Van der Jeugt *et al. BMC Bioinformatics (2022) 23:198*
https://doi.org/10.1186/s12859-022-04736-5

BMC Bioinformatics

SOFTWARE Open Access

FragGeneScanRs: faster gene prediction for short reads

Felix Van der Jeugt[*], Peter Dawyndt and Bart Mesuere

*Correspondence: unipept@ugent.be

Department of Applied Mathematics, Computer Science and Statistics, Ghent University, Ghent, Belgium

Abstract

Background: FragGeneScan is currently the most accurate and popular tool for gene prediction in short and error-prone reads, but its execution speed is insufficient for use on larger data sets. The parallelization which should have addressed this is inefficient. Its alternative implementation FragGeneScan+ is faster, but introduced a number of bugs related to memory management, race conditions and even output accuracy.

Results: This paper introduces FragGeneScanRs, a faster Rust implementation of the FragGeneScan gene prediction model. Its command line interface is backward compatible and adds extra features for more flexible usage. Its output is equivalent to the original FragGeneScan implementation.

Conclusions: Compared to the current C implementation, shotgun metagenomic reads are processed up to 22 times faster using a single thread, with better scaling for multithreaded execution. The Rust code of FragGeneScanRs is freely available from GitHub under the GPL-3.0 license with instructions for installation, usage and other documentation (https://github.com/unipept/FragGeneScanRs).

Keywords: Shotgun metagenomics, Gene prediction, Hidden markov model, Rust

Background

Studying environmental communities of archaea, bacteria, eukaryotes, and viruses is hampered by problems with isolating and culturing most of these organisms in lab conditions [1–5]. Metagenomics has therefore become a routine technique to bypass the cultivation step with a combination of high-throughput DNA sequencing and computational methods [6, 7]. Non-targeted sequencing of genomes in environmental samples, called shotgun metagenomics, in particular allows profiling of both the taxonomic composition and the functional potential of the samples [8]. Identification of protein coding sequences from shotgun metagenomic reads has therefore become an important precursor to gain insight in the taxonomic and functional diversity of an environmental community [9].

One approach for shotgun metagenomics gene prediction is assembling reads into longer contiguous sequences, called contigs, prior to running traditional gene prediction tools [10]. This eases gene prediction, but assembling reads from complex samples

© The Author(s) 2022. **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/. The Creative Commons Public Domain Dedication waiver (http://creativecommons.org/publicdomain/zero/1.0/) applies to the data made available in this article, unless otherwise stated in a credit line to the data.

with many species in different abundances is a challenging problem. It requires special-purpose algorithms that can be slow, produce artificial contigs, and miss low-abundance genomes [11, 12]. Direct gene prediction on individual reads can mitigate assembly problems, speed up computations, and enable profiling of low-abundance organisms that cannot be assembled de novo. But, it does have to face partial protein coding fragments with missing start/stop codons and read errors [8]. The choice between assembly-based versus read-based gene prediction may depend on the sample type and the research question at hand.

Various gene prediction tools specialize in directly calling short reads [13–16]. FragGeneScan (FGS) is the most accurate and popular tool that is currently available [17]. It uses a hidden Markov model (HMM) that incorporates codon usage bias, start/stop codon patterns, and sequencing error models to predict complete or partial genes in short error-prone reads. Back in 2010, the gene prediction model of FGS was implemented in C and Perl [18]. Multithreading support was added in release 1.19 (August 2014) and most Perl code was replaced with C functions in release 1.30 (April 2016). Because FGS was rather slow, a team of researchers at the University of British Columbia (Canada) forked FGS (presumably from release 1.19) to implement FragGeneScan-Plus (FGS+): a pure C implementation with speedups for single threaded execution and better scaling for multithreading [19].

Both FGS and FGS+ now have pure C implementations that support parallel execution, but their latest releases suffer from their own issues. FGS implements multithreading in a very inefficient way, making it much slower than FGS+. The implementation doesn't preserve input order, breaking for example the synchronisation between pair-end read files without an extra sorting step in postprocessing. Bugs have also been introduced when replacing Perl code with C functions. In addition, out-of-bound memory access may corrupt its results and cause the software to crash.

