# Algorithm design and implementation for the scale of sequencing data

Igor MARTAYAN

Supervised by Camille MARCHET

Committee: Sven RAHMANN, Alexandru TOMESCU, Giulio Ermanno PIBIRI, Éric RIVALS & Sylvain SALVATI

PhD defense, September 4th 2026, Lille

martayan.org/slides-phd.pdf

Université de Lille

CRIStAL
Centre de Recherche en Informatique, Signal et Automatique de Lille

Graduate Programmes Lille

**Introduction**

Quickly (pre)processing sequences

Representing $k$-mers with good locality

Using our toolbox for real-world applications

Conclusion

# What motivates this work? Applications in biology

## Assembling complete genomes

HOME > SCIENCE > VOL. 376, NO. 6588 > THE COMPLETE SEQUENCE OF A HUMAN GENOME

SPECIAL ISSUE RESEARCH ARTICLE | HUMAN GENOMICS

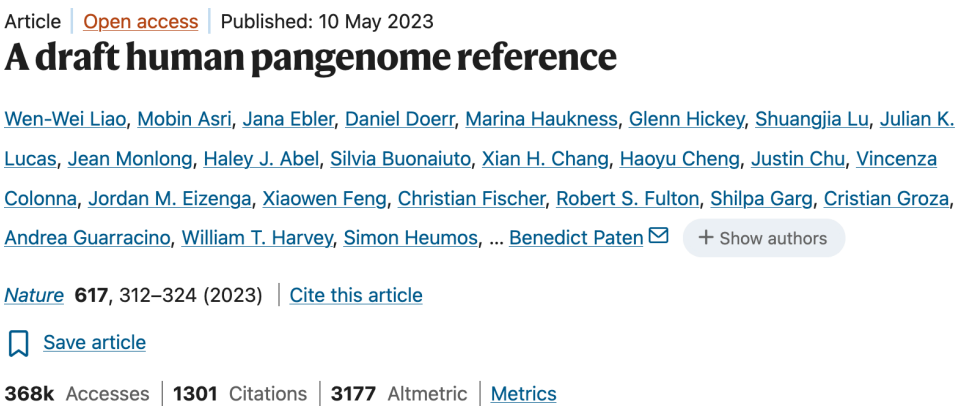
**The complete sequence of a human genome**

SERGEY NURK, SERGEY KOREN, ARANG RHIE, MIKKO RAUTIAINEN, ANDREY V. BZIKADZE, ALLA MIKHEENKO, MITCHELL R. VOLLGER, NICOLAS ALTEMOSE, LEV URALSKY, [...], AND ADAM M. PHILLIPPY +90 authors Authors Info & Affiliations

*SCIENCE* · 31 Mar 2022 · Vol 376, Issue 6588 · pp. 44-53 · DOI: 10.1126/science.abj6987

703,489 2,244

## Building pangenome reference



Article | Open access | Published: 10 May 2023

**A draft human pangenome reference**

Wen-Wei Liao, Mobin Asri, Jana Ebler, Daniel Doerr, Marina Haukness, Glenn Hickey, Shuangjia Lu, Julian K. Lucas, Jean Monlong, Haley J. Abel, Silvia Buonaiuto, Xian H. Chang, Haoyu Cheng, Justin Chu, Vincenza Colonna, Jordan M. Eizenga, Xiaowen Feng, Christian Fischer, Robert S. Fulton, Shilpa Garg, Cristian Groza, Andrea Guarracino, William T. Harvey, Simon Heumos, ... Benedict Paten + Show authors

*Nature* **617**, 312–324 (2023) | Cite this article

Save article

**368k** Accesses | **1301** Citations | **3177** Altmetric | Metrics

- genome: complete set of DNA of an organism
- pangenome: collection of genomes from many individuals

# What motivates this work? Applications in medicine

## Outbreak surveillance

Letter | Published: 03 February 2016

### Real-time, portable genome sequencing for Ebola surveillance

Joshua Quick, Nicholas J. Loman, Sophie Duraffour, Jared T. Simpson, Ettore Severi, Lauren Cowley, Joseph Akoi Bore, Raymond Koundouno, Gytis Dudas, Amy Mikhail, Nobila Ouédraogo, Babak Afrough, Amadou Bah, Jonathan H. J. Baum, Beate Becker-Ziaja, Jan Peter Boettcher, Mar Cabeza-Cabrerizo, Álvaro Camino-Sánchez, Lisa L. Carter, Juliane Doerrbecker, Theresa Enkirch, Isabel García- Dorival, Nicole Hetzelt, Julia Hinzmann, ... Miles W. Carroll + Show authors

*Nature* **530**, 228–232 (2016) | Cite this article

Save article



## Vaccine design

ORIGINAL ARTICLE

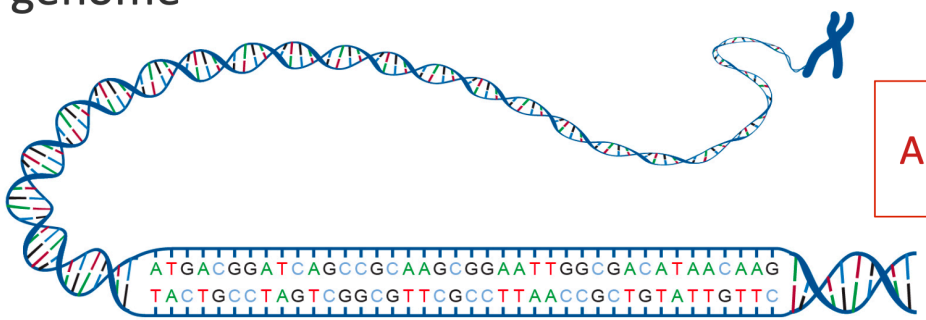
### An mRNA Vaccine against SARS-CoV-2 — Preliminary Report

**Authors**: Lisa A. Jackson, M.D., M.P.H. , Evan J. Anderson, M.D., Nadine G. Rouphael, M.D., Paul C. Roberts, Ph.D., Mamodikoe Makhene, M.D., M.P.H., Rhea N. Coler, Ph.D., Michele P. McCullough, M.P.H., +26 , for the mRNA-1273 Study Group* Author Info & Affiliations

**Published July 14, 2020 | N Engl J Med 2020;383:1920-1931 | DOI: 10.1056/NEJMoa2022483 | VOL. 383 NO. 20**

# How do we read these genomic sequences?

- The genome




A long text over {A,C, G, T}

- Accessing the sequence through DNA sequencing

AACTGACGACG
AGCAGGCAGCAGCA
CAGCAGCATCAGCATAACGACTC
ACGCAGGCAGCATCAGCAGACGACTACGACT

*Credits: Claire Lemaitre*

# Main challenges

DNA sequencers output noisy data:

- sequences are split into incomplete fragments (“reads”)
- they can be altered (sequencing errors, chimera, coverage gap)

**Huge amount of data** doubling every 2–3 years:

- A single human genome contains $3.2 \times 10^9$ base pairs (bp).
- RNA Sequencing (RNA-seq) typically produces ~10 Gbp of reads.
- Whole Genome Sequencing (WGS) produces ~100 Gbp of reads.
- The Sequence Read Archive (SRA) approaches $10^{17}$ = 100 Pbp.
  (→ 100 *million* gigabytes)

