

---

**Supplementary information**

---

**A Python-based programming language for  
high-performance computational genomics**

---

In the format provided by the  
authors and unedited

# Supplementary Notes for **A Python-based programming language for high-performance computational genomics**

Ariya Shajii<sup>1</sup>, Ibrahim Numanagic<sup>1,3</sup>, Alexander T. Leighton<sup>1</sup>, Haley Greenyer<sup>3</sup>, Saman Amarasinghe<sup>1</sup>,  
and Bonnie Berger<sup>\*1,2</sup>

<sup>1</sup>Computer Science and AI Lab, Massachusetts Institute of Technology, Cambridge, MA, USA

<sup>2</sup>Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA, USA

<sup>3</sup>Department of Computer Science, University of Victoria, Victoria, BC V8P 5C2, Canada

## **Note 1: A short, practical guide to Seq**

Here we will demonstrate the capabilities of Seq by implementing a short and fast read alignment pipeline, named SeqMap, with only 35 lines of high-level Python code. Further details, together with the links to the test datasets, can be found <https://docs.seq-lang.org/tutorial/workshop.html>.

We will test SeqMap on the following data:

1. Chromosome 22 as the reference sequence (`chr22.fa`), and
2. A simulated set of 1 million 101bp reads (`reads.fastq`).

We assume that Seq has been properly installed and configured; for more details on how to do this, please visit <https://seq-lang.org>. We also assume that a reader is familiar with the basic bioinformatics formats (FASTA, FASTQ, etc.).

## **Reading sequences from disk**

The first step of processing any kind of sequencing data is to read it from disk. Seq has builtin support for many of the standard file formats in bioinformatics, such as FASTA, FASTQ, SAM, BAM and CRAM.

Let's write a program to read our FASTQ file and print each record's name and sequence on a single line:

```
from sys import argv
from bio import *
```

---

\*Correspondence: [bab@mit.edu](mailto:bab@mit.edu)

```
for record in FASTQ(argv[1]):
    print record.name, record.seq
```

Now we can run this Seq program as follows:

```
seqc run section1.seq data/reads.fq > out.txt
```

and peek into the out.txt:

```
chr22_16993648_16994131_1:0:0_2:0:0_0/1 CTACCAACACCTACTTCGTTTCTTAACATCACTTTAATTTTATCTTAGAGGAATCTTTTCCCTATCCCATTAAGTTATGGGAGATGGGGCCAGGCATGG
chr22_28253010_28253558_1:0:0_0:0:0_1/1 AGTGTTTTGGCTGTGGCTAGACTAAAAATAAGGAATGAGGGGGGTATCTTCCACTCTTGGCCCTCTCATCACCTATTCCCTATATCCAGAACTCAGAGTCC
chr22_21656192_21656802_0:0:0_2:0:0_2/1 ATAGCGTGGATTCCCTATGACATCAAGGAGCTATTTTATTTGGTAAAAACGAAAAAGCACAATAATGAACGAAACGCAAGCACTGAAACAGTGGAGACACCTAG
chr22_44541236_44541725_0:1:0_0:0:0_3/1 CTCTCTGTCTCTCTCTCTCCCTAGGTGAGGGTGGTCCCTGGGGAGGCCCCCTGGGTACCCCAAGACAGGTGGGAGGTGCTTCTACCCGACCCCTCTTCTCT
chr22_39607671_39608139_0:0:0_2:0:0_4/1 ATTGGCTCAGAGTTACGACAGGCTGTACCGCATGGCGCCAGTGTCTGCTCTGGTGAGGCCTTACGGACGTTACAATAACGGCGGAAGGCAAGGGCGGAGC
chr22_35577703_35578255_3:0:0_1:0:0_5/1 TGCCATGGTGGTTAGCTGCACCCATCAACCTGTCTATCATATTAGTTATTTTCTAATGCTATCCCTCCCTAGCACCCCTACCCCTCTGATAGGCCCTGGT
chr22_46059124_46059578_1:0:0_1:0:0_6/1 AATCAGTACCAACAATATATGGATATTATTGGCATTCTTGTCTCCCTCTGCTGAACTGGGAATTCCTCTATTAGTTTGGACATTATCTGGTATTGAACC
chr22_31651867_31652385_2:0:0_2:0:0_7/1 ATCTAGTGACAGTAAGTGGCTGATAAAGTGAGCTGCCATTACATAGTCATCATCTTTAATAGAAGTTAACACATACTGAGTTTCTACTATATTGGGTCTTT
chr22_24816466_24817026_1:0:0_1:0:0_8/1 CACCTCTAGGGCTCAAGGGGCGAGTTCTCCATTCTCTCAGCAGTGGCGCTGTGGAACCTGTCTCTGAGGCCAGGGGGTGGTCAGGCAGGGCTGGAGTGGC
chr22_27496272_27496752_1:0:0_1:0:0_9/1 CTTAGCCCCATTAAACACTGGCAGGGCTGAATTGTCTGTGCCATCCATCACCTTCTCCCTAGCCTGGTTTCTTACCTACCTGGAAGCCGCTCCCTTTT
...
```

That was it! FASTA, BAM and other file formats can be read in a very similar way. Note that since this data is simulated, each read's name (e.g. chr22\_16993648\_...) indicates the locus at which the read was obtained.

## Building an index

Our goal is to find a “mapping” to the genome for each read in a FASTQ file. Comparing to every position on the reference sequence would take far too long. An alternative is to create an index of the  $k$ -mers from the reference sequence and use it to guide the mapping process [46].

Let's build a dictionary that links each 32-mer to its position (*locus*) on the reference sequence:

```
K = Kmer[32] # make a type alias to improve the readability
index = {}
for record in FASTA(argv[1]):
    for pos, kmer in record.seq.kmers_with_pos[K] (step=1):
        index[kmer] = pos
```

Of course, there will be  $k$ -mers that appear multiple times, but let's ignore this detail for now and just store the latest position we see for each  $k$ -mer.

Another important issue is reverse complementation: some of our reads will map in the reverse direction rather than in the forward direction. For this reason, we will build our index in such a way that a  $k$ -mer is considered “equal” to its reverse complement. One easy way to do this is by using *canonical k-mers*, i.e. the minimum of a  $k$ -mer and its reverse complement. For example, the canonical form of GCT is its reverse complement, AGC, since the reverse complement is lexicographically smaller (i.e. comes first alphabetically). On the other hand, the canonical form of CGC is itself, since it is lexicographically smaller than its reverse complement, GCG. Here is the new code:

```
index = {}
for record in FASTA(argv[1]):
    for pos, kmer in record.seq.kmers_with_pos[K] (step=1):
        index[min(kmer, ~kmer)] = pos
```

Note that we will have to use canonical  $k$ -mers when querying the index as well.

Now we have our index as a dictionary (`index`), but we do not want to build it each time we perform read mapping, since it only depends on a (fixed) reference sequence. So, as a last step, let's dump the index to a file using the `pickle` module:

```
import pickle, gzip
with gzip.open(argv[1] + '.index', 'wb') as jar:
    pickle.dump(index, jar)
```

Let us run the program from the command line:

```
seqc run section2.seq data/chr22.fa
```

We should now be able to see a new file `data/chr22.fa.index` which stores our serialized index.

Note that thus far, Seq both feels and behaves like a typical Python program.

## Finding $k$ -mer matches

At this point, we have an index we can load from disk. We will use it to find candidate mappings for our reads. Let us first split each read into  $k$ -mers, and report a mapping if at least two  $k$ -mers “support” a particular locus.

The first step is to load the index:

```
index = None
with gzip.open(argv[1] + '.index', 'rb') as jar:
    # pickle needs to know the original structure type
    index = pickle.load[Dict[K,int]](jar)
```

Now we can iterate over our reads and query  $k$ -mers in the index. We need a way to keep track of candidate mapping positions as we process the  $k$ -mers of a read: we can do this using a new dictionary `candidates` that maps candidate alignment positions to the number of  $k$ -mers supporting the given position [46]. Then, we just iterate over `candidates` and output positions supported by 2 or more  $k$ -mers. Finally, we clear `candidates` before processing the next read:

```
candidates = {} # map position to a count
for record in FASTQ(argv[2]):
    for pos, kmer in record.read.kmers_with_pos[K](step=1):
        found = index.get(min(kmer, ~kmer), -1)
        if found > 0:
            candidates.increment(found - pos)
    for pos, count in candidates.items():
        if count > 1:
            print record.name, pos + 1
    candidates.clear()
```

Let us run the program and peek at the output:

```
seqc run section3.seq data/chr22.fa data/reads.fq | head
chr22_16993648_16994131_1:0:0_2:0:0_0/1 16993648
chr22_28253010_28253558_1:0:0_0:0:0_1/1 28253010
chr22_44541236_44541725_0:1:0_0:0:0_3/1 44541236
chr22_31651867_31652385_2:0:0_2:0:0_7/1 31651867
chr22_21584577_21585142_1:0:0_1:0:0_a/1 21584577
chr22_46629499_46629977_0:0:0_2:0:0_b/1 47088563
chr22_46629499_46629977_0:0:0_2:0:0_b/1 51103174
chr22_46629499_46629977_0:0:0_2:0:0_b/1 46795988
chr22_16269615_16270134_0:0:0_1:0:0_c/1 50577316
chr22_16269615_16270134_0:0:0_1:0:0_c/1 16269615
```

Notice that most positions we reported match the position from the simulated read name (the first integer after the \_); not bad for such a short program!

## Smith-Waterman alignment and CIGAR string generation

We now have the ability to report approximate mapping positions for each read. However, we usually need more precise alignments, which include information about mismatches, insertions and deletions. Luckily, Seq makes sequence alignment easy: to align sequence  $q$  against sequence  $t$ , you can just do:

```
aln = q @ t
```

`aln` is a tuple of alignment score and CIGAR string (a CIGAR string is a way of encoding an alignment result, and consists of operations such as M for match/mismatch, I for insertion and D for deletion, accompanied by the number of associated bases; for example, 3M2I4M indicates 3 (mis)matches followed by a length-2 insertion followed by 4 (mis)matches).

By default, Levenshtein distance is used, meaning mismatch and gap costs are both 1, while match costs are zero. More control over alignment parameters can be achieved using the `align` method:

```
aln = q.align(t, a=2, b=4, ambig=0, gapo=4, gape=2)
```

where  $a$  is the match score,  $b$  is the mismatch cost,  $ambig$  is the ambiguous base (N) match score,  $gapo$  is the gap open cost, and  $gape$  the gap extension cost (i.e. a gap of length  $k$  costs  $gapo + (k \cdot gape)$ ). There are many more parameters as well, controlling factors like alignment bandwidth, Z-drop, global/extension alignment and more; please check the standard library reference for further details.

To align a read to the reference, we will use a simple `query.align(target)`:

```
candidates = {}
for record in FASTQ(argv[2]):
    for pos, kmer in record.read.kmers_with_pos[K](step=1):
        found = index.get(min(kmer, ~kmer), -1)
        if found > 0:
            candidates.increment(found - pos)
    for pos, count in candidates.items():
```

```

    if count > 1:
        # get query, target and align:
        query = record.read
        target = reference[pos:pos + len(query)]
        alignment = query.align(target)
        print record.name, pos + 1, alignment.score, alignment.cigar
    candidates.clear()

```

Now, run the program and observe the output:

```

seqc run section4.seq data/chr22.fa data/reads.fq | head
chr22_16993648_16994131_1:0:0_2:0:0_0/1 16993648 196 101M
chr22_28253010_28253558_1:0:0_0:0:0_1/1 28253010 196 101M
chr22_44541236_44541725_0:1:0_0:0:0_3/1 44541236 196 101M
chr22_31651867_31652385_2:0:0_2:0:0_7/1 31651867 190 101M
chr22_21584577_21585142_1:0:0_1:0:0_a/1 21584577 196 101M
chr22_46629499_46629977_0:0:0_2:0:0_b/1 47088563 110 20M1I4M1D76M
chr22_46629499_46629977_0:0:0_2:0:0_b/1 51103174 134 20M1I4M1D76M
chr22_46629499_46629977_0:0:0_2:0:0_b/1 46795988 128 20M1I4M1D76M
chr22_16269615_16270134_0:0:0_1:0:0_c/1 50577316 118 101M
chr22_16269615_16270134_0:0:0_1:0:0_c/1 16269615 202 101M

```

Most of the alignments contain only matches or mismatches (M), which is to be expected as insertions and deletions are far less common. In fact, the three mappings containing indels appear to be incorrect!

A more thorough mapping scheme would also look at alignment scores before reporting mappings, although for the purposes of this tutorial we will ignore such improvements.

## Pipelines

Pipelines are a very convenient Seq construct for expressing a variety of algorithms and applications. In fact, SeqMap can be thought of as a pipeline with the following stages:

1. read a record from the FASTQ file,
2. find candidate alignments by querying the index, and
3. perform alignment for mappings supported by 2+ k-mers and output results.

