Bioinformatics, 32(3), 2016, 444–446
doi: 10.1093/bioinformatics/btv573
Advance Access Publication Date: 6 October 2015
Applications Note

Sequence analysis

Rust-Bio: a fast and safe bioinformatics library

Johannes Köster

Center for Functional Cancer Epigenetics, Dana-Farber Cancer Institute, Department of Biostatistics and Computational Biology, Dana-Farber Cancer Institute, Harvard School of Public Health and Department of Medical Oncology, Dana-Farber Cancer Institute, Harvard Medical School, MA02215, Boston, USA.

Associate Editor: John Hancock

Received on June 20, 2015; revised on September 14, 2015; accepted on September 28, 2015

Abstract

Summary: We present Rust-Bio, the first general purpose bioinformatics library for the innovative *Rust* programming language. Rust-Bio leverages the unique combination of speed, memory safety and high-level syntax offered by Rust to provide a fast and safe set of bioinformatics algorithms and data structures with a focus on sequence analysis.
Availability and implementation: Rust-Bio is available open source under the MIT license at https://rust-bio.github.io.
Contact: koester@jimmy.harvard.edu
Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

With ever increasing amounts of experimental data being generated, their computational analysis becomes increasingly challenging. For novel or custom problems where carefully engineered high-performance standalone tools (like read mappers) are not yet available, general purpose bioinformatics libraries can help to minimize the coding effort. Bioinformatics libraries are published for many popular programming languages, e.g. SeqAn for C++, Biopython, Bioperl and BioRuby (Cock *et al.*, 2009; Döring *et al.*, 2008; Goto *et al.*, 2010; Stajich *et al.*, 2002). Choosing the programming language for a specific task usually entails a tradeoff between execution and development speed. Low-level system programming languages like C or C++ provide optimal performance at the cost of increased complexity. Higher level languages like Python or Perl provide a more concise syntax while leading to computational overhead introduced by online memory management (e.g. reference counting or garbage collection), type inference and not being compiled but interpreted during execution. Often, the combination of a high-level language with some carefully engineered implementations of a bioinformatics library is a good choice to quickly solve a problem with reasonable performance. However, the amounts of data the bioinformatics community is facing in the coming years and the need to handle nature's resources carefully implies that using a high-performance, compiled language is still beneficial for certain problems.

Recently, *Rust* (http://www.rust-lang.org) has gained attention as a new programming language combining speed with memory safety and high-level syntactical features. Being compiled with LLVM (Lattner and Adve, 2004), Rust has many advantages of low-level, system programming languages, such as speed and a small memory footprint. Supporting automatic type inference, its code is often less verbose than C or C++ code. With Rust, type inference happens at compile time, such that runtime overhead (appearing with scripting languages like Python) can be avoided. The key feature of Rust is a concept of ownership and borrowing of variables, that enables the compiler to automatically decide about lifetime of objects during compile time, making an online memory management superfluous without requiring manual freeing of resources. At the same time, this concept prevents common sources of errors with low-level languages like accessing invalid memory regions. Finally, the ownership concept enforces thread-safety, such that race conditions cannot occur. These features make Rust a promising solution to above tradeoff problem.

In this work, we present Rust-Bio, the first general purpose bioinformatics library for the Rust programming language. Rust-Bio provides a high-level, fast and safe API for many state-of-the-art data structures and algorithms used in bioinformatics.

2 Library

Rust-Bio is built with the following principles in mind. Where possible, iterators are returned. This allows to process streams of data with minimal memory footprint. On top, using the extensive set of iterator tools available in Rust, iterators can be, e.g. filtered, modified, chained or combined in an easy way. If a language data type appears suitable, we avoid to enclose data into a custom object. This mimimizes memory usage and increases flexibility when handling the data, e.g. biological sequences are represented as vectors or slices of bytes in ASCII encoding. This allows to use sequences with all algorithms and functions in, e.g. the Rust standard library that work with byte vectors or slices. Each implemented algorithm is automatically tested via continuous integration (https://travis-ci.org). For each algorithm and data structure, we provide complexities in the documentation. Where more than one alternative is available, the documentation tries to highlight distinguishing use cases. So far, Rust-Bio is focused on algorithms and data structures for biological sequences. A central component of Rust-Bio are *alphabets*, which, e.g. allow to check in linear time whether a given sequence is a word over the alphabet, transform symbols to their lexicographical ranks and perform bit-encoding to save memory or iterate over q-grams. Rust-Bio can read and write common file formats like FASTA, FASTQ and BED. For SAM/BAM, CRAM and VCF/BCF support, it is complemented by Rust-HTSlib.

Listing 1 Creating an FM-Index for a given sequence with an occurrence table sampling rate of 3. Here, the alphabet is used to provide guarantees for being able to limit memory usage during FM-Index construction. Afterward, we iterate over a FASTQ file, use the alphabet to validate read sequences and search for exact matches in the FM-Index. This example illustrates how to create a simple read mapper with Rust-Bio.

```
let alphabet = alphabets::dna::iupac_alphabet();
let pos = suffix_array(text);
let bwt = bwt(text, &pos);
let fmindex = FMIndex::new(&bwt, 3, &alphabet);

let reader = fastq::Reader::from_file('reads.fastq');
for record in reader.records() {
    let seq = record.seq();
    if alphabet.is_word(seq) {
        let interval = fmindex.backward_search(seq.
          iter());
        let positions = interval.occ(&pos);
    }
}
```

Especially when considering sequencing data, many problems can be solved with a set of well-established data structures like suffix arrays (Manber and Myers, 1990), the Burrows-Wheeler Transform (Burrows and Wheeler, 1994), rank/select data structures (Jacobson, 1988) and *q*-gram indices. In line with that, Rust-Bio implements induced sorting for suffix array construction (Nong *et al.*, 2009), the FM-Index (Ferragina and Manzini, 2000) for pattern matching on top of the Burrows-Wheeler Transform, a practical variant of a rank/select data structure (González *et al.*, 2005) and a *q*-gram index for arbitrary alphabets and $q \leq 32$. Further, Rust-Bio implements the FMD-Index (Li, 2012), that allows to find supermaximal exact matches in DNA sequences and their reverse complements in linear time.