# A common approach: tokenizing DNA into *k*-mers

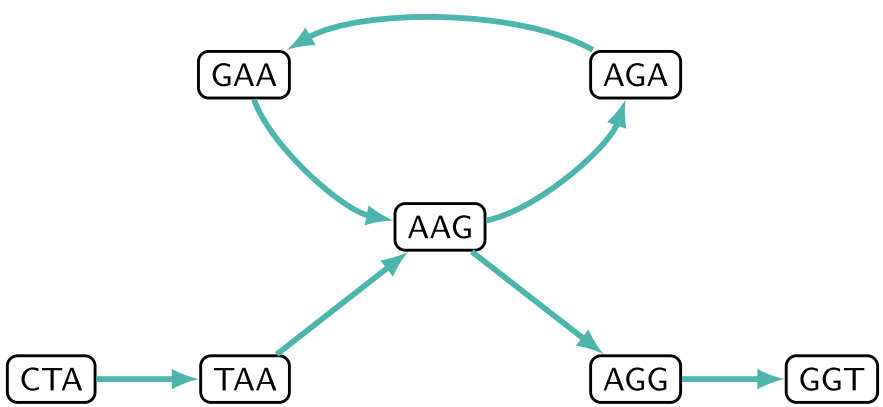
*k*-mer: substring of fixed size *k*

- simple intermediate representation
- collapses repetitions
- we lose positions & number of occurrences
- we usually prefer $k \geq 19$ to have sparse sets

```
CTAAGAAGGT
CTA
 TAA
  AAG
   AGA
    GAA
     AAG
      AGG
       GGT
```

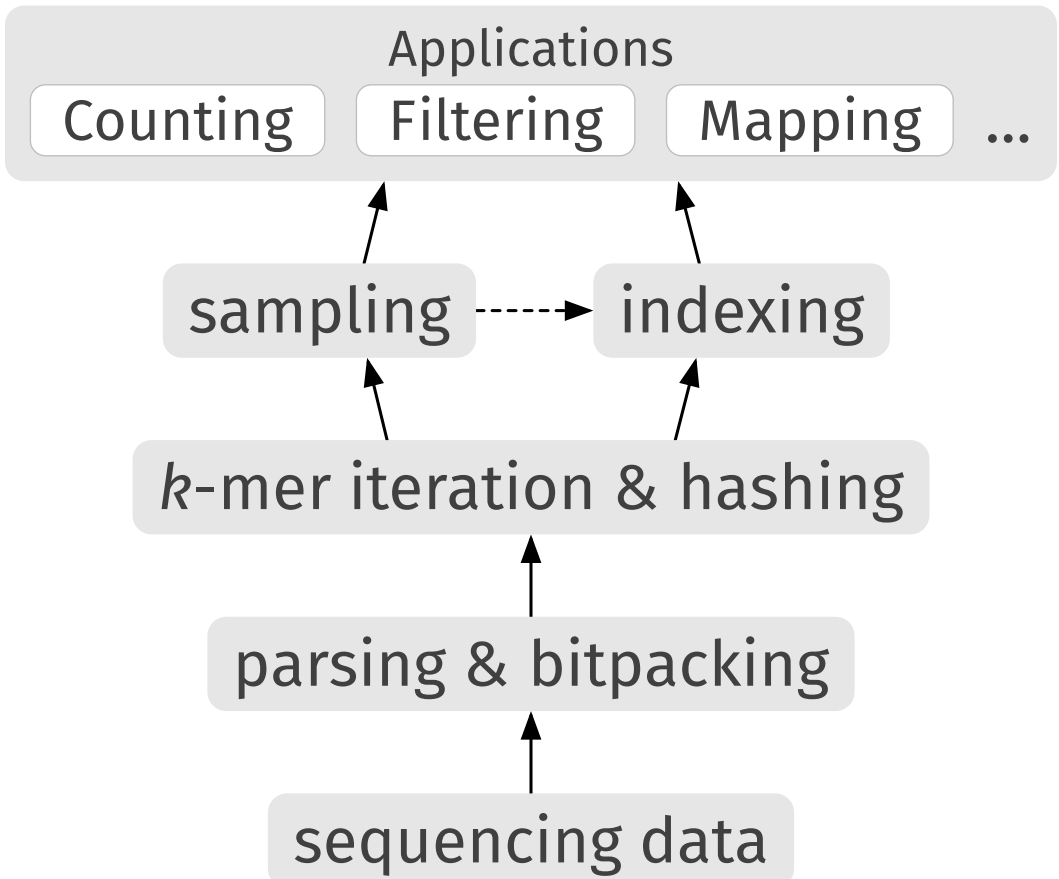
De Bruijn graph of order *k*:

- nodes = *k*-mers of the seq
- edges = overlaps of *k* – 1 bp




Example of node-centric DBG of order 3.

# Overview of the next parts



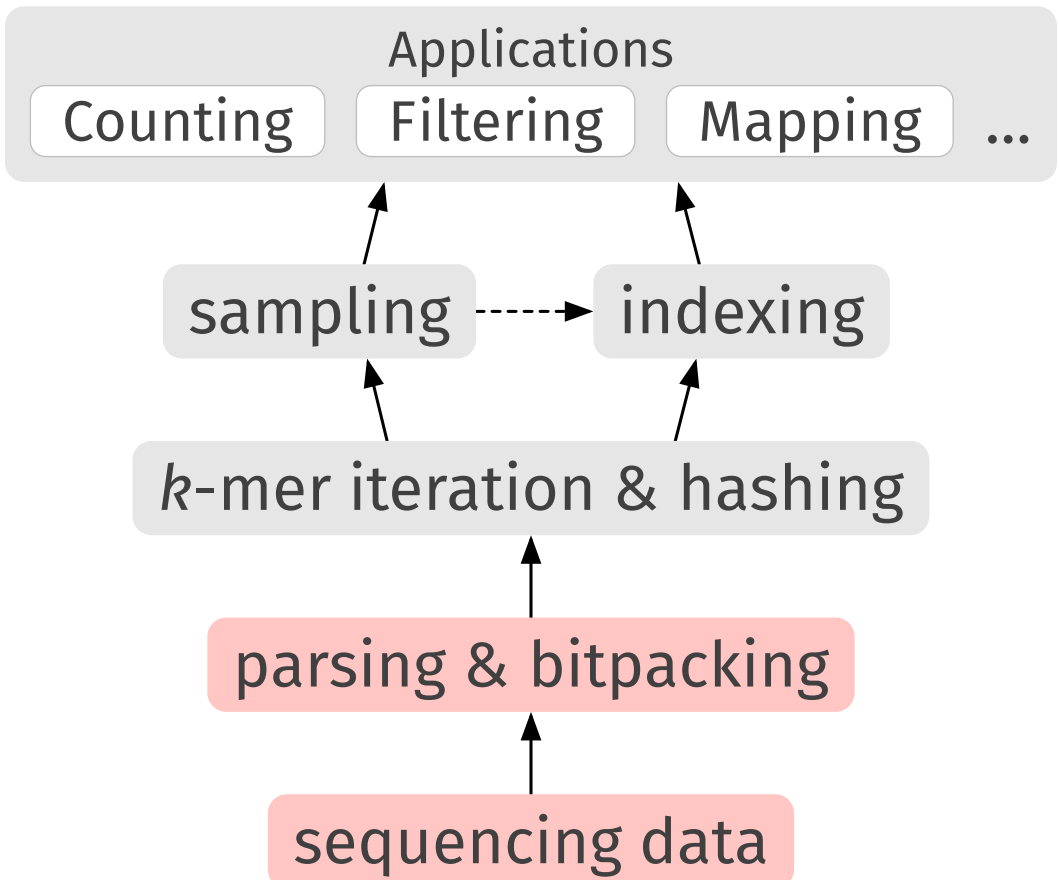
Introduction

**Quickly (pre)processing sequences**

Representing *k*-mers with good locality

Using our toolbox for real-world applications

Conclusion


Applications
Counting
Filtering
Mapping
...
sampling
indexing
k-mer iteration & hashing
parsing & bitpacking
sequencing data