We can write these stages as separate functions in Seq as follows:

```

def find_candidates(record):
    candidates = {}
    for pos, kmer in record.read.kmers_with_pos[K] (step=1):
        found = index.get(min(kmer, ~kmer), -1)
        if found > 0:
            candidates.increment(found - pos)
    for pos, count in candidates.items():
        if count > 1:

```

```

        yield record, pos
def align_and_output(t):
    record, pos = t
    query = record.read
    target = reference[pos:pos + len(query)]
    alignment = query.align(target)
    print record.name, pos + 1, alignment.score, alignment.cigar

```

and then chain them into a pipeline:

```
FASTQ(argv[2]) |> iter |> find_candidates |> align_and_output
```

Notice that `find_candidates` yields candidate alignments to `align_and_output`, which then performs alignment and prints the results. In Seq, all values generated from one stage of a pipeline are lazily passed to the next stage. Lazy passing allows the Seq pipeline to behave like Bash streams: no new item will be generated until the current item is processed. The Seq compiler performs many domain-specific optimizations on pipelines, one of which we focus on next.

## Parallelism

Pipelines can be easily parallelized with the parallel pipe operator, `||>`, which tells the compiler that all subsequent stages can be executed in parallel:

```
FASTQ(argv[2]) |> iter ||> find_candidates |> align_and_output
```

Since the full program also involves loading the index, let us time the main pipeline using the timing module to see if parallelism helps or not:

```

import timing
with timing('mapping'):
    FASTQ(argv[2]) |> iter ||> find_candidates |> align_and_output

```

Let us run this for different numbers of threads (e.g., 1 and 2):

```

export OMP_NUM_THREADS=1 # this sets the number of threads Seq can use
seqc run section5.seq data/chr22.fa data/reads.fq > out.txt
# mapping took 48.2858s

```

```

export OMP_NUM_THREADS=2
seqc run section5.seq data/chr22.fa data/reads.fq > out.txt
# mapping took 35.886s

```

Often, batching reads into larger blocks and processing those blocks in parallel can yield better performance, especially if each read is quick to process. This is also very easy to do in Seq:

```

def process_block(block):
    block |> iter |> find_candidates |> align_and_output
with timing('mapping'):
    FASTQ(argv[2]) |> blocks(size=2000) ||> process_block

```

The effect of batching is clearly visible now:

```

export OMP_NUM_THREADS=1
seqc run section5.seq data/chr22.fa data/reads.fq > out.txt
# mapping took 48.2858s

export OMP_NUM_THREADS=2
seqc run section5.seq data/chr22.fa data/reads.fq > out.txt
# mapping took 25.2648s

```

## Domain-specific optimizations

Seq already performs numerous domain-specific optimizations under the hood. However, we can give the compiler a hint in this case to perform one more: inter-sequence alignment (see Supplementary Note 3 for more details). This optimization entails batching sequences prior to alignment, then aligning multiple pairs using a very fast vectorized (SIMD-optimized) alignment kernel.

In Seq, we just need one additional function annotation to tell the compiler to perform this optimization:

```

@inter_align
def align_and_output(t):
    ...

```

Let us run the program with and without this optimization:

```

# without @inter_align
seqc run section5.seq data/chr22.fa data/reads.fq > out.txt
# mapping took 43.4457s

# with @inter_align
seqc run section6.seq data/chr22.fa data/reads.fq > out.txt
# mapping took 32.3241s

```

The timings with inter-sequence alignment will depend on the SIMD instruction sets your CPU supports; we ran our experiments on an AVX2-compatible machine.

## Final code listing

Here is the final 35-line long code listing for our toy mapper. Note that achieving the same result in general-purpose languages—particularly with inter-sequence alignment—would require several hundred or potentially thousands of lines of code.

```

1  # Usage: seqc run section6.seq <FASTA path> <FASTQ path>
2  from sys import argv
3  from time import timing
4  from bio import *
5  import pickle
6  import gzip
7

```

```

8 K = Kmer[32]
9 index = None
10 reference = s''
11 for record in FASTA(argv[1]): # Take the first sequence
12     reference = record.seq
13 with gzip.open(argv[1] + '.index', 'rb') as jar: # Load the index
14     index = pickle.load(Dict[K, int])(jar)
15
16 def find_candidates(record):
17     candidates = {}
18     for pos, kmer in record.read.kmers_with_pos[K](step=1):
19         found = index.get(min(kmer, ~kmer), -1)
20         if found > 0:
21             candidates.increment(found - pos)
22     for pos, count in candidates.items():
23         if count > 1:
24             yield record, pos
25
26 @inter_align
27 def align_and_output(t):
28     record, pos = t
29     query = record.read
30     target = reference[pos:pos + len(query)]
31     alignment = query.align(target)
32     print record.name, pos + 1, alignment.score, alignment.cigar
33
34 with timing('mapping'):
35     FASTQ(argv[2]) |> iter |> find_candidates |> align_and_output

```

## Using non-Seq libraries

There are cases where you might want to access some Python (or C) library that is not yet ported to Seq. For example, you might want to use Biopython [12] to parse a rare or non-standard file format. To allow seamless integration with existing ecosystems, Seq supports running pure Python code, as well as using Python libraries and objects directly within Seq code.

For example, suppose you need to analyze the output of a NCBI BLAST run, and extract all BLAST hits within a given e-value threshold. You can use the following code to achieve this, and then use the data from BLAST directly in Seq:

```

from python import Bio.Blast.NCBIXML

# Everything within a @python function will be executed by the Python runtime
@python
def process_blast(file, threshold):
    from Bio.Blast import NCBIXML
    result_handle = open(file)

```

```

for record in NCBIXML.parse(result_handle):
    for alignment in record.alignments:
        for hsp in alignment.hsps:
            if hsp.expect < threshold:
                yield (alignment, hsp)

hits = process_blast('blast.xml', 0.04)
for aln, hsp in hits:
    # Use Python objects directly ...
    print aln.accession, 'of length', aln.length
    # ... or cast them to Seq objects for improved performance:
    print str(hsp.query)[:100]
    print str(hsp.sbjct)[:100]
    e_value = hsp.expect.get[float]()
    print e_value

```

For more details, please consult the official Seq documentation at <https://docs.seq-lang.org>.

## Note 2: Overview of Results

A typical genomics analysis pipeline currently consists of four fundamental types of stages: (i) data pre- and post-processing, (ii) reference sequence processing, (iii) read sequence processing, and (iv) downstream analysis. Each of these stages employs different classes of algorithms and has distinct performance challenges (e.g. processing of reads often involves memory-bound operations whereas downstream analyses are frequently numerically-intensive). Thus, unique optimization techniques must be used at each stage in order to achieve ideal scalability and performance.

To demonstrate the utility of Seq, we re-implemented eight standard, real-world applications to evaluate the performance of the compiler’s domain-specific optimizations and the resulting code complexity across these stages, as compared to existing highly-optimized and widely-used C, C++, Java or Python implementations—and, in some instances, their equivalents using Rust, Julia, and C++ genomics libraries. Here we describe each of these applications in greater detail, as well as what is gained by porting them to Seq. An overview of these applications in the context of the genomics pipeline is given in Figure S1, which also shows the speedups attained by Seq.

Before we begin, let us provide a few notes about the re-implementations described below. First, all the tools we compared against are well-tested and highly optimized (often by a large team of developers), and provide several additional options aside from the functionality being compared against in this work. Our goal here is to show that the same—or even *much* better—results can be achieved in a fraction of the time and development effort with a high-level programming language given the proper compiler support.

To ensure fairness, all of the reported results, unless otherwise specified, were generated in single-threaded mode, and unless explicitly specified, the outputs produced by our re-implementations exactly match those of the tools being compared against.

Finally, we would like to note that our memory measurement metric—the maximum resident memory used by a process—is far from perfect. This metric is heavily impacted by system-specific memory management; for example, macOS automatically compresses the working set of high-memory processes, and thus reports lower memory usages (a side-effect of this strategy is occasional slow-down of such processes). Furthermore, this metric struggles with garbage collection-based runtimes such as Seq’s, Java’s, Python’s and Julia’s, as it often reports memory usages higher than the actual value.

In the results below, runtimes are reported in an “hh:mm:ss” format (or “mm:ss” or simply “ss” for shorter runtimes), and memory usages are given in gigabytes. Experiments for which memory usages are not reported used less than 1 GB of RAM.

### (i) Reference sequence processing

**Genome homology table construction (CORA)** CORA [46] is an all-mapping tool for next-generation sequencing (NGS) reads. In other words, given a read (or a read pair), CORA reports all alignments (up to some edit or Hamming distance) of that read in a reference

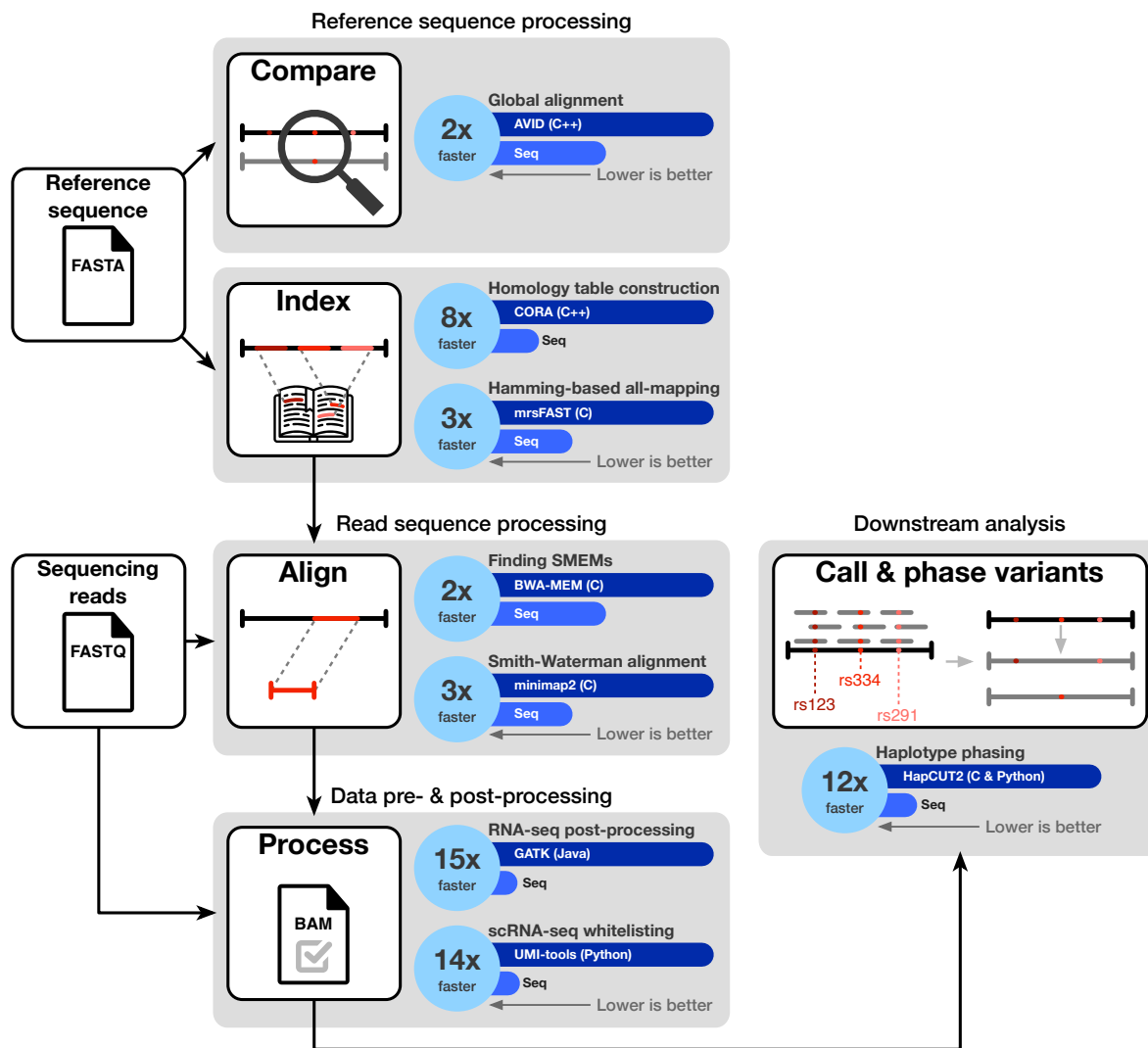


Figure S1: Seq performance improvements demonstrated on eight common, state-of-the-art applications. Comparisons based on a typical genomics pipeline using the Seq language and compiler to implement applications: global alignment (from AVID) under “Compare” for comparative genomics; homology table construction (from CORA) and Hamming distance-based all mapping (from mrsFAST, which also performs alignment) under “Index” for reference sequence indexing; finding super-maximal exact matches, or SMEMs (from BWA-MEM) and Smith-Waterman alignment (from minimap2) under “Align” for read sequence alignment; RNA-seq read post-processing (from GATK *SplitNCigar*) and single-cell RNA-seq barcode whitelisting (from UMI-tools) under “Process” for data pre- and post-processing; and haplotype phasing under “Call & phase variants” for numerically-intensive downstream analysis. The accompanying bar charts show runtimes of the original tools compared to the Seq versions. Detailed comparisons are available in the Supplementary Note 2.

