Seq: a high-performance language for computational biology

Ibrahim Numanagić
University of Victoria

Matrix Institute for Applied Data Science
1. xi 2019.
Sequencing

DNA / RNA
Sequencing
Sequencing: better than Moore’s law

- The development of next-generation sequencing technologies made sequencing fast and really cheap
- Its growth surpassed Moore’s and Kryder’s laws long ago
- 2019: $100 to sequence a human genome
Computing is the bottleneck

• But the computing did not scale as fast

• The biggest bottleneck in sequencing pipelines today:

computational data analysis

image source: Sboner et al. Genome Biology 2011, 12:125
Reason #1: Enormous scale of data

A single sequencing experiment can generate $\frac{1}{2}$ TB of data nowadays!
Reason #2: Rapid technology changes

- Too many platforms to support
- Soon, most of these will be obsolete anyway…
Outdated development practices

• Two camps of developers depending on the language used:

• **Accessible languages**: popular, easy to develop and easy to understand but too slow (Python, R)
Outdated development practices

- Two camps of developers depending on the language used:
  - **Accessible languages**: popular, easy to develop and easy to understand but too slow (Python, R)
Outdated development practices

- Two camps of developers depending on the language used:

  - **Accessible languages**: popular, easy to develop and easy to understand but too slow (Python, R)

  - **Fast languages**: fast… but hard and cumbersome to develop and maintain (C, C++)

```c
#include KSH_SSE2_ONLY
#ifdef __SSE4_1_
#include <ssem intrinsic.h>
#endif

#ifndef KSH_CPU_DISPATCH
#define __SSE4_1_
void ksw_ext02_sse2(void *km, int qlen, const uint8_t *query, int tlen, const uint8_t *target, int8_t *e, const int8_t *q, int8_t *o, int8_t *d2, int8_t d2, int w, int zq, int end_bonus);
#else
define __SSE4_1_
void ksw_ext02_sse2(void *km, int qlen, const uint8_t *query, int tlen, const uint8_t *target, int8_t *e, const int8_t *q, int8_t *o, int8_t *d2, int8_t d2, int w, int zq, int end_bonus);
#endif

#endif

// -KSH_CPU_DISPATCH

#define __ncode_block1

z = _mm_load_si128((__m128*)t1);
xt1 = _mm_load_si128((__m128*)t1);

// x1 = x[-1](t..t+15) */

tmp = _mm_srl_epi16(xt1, 15);
	/* tmp = x[-1](t+15) */

x1 = tmp;
v11 = _mm_srl_epi16(vt1, 15);

tmp = _mm_srl_epi16(vt1, 15);
	/* tmp = v[-1](t+15) */

v11 = tmp;
u1 = _mm_load_si128((__m128*)t1);

// a = _mm_add_epi8(xt1, vt1);

// ut = _mm_load_si128 neu1);

b = _mm_add_epi8(_mm_load_si128 (vy1), ut1);

x21c = _mm_load_si128 (x21c1);

tmp = _mm_srl_epi16(x21c, 15);

x21c = _mm_srl_epi16(x21c, x21c1, 1), x21c1;

x21c = tmp;

// b = _mm_add_epi8(_mm_load_si128 (vy1), ut1);
```
What do people use?

Fig 3. Number and length of source files by programming language. Languages included in at least 50 main repositories are shown. Each dot corresponds to one repository and indicates the number of files in the language and the mean number of lines of code per file not including comments. The data are provided as Table S8.

Developer communities

For version control systems such as GitHub, a core team of developers with commit access; these developers can push changes directly to the repository. In addition, GitHub facilitates community collaboration through a system of forks and pull requests. Anyone can create a public user, called a fork, and make changes to their fork. If an outside developer feels their changes could benefit the main project, they can create.
Anything bio-specific?

- No language that specifically targets bioinformatics constructs
  - Libraries are either absent or don’t cut it
  - Constant reimplementations needed for each new iteration in sequencing technologies
A typical example of a pipeline

Bash-based DSL (why?) → Python processor

import (  
"encoding/binary"
)  
"ret"

"ret"

"CommaFmt"
  "github.com/bioeg/hits/bam"
  "github.com/bioeg/hits/sam"
)

var {  
//MaxBlock len = 1000000  
MinGapLen = 10000  
// minimum gap required to  
}

fn log_sum_exp(p: Vec<f64>) -> f64{  
let max_p: f64 = p.iter().cloned().fold(f64::NEG_INFINITY, f64::max);  
let sum_rst: f64 = p.iter().map(|x| x - max_p).exp().sum();  
max_p + sum_rst}.ln()}  
}

pub struct SmoothingInfo {  
pub num_states: usize,  
pub num_positions: usize,  
pub forward_prob: Vec<Vec<f64>>,  
pub backward_prob: Vec<Vec<f64>>,  
pub status_prob: Vec<Vec<f64>>, // prob\[x, n | a, 1, e, 2, \ldots, n, 0\]  
pub null_state: usize,  
pub compared_to_second_best: bool,  