# What does sequencing data look like in practice?

Historical formats: FASTA & FASTQ, designed in the 1980s to be *human-readable*

fasta

```
1  >header1
2  GATTAATCAC
3  >a longer header <owo>
4  TTAGCTGTGC
5  GANNNTAGAT
6  CTA
7  >header3 ...
```

fastq

```
1  @header1
2  GATTAATCAC
3  +
4  aneerol~!!
5  @header2
6  TTAGCTGTGCGANNNTAGATCTA
7  +
8  @This+is+a+valid+score!
```

# What does sequencing data look like in practice?

Historical formats: FASTA & FASTQ, designed in the 1980s to be *human-readable*

fasta

```
>header1
GATTAATCAC
>a longer header <owo>
TTAGCTGTGC
GANNNTAGAT
CTA
>header3 ...
```

fastq

```
@header1
GATTAATCAC
+
aneerol~!!
@header2
TTAGCTGTGCGANNNTAGATCTA
+
@This+is+a+valid+score!
```

# What does sequencing data look like in practice?

Historical formats: FASTA & FASTQ, designed in the 1980s to be *human-readable*

fasta

```
1  >header1
2  GATTAATCAC
3  >a longer header <owo>
4  TTAGCTGTGC
5  GANNNTAGAT
6  CTA
7  >header3 ...
```

non-ACTG bases (IUPAC)

fastq

```
1  @header1
2  GATTAATCAC
3  +
4  aneerol~!!
5  @header2
6  TTAGCTGTGCGANNNTAGATCTA
7  +
8  @This+is+a+valid+score!
```

# Turning FASTA/Q files into bitpacked representations

| A | C | T | G |
|---|---|---|---|
| 01000001 | 01000011 | 01010100 | 01000111 |

Packed and columnar representations:

| sequence | G | A | T | T | A | C | A |
|---|---|---|---|---|---|---|---|
| packed | 11 | 00 | 10 | 10 | 00 | 01 | 00 |
| columnar high | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| columnar low | 1 | 0 | 0 | 0 | 0 | 1 | 0 |

Goal: parse the file and bitpack bases *as efficiently as possible* (this is the first step of *many* analysis pipelines run everyday)

# Accelerating parsing with SIMD vectorization

Core design principle: **use the same instructions for every bytes**

→ the CPU can execute multiple of them at the same time

Main constraints: no conditions, bytes processed independently

github.com/imartayan/helicase

# Accelerating parsing with SIMD vectorization

Core design principle: **use the same instructions for every bytes**

→ the CPU can execute multiple of them at the same time

Main constraints: no conditions, bytes processed independently

| input | > | c | h | r | 1 | ↵ | C | G | G | A | C | ↵ | A | C | G | T | ↵ | > | u | w | u | ↵ | C |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ↵ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| > | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| !↵ | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |

github.com/imartayan/helicase

# Accelerating parsing with SIMD vectorization

Core design principle: **use the same instructions for every bytes**

→ the CPU can execute multiple of them at the same time

Main constraints: no conditions, bytes processed independently

| input | > | c | h | r | 1 | ↵ | C | G | G | A | C | ↵ | A | C | G | T | ↵ | > | u | w | u | ↵ | C |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ↵ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| > | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| !↵ | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| > + !↵ | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

github.com/imartayan/helicase

# Accelerating parsing with SIMD vectorization

Core design principle: **use the same instructions for every bytes**

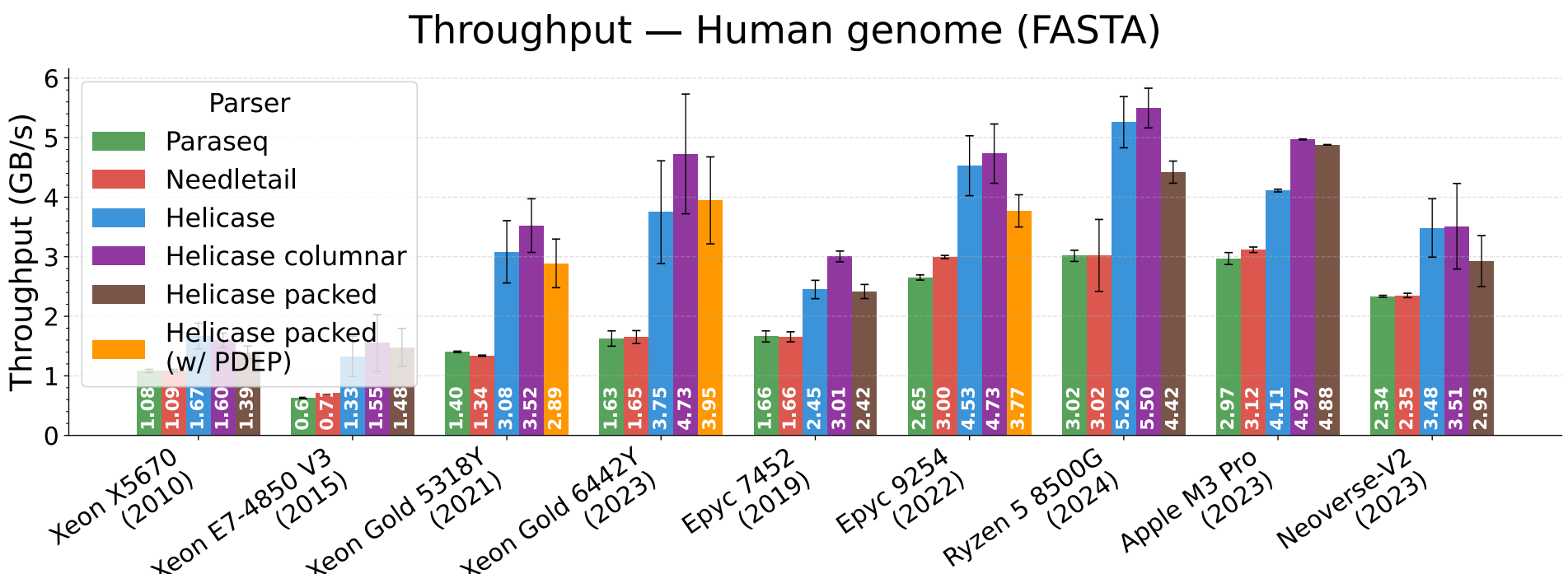
→ the CPU can execute multiple of them at the same time

Main constraints: no conditions, bytes processed independently

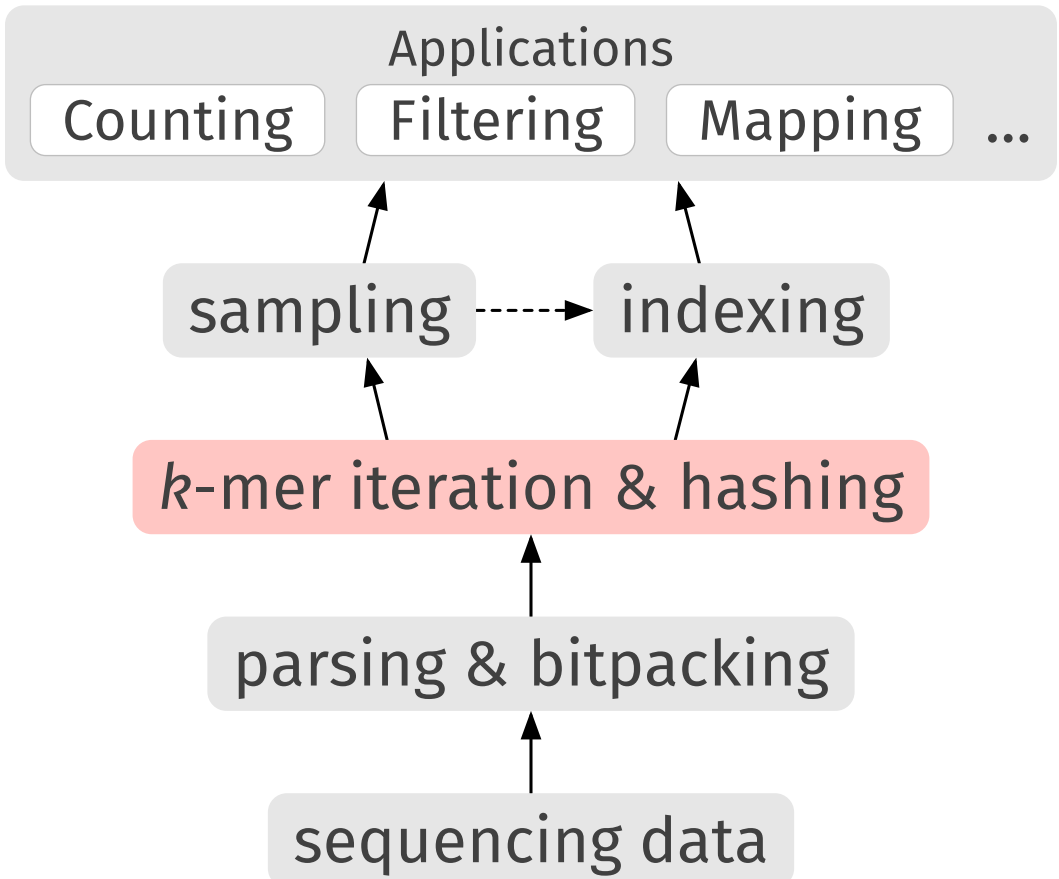
| input | > | c | h | r | 1 | ↵ | C | G | G | A | C | ↵ | A | C | G | T | ↵ | > | u | w | u | ↵ | C |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ↵ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| > | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| !↵ | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| > + !↵ | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| ⊕ !↵ | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |

github.com/imartayan/helicase

# How fast can we parse multiline FASTA from disk?




→ Helicase is 1.5–2× faster than existing parsers


Applications
Counting
Filtering
Mapping
...
sampling
indexing
k-mer iteration & hashing
parsing & bitpacking
sequencing data

# Quickly computing rolling hashes of *k*-mers (*ntHash*)

| … | G | G | C | A | T | C | … |
|---|---|---|---|---|---|---|---|

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| hash of GGCAT | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| rotated left by 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| xor | | | | | | | | |
| h(C) | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| xor | | | | | | | | |
| h(G) rotated *k* times | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| hash of GCATC | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |

Benefits:

- each new *k*-mer is hashed in constant-time (3 ops)
- no need to keep the whole *k*-mer in memory
- **we can vectorize it**

Vectorized implementation: github.com/rust-seq/seq-hash

# How fast can we hash all *k*-mers of a sequence?

Throughput using a single thread:

| Implementation | on x86 (Xeon 6430) | on ARM (Apple M1) |
|---|---|---|
| ASCII + rapidhash | 0.47 Gbp/s | 0.65 Gbp/s |
| packed + fxhash | 0.71 Gbp/s | 1.25 Gbp/s |
| original nthash | 1.03 Gbp/s | 1.00 Gbp/s |
| **seq-hash** | **2.65 Gbp/s** | **2.28 Gbp/s** |

→ seq-hash is 2.5–5× faster than existing hash functions

(this can be further accelerated using multiple threads)

# Some perspectives

Our hash functions are fast but have little formal guarantees:

- ntHash can have collisions and bias for consecutive $k$-mers
- can we design a rolling hash that's both fast & well-behaved? (ongoing work w/ P. Kazemi and R. Groot Koerkamp)

SIMD code brings significant gains but is difficult to maintain:

- portable abstractions are not as performant yet
- new languages aim to make it easier
- could this code be suitable for GPUs?

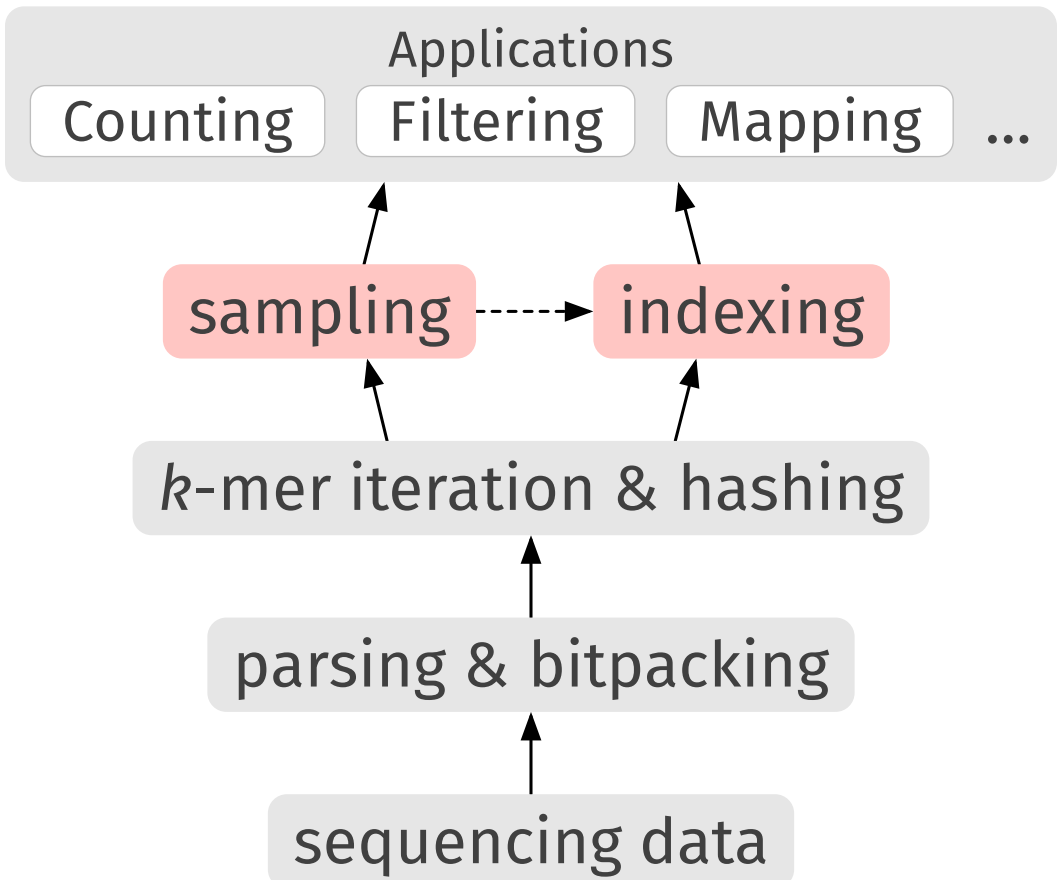
Introduction

Quickly (pre)processing sequences

**Representing *k*-mers with good locality**

Using our toolbox for real-world applications

Conclusion


Applications
Counting
Filtering
Mapping
...
sampling
indexing
*k*-mer iteration & hashing
parsing & bitpacking
sequencing data

# What do we mean by *good locality*?

We want to organize *k*-mers so that “similar ones” are grouped (this helps keeping related data in cache to avoid memory load)