### Genome homology table construction (CORA)

| <i>Linux</i>      | Time                      | Memory      | Seq imprv. | Time                        | Memory      | Seq imprv. |
|-------------------|---------------------------|-------------|------------|-----------------------------|-------------|------------|
| <b>CORA (C++)</b> | 1:32:09                   | 206.7       | 7.9×       | 11:42:44                    | 329.6       | 3.7×       |
| <b>Seq</b>        | <b>11:44</b>              | <b>72.5</b> | —          | <b>3:07:47</b>              | <b>50.8</b> | —          |
|                   | <i>(a) Exact matching</i> |             |            | <i>(b) Inexact matching</i> |             |            |

Table S1: Performance improvements in homology table construction from using the Seq language and compiler. Due to the memory requirements, we only ran this experiment on a large-memory Linux machine. The Seq version achieves nearly an 8× speed improvement while using 3–6× less memory than the original version.

sequence. To achieve this goal, the primary data structure used by CORA is the *homology table*, which stores groups of “homologous” regions in the reference, and is used to convert a single alignment as reported by an off-the-shelf mapper like BWA, into a list of all mappings satisfying the distance criteria. Two versions of the homology table are built: *exact* and *inexact*. The exact homology table stores in a compressed form pairs of equal  $k$ -mers (with  $33 \leq k \leq 64$ ) from the reference, along with their loci (where “compression” is done by merging two or more consecutive homologous pairs into a single group). Similarly, the inexact homology table stores pairs of unequal  $k$ -mers whose Hamming distance is less than some threshold. CORA’s definition of homology also allows for cases where one  $k$ -mer is homologous to the reverse complement of another  $k$ -mer.

The process of constructing the homology table involves many of the operations that Seq optimizes, including  $k$ -merization, reverse complementation,  $k$ -mer matching and Hamming distance calculations. Indeed, we implemented CORA’s homology table construction in Seq, and compared both performance and lines of code with the original, highly-optimized C++ implementation. We used a  $k$ -mer length of 64 and a maximum Hamming distance threshold of 1 for the inexact homology table. Exact homology table construction was single-threaded, whereas the inexact table’s construction was allowed to use all 48 threads of our machine.

Table S1 shows the results of the comparison. The Seq versions were substantially smaller in terms of lines of code (over 2,500 in the original C++ version versus less than 300 in Seq—nearly 10× fewer lines) and ran substantially faster (4–8×) while using 3–4× less memory, which has been somewhat of a barrier to CORA’s use (both CORA and the Seq re-implementation were run with 24 threads for inexact construction). Lastly, we found the outputs of the original and Seq implementations to be very slightly different; in particular, the original version found 4,186,875 exact homologies in hg19 with a  $k$ -mer length of 64, whereas the Seq implementation found 4,372,977 (roughly 4% more) for the same  $k$ -mer size; all of the additional homologies were found to be correct by a manual inspection. The additional homologies have now been made publicly available at <https://github.com/seq-lang/seq-benchmarks>. Therefore, we attribute this discrepancy to a bug in the original version. Note that while such bugs are hard to find in large and tightly optimized C/C++ codebases, they often become self-evident in high-level codebases that focus only on the core algorithm, such as those produced with Seq.

### Global sequence alignment (AVID)

| <i>Linux</i>        | Time                      | Memory     | Seq imprv. | Time                      | Memory     | Seq imprv. |
|---------------------|---------------------------|------------|------------|---------------------------|------------|------------|
| <b>AVID</b> (C/C++) | 35:05                     | <b>1.0</b> | 2.0×       | 34:24                     | <b>2.0</b> | 2.0×       |
| <b>Seq</b>          | <b>17:44</b>              | 2.2        | —          | <b>16:52</b>              | 5.3        | —          |
|                     | <i>(a) 90% similarity</i> |            |            | <i>(b) 75% similarity</i> |            |            |

Table S2: Performance improvements in global sequence alignment from using the Seq language and compiler. As we only had access to the Linux binary distribution of AVID, we had to restrict this benchmark to the Linux machine. The Seq version achieves nearly a 2× speed improvement. Note that the results are not identical, as the original source code is not available. However, we found out that  $\geq 90\%$  of all generated alignment scores are nearly identical (within 10% difference) between the tools.

**Global sequence alignment (AVID)** AVID [9] is one of the fastest tools for global alignment of large nucleotide and amino acid sequences (see also Batzoglu et al. [5]). Although the canonical global alignment algorithm (Needleman-Wunsch) is a slight variant of the local alignment algorithm (Smith-Waterman), only a few tools such as FASTA [34] and AVID (released in 1989 and 2002, respectively) are able to quickly produce global alignments between two sequences. Unfortunately, the source code of AVID is not available, and pre-built binaries are only available for a limited number of platforms.

We re-implemented AVID’s algorithm as described in the original publication [9]. The core of AVID’s method consists of finding “Maximal Exact Matches” (MEMs)—identical sub-sequences between the two sequences of interest that cannot be further extended—and using them to guiding the global alignment process. MEMs are commonly found by recursively traversing a generalized suffix tree made from the subject sequences. In our implementation, we used suffix arrays (SA) instead of suffix trees (as Seq supports efficient SA operations out of the box), and a bottom-up traversal algorithm over SAs to find all MEMs [1].

Overall, we were able to prototype the whole pipeline as described in the paper in 170 lines of high-level Seq code. We ran AVID and Seq-AVID on the large set of similar (similarity  $\geq 90\%$ ) and less-similar sequences ( $\geq 70\%$ ) from the human genome segmental duplications database [4, 32], and found that the Seq implementation provided a 2× speedup over the original binary while maintaining a similar accuracy (note that our re-implementation is not an exact reproduction of the original software as the exact algorithm used by the currently available AVID binary differs slightly from the published description; nevertheless, our implementation produces alignments that are highly concordant with the alignments produced by the AVID binary). The results of this comparison are shown in Table S2.

## (ii) Read sequence processing

**Finding SMEMs (BWA-MEM)** Super-Maximal Exact Matches, or SMEMs, are a subset of MEMs such that no two SMEMs overlap within the read sequence. SMEMs are an integral component of many important genomics algorithms, ranging from sequence alignment to as-

sembly [25, 26, 44]. Because the SMEM detection algorithm primarily involves querying an FMD-index data structure (unlike the MEM detection algorithm, which relies on suffix tree traversal), it is inherently memory-bound, and can be greatly accelerated by several of Seq’s compiler optimizations.

To this end, we implemented BWA-MEM’s SMEM-finding algorithm in Seq, with roughly 135 lines of code (compared to the over 500 lines of C code comprising the original). The standard algorithm involves repeatedly querying an FMD-index to extend matches between a read and the reference, which causes expensive memory stalls—indeed, this is the main bottleneck in the algorithm. As querying large genomic indices is itself a prevalent operation in genomics algorithms, the Seq compiler automatically performs several optimizations to greatly speed up such queries. Rather than issuing memory requests serially and forcing the entire program to stall, Seq makes use of coroutines and software prefetching to *overlap* the memory-bound stall from one query with other useful work, greatly reducing the effective stall time. This process is shown in Figure S5 (left half).

While implementing these transformations by hand in existing codebases would require substantial effort (including implementing coroutines, scheduling and managing coroutine state, dispatching queries and returning results, debugging, etc.), in Seq it only requires a single additional `@prefetch` annotation—everything else is handled automatically by the compiler. Applying this optimization to the SMEM algorithm resulted in a roughly  $2\times$  speedup over BWA-MEM (Table S3).

Software prefetching is also used in BWA-MEM2, the next iteration of BWA-MEM, which adds numerous manual, low-level performance optimizations [44]. However, BWA-MEM2’s implementation does not perform a coroutine-based software context switch like Seq compiler does; consequently, as the authors describe, it cannot fully remedy memory-bound stalls. By contrast, Seq’s use of coroutines to overlap stalls with useful work can often ensure that a particular address is cache-resident by the time it needs to be accessed (i.e. when a suspended coroutine is resumed, as described in detail in Methods). We also evaluated software prefetching *without* the use of coroutines—much like what BWA-MEM2 does—but observed no appreciable performance improvements over the coroutine-based prefetching.

In order to compare the performance of Seq with other high-performance genomics libraries, we also implemented this experiment in Rust using the Rust-Bio library [23] and in C++20 using the SeqAn3 library [14]. In both cases, the Seq version takes roughly  $6\text{--}9\times$  less time to process the reads than the alternate implementations, regardless of whether Seq’s prefetch optimization had been enabled. Finally, we would like to point out that the Seq version is also cleaner and more adaptable: Rust-Bio had to be manually patched<sup>1</sup> to support exact SMEM extraction, while the SeqAn version had lengthier code, was more verbose, and took  $6\times$  longer to compile.

**Hamming-distance based all-mapping (mrsFAST)** mrsFAST [18] is a perfect-sensitivity all-mapping tool based on Hamming distance. mrsFAST employs a hash table to index  $k$ -mers from a reference genome, and uses that table to find all mappings of a given read that

---

<sup>1</sup><https://github.com/rust-bio/rust-bio/pull/392>

### Finding SMEMs (BWA-MEM)

| <i>Linux</i>           | Time      | Memory      | Seq imprv. |
|------------------------|-----------|-------------|------------|
| <b>BWA</b> (C)         | 1:37      | <b>0.4*</b> | 1.7×       |
| <b>Rust-Bio</b> (Rust) | 6:23      | 81.2        | 6.7×       |
| <b>SeqAn</b> (C++)     | 5:16      | 2.5         | 5.5×       |
| <b>Seq</b>             | 1:01      | 7.2         | 1.1×       |
| <b>Seq</b> (prefetch)  | <b>57</b> | 7.2         | —          |

| <i>macOS</i>           | Time      | Memory      | Seq imprv. |
|------------------------|-----------|-------------|------------|
| <b>BWA</b> (C)         | 1:19      | <b>0.4*</b> | 2.3×       |
| <b>Rust-Bio</b> (Rust) | 4:17      | 61.1        | 7.6×       |
| <b>SeqAn</b> (C++)     | 4:53      | 2.6         | 8.6×       |
| <b>Seq</b>             | 50        | 7.3         | 1.5×       |
| <b>Seq</b> (prefetch)  | <b>34</b> | 7.3         | —          |

Table S3: Performance improvements in SMEM finding from using the Seq language and compiler on Linux and macOS systems. The Seq version achieves nearly a  $2\times$  speed improvement over BWA, and roughly a  $6\text{--}9\times$  improvement over high-performance genomics libraries Rust-Bio and SeqAn. The reported timings represent the time needed for each tool to find SMEMs in a set of FASTQ reads; index building and loading times are not included, as they vary across the tools. Note that Seq, Rust and SeqAn FM-index implementations are not compressed, and as such use more RAM than BWA’s compressed implementation (marked with \*). Also note that prefetch improvements depend on the CPU cache size: large-cache machines (such as our Linux machine) benefit less than machines with smaller CPU caches (macOS).

are below a user-defined Hamming distance threshold  $e$ . The algorithm is centered around the pigeonhole principle: each read is partitioned into  $e + 1$  non-overlapping seeds, which guarantees that at least one seed will map to all positions in the reference at which the Hamming distance is at most  $e$ . mrsFAST-Ultra, the latest version of the tool, is implemented in C, and involves many of the operations discussed here, including  $k$ -merization, reverse complementation,  $k$ -mer hashing, indexing and of course Hamming distance calculations.

We implemented a subset of mrsFAST-Ultra for single-end mapping in Seq to further test the effectiveness of the compiler’s optimizations, the results of which are shown in Table S4. With a distance threshold  $e = 2$  on chromosome 1, the Seq implementation was  $\approx 30\%$  faster than the original hand-optimized C implementation, with a third of the code size. We also implemented a version using an FM-index instead of a hash table—a modification that requires changing only a few lines of code in Seq—and tested it with exact matching ( $e = 0$ ) on the entire genome, resulting in a  $2.5\times$  improvement, which is particularly notable given that mrsFAST-Ultra’s is a cache-oblivious algorithm. Ultimately, the original version is comprised of well over 1000 lines of code and takes over an hour to perform exact matching, whereas the Seq version is comprised of 100 lines of code and completes in about 15 minutes.