FGS+ has inherited inefficient memory usage from FGS and made it worse by copying immutable data structures to individual threads and introducing leaks in the allocated memory. Its multithreading model is overly complex and may deadlock due to race conditions in thread semaphores especially when using a larger number of threads. Although FGS+ is faster than FGS, its results may significantly deviate due to bugs introduced in the reimplementation and missing bug fixes from later FGS versions. For example, FGS+ systematically makes wrong translations of genes encoded on the reverse strand, may result in out-of-bound memory access when copying FASTA headers from a dynamically allocated global array to thread-specific arrays that are statically allocated, uses fixed-length string buffers of 1MB for DNA sequences that overflow for complete genomes, and also crashes when reading from standard input and writing to standard output. A list of problematic invocations is included in Additional file 1. In conclusion, FGS generates more accurate results but is slower, whereas FGS+ is considerably faster but generates wrong output.

The source code of FGS and FGS+ is no longer actively maintained, but as metagenomics datasets continue to grow in size, we still need faster gene predictors. In this manuscript we present FragGeneScanRs (FGSrs), a reliable, high-performance, and accurate Rust implementation of the FGS gene prediction model. We ran a benchmark to show that FGSrs produces the same results as FGS and is faster than both FGS and FGS+.

Implementation

FGSrs is implemented in Rust, a programming language known for its focus on speed and memory-efficiency. In addition, segmentation faults that occur while running FGS or FGS+ are automatically avoided because memory-safety and thread-safety are guaranteed by Rust's type system and ownership model. Its zero-cost abstractions yield more readable code and put optimizations in the hands of the compiler.

We started off with a Rust implementation that was equivalent to FGS release 1.31. Afterwards, we gradually optimized performance and improved the quality of the software, while monitoring equivalence with the original implementation. We outline some of these optimizations and improvements in what follows and refer to the source code and its documentation for more details.

FGS uses statically allocated 300 KB buffers for different representations of reads and 100 KB buffers for protein translations. By using dynamically allocated buffers that grow when needed instead, we avoid static buffer overflows, increase speed, and reduce the memory footprint.

FGS stores default training data for its HMM in separate text files. We include a binary representation of default data in the executable during compilation, but still support passing custom training data as command line arguments. This improves usability by reducing dependencies during installation of the software and allowing to execute FGSrs with default training data anywhere on the system without the need to run FGS in its own directory or explicitly pass the path to the default training files (new option `-r`).

The HMM configurations used by FGS are immutable after initialization, but the original implementation wastes memory by copying them to each thread. We store this data in shared memory that threads can access concurrently. We use mutexes for protected access to shared input and output file handles. This avoids the need to split input in chunks upfront, have threads that store results per chunk in a separate file, and merge these files afterwards. This eliminates disk overhead and speeds up I/O.

HMM gene regions have six inhomogeneous sets of states that represent matches, insertions and deletions for two successive codons in a read. FGS processes the six states in a loop, combined with conditional execution to handle topological differences between state transitions. Unrolling these loops not only makes the condition-less code more readable, it is also significantly faster. We have further improved readability of the code by replacing `#define` constants in C with Rust enum types.

Where FGS uses row-major order to store dynamic programming matrices, we switched to column-major order for improved locality when accessing matrix elements in the Viterbi algorithm. This results in a speedup because the reordered memory layout causes less cache misses.

FGS outputs the DNA sequence, its translated amino acid sequence, and additional metadata for each coding region found in a read, with optional formatting to indicate insertions and deletions in the DNA sequence. This requires FGS to compute and store the DNA sequence twice during the backtracking step at the end of the Viterbi algorithms. Once for reconstructing the formatted DNA sequence and once for reconstructing the unformatted DNA sequence. We produce a unified representation and delay formatting until output is generated. We also note that a bug was introduced in FGS

(release 1.30) when the backtracking step was converted from Perl to C, which generates DNA and protein sequences for complete genomes that are incorrect.