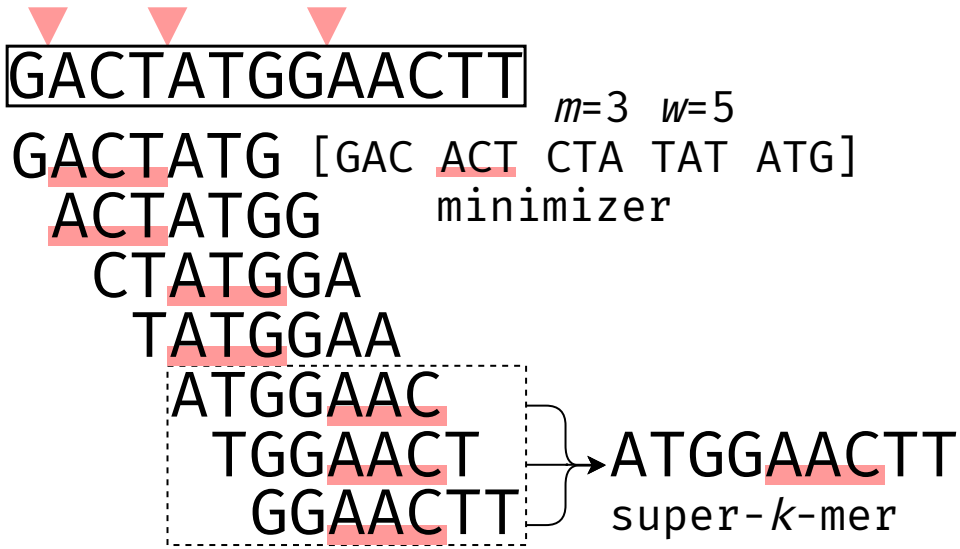
- *k*-mers that are consecutive
  ⇒ lexicographic order is not suitable ✗
- *k*-mers that differ by one substitution
  ⇒ position in reference is not suitable ✗

```
 TTACCATA
ATTACCAT
ATTACCGT
```

→ *What kind of transformation can we apply?*

# (Random) Minimizers

Given a **window of *w* consecutive *m*-mers**, select the **"smallest" *m*-mer**.
Random minimizer: select the *m*-mer with the **smallest hash**.




- consecutive *k*-mers often share the same minimizer
- they can be grouped into "super-*k*-mers"

# A branchless sliding minimum algorithm

**"Two stacks" algorithm:**

Split into chunks of size $w$,

Given 2 chunks $\boxed{x_1, \ldots, x_w}\ \boxed{y_1, \ldots, y_w}$

- suffix stack: $S[i] = \min(x_i, \ldots, x_w)$
- prefix stack: $P[i] = \min(y_1, \ldots, y_{i-1})$

$i$-th window min: $\min(S[i], P[i])$

→ **linear and branchless!**

Example for $w = 5$

| 2 | 1 | 9 | 5 | 6 | 7 | 4 | 9 | 3 | 8 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 1 | 5 | 5 | 6 | | | | | | |

github.com/rust-seq/simd-minimizers

# A branchless sliding minimum algorithm

**"Two stacks" algorithm:**

Split into chunks of size $w$,

Given 2 chunks $\boxed{x_1, \ldots, x_w}$ $\boxed{y_1, \ldots, y_w}$

- suffix stack: $S[i] = \min(x_i, \ldots, x_w)$
- prefix stack: $P[i] = \min(y_1, \ldots, y_{i-1})$

$i$-th window min: $\min(S[i], P[i])$

→ **linear and branchless!**

Example for $w = 5$

| 2 | 1 | 9 | 5 | 6 | 7 | 4 | 9 | 3 | 8 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 1 | 5 | 5 | 6 | | | | | | |
| | **1** | 5 | 5 | 6 | 7 | | | | | |

github.com/rust-seq/simd-minimizers

# A branchless sliding minimum algorithm

**"Two stacks" algorithm:**

Split into chunks of size $w$,

Given 2 chunks $\boxed{x_1, \ldots, x_w}$ $\boxed{y_1, \ldots, y_w}$

- suffix stack: $S[i] = \min(x_i, \ldots, x_w)$
- prefix stack: $P[i] = \min(y_1, \ldots, y_{i-1})$

$i$-th window min: $\min(S[i], P[i])$

→ **linear and branchless!**

Example for $w = 5$

| 2 | 1 | 9 | 5 | 6 | 7 | 4 | 9 | 3 | 8 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 1 | 5 | 5 | 6 | | | | | | |
| | **1** | 5 | 5 | 6 | 7 | | | | | |
| | | 5 | 5 | 6 | 7 | **4** | | | | |

github.com/rust-seq/simd-minimizers

# A branchless sliding minimum algorithm

**"Two stacks" algorithm:**

Split into chunks of size $w$,

Given 2 chunks $\boxed{x_1, \ldots, x_w}$ $\boxed{y_1, \ldots, y_w}$

- suffix stack: $S[i] = \min(x_i, \ldots, x_w)$
- prefix stack: $P[i] = \min(y_1, \ldots, y_{i-1})$

$i$-th window min: $\min(S[i], P[i])$

→ **linear and branchless!**

Example for $w = 5$

| 2 | 1 | 9 | 5 | 6 | 7 | 4 | 9 | 3 | 8 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 1 | 5 | 5 | 6 | | | | | | |
| | **1** | 5 | 5 | 6 | 7 | | | | | |
| | | 5 | 5 | 6 | 7 | **4** | | | | |
| | | | 5 | 6 | 7 | 4 | **4** | | | |

github.com/rust-seq/simd-minimizers

# A branchless sliding minimum algorithm

**"Two stacks" algorithm:**

Split into chunks of size $w$,

Given 2 chunks $\boxed{x_1, \ldots, x_w}$ $\boxed{y_1, \ldots, y_w}$

- suffix stack: $S[i] = \min(x_i, \ldots, x_w)$
- prefix stack: $P[i] = \min(y_1, \ldots, y_{i-1})$

$i$-th window min: $\min(S[i], P[i])$

→ **linear and branchless!**

Example for $w = 5$

| 2 | 1 | 9 | 5 | 6 | 7 | 4 | 9 | 3 | 8 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 1 | 5 | 5 | 6 | | | | | | |
| | **1** | 5 | 5 | 6 | 7 | | | | | |
| | | 5 | 5 | 6 | 7 | **4** | | | | |
| | | | 5 | 6 | 7 | 4 | **4** | | | |
| | | | | 6 | 7 | 4 | 4 | **3** | | |

github.com/rust-seq/simd-minimizers

# A branchless sliding minimum algorithm

**"Two stacks" algorithm:**

Split into chunks of size $w$,

Given 2 chunks $\boxed{x_1, \ldots, x_w}$ $\boxed{y_1, \ldots, y_w}$

- suffix stack: $S[i] = \min(x_i, \ldots, x_w)$
- prefix stack: $P[i] = \min(y_1, \ldots, y_{i-1})$

$i$-th window min: $\min(S[i], P[i])$

→ **linear and branchless!**

Example for $w = 5$

| 2 | 1 | 9 | 5 | 6 | 7 | 4 | 9 | 3 | 8 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 1 | 5 | 5 | 6 | | | | | | |
| | **1** | 5 | 5 | 6 | 7 | | | | | |
| | | 5 | 5 | 6 | 7 | **4** | | | | |
| | | | 5 | 6 | 7 | 4 | **4** | | | |
| | | | | 6 | 7 | 4 | 4 | **3** | | |
| | | | | | **3** | 3 | 3 | 3 | 8 | |

github.com/rust-seq/simd-minimizers

# A branchless sliding minimum algorithm

**"Two stacks" algorithm:**

Split into chunks of size $w$,

Given 2 chunks $\boxed{x_1, \ldots, x_w}$ $\boxed{y_1, \ldots, y_w}$

- suffix stack: $S[i] = \min(x_i, \ldots, x_w)$
- prefix stack: $P[i] = \min(y_1, \ldots, y_{i-1})$

