as a conda package. This turned out to be a nontrivial task, as `conda skeleton pypi milksnake` created a package conda could not build, the issue here being that milksnake was only provided as a `*.zip` file and conda had hardcoded the format `*.tar.gz`.

Additionally I set up a buildserver, adding some tests and fixing smaller bugs.

4.7.7 Parallelizing

Nearing the end of my work, I set up ways to test and compare my implementation with the other. One of the last things I did was adding Parallelization.

4.8 Testing

I ran a total of 416 different configurations to test for a total of 3 different parameters. The first two are the same for all, the third applies only to this implementation. I tested the original Python-implementation as well as the new KR in C++.

- Size of Matrix (four different ones)
- Number of chromosomes (8 different sizes)
- Number of threads (11 different numbers)

The sizes of the matrices for reference (biggest to smallest):

```
# Matrix information file. Created with HiCExplorer's hicInfo version 3.0
File:   matrix.h5
Size:   309,581
Bin_length:     10000
Sum of matrix:  2416588411.2530212
Chromosomes:    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                14, 15, 16, 17, 18, 19, 20, 21, 22, X, Y, MT
Non-zero elements:      2,111,867,476
Minimum (non zero):     0.008667398294551536
Maximum:        139544.65657933566
NaN bins:       25948

# Matrix information file. Created with HiCExplorer's hicInfo version 3.0
File:   25kb_raw.h5
Size:   123,841
Bin_length:     25000
Sum of matrix:  2378265786.0
Chromosomes:    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                14, 15, 16, 17, 18, 19, 20, 21, 22, X, Y, MT
Non-zero elements:      1,530,533,003
Minimum (non zero):     1
Maximum:        320932
NaN bins:       9290
```

```
# Matrix information file. Created with HiCExplorer's hicInfo version 3.0
File:   50kb_raw.h5
Size:   61,928
Bin_length:     50000
Sum of matrix:  2333794628.0
Chromosomes:    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                14, 15, 16, 17, 18, 19, 20, 21, 22, X, Y, MT
Non-zero elements:      1,053,216,825
Minimum (non zero):     1
Maximum:        320932
NaN bins:       4514

# Matrix information file. Created with HiCExplorer's hicInfo version 3.0
File:   small_test_matrix.h5
Size:   33,754
Bin_length:     5000
Sum of matrix:  35778.0
Chromosomes:    chr2RHet, chr3RHet, chr2LHet, chr4, chrYHet, chr3L, chr2L,
                chrU, chrX, chrXHet, chr2R, chr3R, chrUextra, chrM, chr3LHet
Non-zero elements:      69,213
Minimum (non zero):     1
Maximum:        8
NaN bins:       0
```

(EXTEND: building matrix-test-suite and getting results)

5 Experiments

Experiments were run on a Server with the following specs (Specifics here[1]):

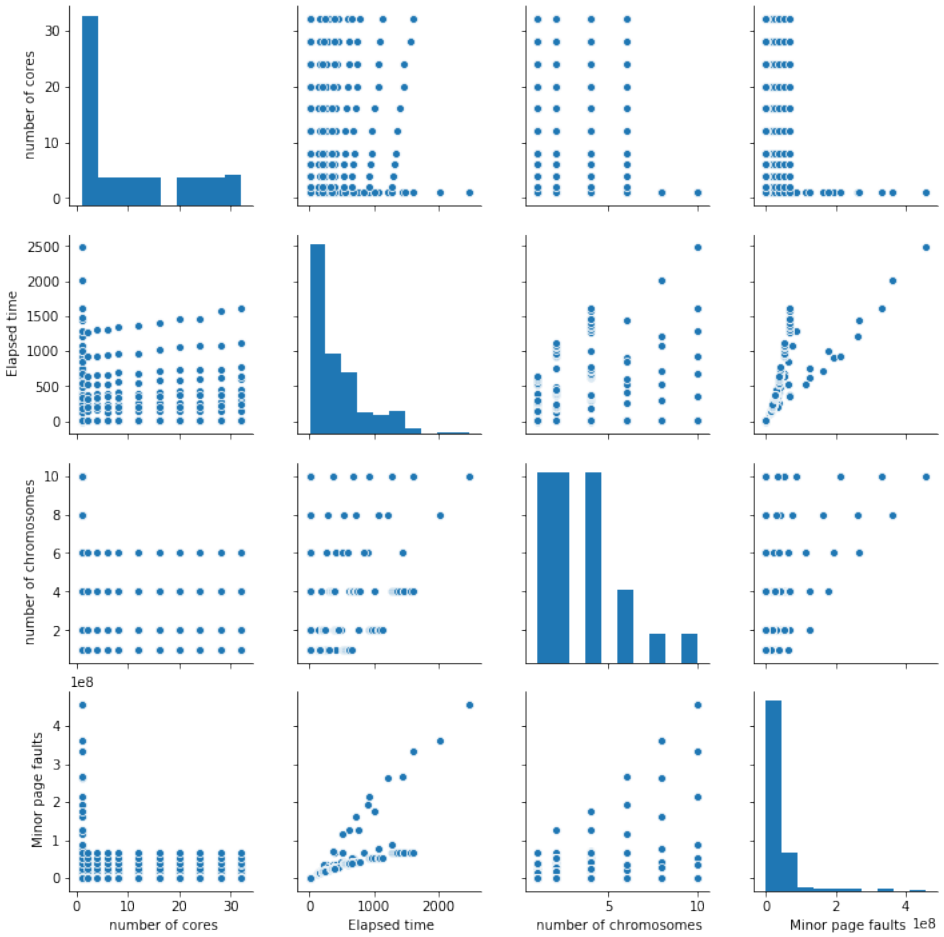
Processor	Intel® Xeon® Processor E5 v4 Family
Number	E5-2630V4
Performance	
Number of Cores	10
Number of Threads	20
Base frequency	2.2 GHz
Max Turbo frequency	3.1 GHz
Working Memory (RAM)	120 GByte

(TODO: The time-resource-measures done)

(DRAFT: Something is off with my numbers, they show something entirely different from earlier. I'll check that again.)

See Figure 2 (more of that will follow).

[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



(a) Pairplot over some variables

Figure 2: Pairplot over some of the variables ... moar info here

6 Conclusion

7 Acknowledgments

First and foremost, I would like to thank my primary supervisor, Joachim Wolff, for all the advice given.

(EXTEND: this!!)

ToDo Counters

To Dos: 10; 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Parts to extend: 4; 1, 2, 3, 4

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

Bibliography

[1] 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].

[2] 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].

[3] 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.

[4] 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.

[5] 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.

[6] 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.