### Hamming-distance based all-mapping (mrsFAST)

| <i>Linux</i>          | Time        | Memory      | Seq imprv. | Time         | Memory      | Seq imprv. |
|-----------------------|-------------|-------------|------------|--------------|-------------|------------|
| <b>mrsFAST (C)</b>    | 10:39       | <b>13.1</b> | 1.3×       | 27:03        | 34.5        | 2.7×       |
| <b>Seq</b>            | <b>8:30</b> | 17.8        | —          | 17:25        | <b>25.1</b> | 1.7×       |
| <b>Seq (prefetch)</b> |             | N/A         |            | <b>10:11</b> | 25.2        | —          |

| <i>macOS</i>          | Time        | Memory      | Seq imprv. | Time         | Memory      | Seq imprv. |
|-----------------------|-------------|-------------|------------|--------------|-------------|------------|
| <b>mrsFAST (C)</b>    | 8:47        | <b>13.2</b> | 1.1×       | 17:41        | 34.1        | 1.3×       |
| <b>Seq</b>            | <b>8:08</b> | 19.0        | —          | 20:33        | <b>25.1</b> | 1.6×       |
| <b>Seq (prefetch)</b> |             | N/A         |            | <b>13:17</b> | 25.2        | —          |

(a) *Inexact matching (e = 2, chr1)*      (b) *Exact matching (hg19)*

Table S4: Performance improvements in Hamming distance-based read alignment from using the Seq language and compiler on Linux and macOS systems. The Seq version achieves up to a 2.5× speed improvement over mrsFAST. The reported timings represent the time needed for each tool to find all-mappings for a set of FASTQ reads; index building and loading times are not included. Note that Seq uses mrsFAST’s *k*-mer index for inexact matching, and an FM-index for exact matching, which explains the somewhat varying RAM usages across the tools in different experiments.

**Long-read mapping (minimap2)** As third-generation sequencing becomes more widespread, high-performance methods and tools for processing long-reads become increasingly important [39]. minimap2 [27] is the current state of the art in terms of long-read alignment, both in terms of performance and accuracy. The algorithm used by minimap2 follows similar steps as AVID, and involves chaining to find overlaps between reads and the genome, followed by dynamic programming alignment between the seeds of a chain to obtain the final alignments.

Smith-Waterman sequence alignment is an essential kernel in many genomics applications, and is consequently a heavily researched area [17, 36, 27]. Most hand-optimized implementations use instruction-level SIMD parallelism to accelerate the alignment of a single pair of sequences—an approach referred to as intra-sequence alignment. However, yet another approach is to use SIMD to accelerate aligning *multiple* pairs of sequences, referred to as inter-sequence alignment [36]. While inter-sequence alignment can be substantially faster than intra-sequence alignment, it is typically cumbersome to implement with general-purpose programming languages due to the need to batch sequences, manage state, dispatch alignment results and so on. In Seq, however, the compiler performs several pipeline transformations to convert standard alignment to inter-sequence alignment with just a single-line code annotation.

We implemented minimap2’s Smith-Waterman alignment step in Seq to test the compiler’s inter-sequence alignment optimization relative to minimap2’s SIMD-optimized intra-sequence alignment kernel (in fact, Seq’s default alignment kernel is the same one used by minimap2; minimap2 does not support inter-sequence alignment, however). Results of this comparison are shown in Table S5, where Seq’s inter-sequence alignment optimization is up to 2.5× faster than the intra-sequence implementation. Note that the traceback step (required to produce CIGAR strings) is not vectorized in minimap2 nor Seq, meaning timings including CIGAR gen-

eration are expected to be comparatively strictly slower than those without CIGAR generation. Additionally, Seq automatically demotes sequences that do not benefit from inter-sequence optimization<sup>2</sup> to intra-sequence alignment (i.e., the same method is used to align such pairs in both the Seq implementation and in minimap2), so consequently timings including “all” sequences will also be comparatively slower. Nevertheless, even when including CIGAR generation and all sequence pairs, Seq performs up to 45% better than intra-sequence alignment.

We also implemented a simplified version of this experiment in Rust using Rust-Bio, SeqAn3, and Julia using the BioJulia library [45], in order to compare our implementation with highly-optimized genomics libraries. For a fair comparison, we just compared the speed of standard global alignment without the additional parameters that Seq offers (e.g. bandwidth, Z-drop and a couple other parameters were disabled as other libraries do not support them). Seq is roughly 7–34× faster than the alternatives, even without the inter-sequence alignment optimization. In more concrete terms, Seq can find the full alignments for 50,000 sequence pairs in a few seconds, as compared to couple minutes needed by the other tools. As such, the Seq version is substantially more scalable on real-world datasets.

One distinct advantage of the Seq language over general-purpose languages is the ability to incorporate complex pipeline optimizations—like those for inter-sequence alignment and prefetching—with minimal code changes. For example, adding inter-sequence alignment to the current C implementation of minimap2 would require large-scale code refactoring to batch sequences (and other state information) before performing the alignment, and the complete rewrite of a complex SIMD-based alignment kernel, whereas in Seq all of this is handled implicitly by the compiler using coroutines, and usually requires only a single-line code annotation, as shown in Figure S5 (right half).

### (iii) Data pre- and post-processing

**RNA-seq data clean-up (GATK SplitNCigar)** The most time-consuming steps in a NGS (next-generation sequencing) processing pipeline often consist of mundane data transformations performed on large datasets prior to, during and after the alignment or downstream analysis steps [33]. These tasks typically employ string operations (e.g. duplicate marking, sample tagging), score recalculations (e.g. base quality score re-calibration), and in the case of RNA-seq samples, splitting reads that span large intronic regions. Such tasks are commonly performed by GATK and Picard tools [10, 31]. However, these tools leave a lot to be desired in terms of performance, mainly due to the high-level language they are written in (Java). While data HLpost-processing tasks are often conceptually simple, the sheer volume of data (often on the order of terabytes) heavily penalizes even minor performance shortcomings that high-level languages often bear (e.g. an unnecessary memory copy can easily add several additional hours to the total runtime on real-world datasets). For instance, a relatively simple tool for splitting intron-spanning reads—SplitNCigar—from the GATK suite takes more than 10 hours to complete on a medium-sized 20GB BAM file.

We show that such post-processing can be done faster and easier by re-implementing GATK’s

---

<sup>2</sup>We found that sequences >512 bases did not benefit from inter-sequence alignment; this limit is adjustable.

### Long-read mapping (minimap2; $\approx 1$ million sequences)

| <i>Linux</i>                      | CIGAR       | Score-only | CIGAR                                          | Score-only | Seq imprv.       |
|-----------------------------------|-------------|------------|------------------------------------------------|------------|------------------|
| <b>minimap2 / KSW2</b> (Seq/C)    | 1:28        | 1:14       | 55                                             | 46         | 1.3–2.6 $\times$ |
| <b>Seq</b> (SSE4 inter-align)     | 1:23        | 1:02       | 50                                             | 35         | 1.2–1.9 $\times$ |
| <b>Seq</b> (AVX2 inter-align)     | 1:07        | 51         | 30                                             | 23         | 1.0–1.3 $\times$ |
| <b>Seq</b> (AVX-512F inter-align) | <b>1:07</b> | <b>50</b>  | <b>27</b>                                      | <b>18</b>  | —                |
|                                   |             |            |                                                |            |                  |
| <i>macOS</i>                      | CIGAR       | Score-only | CIGAR                                          | Score-only | Seq imprv.       |
| <b>minimap2 / KSW2</b> (Seq/C)    | 1:06        | 56         | 41                                             | 34         | 1.5–2.3 $\times$ |
| <b>Seq</b> (SSE4 inter-align)     | 59          | 47         | 34                                             | 26         | 1.3–1.7 $\times$ |
| <b>Seq</b> (AVX2 inter-align)     | <b>45</b>   | <b>36</b>  | <b>20</b>                                      | <b>15</b>  | —                |
|                                   |             |            |                                                |            |                  |
| <i>All sequences</i>              |             |            | <i>Small sequences (<math>\leq 512</math>)</i> |            |                  |

### Long-read mapping (minimap2; 50,000 sequences)

| <i>Linux</i>                      | CIGAR     | Score-only | Seq imprv.     |
|-----------------------------------|-----------|------------|----------------|
| <b>minimap2 / KSW2</b> (Seq/C)    | 16        | 12         | 1.1 $\times$   |
| <b>Seq</b> (AVX-512F inter-align) | <b>15</b> | <b>11</b>  | —              |
| <b>Rust-Bio</b> (Rust)            | 6:22      | 3:16       | 19–25 $\times$ |
| <b>SeqAn</b> (C++)                | 3:13      | 2:17       | 12–13 $\times$ |
| <b>BioJulia</b> (Julia)           | 1:52      | 1:52       | 7–10 $\times$  |
|                                   |           |            |                |
| <i>macOS</i>                      | CIGAR     | Score-only | Seq imprv.     |
| <b>minimap2 / KSW2</b> (Seq/C)    | 12        | 10         | 1.1 $\times$   |
| <b>Seq</b> (AVX2 inter-align)     | <b>11</b> | <b>9</b>   | —              |
| <b>Rust-Bio</b> (Rust)            | 6:10      | 3:16       | 21–34 $\times$ |
| <b>SeqAn</b> (C++)                | 2:35      | 1:52       | 12–14 $\times$ |
| <b>BioJulia</b> (Julia)           | 1:24      | 1:24       | 7–9 $\times$   |

Table S5: Performance improvements in long-read mapping and sequence alignment from using the Seq language and compiler on Linux and macOS systems. Seq versions achieves up to a 2.5 $\times$  speed improvement over standard SIMD-optimized alignment algorithms, and up to a 34 $\times$  improvement over their non-SIMD counterparts. The reported timings represent the average time for 3 separate runs to account for SIMD-based time variation. “CIGAR” refers to CIGAR string generation (i.e. the backtrace step), and the “ $\leq 512$ ” experiments were run on sequences shorter than that limit. Seq’s inter-sequence optimization (“inter-align”) was applied on several instruction sets (SSE4, AVX2 and AVX-512F). Our macOS machine did not support the AVX-512F instruction set.

SplitNCigar tool in Seq. By relying on automatic, conservative memory management and thus avoiding unnecessary memory copy operations, we were able to obtain 14 $\times$  faster run-times while using up to 10 $\times$  less memory than the original GATK implementation (note that both Java and Seq rely on a garbage collector for managing memory). In concrete terms, this

| RNA-seq data clean-up (GATK SplitNCigar) |              |            |            |              |            |            |              |            |            |
|------------------------------------------|--------------|------------|------------|--------------|------------|------------|--------------|------------|------------|
| <i>Linux</i>                             | Time         | Memory     | Seq imprv. | Time         | Memory     | Seq imprv. | Time         | Memory     | Seq imprv. |
| <b>GATK</b> (Java)                       | 2:30:55      | 11.0       | 14.5×      | 7:29:52      | 11.8       | 9.8×       | 10:09:08     | 11.6       | 10.4×      |
| <b>Seq</b>                               | <b>10:23</b> | <b>0.7</b> | —          | <b>45:54</b> | <b>4.0</b> | —          | <b>58:36</b> | <b>2.0</b> | —          |
| (a) 1.8 GB BAM                           |              |            |            |              |            |            |              |            |            |
| <i>macOS</i>                             | Time         | Memory     | Seq imprv. | Time         | Memory     | Seq imprv. | Time         | Memory     | Seq imprv. |
| <b>GATK</b> (Java)                       | 1:42:06      | 8.2        | 13.4×      | 5:05:12      | 10.0       | 10.4×      | 8:28:29      | 11.7       | 12.6×      |
| <b>Seq</b>                               | <b>7:38</b>  | <b>0.6</b> | —          | <b>29:16</b> | <b>4.1</b> | —          | <b>40:19</b> | <b>1.9</b> | —          |
| (b) 13 GB BAM                            |              |            |            |              |            |            |              |            |            |
| (c) 20 GB BAM                            |              |            |            |              |            |            |              |            |            |

Table S6: Performance improvements in RNA-seq data post-processing from using the Seq language and compiler on Linux and macOS systems. The Seq version achieves up to a 14× speed improvement over GATK SplitNCigar with significantly lower memory usage. Both tools were run on single-ended BAM files with no optional tags.

| Single-cell data pre-processing (UMI-tools) |          |            |            |
|---------------------------------------------|----------|------------|------------|
| <i>Linux</i>                                | Time     | Memory     | Seq imprv. |
| <b>UMI-tools</b> (Python)                   | 1:01     | 0.1        | 10.2×      |
| <b>Seq</b>                                  | <b>6</b> | <b>0.0</b> | —          |
| <i>macOS</i>                                | Time     | Memory     | Seq imprv. |
| <b>UMI-tools</b> (Python)                   | 56       | 0.1        | 14.0×      |
| <b>Seq</b>                                  | <b>4</b> | <b>0.0</b> | —          |

Table S7: Performance improvements in single-cell barcode whitelisting from using the Seq language and compiler on Linux and macOS systems. The Seq version achieves a  $\geq 10\times$  speed improvement over UMI-tools.

means that a 10-hour post-processing step can be completed in a less than an hour with Seq. Results of this comparison are shown in Table S6.

**Single-cell data pre-processing (UMI-tools)** Single-cell sequencing technologies typically employ short sequence barcodes to differentiate cells within the sequenced sample. However, a sequencing dataset often contains more barcodes than the sequenced cells due to sequencing errors and biases. Thus, the first step in any single-cell RNA-seq pipeline consists of whitelisting *true* barcodes that identify cells, and discarding others based on abundance levels. This task is commonly performed with Python-based tools such as UMI-tools [40].

We have ported this pre-processing step to Seq in less than 90 lines of code, achieving more than a 10× speedup over the original version. Results of this comparison are shown in Table S7.