$i$-th window min: $\min(S[i], P[i])$

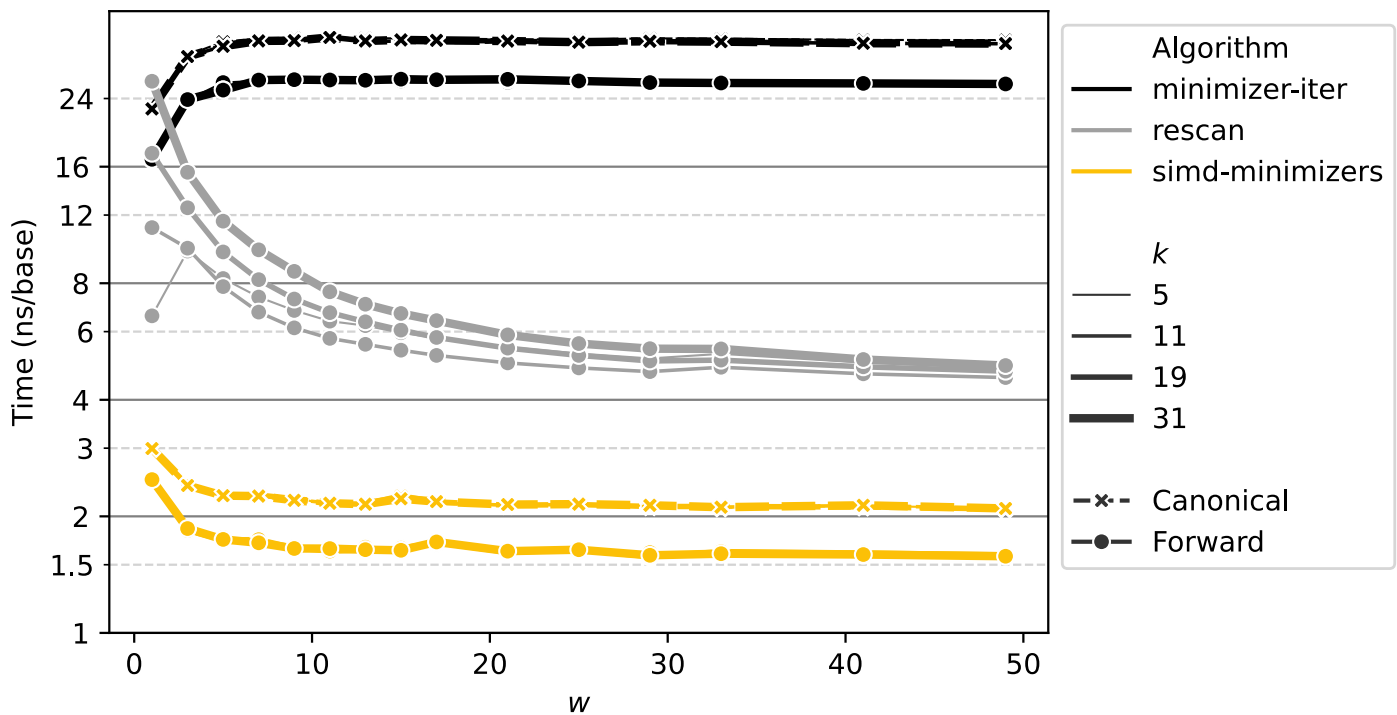
→ **linear and branchless!**

Example for $w = 5$

| 2 | 1 | 9 | 5 | 6 | 7 | 4 | 9 | 3 | 8 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 1 | 5 | 5 | 6 | | | | | | |
| | **1** | 5 | 5 | 6 | 7 | | | | | |
| | | 5 | 5 | 6 | 7 | **4** | | | | |
| | | | 5 | 6 | 7 | 4 | **4** | | | |
| | | | | 6 | 7 | 4 | 4 | **3** | | |
| | | | | | **3** | 3 | 3 | 3 | 8 | |
| | | | | | | 3 | 3 | 3 | 8 | **2** |

github.com/rust-seq/simd-minimizers

# How fast can we compute minimizer positions?

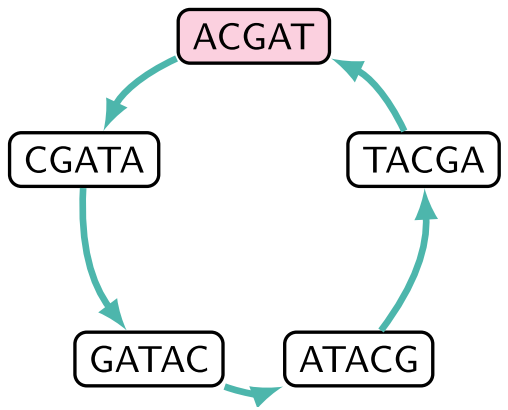



→ simd-minimizers is 3–7× faster than existing algorithms

# Necklaces as an alternative to minimizers

Necklace: smallest cyclic rotation of a word (using lexicographic order)

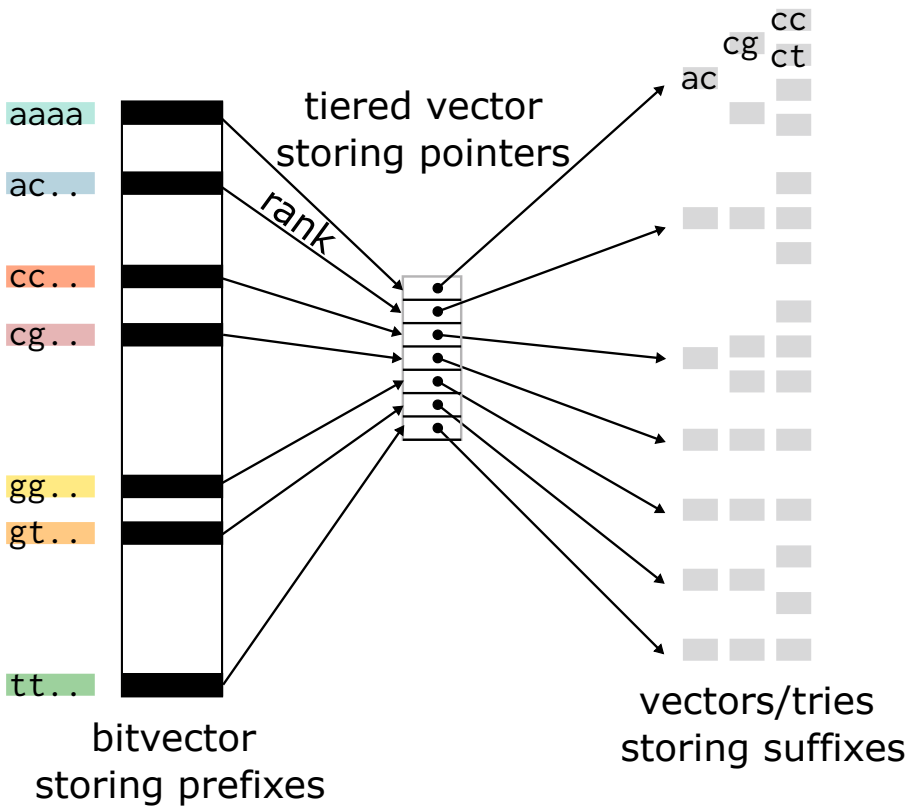
e.g. the necklace of GATAC is AC<u>GAT</u>




Necklaces of consecutive *k*-mers typically share long prefixes, and can be grouped into *super-necklaces*.

e.g. for the 5-mers in GATACTC:
(AC<u>GAT</u>, ACT<u>AT</u>, ACTC<u>T</u>) can be compacted as (ACTC, <u>GAT</u>)

# CBL: indexing *k*-mers by quotienting necklaces




Query: given a *k*-mer $x$,

1. compute its necklace $\langle x \rangle$
2. split $\langle x \rangle$ as $q \| r$
3. query $r$ in the bucket of $q$

→ faster for consecutive *k*-mers (likely in the same bucket $q$)

# More perspectives

Beyond static indexes and *k*-mer-level queries:

- **incremental updates** are valuable as we get more sequences
- **set-level operations** can be faster and enable richer queries (worth discussing with the database community)

Different “atomic” building blocks for our indexes:

- **indexing the DNA content *between* some *k*-mers** rather than the *k*-mers themselves could significantly reduce space usage (ongoing work w/ R. Patro)

