List of Figures

1 Introduction

(DRAFT: Okay like this?)

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 however, which is probably due to PCR artifacts [3].

Now our approach is to just take this assumption, and "normalize" 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 [4].

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

1.1 Core setup

(DRAFT: Okay like this?)

This work is part of the HiCExplorer (Section 3.2) tool from the Deeptools (Section 3.1) framework. HiCExplorer is mainly written in Python, with this implementation needing too much working memory (RAM) for the iterative correction. For some time, the goal was to reduce memory usage by not copying a huge matrix a couple of times. From the Rust side, it would have been possible to read and write to the matrix in-memory, but we decided against doing so due to concerns regarding correctness. Still, the focus is to investigate which version requires less resources (CPU Time/RAM/...).

This is going to be an interesting comparison, since, the (before) default Python-implementation was only using one core. Rust code written for single-core applications can easily be turned to multicore code. During the work of this project, an implementation of a different algorithm (Section 3.4) written in C++ got added, so We'll compare with this one now as well.

(EXTEND: maybe add more details)

1.2 Algorithm

(TODO: describe the algorithm)

(TODO: Reference description from additional notes from the 2012-paper and) (EXTEND: the description of the algorithm to be easily undarstandable, include code (probably pseudocode, not python or rust)) Our fundamental assumption is that every location in our Matrix has in total as many interactions (with other locations) as every other location. Taking this in mind, the algorithm itself is pretty straightforward.

1.3 Operation

(DRAFT: Okay like this?)

(DRAFT: Change Name!) smb can be run on any Unix-based operating system (tested using ubuntu-18.04) with Python, Rust and common development packages installed (e.g. libopenssl-dev, python3-dev, build-essential, ...). For best performance, the size of the matrix should correlate with the number of available cores and the amount of available RAM **(EXTEND: Give rough factors!)**.

1.4 Motivation

(DRAFT: Okay like this?)

There is several reasons for using Rust, including faster development than C++ and better tooling, while still having the same if not better performance. A killer feature here is the easiness of adding parallelism, and the modularity of Rust code. Libraries in Rust do not do weird things, and the compiler will complain if the memory is not handled the way it should be (in C++ it is common practise to misunderstand which part should care about what memory, easily resulting in Segfaults and other hard to debug issues). 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.

(TODO: Add motivation for this whole Hi-C part)

2 Background

Explain the math and notation.

(TODO: Add section about DNA?)

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 seems to be kind of like a big ball of wool. With the help of Hi-C (and similar methods) we can visualize spacial proximity. Regulating **(DRAFT: correct wording?)** genes also have effect on their spatial neighbourhood, not only on the neighbourhood going up and down the strand of DNA.

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

- creating chromatin cross-links
- digesting the cross-linked chromatin and

- ligating the ends

to get a reversed cross-link sequence.

2.1.2 Cross-linking DNA

The first step, as shown in Figure 1 is to cross-link DNA strands that are close to each other spatially. This is done by adding formaldehyde, which bonds 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 whole ball of wool apart in certain intervals. For this, restriction-enzymes are used (specifically restriction endonuclease). Commonly used enzyme here would be EcoR1 or HindIII **(TODO: add reference or sth)**, cutting the genome about every 4000 base-pairs.

This is cutting the whole genome apart, leaving some (cross-) linked and some unlinked parts.

2.1.4 Ligation

Next, we randomly ligate (connect together) ends which we previously cut apart. This is done using biotin **(TODO: more details!)**. Some of the reconnected will have

been together before as well, so this is usually done in less concentrated environments. The goal here is to get the previously linked parts to be connected.

2.1.5 Reverse Cross-links

(EXTEND: This) As soon as the formaldehyde is removed, reverse cross-links remain. **(TODO: where does the name come from / what does it mean) (TODO: How is the formaldehyde removed)**

2.1.6 Sonication

(TODO: actually write this.)

2.1.7 Removal of biotin

(TODO: actually write this.)

2.1.8 Sequencing

(TODO: actually write this.)

(TODO: Cite/introduce/... the given papers, and introduce the required concepts)

Required concepts:

- Biology:
- Chromosome Conformation Capture
- Hi-C + pipeline

- The iterative algorithm (again ?)
- analysis that can be done further
- outlook. Meaning: What can be done, when having the 3C-Data?