Results

The FGSrs command line interface is backward compatible with FGS, so it can be used as a faster and memory-friendly drop-in replacement for FGS in bioinformatics pipelines. FGSrs also has some additional features that enable more flexible usage. We support the conventional standard input and output channels as alternatives for passing files as arguments to the `-s` and `-o` options. This enables embedding FGSrs in POSIX pipes without storing intermediate results on the file system. Storage locations for generated DNA sequences (option `-n`), translated amino acid sequences (option `-a`) and metadata (option `-m`) can also be specified individually, which may give additional speedups because unspecified information does not need to be computed. Where FGS and FGS+ report gene predictions out-of-order, FGSrs by default preserves read order. The priority queue that guarantees in-order reporting comes with a small overhead on speed and memory usage, but can be disabled with the option `-u`.

We ran two benchmarks on a 16-core Intel® Xeon® CPU E5-2650 v2 at 2.60 GHz, to evaluate the performance improvements of FGSrs (release 1.0.0) over FGS (release 1.31) and FGS+ (git version 91b0ab6). The first benchmark uses the sample datasets included in the FGS and FGSrs repositories as input data and measurements are averaged over 5 runs. When all three implementations are executed single threaded (Fig. 1), FGSrs processes short reads (80 bp) 22.6× faster than FGS and 1.2× faster than FGS+. Long reads (1328 bp) are processed 4.2× times faster than FGS and 1.6× faster than FGS+. The bulk of the runtime is consumed by the Viterbi algorithm, having a time complexity of $O(s^2 \times n)$ with s the number of HMM states and n the length of the sequence that needs to be processed. Because the HMM of the gene prediction model has a fixed number of states ($s = 49$), we can thus expect runtime to grow linearly for reads that are

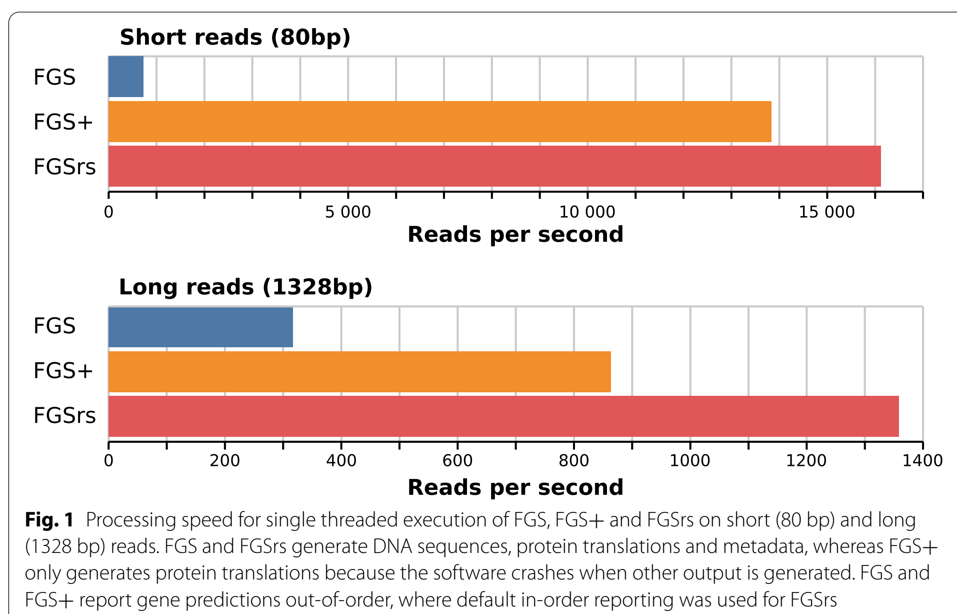

Fig. 1 Processing speed for single threaded execution of FGS, FGS+ and FGSrs on short (80 bp) and long (1328 bp) reads. FGS and FGSrs generate DNA sequences, protein translations and metadata, whereas FGS+ only generates protein translations because the software crashes when other output is generated. FGS and FGS+ report gene predictions out-of-order, where default in-order reporting was used for FGSrs