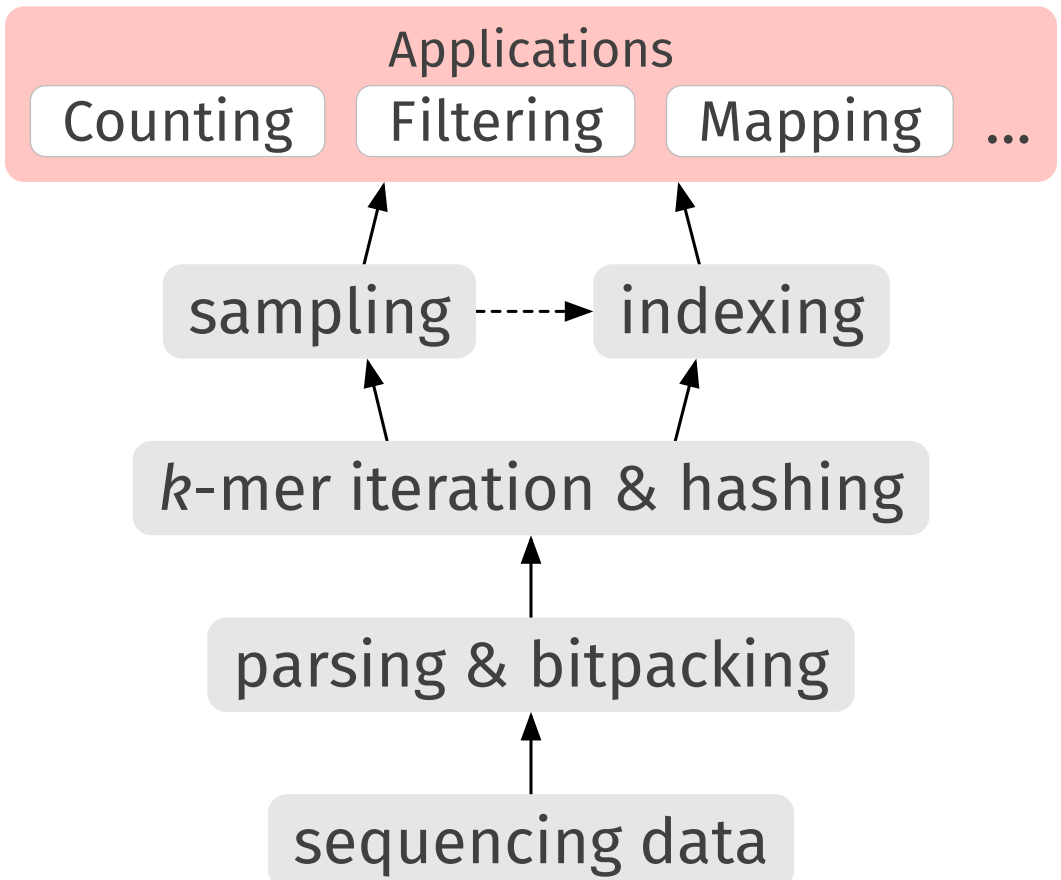
Introduction

Quickly (pre)processing sequences

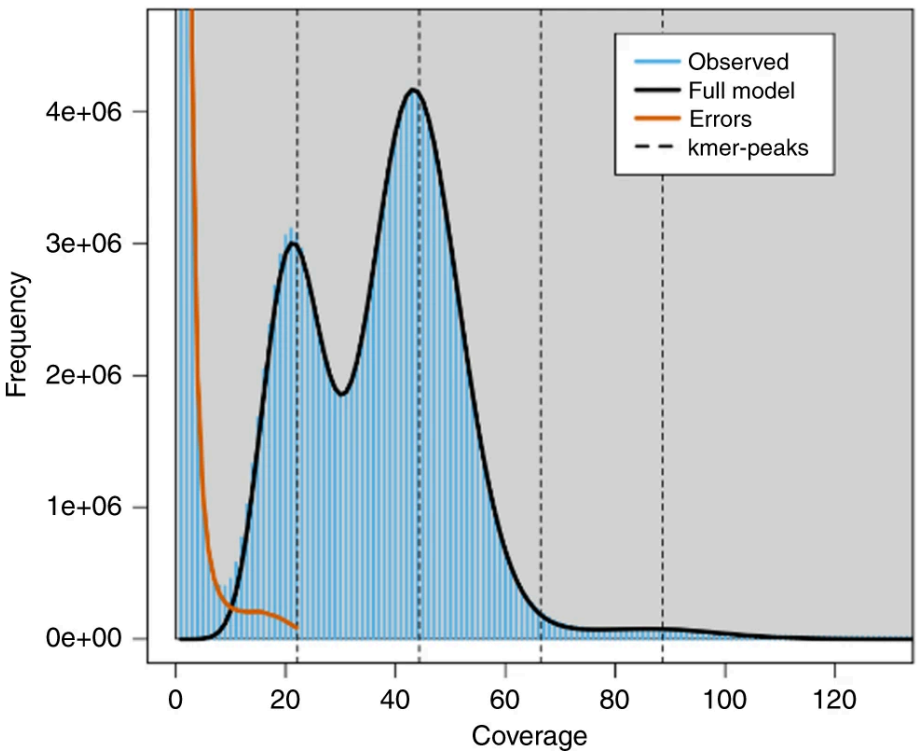
Representing $k$-mers with good locality

**Using our toolbox for real-world applications**

Conclusion


Applications
Counting
Filtering
Mapping
...
sampling
indexing
k-mer iteration & hashing
parsing & bitpacking
sequencing data

# Counting *k*-mers to recover the spectrum



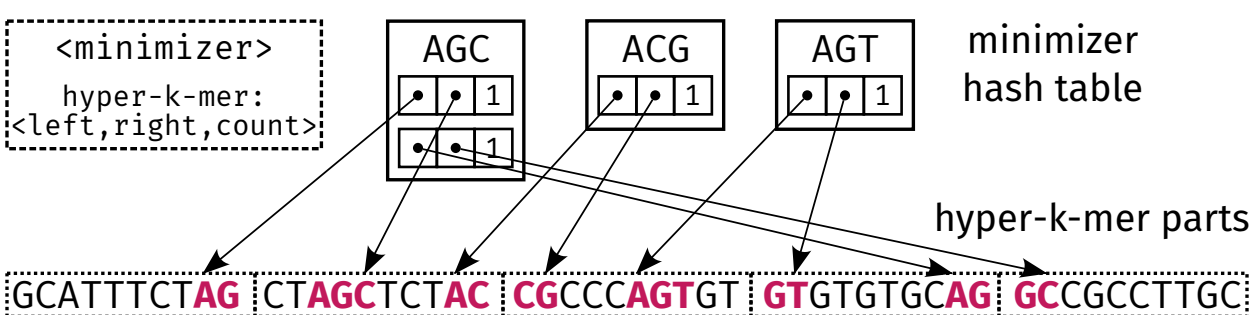

*Credits: GenomeScope 2.0*

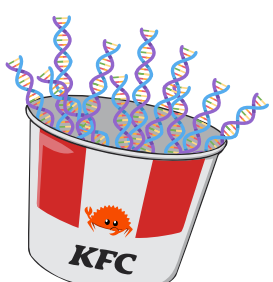
*k*-mer counts are useful to:

- discard erroneous bases
- estimate genome size
- identify variants and repeats

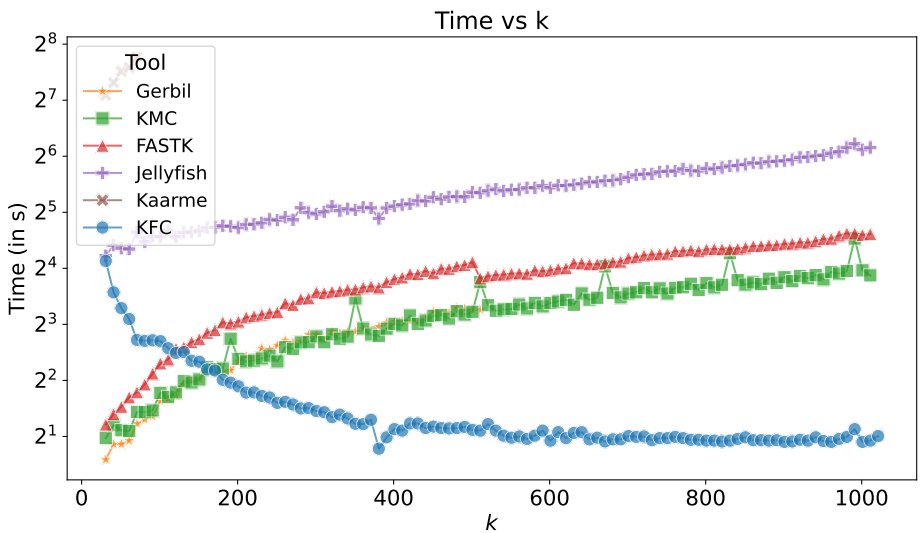
*Can we afford using larger k-mers as reads get longer & more accurate?*

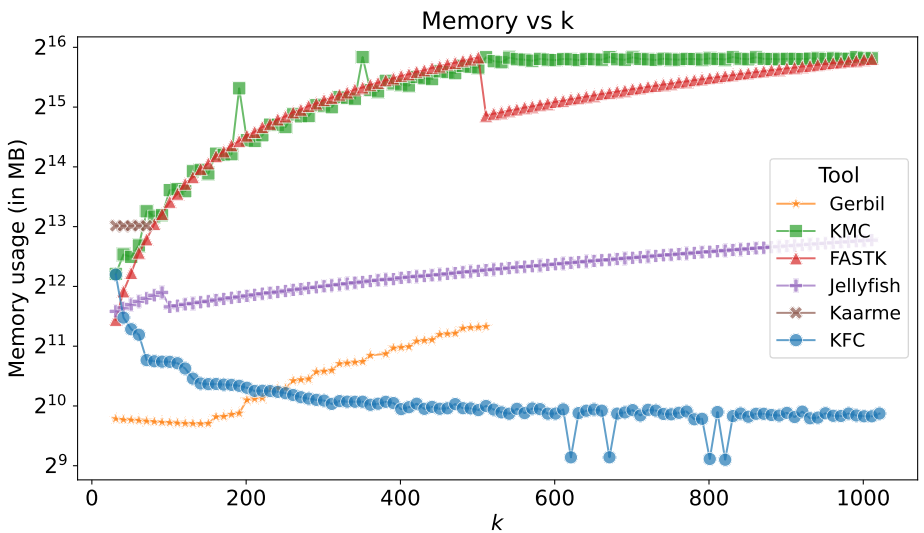
# Counting *k*-mers with hyper-*k*-mers

# Hyper-*k*-mers are especially efficient for large *k*-mers




Experiments on ONT Simplex E. coli dataset (SRR28370668)

→ KFC is competitive for $k \geq 63$ and is the best for $k \geq 200$

github.com/lrobidou/KFC

# Filtering reads with minimizers

We can quickly estimate whether a read belongs to a known organism by simply looking at its minimizers. Useful to:

- *decontaminate* samples (e.g. removing human artifacts)
- identify samples of unknow origin (e.g. new bacterial variants)

Article | Open access | Published: 14 June 2022

**The human "contaminome": bacterial, viral, and computational contamination in whole genome sequences from 1000 families**

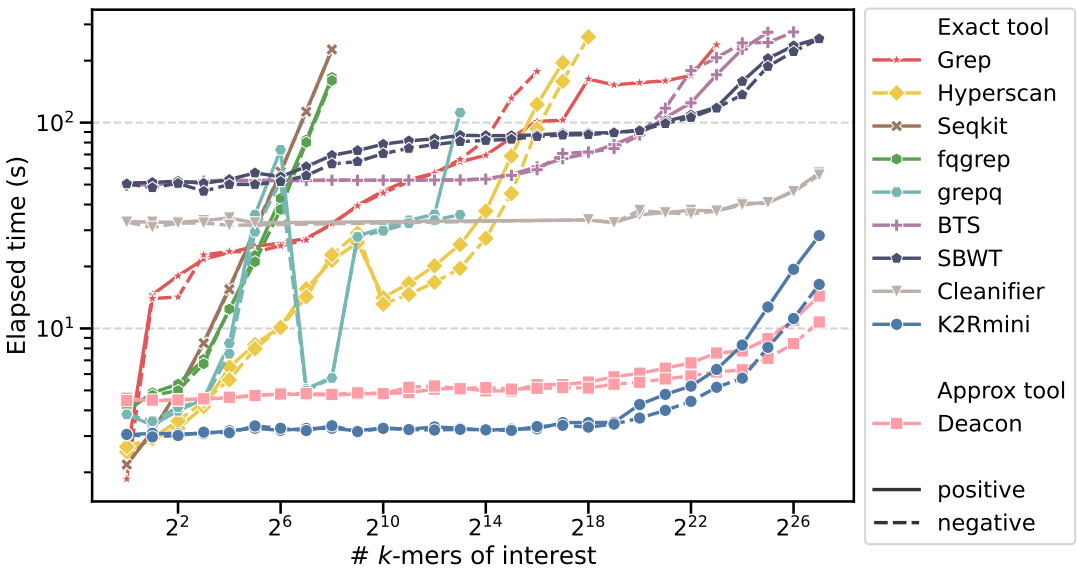
Brianna Chrisman, Chloe He, Jae-Yoon Jung, Nate Stockham, Kelley Paskov, Peter Washington & Dennis P. Wall

*Scientific Reports* **12**, Article number: 9863 (2022) | Cite this article

Save article

**17k** Accesses | **67** Citations | **56** Altmetric | Metrics

# Comparison of *k*-mer filtering tools




Main idea of K2Rmini:
count minimizer matches to upperbound *k*-mer matches, then apply an early filter

→ K2Rmini has the lowest time & memory footprint among exact tools

github.com/Malfoy/K2Rmini

Introduction

Quickly (pre)processing sequences

Representing $k$-mers with good locality

Using our toolbox for real-world applications

**Conclusion**

# Closing thoughts: on low-level optimizations

- Careful implementation is just as important as good algorithm design when it comes practical performances.
- Not using vectorization & parallelism is a missed opportunity (recent hardware improvements focused on these).
- Tooling for developpers is progressively making it easier.

# Closing thoughts: towards “sketch-space” algorithms

- While we have good and scalable centralized indexes, more computation could be done directly on sketches locally.
- So far, sketching was mostly use as a light preprocessing step. Data structures built directly on sketches can be much lighter.
- MinHash is not always the best. Recent work w/ Jules Le Prince suggests that “lexicographic-informed” sketching can be more space-efficient & tolerate more errors.

TLDR: we can afford to discard information in many use cases.

# Closing thoughts: working w/ both DNA & proteins

We focused on optimizing DNA analysis throughout this thesis, but proteins also provide a lot of insights and hybrid approaches are worth exploring.

Brief Communication | Published: 20 May 2024

**Metabuli: sensitive and specific metagenomic classification via joint analysis of amino acid and DNA**

Jaebeom Kim & Martin Steinegger

*Nature Methods* **21**, 971–973 (2024) | Cite this article

Save article

**8796** Accesses | **47** Citations | **64** Altmetric | Metrics

JOURNAL ARTICLE

**Exploring gene content with pangene graphs**

Heng Li, Maximillian Marin, Maha R Farhat

*Bioinformatics*, Volume 40, Issue 7, July 2024, btae456, https://doi.org/10.1093/bioinformatics/btae456

**Published:** 23 July 2024 **Article history**

# Thank you!