#### (iv) Downstream analysis

**Haplotype phasing (HapTree-X)** Haplotype phasing is the process of separating variants into a set of maternal and paternal alleles, and can provide valuable biological insights beyond what can be gained from raw genotypes alone [2]. HapTree-X is a phasing algorithm

| Haplotype phasing (HapTree-X) |      |        |                            |      |        |                            |         |        |            |
|-------------------------------|------|--------|----------------------------|------|--------|----------------------------|---------|--------|------------|
| Linux                         | Time | Memory | Seq imprv.                 | Time | Memory | Seq imprv.                 | Time    | Memory | Seq imprv. |
| HapCUT2 (C)                   | 1:23 | 0.1    | 1.0×                       | 4:16 | 1.4    | 0.7×                       | 1:53:00 | 7.2    | 11.5×      |
| phASER (Python)               | 1:53 | 0.7    | 1.3×                       |      |        |                            |         |        |            |
| HapTree-X (Seq)               | 1:25 | 2.9    | —                          | 5:47 | 1.0    | —                          | 9:49    | 5.6    | —          |
|                               |      |        |                            |      |        |                            |         |        |            |
| macOS                         | Time | Memory | Seq imprv.                 | Time | Memory | Seq imprv.                 | Time    | Memory | Seq imprv. |
| HapCUT2 (C)                   | 1:29 | 0.1    | 1.7×                       | 4:14 | 1.1    | 1.1×                       | 1:29:19 | 7.9    | 12.1×      |
| phASER (Python)               | 1:14 | 0.7    | 1.4×                       |      |        |                            |         |        |            |
| HapTree-X (Seq)               | 52   | 2.9    | —                          | 3:47 | 1.0    | —                          | 7:24    | 5.9    | —          |
| (a) Small RNA-seq BAM (3 GB)  |      |        | (b) Medium WXS BAM (17 GB) |      |        | (c) Medium 10X BAM (13 GB) |         |        |            |

Table S8: Performance improvements in haplotype phasing from using the Seq language and compiler on Linux and macOS systems. The Seq version achieves up to a 12× speed improvement over other tools.

implemented in Seq on top of HapTree [7], which utilizes Bayesian optimization to infer optimal haplotypes from aligned genomic data. In contrast to other case studies that focus mostly on discrete optimization and sequence operations, the HapTree-X phasing algorithm is a probabilistic optimization framework that spends most of its runtime doing numerical computations and graph traversals. Furthermore, HapTree-X is a complete genomics analysis pipeline that needs to pre-process and analyze large BAM files prior to the haplotype phasing step. As such, it is an excellent test of Seq’s versatility and applicability to applications that require both efficient sequence processing *and* high-performance numerical algorithms.

Performance of Seq/HapTree-X on three diverse datasets spanning multiple sequencing technologies, relative to another two state-of-the-art phasing tools, is shown in Table S8. HapTree-X was faster than phASER (Python), and had comparable performance to that of HapCUT2 (C), on smaller RNA-seq and medium-sized exome datasets. However, we found that the performance gains increased with the scale of the data: on a large 120GB WGS BAM, HapTree-X was 3× faster than HapCUT2, while on a high-coverage 10X Genomics dataset (≈200GB BAM file), HapTree-X achieved up to a 12× speed up over HapCUT2 (C++, Python). Furthermore, parallelizing the HapTree-X pipeline and running it on four threads (by simply converting a few nested function calls into a Seq pipeline) further decreased the total runtime over 2×, totalling in a nearly 25× speed increase. In absolute terms, HapTree-X was able to phase a 200GB file in less than an hour as compared to the 22 hours needed by the fastest alternative tool [7] (note that Table S8 reports the measurements on a smaller subset of the original 120GB 10X BAM file; the results for the original dataset, as well as other datasets, are available in [7]). We should mention that HapTree-X is *not* a re-implementation of HapCUT2 or a similar tool, but a completely new tool that uses a different algorithm [7]; however, as all tools utilize many common data-processing steps that take the vast majority of time (SAM/BAM pre-processing, read analysis, etc.), it is fair to say that most of the speed improvements are due to Seq’s optimizations.

## Note 3: Supplementary Methods

### Seq design and implementation

Seq is designed to be easy to learn, use and maintain, without sacrificing performance. To this end, it borrows Python’s syntax, semantics, and standard libraries, but performs ahead-of-time type checking and compilation to avoid Python’s (specifically, CPython’s) substantial runtime performance overhead. Unlike Python, which compiles to bytecode and runs inside a virtual machine, Seq uses the LLVM compiler infrastructure [24] to perform general-purpose optimizations and generate native machine code, leading to a more than  $100\times$  performance improvement over standard Python in many cases [38].

The fundamental datatype underlying most bioinformatics or genomics applications is the “sequence”, which is conceptually a string of nucleotide (“A”, “C”, “G”, “T”) bases (or possibly amino acids in the case of protein sequences; while we focus on DNA sequences here, the same principles can be carried over to RNA or protein sequences, which Seq also has full support for). Below, we lay out how Seq represents and manipulates sequences and  $k$ -mers to optimize genomics applications. We also discuss the notion of a “pipeline”, including how Seq represents pipelines at the language level as well as the numerous pipeline optimizations the compiler performs.

### Preliminaries

For a sequence  $s$  over an alphabet  $\Sigma$  (where usually  $\Sigma = \{A, C, G, T\}$ ), we let  $|s|$  be the length of  $s$ ,  $s[i]$  be the character at 0-based index  $i$ ,  $s[i : j]$  be the subsequence from 0-based index  $i$  (inclusive) to 0-based index  $j$  (exclusive), and  $s \parallel t$  be the concatenation of  $s$  with another sequence  $t$ . Further, let  $\bar{s}$  be the *reverse complement* of  $s$ . Reverse complementation is a common operation in computational genomics where a given sequence  $s$  is transformed into a new sequence  $t$  such that  $t[i] = \text{RevComp}(s[|s| - i - 1])$  for  $0 \leq i < |s|$ , where

$$\text{RevComp}(c) = \begin{cases} T & c = A \\ G & c = C \\ C & c = G \\ A & c = T \end{cases} \quad (1)$$

We define a  $k$ -mer to be a sequence  $s$  of fixed length  $|s| = k$ .  $k$  is typically on the order of 10 to 100. Let  $\kappa_k(s)$  be the set of all  $k$ -mers that exist as subsequences in sequence  $s$ .

These definitions give rise to various algebraic rules, each of which Seq uses to perform various optimizations. For example:

1.  $\bar{\bar{s}} = s$ : “the reverse complement of the reverse complement is the original sequence” – enables efficient  $O(1)$  lazy sequence reverse complementation.
2.  $\bar{s} = \overline{s[i : |s|]} \parallel \overline{s[0 : i]}, 0 \leq i \leq |s|$ : “the reverse complement is the concatenation of the reverse complements of the two halves of the sequence, in reverse order” – enables

```

s = s'ACGTACGT'      # sequence literal
print s[2:5]          # subsequence
print ~s              # reverse complement
kmer = Kmer[8](s)     # convert to k-mer

# iterate over length-3 subsequences
# with step 2
for sub in s.split(3, step=2):
    print sub[-1]     # last base

# iterate over 4-mers with step 1
for kmer in s.kmers[Kmer[4]](step=1):
    print ~kmer       # '~' also works on k-mers

```

Figure S2: Examples of using sequence and  $k$ -mer types in Seq.

efficient  $k$ -mer reverse complementation via a lookup table.

3.  $\{\bar{t} \mid t \in \kappa_k(s)\} = \{t \mid t \in \kappa_k(\bar{s})\}$ : “iterating over reverse complemented  $k$ -mers can be done by iterating over  $k$ -mers of the reverse complemented sequence” – enables loop reordering for  $k$ -mer iteration followed by reverse complementation.

Below, we discuss in greater detail how each of these types and operations is used and implemented in Seq, and provide numerous benchmarks as to their efficacy.

## The sequence type

It goes without saying that computational genomics applications largely deal with processing and operating on sequences, and so an efficient *sequence type* implementation is invaluable. Seq provides a sequence type `seq` and a separate  $k$ -mer type `Kmer[k]` for  $1 \leq k \leq 1024$ . While the sequence type and the  $k$ -mer type are conceptually both strings of nucleotide bases, they differ largely in where and how they are used: sequences have arbitrary length and can be accompanied by various metadata like an error profile, quality scores or ambiguous bases;  $k$ -mers on the other hand, at least in how they are frequently used in practice, are fixed-size segments of some larger sequence. For example, a typical alignment algorithm would employ short  $k$ -mers obtained from an arbitrary-length sequencing read, in conjunction with some  $k$ -mer index, to obtain candidate alignments for that read, where  $k$  is usually chosen as a fixed parameter beforehand, but read lengths are not known until runtime. In this case, the reads would be of type `seq` and the  $k$ -mers would, unsurprisingly, be of type `Kmer[k]` for some  $k$ . In practice, general sequences can also have a larger alphabet than  $k$ -mers (which are restricted to just the four nucleotide bases); for instance, N indicates an ambiguous base, R indicates an A or a G, etc. Figure S2 gives a few examples of using sequence and  $k$ -mer types in Seq, showing the general syntax as well as some common operations and how to convert between the two.

The following operations on sequences and  $k$ -mers are common, and are the subject of several

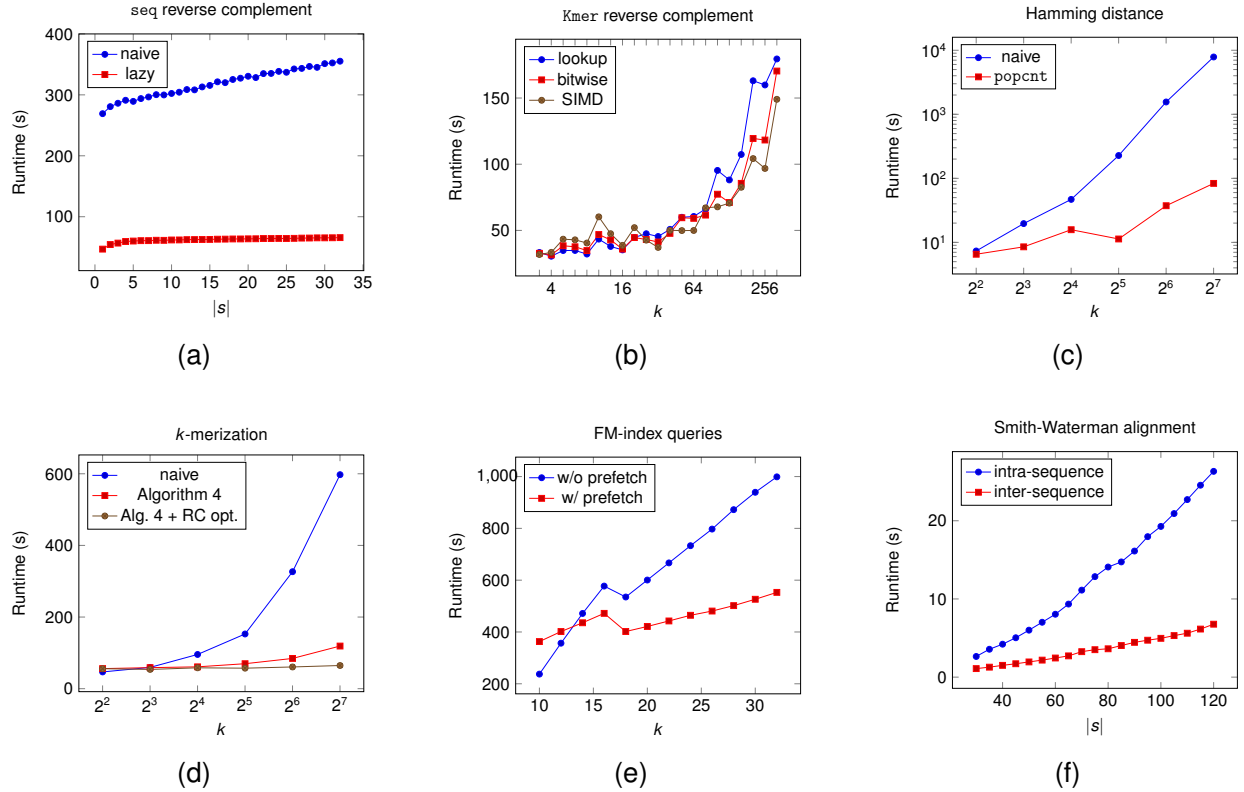


Figure S3: Effect of several compiler optimizations performed by the Seq compiler on various small benchmarks. Reverse complement performance was measured by counting sequences (or  $k$ -mers) lexicographically larger than their reverse complements; Hamming distance performance by computing distances between  $k$ -mers and their reverse complements; FM-index query performance by querying  $k$ -mers from a set of Illumina reads; Smith-Waterman alignment by aligning one million simulated sequence pairs of maximum length  $|s|$  with 90% identity and maximum indel size of 3. The reference genome used in each applicable benchmark was hg19.

compiler optimizations performed by Seq: *subsequence extraction/iteration*, *reverse complementation*, *k-merization* (the process of iterating over a sequence's constituent  $k$ -mers), *Hamming distance computation*, *hashing / indexing*, and *dynamic programming alignment* (e.g. Smith-Waterman or Needleman-Wunsch).