even longer. FGSrs processes the complete genome sequence of *Escherichia coli* str. K-12 subst. MG1665 (NC_000913; 4,639,675 bp) 2.2× faster than FGS and 347.6× faster than FGS+. The latter measurement essentially shows that FGS+ is not fit for processing complete genomes in practice. FGSrs and FGS+ scale better for multithreaded execution than FGS (Fig. 2). Increasing the thread count from 1 to 8 results in a speedup factor of 6.5 for both FGSrs and FGS+ and only 5.4 for FGS. The difference increases for higher thread counts and the execution even consistently halts due to race conditions for FGS+ when using more than 10 threads. FGSrs and FGS have a comparable memory footprint of about $(80 + 70t)$ MB for t threads (Fig. 3). FGS+ consumes more memory with about $(265 + 85t)$ MB for t threads. However, the memory requirements are not a limiting factor to run any of these tools on a standard laptop, with 4 threads needing between 350MB and 520MB RAM.

The second benchmark uses a collection of data sets simulated using Mason [21]. Each data set contains 10K Illumina reads of varying read length simulated from *Geobacter anodireducens* str. SD-1. The results for these datasets may be found in Table 1. More details and a comparison of the predictive performance of FGS and FGSrs on one of these data sets may be found in Additional file 2.

Conclusions

In conclusion we can state that FGSrs is a reliable implementation of the FGS gene prediction model that is an order of magnitude faster than the original implementation. Its command line interface is backward compatible with extensions for more flexible usage.

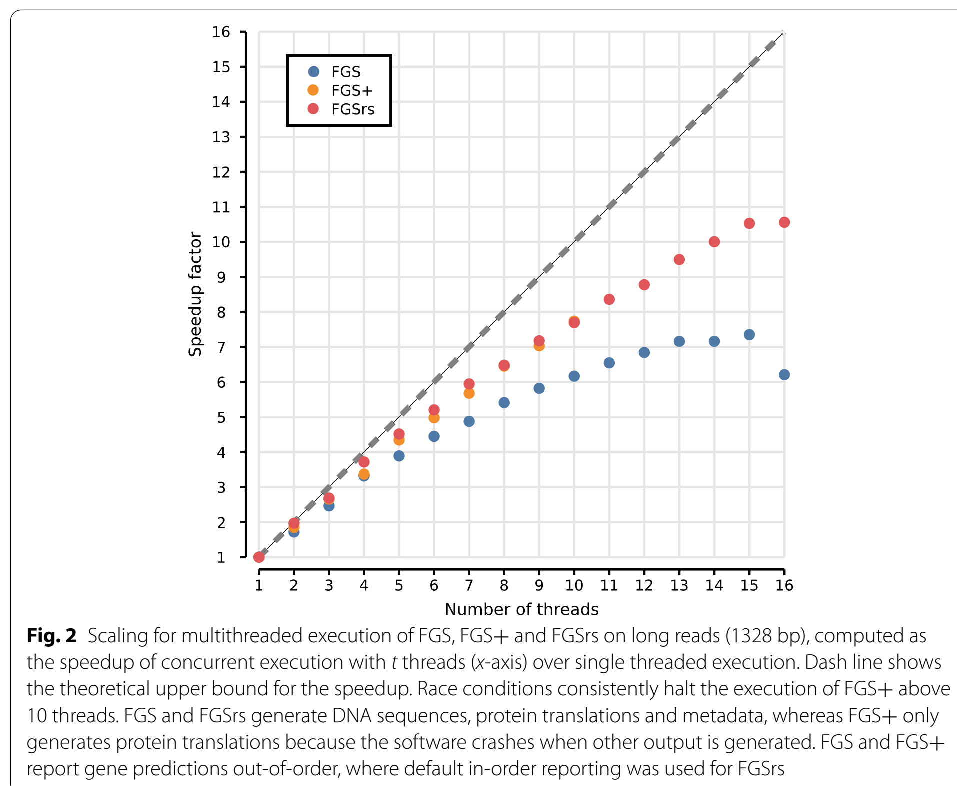

Fig. 2 Scaling for multithreaded execution of FGS, FGS+ and FGSrs on long reads (1328 bp), computed as the speedup of concurrent execution with *t* threads (*x*-axis) over single threaded execution. Dash line shows the theoretical upper bound for the speedup. Race conditions consistently halt the execution of FGS+ above 10 threads. FGS and FGSrs generate DNA sequences, protein translations and metadata, whereas FGS+ only generates protein translations because the software crashes when other output is generated. FGS and FGS+ report gene predictions out-of-order, where default in-order reporting was used for FGSrs