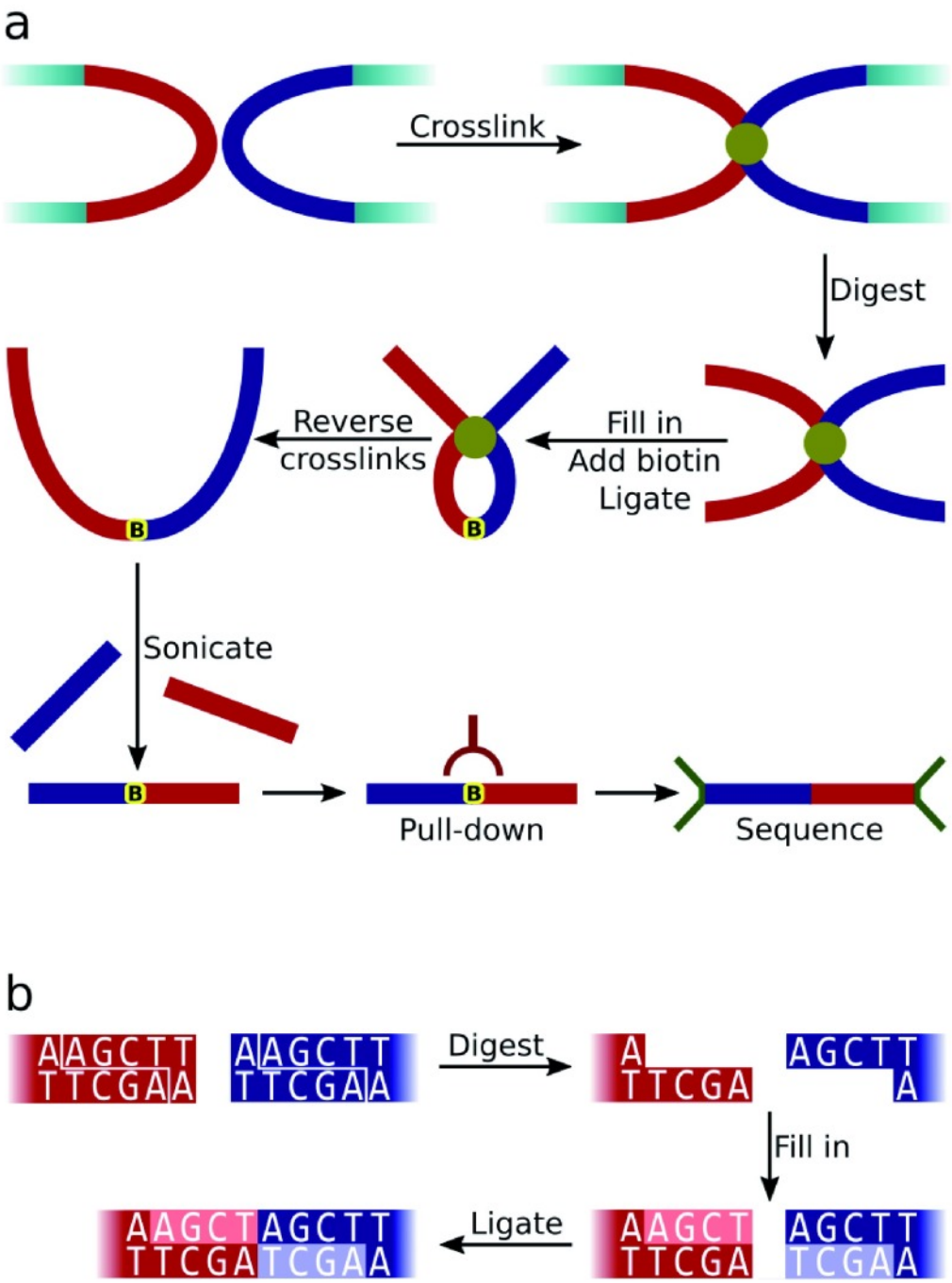

Figure 1: **a)** Diagram summarising the Hi-C experimental protocol. The red and blue rectangles represent cross-linked restriction fragments while the yellow marker shows the position of biotin incorporation. **b)** Generation of the Hi-C ligation junction sequence by successive digestion (with HindIII in this example), fill in and blunt-ended ligation steps. The modified restriction site sequence is not found in the original genomic sequence. Source: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4706059/?report=reader (HiCUP-paper)

3 Related Work

(TODO: introduce original implementation in python and KR-algorithm in C++)

3.1 Deeptools

(TODO: introduce the whole deeptools framework)

3.2 HiCExplorer

(TODO: introduce hicexplorer)

3.3 Python implementation

As the deeptools-framework is mainly written in Python, and this algorithm is a fundamentally important one, it was originally implemented in Python. This implementation however requires too much working memory, limiting the size of usable matrices, which is why a different implementation was asked for.

Rust and Python are two quite different programming languages, which is why a direct “translation” is not even possible. Both are semantically speaking the same,

but details differ. Since Rust has a much finer control about memory and the applying of functions to datastructures, some operations have been seperated 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. For smaller matrices the Overhead can be quite big,p

3.4 KR-algorithm

(TODO: read paper + introduce here)

(TODO: maybe also implement in Rust as well)

(TODO: is that really everything related?)

4 Approach

4.1 Problem

4.2 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:

```
#[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 Py
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'. This has been updated to '1.34.0-nightly 2019-02-06' with the most recently update. This is due to the usage of several unstable features, most of which have recently been able to be promoted to stable. Still missing is Specialisation though, 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 and the length of it. Other than that, the Rust code has some additional `\#[no_mangle]` and `\#[repr(C)]` (procedural) macros, which result in these parts actually accessible from e.g. Python or 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.

(TODO: Introduce main approach, problems I came across and more)

5 Experiments

times (precomputed correction factors):

```
542.32user 447.26system 16:26.02elapsed 100\%CPU (0avgtext+0avgdata 116107220maxresident)
14528inputs+8outputs (20major+159388812minor)pagefaults 0swaps
```

(TODO: The time-resource-measures done)

6 Conclusion

7 Acknowledgments

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

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: 4; 1, 2, 3, 4

Draft parts: 6; 1, 2, 3, 4, 5, 6

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] 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.

[4] 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.