## Implementation

Seq's sequence type is implemented as a 16-byte pass-by-value structure containing an 8-byte character pointer and an 8-byte length. The C equivalent would be:

```
struct seq_t { char *ptr; int64_t len; };
```

Subsequence operations do not copy, but rather just return a new sequence instance whose pointer is at the appropriate offset from the original's. Seq then uses a conservative garbage

collector to ensure that memory is deallocated as needed, and that nothing is prematurely deallocated, which is especially important when employing this scheme as many pointers will be referring to the middle of some larger allocated block. (In general, there are many ways to implement a sequence data type, and future work entails choosing the best variant based on context at compile-time, be it an ASCII representation, a 2-bit encoding or something else. We found the described implementation to work well in a wide range of cases, however.)

Sequence reverse complementation is an  $O(1)$  operation in Seq, implemented not by copying and physically reverse complementing, but by simply, “lazily”, flipping the sign of the sequence length. Then, each sequence method (e.g. for subsequence,  $k$ -merization, etc.) first checks the sign of the length, and uses it to determine if the sequence should be treated as reverse complemented or not (of course, the actual length is given by the absolute value). Conceptually, therefore, each sequence  $s$  is a 2-tuple of the underlying “logical” sequence  $s_{\log}$  and a length  $s_{\text{len}}$ , such that  $|s| = |s_{\log}| = |s_{\text{len}}|$ , and  $s_{\text{len}} < 0$  if and only if  $s$  is really reverse complemented with respect to  $s_{\log}$ . The reverse complementation algorithm is shown in Algorithm 1.

---

**Algorithm 1:** Reverse complement of sequence  $s = (s_{\log}, s_{\text{len}})$ .

---

**Result:**  $\bar{s}$

1 **return**  $(s_{\log}, -s_{\text{len}})$ ;

---

An example implementation of  $s[\cdot]$  under this scheme is shown in Algorithm 2; notice that the sign of  $s_{\text{len}}$  must first be checked in order to determine how to index into  $s_{\log}$ . If  $s_{\text{len}}$  is positive, we index regularly; if it is negative, on the other hand, we treat  $s_{\log}$  as reverse complemented and index correspondingly.

---

**Algorithm 2:** Sequence indexing for sequence  $s = (s_{\log}, s_{\text{len}})$ .

---

**Result:**  $s[i]$

1 **if**  $s_{\text{len}} > 0$  **then**  
 2     **return**  $s_{\log}[i]$ ;  
 3 **else**  
 4     **return**  $\text{RevComp}(s_{\log}[(-s_{\text{len}}) - i - 1])$ ;  
 5 **end**

---

The appeal of this approach is that it not only saves memory and copying, but can also be applied multiple times without issue. For instance, clearly  $\bar{\bar{s}} = (s_{\log}, -(-s_{\text{len}})) = s$ . A more complicated example is  $\bar{s}[i : j]$ , which also works as shown below, along with a concrete example where  $s = \text{AAAGGGTTTCCC}$  ( $|s| = 12$ ) with  $i = 2$  and  $j = 5$ :

$$\begin{aligned} \bar{s}[i : j] &= \overline{(s_{\log}, -s_{\text{len}})[i : j]} \\ &= \overline{(s_{\log}[|s| - j : |s| - i], -s_{\text{len}})} \\ &= (s_{\log}[|s| - j : |s| - i], s_{\text{len}}). \end{aligned}$$

$$\begin{aligned} s &= \text{AAAGGGTTTCCC}, \\ \bar{s} &= \text{GGGAAACCCTTT}, \\ \bar{s}[2 : 5] &= \text{GAA}, \\ \overline{\bar{s}[2 : 5]} &= \text{TTC}, \\ s[|s| - 5 : |s| - 2] &= s[7 : 10] = \text{TTC}. \end{aligned}$$

The drawback of this approach is that all sequence operations must include a preliminary check of  $s_{\text{len}}$ 's sign, although we have found the overhead of this to be minimal, and greatly outweighed by the benefits (Figure S3a). In fact, this check can be elided altogether by the compiler in many cases (e.g. when sequences are first read from disk, at which point nothing has yet been reverse complemented).

Figure S3a shows the performance of this approach compared to the naive method of simply copying and reverse complementing. The benchmark shown in Figure S3a entails counting all subsequences of a given length in the human reference genome that are lexicographically larger than their reverse complement. Notice that the naive implementation is  $O(|s|)$  in the best case, whereas the lazy approach is best-case  $O(1)$  since bases must only be read until the lexicographic order of the given sequence and its reverse complement can be determined (by contrast, the naive approach must always read all bases of the subsequence to reverse complement it). Hence, the lazy approach not only performs better, but also scales better, as shown in the figure.

### ***k*-mers**

In Seq, a *k*-mer's length is defined by its type (e.g. `Kmer[5]` is a different type than `Kmer[32]`), so unlike sequences the length does not need to be stored explicitly. Seq stores *k*-mers in a 2-bit encoded format, where `Kmer[k]` maps to LLVM IR type `iN` with  $N = 2k$ . LLVM then maps that integral type to hardware; for  $k \leq 32$ , a *k*-mer fits in a single machine register on 64-bit systems, whereas larger *k*-mers may be spilled to the stack. Notationally, let  $\xi_k(\cdot)$  be the bijection from the set of all *k*-mers to unique 2-bit encodings.

### **Reverse complement**

The Seq compiler uses three different approaches to reverse complement *k*-mers:

- *Lookup*: The lookup method uses a lookup table of hard-coded 4-mer reverse complements (taking up  $4^4 \times (2 \cdot 4)$  bits, or 256 bytes). Reverse complements of longer *k*-mers are then constructed using the fact that  $\bar{s} = \bar{s}[4 : |s|] \parallel \bar{s}[0 : 4]$  (recursively), i.e. the first four bases are reverse complemented, followed by the next four and so on, after which they are concatenated (in this case via bitwise shifts) in reverse order. For  $k < 4$ , the 2-bit encoding can simply be padded with A bases to construct a 4-mer, and the resulting T bases can be trimmed after reverse complementation.
- *Bitwise*: The bitwise approach uses a series of bitwise operations to reverse the 2-bit pairs of the encoded *k*-mer, then applies a bitwise-NOT to obtain the complement, the end result of which is the reverse complement. This is a generalization of the approaches used in GATB [15] and Jellyfish [30] for arbitrary-width integer encodings.
- *SIMD*: The SIMD approach uses a vector shuffle instruction to reverse the bytes of the encoded *k*-mer, then uses bitwise operations on each byte of the reversed vector in a similar fashion as the bitwise approach to obtain the final reverse complement. A similar method is implemented in MMseqs2 [42], although this implementation uses x86-specific `pshufb` instructions to reverse complement individual bytes in the reversed

vector, whereas Seq’s LLVM IR implementation uses just the target-agnostic LLVM IR `shufflevector` instruction followed by a series of shifts and bitmasks to achieve the same effect.

The performance of each of these approaches for a wide range of  $k$  is shown in Figure S3b. As shown, there is no clear winner between these three methods. For smaller  $k$  ( $\leq 20$ ), the lookup method is almost always best; for larger  $k$  ( $\geq 32$ ), the SIMD approach is almost always best. For  $k$  values between these two cutoffs, the optimal algorithm varies, but we found the bitwise approach to be consistently close to, if not the best option. Because of this, the Seq compiler employs this heuristic to decide how to reverse complement a given  $k$ -mer.

### **$k$ -mer hashing**

$k$ -mer hashing is an extensively researched area [29]. For  $k \leq 32$ , Seq simply uses the encoded 2-bit value as the hash by default. For  $k > 32$ , on the other hand, the situation is somewhat more complicated. One option is to use the hash of the first 32 bases, but due to the highly non-uniform structure of the genome, this leads to excessive collisions; for example, all  $k$ -mers consisting of a string of 32 As followed by unique bases would hash to the same value. For this reason, hashing  $k$ -mers for  $k > 32$  is done by hashing the first and last 32 bases, and XORing the two hash values. Compared to the naive approach, this method reduces the average collisions per unique 64-mer in hg19 from 1.089 to 1.007, and more than halves the size of the largest bucket. The best way to hash  $k$ -mers can vary based on the application, so Seq enables the user to override the  $k$ -mer hash function if needed.

### **Hamming distance**

The Hamming distance of two  $k$ -mers  $s$  and  $t$  is defined to be  $|\{i \mid s[i] \neq t[i], 0 \leq i < k\}|$ ; i.e. the number of positions at which corresponding bases differ. A naive Hamming distance algorithm is to iterate over the bases of the two  $k$ -mers one at a time, and count how many differ. Seq employs a different algorithm using the `popcnt` instruction (which computes the number of 1 bits in an integer’s binary representation), shown in Algorithm 3.

---

#### **Algorithm 3:** Hamming distance between two $k$ -mers $s$ and $t$ .

---

**Result:**  $|s - t|$

- 1  $m_1 \leftarrow \overbrace{010101 \dots 01}_k;$
- 2  $m_2 \leftarrow \overbrace{101010 \dots 10}_k;$
- 3  $r_1 \leftarrow (\xi_k(s) \wedge m_1) \oplus (\xi_k(t) \wedge m_1);$
- 4  $r_2 \leftarrow (\xi_k(s) \wedge m_2) \oplus (\xi_k(t) \wedge m_2);$
- 5  $d \leftarrow (r_1 \ll 1) \vee r_2;$
- 6 **return** `popcnt`( $d$ )

---

Algorithm 3 works by first checking for differences in the odd bits via the  $m_1$  bitmask, then checks the even bits using  $m_2$ , and finally ORs the results to obtain a bit-vector with 1s cor-

```

from sys import argv
K = Kmer[20] # seed length
...
index = read_genome_index_from_file(argv[1])
(FASTQ(argv[2])) |>
  iter |>
    find_candidate_mappings[K](index) |>
      filter_candidate_mappings |>
        smith_waterman_align |>
          output_to_disk(argv[3]))

```

Figure S4: Hypothetical pipeline for read mapping in Seq, where the first argument (`argv[1]`) is some serialized genome index file, the second is a FASTQ file containing input reads, and the third is the output file. The shown pipeline is parallelized using the parallel pipe operator, `|>`, and each read can be processed in parallel.

responding to different bases, a population count of which is the desired result. We leave to LLVM the decision of how to lower the logical `popcnt` instruction to actual machine instructions. For longer  $k$ -mers, the `popcnt` version can be substantially faster than the naive version, as shown in Figure S3c. For example, for 128-mers, the `popcnt` approach is nearly  $100\times$  faster.

## Pipelines

Pipelining is a natural model for thinking about processing genomic data. For example, a typical read mapping algorithm can be formulated as the pipeline:

1. Read input reads from FASTQ file
2. Extract seeds and lookup in some genomic index
3. Filter candidate alignment positions
4. Perform full Smith-Waterman alignment on remaining candidates
5. Format and output results to SAM file

Pipelining is supported natively in Seq via the pipe operator, `|>`. Pipelines can be parallelized via the parallel pipe operator, `||>`, which allows all subsequent stages to be executed in parallel (the Seq compiler uses an OpenMP task backend to implement this; future work entails implementing other backends, such as GPU for highly parallelizable tasks or MPI for distributed computing). In Seq, a `|> b` simply passes the output of `a` to `b` if `a` is a function; if `a` is a generator, all values yielded by `a` are passed to `b`. An example of how pipelines might be used to implement a typical read mapping application with the stages described above is shown in Figure S4.

The Seq compiler performs various domain-specific optimizations on pipelines, which we describe here.

## ***k*-merization**

An extremely common pattern is to iterate over the  $k$ -mers of some sequence with a particular step size  $r$ . In Seq, this can be achieved using the `kmers[K]` function, where  $K$  is the target  $k$ -mer type. For instance, the following code:

```
K = Kmer[2]
s'ACGT' |> kmers[K](step=1) |> echo
```

prints all 2-mers in the sequence ACGT (i.e. AC, CG and GT). Obtaining  $k$ -mers with positions as a tuple can be done with the analogous `kmers_with_pos[K]` function.

The best way to perform this  $k$ -merization depends on  $k$  and the step size  $r$ . For example, if  $r \geq k$ , the best we can do is to simply extract the length- $k$  subsequences at offsets 0,  $r$ ,  $2r$ , etc. and convert them to  $k$ -mers. However, if  $r < k$ , then  $k$ -mers overlap, so we can reuse the previous  $k$ -mer to obtain the next, similar to a rolling hash. The situation is complicated somewhat by the presence of non-ACGT bases in the sequence;  $k$ -mers spanning such bases are skipped by `kmers`. A simplified version of the  $k$ -merization algorithm used in Seq is shown in Algorithm 4. In this algorithm, the two main branches inside the loop dictate whether the current  $k$ -mer must be “refreshed” because a non-ACGT base was encountered; if so, the  $k$ -mer is re-encoded afresh via `SequenceToKmer`, otherwise the previous  $k$ -mer is “shifted” to the right via `ShiftKmer` to cover the new bases, without doing a full re-encoding from scratch. Also notice that if  $r \geq k$ , the second branch is never entered, and we perform a full re-encoding for every  $k$ -mer. This style of  $k$ -merization is not unique to Seq, and is often used in performance-sensitive genomics applications; the optimizations we discuss below, however, specifically require a domain-specific compiler like Seq to be performed automatically.

