

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 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

This work is part of the HiCExplorer (Section 2.1) tool from the Deeptools (Section 2.2) 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 ??) written in C++ got added, so We'll compare with this one now as well.

(EXTEND: describe the core setup better) (DRAFT: actually put in more words)

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 understandable, 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: Change Name!) smb can be run on any Unix-based operating system (tested using ubuntu-18.04 and macOS) 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!)**.

(TODO: Motivation as to why you'd want to do it at all and like this)
(EXTEND: The motivation being: Rust is fast and still memory-safe. Also can easily be parallelized.)

2 Related Work

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

2.1 HiCExplorer

(TODO: introduce hicexplorer and)

2.2 Deeptools

(TODO: introduce the whole deeptools framework)

(TODO: is that really everything related?)

3 Background

Explain the math and notation.

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

Required concepts:

- Biology:
- (Chromosome Conformation Capture - what is that even?)
- Hi-C + pipeline
- The iterative algorithm (again ?)
- analysis that can be done further

4 Approach

The approach usually starts with the problem definition and continues with what you have done. Try to give an intuition first and describe everything with words and then be more formal like ‘Let g be ...’.

4.1 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)k
14528inputs+8outputs (20major+159388812minor)pagefaults 0swaps
```

(TODO: The time-resource-measures done)

6 Conclusion

7 Acknowledgments

First and foremost, I would like to thank...

- advisers
- examiner
- person1 for the dataset
- person2 for the great suggestion
- proofreaders

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: 2; 1, 2

Literaturverzeichnis

- [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.