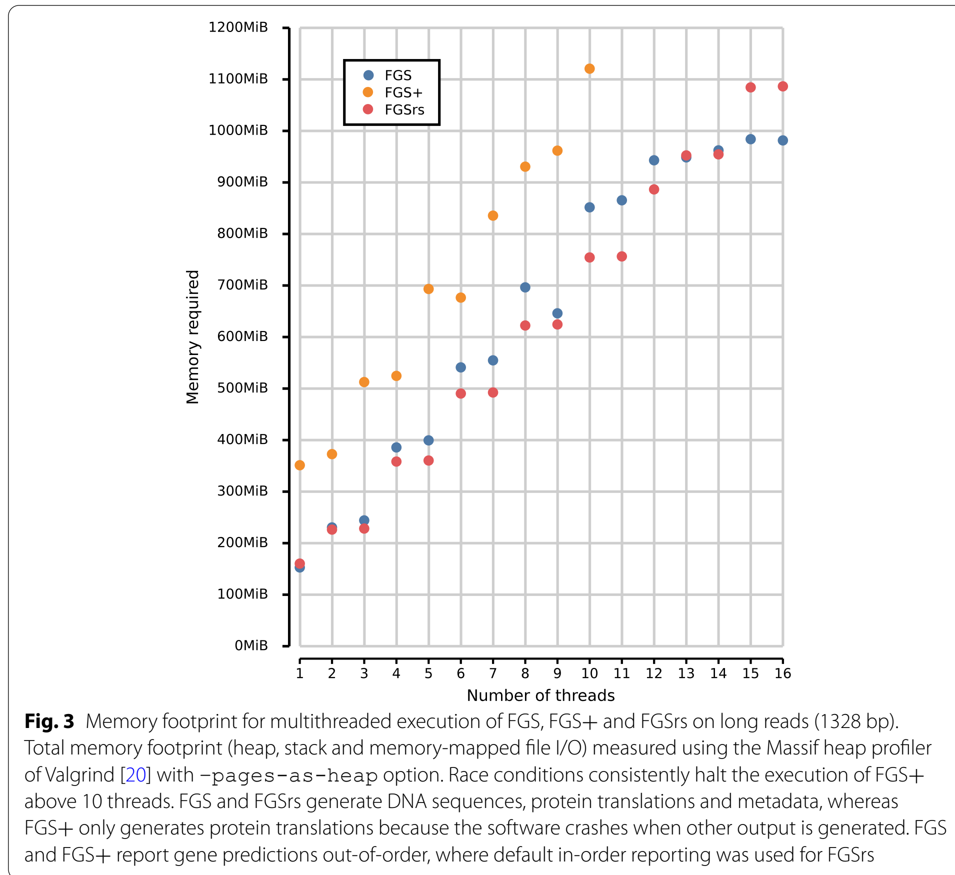

Fig. 3 Memory footprint for multithreaded execution of FGS, FGS+ and FGSrs on long reads (1328 bp). Total memory footprint (heap, stack and memory-mapped file I/O) measured using the Massif heap profiler of Valgrind [20] with `--pages-as-heap` option. Race conditions consistently halt the execution of FGS+ above 10 threads. FGS and FGSrs generate DNA sequences, protein translations and metadata, whereas FGS+ only generates protein translations because the software crashes when other output is generated. FGS and FGS+ report gene predictions out-of-order, where default in-order reporting was used for FGSrs