## **Reverse complementation**

$k$ -merization followed by reverse complementation can be implemented more efficiently by using property  $\{\bar{t} \mid t \in \kappa_k(s)\} = \{t \mid t \in \kappa_k(\bar{s})\}$ , as described above. Because of this, the Seq compiler internally makes transformations of the following form:

```
A |> kmers[K](step) |> revcomp |> B
```

↓

```
A |> _kmers_revcomp[K](step) |> B
```

where `_kmers_revcomp[K]` is a function that iterates over reverse complemented  $k$ -mers by first reverse complementing the entire sequence *then*  $k$ -merizing.

The effect of these optimizations is shown in Figure S3d, which shows the performance of a  $k$ -merization and reverse complementation pipeline for a naive implementation (i.e. every  $k$ -mer is encoded and reverse complemented individually), an implementation using Algorithm 4 for  $k$ -merization, and finally Seq’s implementation, which uses both Algorithm 4 and the reverse complement optimization described above.

---

**Algorithm 4:**  $k$ -mers contained in sequence  $s$  at offsets a multiple of  $r$ , skipping non-ACGT bases. For simplicity, we ignore reverse complementation and assume  $s$  is in the forward direction.

---

**Result:**  $k$ -mers in  $s$  with step size  $r$

---

```

1   $\Sigma := \{A, C, G, T\};$ 
2   $i \leftarrow 0;$ 
3   $b \leftarrow \text{False};$ 
4   $K \leftarrow \underbrace{AA \dots A}_k;$ 
5   $\kappa \leftarrow \emptyset;$ 
6  while  $i + k \leq |s|$  do
7      if  $b = \text{True}$  then
8           $s' \leftarrow s[i : i + k];$ 
9          if  $c \in \Sigma, \forall c \in s'$  then
10              $K \leftarrow \text{SequenceToKmer}(s', k);$ 
11              $b \leftarrow r \geq k;$ 
12              $\kappa \leftarrow \kappa \cup \{(i, K)\};$ 
13         end
14     else
15          $s' \leftarrow s[i + k - r : i + k];$ 
16         if  $c \in \Sigma, \forall c \in s'$  then
17              $K \leftarrow \text{ShiftKmer}(K, s');$ 
18              $\kappa \leftarrow \kappa \cup \{(i, K)\};$ 
19         else
20              $b \leftarrow \text{True};$ 
21         end
22     end
23      $i \leftarrow i + r;$ 
24 end
25 return  $\kappa$ 

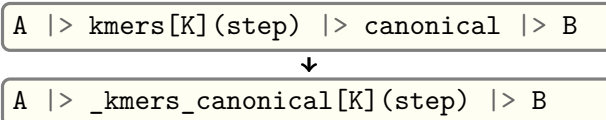
```

---

## Canonical $k$ -mers

A “canonical”  $k$ -mer is the minimum (often, lexicographic minimum) of a  $k$ -mer and its reverse complement. Canonical  $k$ -mers are frequently used in  $k$ -mer counting algorithms as well as various hashing schemes to ensure that  $k$ -mers are treated as equal to their reverse complements [29, 46]. For example, many  $k$ -mer counting methods report counts of canonical  $k$ -mers rather than raw  $k$ -mer counts. Consequently, a common pipeline pattern is to pipe the output of `kmers` to the `canonical` function, which simply returns the canonicalization of its argument.

A naive implementation of such a pipeline might reverse complement each  $k$ -mer individually, but a better approach is to use two sliding windows when iterating over  $k$ -mers: one for the forward-direction  $k$ -mer, and one for the reverse complement. When the step size is 1, for instance, the next base is shifted in on the right of the forward window, while the *complement* of the base is shifted in on the *left* of the reverse window, thereby ensuring the reverse window always holds the reverse complement of the forward window. Using this approach, no  $k$ -mer ever actually needs to be reverse complemented during the iteration. The Seq compiler recognizes this pattern and makes use of a double-sliding-window implementation, `_kmers_canonical`:



## Software prefetching

Large genomic indices, coupled with cache-unfriendly access patterns, often result in a substantial fraction of stalled memory-bound cycles in many genomics applications [38, 3, 47]. As a result, the Seq compiler performs optimizations to overlap the cache miss latency of an index lookup with other useful work, similar to the general-purpose approaches described in [20] and [11].

To illustrate the effect of this optimization, consider the pipeline `A |> B |> C` where function `B` performs an index lookup. In Seq, by using the `@prefetch` annotation on `B`, the function is implicitly transformed into a coroutine which performs a software prefetch, yields, then commences with the actual index lookup once it is resumed. Code for the pipeline itself is generated to include a dynamic scheduler, which manages multiple instances of the `B` coroutine, suspending and resuming them as needed until all are complete. In this way, while one instance of `B` is performing its prefetch, another can continue doing useful work, compared to the original pipeline where progress would be stalled until a given index lookup completes. A concrete example for FM-indices is shown in Figure S5, wherein the typical backwards-extension FM-index search algorithm is applied to count occurrences of subsequences from an input FASTQ file.

The performance of the code from Figure S5 for various values of  $k$ , both with and without the `@prefetch` line, is shown in Figure S3e. For smaller  $k$ , the “head” of the FM-index is largely cache-resident, so the prefetch optimization has no benefit. For larger  $k$ , however, the effect of cache misses increases and the optimization significantly reduces runtime (by almost  $2\times$  at

$k = 32$ ). The prefetch optimization also works with other types of indices, such as hash tables, although we found it to be most effective on tree-like structures like FM-indices.

## Inter-sequence alignment

Sequence alignment via a dynamic programming algorithm like Smith-Waterman is arguably one of the most prevalent operations in genomics applications, and has been the subject of much research and optimization, particularly with regards to SIMD parallelization [43, 13, 36]. There are two approaches to SIMD alignment, referred to as *intra-sequence* and *inter-sequence*. An intra-sequence implementation uses SIMD to align a single pair of sequences, whereas an inter-sequence approach aligns a batch of pairs at once, and uses SIMD across different sequence pairs rather than within one pair. Inter-sequence alignment can be substantially faster than intra-sequence, but is rarely used in practice due to programming difficulty [44].

In Seq, however, inter-sequence alignment is just as easy to program as intra-sequence, as shown in Figure S5. When a function marked `@inter_align` is used in a pipeline, the Seq compiler will perform transformations similar to those used for software prefetching; specifically, the function is transformed into a coroutine and suspended just before the alignment. During execution of the pipeline, several hundred suspended coroutines are batched, have their sequences aligned using inter-sequence alignment, and are resumed after the alignment result is returned to them. Similar to the prefetch optimization, this is orchestrated by a dynamic scheduler of coroutines, the code for which is generated in the body of the pipeline. An illustration of such a pipeline before and after this transformation is shown in Figure S5 (bottom right).

Performance of inter-sequence alignment compared to intra-sequence is shown in Figure S3f, where inter-sequence is shown to be almost  $4\times$  faster in some cases. The code difference between the two implementations used in this figure is just a single line: the `@inter_align` annotation on the function performing the alignment. Note that Seq uses KSW2 [27, 43] as its intra-sequence alignment kernel.

This approach to alignment also lends itself well to other backends. For example, it would be possible to instead perform inter-sequence alignment on a GPU or FPGA rather than a CPU, and tune the parameters (e.g. sequence buffer size) for each backend. A domain-specific language and compiler such as Seq makes it possible to explore various backends and their benefits in a systematic way, and we plan to pursue this as future work.

## Note 4: Supplementary Discussion

The creation of high-performance, maintainable bioinformatics software is undoubtedly a difficult task, as evidenced by the field's unfortunate track record of poor software development, maintenance and testing practices. A key cause of this, among other factors, is a lack of domain-specific development tooling, debugging facilities and general programming infrastructure. While several software libraries have been developed as an attempt at answering

```

from sys import argv
from bio.fminindex import FMIndex
fmi = FMIndex(argv[1])
k, step, n = 20, 20, 0
def add(count): n += count

@prefetch
def search(s, fmi):
    intv = fmi.interval(s[-1])
    s = s[:-1] # trim last base
    while s and intv:
        # backwards-extend intv
        intv = fmi[intv, s[-1]]
        s = s[:-1] # trim last
    # return count of occurrences
    return len(intv)

FASTQ(argv[2]) |> seqs |> \
    split(k, step) |> search(fmi) |> add
print 'n:', n

```

```

from sys import argv
in1 = argv[1] + '.1.txt'
in2 = argv[1] + '.2.txt'

@inter_align
def process(t):
    query, target = t
    # align and get score
    score = query.align(target).score
    print query, target, score

zip(seqs(in1), seqs(in2)) |> process

```

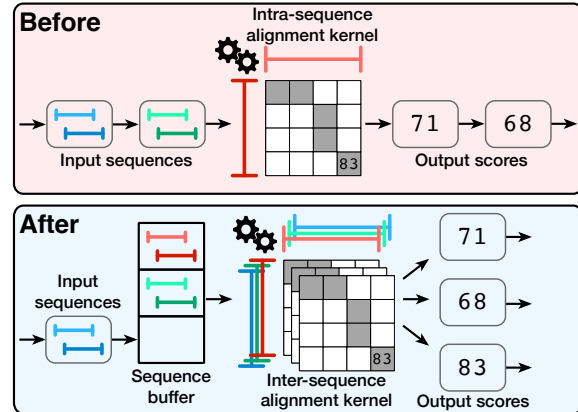
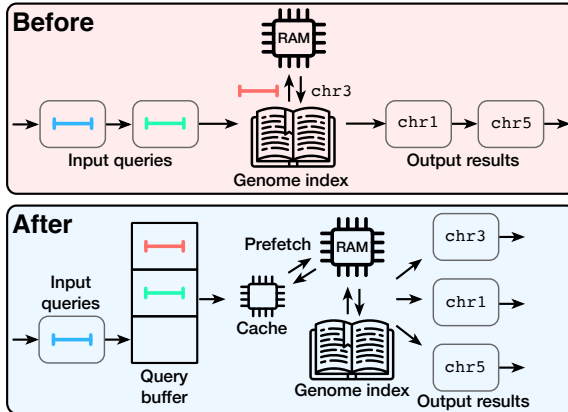


Figure S5: Coroutine-based pipeline optimizations in Seq. The top left snippet gives an example of using the `@prefetch` annotation with a pipeline performing FM-index queries; the depicted code simply counts occurrences of length-20 subsequences from a given FASTQ using the index. The top right snippet gives an example of inter-sequence alignment using the `@inter_align` annotation, where sequence pairs are read from two text files, aligned, and printed. The bottom row illustrates the resulting pipeline optimizations performed by the Seq compiler. For genomic index prefetching, pipeline stages that query a large index are converted to coroutines and suspended before querying the index, after issuing a corresponding software prefetch; a dynamic scheduler manages the coroutines and ultimately ensures that any cache miss latency is overlapped with useful work by another coroutine. For inter-sequence alignment, in the original pipeline, sequence pairs are passed through the pipeline one at a time and aligned individually; in the transformed pipeline, sequence pairs are batched by a dynamic scheduler, aligned with inter-sequence alignment, and returned to their respective suspended coroutines.

this problem, libraries are unable to perform many of the optimizations discussed in this work and hence cannot achieve optimal performance in many key situations. Additionally, these libraries are still bound to their host languages, usually at the detriment of either performance or software maintainability.

The fundamental difference with Seq is that its entire software stack, from the language itself to the standard library to the compiler and even runtime, is designed and implemented with the domain of bioinformatics in mind. “Owning” the entire stack in this way comes with numerous distinct advantages. For example, datatypes or compiler optimizations relevant to new sequencing platforms can be incorporated seamlessly into the framework, circumventing large code rewrites or potentially even algorithmic changes. Additionally, as GPUs, FPGAs and cloud computing become increasingly prevalent in the field, new compiler backends for Seq—including new compiler optimizations pertinent to each computing platform—would enable existing code to run without changes on these backends, and is ongoing work within the Seq project. Indeed, many novel, promising computer architectures such as memory-driven computing [6] require specific optimizations and program transformations to be fully taken advantage of; these can be performed automatically by the Seq compiler.

Yet another unique advantage of a framework like Seq lies in its ability to provide domain-specific debugging or visualization support; as the compiler has knowledge of sequences,  $k$ -mers and operations involving them, debugging or visualization tools can be seamlessly integrated into the Seq framework, allowing users to track and observe in a meaningful way how data is transferred and manipulated throughout a program. None of these possibilities are easily attainable without having a software stack designed from the ground up for genomics and bioinformatics. In fact, such domain-specific languages are widely used in a variety of other fields for similar reasons, including computer graphics [35], tensor algebra computing [21] and physical simulations [22].