Implementations for many classical pattern matching algorithms are provided, including the algorithm of Knuth, Morris and Pratt, Backward Nondeterministic DAWG Matching, Backward Oracle Matching, the algorithm of Horspool and the Shift-And algorithm (Allauzen *et al.*, 1999; Gonzalo Navarro, 1998; Horspool, 1980; Knuth *et al.*, 1977; Wu and Manber, 1992). In the Supplementary Material, we compare the speed of these algorithms against the C++-based Seqan, which is among the fastest bioinformatics libraries (Döring *et al.*, 2008). The benchmarks exemplify that the speed of Rust-Bio is comparable to that of C++-based implementations. For approximate pattern matching, Ukkonen's dynamic programming-based algorithm (Ukkonen, 1985) and Myer's bit-parallel algorithm (Myers, 1999) are provided. Finally, Rust-Bio implements local, global and semi-global pairwise sequence alignment as variants of the Smith–Waterman and Needleman–Wunsch algorithms (Needleman and Wunsch, 1970; Smith and Waterman, 1981). An example for using the Rust-Bio API can be seen in Listing 1.

3 Conclusion

Rust-Bio is a general purpose bioinformatics library. Building on the innovative Rust programming language, Rust-Bio combines memory safety with speed, complemented by rigorous continuous integration tests. So far, a wide set of algorithms and data structures for biological sequences is provided, ranging from index data structures to pattern matching and alignment, complemented by readers and writers for common file formats.

Acknowledgements

We thank Christopher Schröder and Peer Aramillo Irizar for code contributions as well as Sven Rahmann, Dominik Kopczynski, Tobias Marschall and Marcel Martin for their inspiring lecture notes 'Algorithms on Sequences'.

Conflict of Interest: none declared.

References

Allauzen,C. *et al.* (1999) Factor oracle: a new structure for pattern matching. *Lect. Notes Comput. Sci.*, **1725**, 1–16.

Burrows,M. and Wheeler,D. (1994) A block-sorting lossless data compression algorithm. *Algorithm Data Compression*, (124), 18.

Cock,P.J.A. *et al.* (2009) Biopython: freely available python tools for computational molecular biology and bioinformatics. *Bioinformatics*, **25**, 1422–1423.

Döring,A. *et al.* (2008) Seqan an efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, **9**, 11.

Ferragina,P. and Manzini,G. (2000) Opportunistic data structures with applications. In: Young,D.C. (ed.), *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, Redondo Beach, pp. 390–398.

González,R. *et al.* (2005) Practical implementation of rank and select queries. In: *Poster Proceedings Volume of the 4th Workshop on Efficient and Experimental Algorithms (WEA05)*, pp. 27–38.

Gonzalo Navarro,M.R. (1998) A bit-parallel approach to suffix automata: fast extended string matching. In: Farach-Colton,M. (ed.), *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching, Lecture Notes in Computer Science*, Piscataway, New Jersey, pp. 14–31.

Goto,N. *et al.* (2010) Bioruby: bioinformatics software for the ruby programming language. *Bioinformatics*, **26**, 2617–9.

Horspool,R.N. (1980) Practical fast searching in strings. *Softw. Pract. Exp.*, **10**, 501–506.

Jacobson,G.J. (1988) Succinct static data structures. PhD Thesis, Carnegie Mellon University Pittsburgh.

Knuth,D.E. *et al.* (1977) Fast pattern matching in strings. *SIAM J. Comput.*, **6**, 323–350.

Lattner,C. and Adve,V. (2004) Llvm: A compilation framework for lifelong program analysis & transformation. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE*, San Jose, pp. 75–86.

Li,H. (2012) Exploring single-sample SNP and indel calling with whole-genome de novo assembly. *Bioinformatics*, **28**, 1838–1844.

Manber,U. and Myers,G. (1990) Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, **22**, 935–948.

Myers,G. (1999) A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, **46**, 395–415.

Needleman,S.B. and Wunsch,C.D. (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, **48**, 443–453.

Nong,G. *et al.* (2009) Linear suffix array construction by almost pure induced-sorting. In: Storer,J.A. and Marcellin,M.W. (eds), *2009 Data Compression Conference. IEEE*, Snowbird, Utah, pp. 193–202.

Smith,T.F. and Waterman,M.S. (1981) Identification of common molecular subsequences. *J. Mol. Biol.*, **147**, 195–197.

Stajich,J.E. *et al.* (2002) The bioperl toolkit: Perl modules for the life sciences. *Genome Res.*, **12**, 1611–1618.

Ukkonen,E. (1985) Algorithms for approximate string matching. *Inform. Control*, **64**, 100–118.

Wu,S. and Manber,U. (1992) Fast text searching: allowing errors. *Commun. ACM*, **35**, 83–91.