Table 1 Average execution time and standard deviation for various read lengths

Average read length	FGS Mean [s]	FGS+ Mean [s]	FGSrs Mean [s]
100	4.843 ± 0.018	0.491 ± 0.002	0.396 ± 0.004
200	5.966 ± 0.028	0.842 ± 0.001	0.748 ± 0.002
300	7.020 ± 0.011	1.168 ± 0.001	1.103 ± 0.012
400	8.073 ± 0.020	1.487 ± 0.002	1.451 ± 0.002
500	9.109 ± 0.016	1.803 ± 0.004	1.795 ± 0.002
600	10.122 ± 0.021	2.115 ± 0.002	2.154 ± 0.015
700	11.137 ± 0.026	2.448 ± 0.007	2.505 ± 0.008
800	12.196 ± 0.020	2.790 ± 0.005	2.859 ± 0.010
900	13.218 ± 0.023	3.132 ± 0.003	3.198 ± 0.010
1000	14.309 ± 0.255	3.474 ± 0.018	3.547 ± 0.005
2000	24.443 ± 0.025	7.021 ± 0.020	7.052 ± 0.024
3000	34.655 ± 0.038	10.913 ± 0.014	10.545 ± 0.025
4000	45.037 ± 0.051	14.841 ± 0.020	14.042 ± 0.022
5000	55.315 ± 0.059	18.756 ± 0.034	17.525 ± 0.039
6000	65.537 ± 0.118	22.997 ± 0.298	21.007 ± 0.028
7000	75.791 ± 0.058	27.099 ± 0.080	24.521 ± 0.068
8000	86.172 ± 0.202	31.440 ± 0.051	27.980 ± 0.043
9000	96.354 ± 0.347	35.858 ± 0.040	31.489 ± 0.055
10,000	106.509 ± 0.104	40.513 ± 0.071	34.962 ± 0.050

The source code of FGSrs is freely available from GitHub under the GPL-3.0 license (https://github.com/unipept/FragGeneScanRs), with instructions for installation, usage and other documentation.

Abbreviations
base pair: (bp); FragGeneScan: (FGS); FragGeneScan+: (FGS+); FragGeneScanRs: (FGSrs); Hidden Markov Model: (HMM).

Supplementary Information

The online version contains supplementary material available at https://doi.org/10.1186/s12859-022-04736-5.

Additional file 1. Examples of input problematic for FGS+. A PDF file describing in detail some examples of input problematic for FGS+

Additional file 2. Performance on simulated reads. Detailed benchmark results on a collection of simulated data sets

Acknowledgements
We thank the students of the Computational Biology class of 2019-2020 for scrutinizing issues with the code of FGS and FGS+.

Availability and requirements

Project name FragGeneScanRs
Project home page https://github.com/unipept/FragGeneScanRs
Operating system(s) Platform independent
Programming language Rust
Other requirements Rust edition 2018 or higher
License GNU GPL
Any restrictions to use by non-academics None

Author contributions
BM, PD and FVDJ contributed to the conception of the software. FVDJ created the new software used in the work. BM, PD and FVDJ drafted the work or substantively revised it. All contributors agree to be accountable for all aspects of the work in ensuring that questions related to the accuracy of integrety of any part of the work are appropriately investigated and resolved. All authors read and approved the final manuscript.

Funding
This work was supported by the Research Foundation–Flanders (FWO) [12I5220N to B.M.]. The funding body did not play any role in the design of the study, or collection, analysis and interpretation of data, or in writing the manuscript.

Availability of data and materials
The datasets generated and/or analysed during the current study are available in the GitHub repository, https://github.com/unipept/FragGeneScanRs/tree/main/example.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Received: 9 December 2021 Accepted: 17 May 2022
Published online: 28 May 2022

References

1. Locey KJ, Lennon JT. Scaling laws predict global microbial diversity. Proc Natl Acad Sci. 2016;113(21):5970–5. https://doi.org/10.1073/pnas.1521291113.
2. Rappé MS, Giovannoni SJ. The uncultured microbial majority. Annu Rev Microbiol. 2003;57(1):369–94. https://doi.org/10.1146/annurev.micro.57.030502.090759.
3. Pedrós-Alió C, Manrubia S. The vast unknown microbial biosphere. Proc Natl Acad Sci. 2016;113(24):6585–7. https://doi.org/10.1073/pnas.1606105113.