## Related work

A number of software libraries for bioinformatics and computational genomics have been developed for a variety of programming languages, such as Biopython [12] for Python, BioPerl [41] for Perl, SeqAn [14] for C++, Rust-Bio [23] for Rust or BioJulia [45] for Julia. In the preliminary conference version of this paper, we show that Seq attains substantial speedups over biological packages for Julia and Python [38]. Some of these libraries are bound to low-level host languages (C++, Rust), preventing inexperienced programmers from easily using them; others are bound to high-level languages that suffer from poor performance (Python, Perl). Languages like Julia attempt to bridge the gap between these extremes, but are infrequently used in practice [37]. Seq, by contrast, brings high-performance to a familiar, widely-used language (Python) that is among the most prevalent in the field. Furthermore, many of the optimizations performed by the Seq compiler are simply infeasible for libraries to replicate; for example, the inter-sequence alignment optimization described above requires a global view of the program, the ability to generate arbitrary code, the ability to convert functions to coroutines at compile-time and so on—these transformations are inherently out of reach for libraries, and require a domain-specific compiler. Consequently, libraries are unable to match Seq’s per-

formance on many of the applications re-implemented in this work. Note that we limited our comparisons to existing implementations, none of which actually use these libraries.

We note that some of these libraries, such as Biopython, currently offer more features than the Seq standard library. This is understandable if the age of each project is considered: while Seq is about 2 years old, Biopython is more than 20 years old. However, Seq’s Python interoperability allows one to easily import *any* API (Application Programming Interface) from Biopython and use it in Seq. For example, a user can easily use Biopython to parse some exotic file format, and then use the results of Biopython to build up an efficient post-processing pipeline; see Supplementary Note 1 for more details. We plan for Seq’s standard library to soon catch up with other libraries as well (e.g. support for less common formats, less common string matching techniques, and support for proteomics data).

SARVAVID [28] is a previous attempt at designing a high-level, domain-specific language for computational genomics. However, SARVAVID provides only very high-level, coarse-grained primitives (“kernels”) at the language level (such as “*k*-merization”, “error correction”, “index lookup”, etc.); in practice, these operations are extremely nuanced and the details vary greatly between applications, making SARVAVID unsuitable for writing new applications that do not fit this exact mold. Seq is designed to be a much lower-level language than SARVAVID, allowing users to write *new* genomics kernels to suit the needs of their specific application. Lastly, and perhaps most importantly, we were unable to find any implementation of SARVAVID—neither as source nor pre-compiled binaries—online for testing purposes, and to our knowledge it is not used in practice.

Seq’s syntax and semantics are borrowed from Python. Previous general-purpose attempts at making Python itself more performant include PyPy [8], Shed Skin [16], Nuitka [19] and a few others. Most of these approaches still rely on CPython’s (i.e. the “standard” Python implementation’s) runtime, and therefore bear many of the same overheads. Seq is designed as a standalone language from the ground up, where all type-checking is performed at compile-time and there is zero overhead at runtime, leading to substantial performance improvements not only over CPython but also over these alternate implementations.

**Performance benchmarking.** To evaluate runtime and memory usage of our Seq implementations, all experiments were run on the following systems:

1. dual-socket system with Intel Xeon Gold 5218 CPUs (2.30 GHz) with 16 cores each (totalling 32 cores and 64 hyper-threads) and 768GB DDR4-2933 RAM and Linux OS.
2. desktop iMac with Intel Core i9-9900K CPU (3.60 GHz) with 8 cores (totalling 16 hyper-threads) and 64GB DDR4-2667 RAM and macOS Big Sur.

The Seq version is 0.10.1.

**Code availability.** Seq is freely available at <https://seq-lang.org>. We provide the source code for each of the Seq implementations presented here on GitHub at <https://github.com/seq-lang/seq-benchmarks>, with the exception of HapTree-X, which is available at <https://github.com/HapTree-X>.

`//github.com/0xTCG/haptreex.`

**Acknowledgments.** The authors thank Deniz Yorukoglu for helpful discussions, Brian Hie for invaluable comments and edits on the main text, as well as Jordan Watson, Jodie Weldon and Matthew Ha for their contributions to the Seq standard library. We also thank Mark Henderson for helpful suggestions regarding Seq. A preliminary conference version that showcased Seq and focused on specifics of the programming language itself—without genomics applications—appeared in OOPSLA [38].

**Funding.** A.S. was partially supported by NIH grant R01HG010959 (to B.B. and S.A.). I.N. was supported by NSERC Discovery Grant (RGPIN-2019-04973), Canada Research Chairs Program, and Canada Foundation for Innovation’s John R. Evans Leaders Fund. H.G. was supported by NSERC Discovery Grant (RGPIN-2020-04243).

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

**Correspondence.** Correspondence and requests for materials should be addressed to Bonnie Berger (bab@mit.edu) and Saman Amarasinghe (saman@csail.mit.edu).

## References

- [1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1):53–86, 2004.
- [2] Ziad Al Bkhetan, Justin Zobel, Adam Kowalczyk, Karin Verspoor, and Benjamin Goudey. Exploring effective approaches for haplotype block phasing. *BMC Bioinformatics*, 20(1):540, Oct 2019.
- [3] R. Appuswamy, J. Fellay, and N. Chaturvedi. Sequence alignment through the looking glass. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 257–266, May 2018.
- [4] Jeffrey A Bailey, Amy M Yavor, Hillary F Massa, Barbara J Trask, and Evan E Eichler. Segmental duplications: organization and impact within the current human genome project assembly. *Genome research*, 11(6):1005–1017, 2001.
- [5] S Batzoglou, L Pachter, JP Mesirov, B Berger, and ES Lander. Human and mouse gene structure: comparative analysis and application to exon prediction. *Genome research*, 10(7):950–958, July 2000.
- [6] Matthias Becker, Hartmut Schultze, Kirk Bresniker, Sharad Singhal, Thomas Ulas, and Joachim L. Schultze. A novel computational architecture for large-scale genomics. *Nature Biotechnology*, Sep 2020.
- [7] Emily Berger, Deniz Yorukoglu, Lillian Zhang, Sarah K. Nyquist, Alex K. Shalek, Manolis Kellis, Ibrahim Numanagić, and Bonnie Berger. Improved haplotype inference by exploiting long-range linking and allelic imbalance in rna-seq datasets. *Nature Communications*, 11(1):4662, Sep 2020.
- [8] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy’s tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOLPS ’09*, pages 18–25, New York, NY, USA, 2009. ACM.
- [9] Nick Bray, Inna Dubchak, and Lior Pachter. Avid: A global alignment program. *Genome research*, 13(1):97–102, 2003.
- [10] Broad Institute. *Picard Tools*. <http://broadinstitute.github.io/picard/>.
- [11] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3), August 2007.
- [12] Peter JA Cock, Tiago Antao, Jeffrey T Chang, Brad A Chapman, Cymon J Cox, Andrew Dalke, Iddo Friedberg, Thomas Hamelryck, Frank Kauff, Bartek Wilczynski, et al. Biopython: freely available python tools for computational molecular biology and bioinformatics. *Bioinformatics*, 25(11):1422–1423, 2009.

- [13] Jeff Daily. Parasail: Simd c library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinformatics*, 17(1):81, 2016.
- [14] Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. Seqan an efficient, generic c++ library for sequence analysis. *BMC Bioinformatics*, 9(1):11, 2008.
- [15] Erwan Drezen, Guillaume Rizk, Rayan Chikhi, Charles Deltel, Claire Lemaitre, Pierre Peterlongo, and Dominique Lavenier. Gatb: Genome assembly & analysis tool box. *Bioinformatics (Oxford, England)*, 30(20):2959–2961, Oct 2014. 24990603[pmid].
- [16] Mark Dufour. Shed skin: An optimizing python-to-c++ compiler. Master's thesis, Delft University of Technology, 2006.
- [17] Michael Farrar. Striped Smith–Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, 11 2006.
- [18] Faraz Hach, Iman Sarrafi, Farhad Hormozdiari, Can Alkan, Evan E. Eichler, and S. Cenk Sahinalp. mrsFAST-Ultra: A compact, SNP-aware mapper for high performance sequencing applications. *Nucleic Acids Research*, 42(W1):W494–W500, 2014. <http://nar.oxfordjournals.org/content/42/W1/W494.abstract>.
- [19] K Hayen. Nuitka, 2012.
- [20] Vladimir Kiriansky, Haoran Xu, Martin Rinard, and Saman Amarasinghe. Cimple: Instruction and memory level parallelism: A dsl for uncovering ilp and mlp. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT '18*, pages 30:1–30:16, New York, NY, USA, 2018. ACM.
- [21] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. Taco: A tool to generate tensor algebra kernels. In *Proc. IEEE/ACM Automated Software Engineering*, pages 943–948. IEEE, 2017.
- [22] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David IW Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M Kaufman, Gurtej Kanwar, Wojciech Matusik, et al. Simit: A language for physical simulation. *ACM Transactions on Graphics (TOG)*, 35(2):20, 2016.
- [23] Johannes Köster. Rust-Bio: a fast and safe bioinformatics library. *Bioinformatics*, 32(3):444–446, 10 2015.
- [24] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proc. Int. Sym. on Code Generation and Optimization*, page 75. IEEE Computer Society, 2004.
- [25] Heng Li. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, 05 2012.
- [26] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem, 2013.

- [27] Heng Li. Minimap2: fast pairwise alignment for long dna sequences. *arXiv preprint arXiv:1708.01492*, 2017.
- [28] Kanak Mahadik, Christopher Wright, Jinyi Zhang, Milind Kulkarni, Saurabh Bagchi, and Somali Chaterji. Sarvavid: A domain specific language for developing scalable computational genomics applications. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 34:1–34:12, New York, NY, USA, 2016. ACM.
- [29] Swati C Manekar and Shailesh R Sathe. A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience*, 7(12), 10 2018. giy125.
- [30] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 01 2011.
- [31] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernysky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark A. DePristo. The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20(9):1297–1303, September 2010.
- [32] Ibrahim Numanagić, Alim S Gökkaya, Lillian Zhang, Bonnie Berger, Can Alkan, and Faraz Hach. Fast characterization of segmental duplications in genome assemblies. *Bioinformatics*, 34(17):i706–i714, 2018.
- [33] Takeshi Ogasawara, Yinhe Cheng, and Tzy-Hwa Kathy Tzeng. Sam2bam: High-performance framework for NGS data preprocessing tools. *PloS one*, 11(11):e0167100, 2016.
- [34] William R Pearson and David J Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.
- [35] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.
- [36] Torbjørn Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation. *BMC Bioinformatics*, 12(1):221, 2011.
- [37] Pamela H. Russell, Rachel L. Johnson, Shreyas Ananthan, Benjamin Harnke, and Nicholas E. Carlson. A large-scale analysis of bioinformatics code on github. *PLOS ONE*, 13(10):1–19, 10 2018.
- [38] Ariya Shajii, Ibrahim Numanagić, Riyadh Baghdadi, Bonnie Berger, and Saman Amarasinghe. Seq: A high-performance language for bioinformatics. *Proc. ACM Program. Lang.*, 3(OOPSLA):125:1–125:29, October 2019.

- [39] Ariya Shajii, Ibrahim Numanagić, Christopher Whelan, and Bonnie Berger. Statistical binning for barcoded reads improves downstream analyses. *Cell Systems*, 7(2):219–226, 2018.
- [40] Tom Sean Smith, Andreas Heger, and Ian Sudbery. Umi-tools: Modelling sequencing errors in unique molecular identifiers to improve quantification accuracy. *Genome Research*, 2017.
- [41] Jason E. Stajich, David Block, Kris Boulez, Steven E. Brenner, Stephen A. Chervitz, Chris Dagdigan, Georg Fuellen, James G. R. Gilbert, Ian Korf, Hilmar Lapp, Heikki Lehväslaiho, Chad Matsalla, Chris J. Mungall, Brian I. Osborne, Matthew R. Pocock, Peter Schattner, Martin Senger, Lincoln D. Stein, Elia Stupka, Mark D. Wilkinson, and Ewan Birney. The bioperl toolkit: Perl modules for the life sciences. *Genome research*, 12(10):1611–1618, Oct 2002. 12368254[pmid].
- [42] Martin Steinegger and Johannes Söding. Mmseqs2 enables sensitive protein sequence searching for the analysis of massive data sets. *Nature Biotechnology*, 35(11):1026–1028, 2017.
- [43] Hajime Suzuki and Masahiro Kasahara. Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC Bioinformatics*, 19(1):45, Feb 2018.
- [44] M. Vasimuddin, S. Misra, H. Li, and S. Aluru. Efficient architecture-aware acceleration of bwa-mem for multicore systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 314–324, May 2019.
- [45] Ben J. Ward. *BioJulia*, accessed November 19, 2020. <https://biojulia.net>.
- [46] Deniz Yorukoglu, Yun William Yu, Jian Peng, and Bonnie Berger. Compressive mapping for next-generation sequencing. *Nat Biotech*, 34(4):374–376, 2016. Opinion and Comment.
- [47] Jing Zhang, Heshan Lin, Pavan Balaji, and Wu-chun Feng. Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 377–384. IEEE, 2013.