4. Hofer U. The majority is uncultured. Nat Rev Microbiol. 2018;16:716–7.
5. Hahn MW, Koll U, Schmidt J. Isolation and Cultivation of Bacteria, pp. 313–351. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16775-2_10.
6. Hugenholtz P, Tyson GW. Metagenomics. Nature. 2008;455:481–3. https://doi.org/10.1038/455481a.
7. Thomas T, Gilbert J, Meyer F. MCS), Univ. of New South Wales, S.: Metagenomics - a guide from sampling to data analysis. Microbial Informatics and experimentation **2**(3) (2012). https://doi.org/10.1186/2042-5783-2-3
8. Quince C, Walker A, Simpson J, Loman N, Segata N. Shotgun metagenomics, from sampling to analysis. Nat Biotechnol. 2017;35(9):833–44. https://doi.org/10.1038/nbt.3935.
9. Sharpton TJ. An introduction to the analysis of shotgun metagenomic data. Front Plant Sci. 2014;5:209. https://doi.org/10.3389/fpls.2014.00209.
10. Breitwieser FP, Lu J, Salzberg SL. A review of methods and databases for metagenomic classification and assembly. Brief Bioinform. 2017;20(4):1125–36. https://doi.org/10.1093/bib/bbx120.
11. Ghurye J, Cepeda-Espinoza V, Pop M. Metagenomic assembly: overview, challenges and applications. Yale J Biol Med. 2016;89:353–62.
12. Vollmers J, Wiegand S, Kaster A-K. Comparing and evaluating metagenome assembly tools from a microbiologist's perspective - not only size matters! PLoS ONE. 2017;12(1):1–31. https://doi.org/10.1371/journal.pone.0169662.
13. Hyatt D, LoCascio PF, Hauser LJ, Uberbacher EC. Gene and translation initiation site prediction in metagenomic sequences. Bioinformatics. 2012;28(17):2223–30. https://doi.org/10.1093/bioinformatics/bts429.
14. Hoff KJ, Lingner T, Meinicke P, Tech M. Orphelia: predicting genes in metagenomic sequencing reads. Nucleic Acids Res. 2009;37(suppl2):101–5. https://doi.org/10.1093/nar/gkp327.
15. Zhu W, Lomsadze A, Borodovsky M. Ab initio gene identification in metagenomic sequences. Nucleic Acids Res. 2010;38(12):132–132. https://doi.org/10.1093/nar/gkq275.
16. Noguchi H, Taniguchi T, Itoh T. MetaGeneAnnotator: detecting species-specific patterns of ribosomal binding site for precise gene prediction in anonymous prokaryotic and phage genomes. DNA Res. 2008;15(6):387–96. https://doi.org/10.1093/dnares/dsn027.
17. Trimble WL, Keegan KP, D'Souza M, Wilke A, Wilkening J, Gilbert J, Meyer F. Short read reading-frame predictors are not created equal: sequence error causes loss of signal. BMC Bioinform. 2012. https://doi.org/10.1186/1471-2105-13-183.
18. Rho M, Tang H, Ye Y. FragGeneScan: predicting genes in short and error-prone reads. Nucleic Acids Res. 2010;38(20):191–191. https://doi.org/10.1093/nar/gkq747.
19. Kim D, Hahn AS, Wu S-J, Hanson NW, Konwar KM, Hallam SJ. Fraggenescan-plus for scalable high-throughput short-read open reading frame prediction. In: 2015 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB), pp. 1–8 (2015). https://doi.org/10.1109/CIBCB.2015.7300341
20. Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In: 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07), pp. 89–100. Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1250734.1250746.
21. Holtgrewe M. Mason - a read simulator for second generation sequencing data. Technical Report FU Berlin (2010)

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Ready to submit your research? Choose BMC and benefit from:

- fast, convenient online submission
- thorough peer review by experienced researchers in your field
- rapid publication on acceptance
- support for research data, including large and complex data types
- gold Open Access which fosters wider collaboration and increased citations
- maximum visibility for your research: over 100M website views per year

At BMC, research is always in progress.

Learn more biomedcentral.com/submissions