All this for simple  
read(dna) | correct | split | align | collect  

then a pinch of Go  
and bit of  

dash of Rust  
(everything is re-implemented  
of course)
A curious example of a pipeline

- **Makefile-driven development**
- whole software package is a collection of Makefiles
- This is *industrial grade* bioinformatics software... that almost nobody uses!
Can we make it better?
A genomics primer: building blocks

- Computational genomics applications use the same set of core operations:
  - String manipulation on a limited alphabet (typically A, C, G, T and N)
  - Frequent genome queries and index lookups
  - Dynamic programming algorithms such as string alignment

[Image source: genome.gov]
A genomics primer: alignment

- Each sequencing machine produces reads: short DNA sequences of size ~100

- **Read mapping**: find the origin location of a read in the reference genome
  1. Split a read into $k$-mers: small fixed length-$k$ subsequences)
  2. Find the occurrences of each $k$-mer in the reference genome by querying the genome index
  3. Run dynamic programming to produce the final alignment
  4. Repeat this for 100,000,000+ reads
A genomics primer: assembly

- **Read assembly**: reconstruct the reference genome from sequenced reads
  - Make a de Bruijn graph from read $k$-mers
    - its edges represent $(k - 1)$-length overlaps between the nodes ($k$-mers)
- **Assembly contig**: an Eulerian path in de Bruijn graph
  - NP-hard due to genome repeats
How about domain-specific language?

• Won’t work!

• Computational biology data has general laws, but…

• … the target computational domain is too general

• Any algorithm or a data structure out there— we have it and we use it

• **Example**: EMA aligner for third-generation barcoded sequencing technologies

• Large file processing (splitting, sorting, I/O)

• Alignment and other pattern matching methods

• Probabilistic methods (EM, simulated annealing)

• Integer linear programming
We need a two-tier approach

- **Top-down (high-level):** describe the problem intuitively without thinking about optimizations

- **Bottom-up (low-level):** implement high-performant and scalable components
understand genomics

general purpose

fast and scalable

easy & rapid development
understand genomics

general purpose

fast and scalable

easy & rapid development

Seq
Seq: a language for computational biology

- Our approach: **Seq**, a general language with a host of genomics-related features and optimizations
A critical barrier to any new language’s success in a particular domain is its initial adoption, as most programmers second. For this reason, the Seq language borrows the syntax and semantics of Python—one of the most widely-used languages in bioinformatics—and adds several genomics-specific features.

To achieve this, we designed a compiler with a static type system. It performs Python-style duck typing and runtime type checking at compile time, completely eliminating the substantial runtime overhead imposed by the reference Python implementation, CPython, and most other Python implementations alike. Unlike these, we reimplemented all of Python’s language features used within the genomics community will compile and run without modification.

```
#include <iostream>
#include <fstream>
#include <string>
#include <stdlib>
#include "GenomeIndex.h"

char revcomp(char base) {
    switch (base) {
        case 'A': return 'T';
        case 'C': return 'G';
        case 'G': return 'C';
        case 'T': return 'A';
        default: return base;
    }
}

void revcomp(char *kmer, int k) {
    for (int i = 0; i < k/2; i++) {
        char a = revcomp(kmer[i]);
        char b = revcomp(kmer[k - i - 1]);
        kmer[i] = b;
        kmer[k - i - 1] = a;
    }
}

int main(int argc, char *argv[]) {
    const int k = 20;
    const int stride = 10;
    auto *index = GenomeIndex(argv[1], k);
    std::ifstream fin(argv[2]);
    std::string read;
    long line = -1;
    while (std::getline(fin, read)) {
        line++;
        // skip over non-sequences in FASTQ
        if (line % 4 == 1) continue;
        auto *buf = (char *)read.c_str();
        int len = read.size();
        for (int i = 0; i + k <= len; i += stride)
            process(kmer, k, index);
    }
    return 0;
}
```

```
from sys import argv
from genomeindex import *
type K = Kmer[20]

# index and process 20-mers
def process(kmer: K, index: GenomeIndex[K]):
    prefetch index[kmer], index[-kmer]
    hits_fwd = index[kmer]
    hits_rev = index[-kmer]
    ...

# index over 20-mers
index = GenomeIndex[K](argv[1])

# stride for k-merization
stride = 10

# sequence-processing pipeline
(fastq(argv[2])
    |> kmers[K](stride)
    |> process(index))
```
Implementation: a strongly-typed Python

• Python is duck-typed dynamic language that completely relies on runtime

• We want the syntax and clarity of Python with none of the runtime overhead

  • Most of the Python’s overhead stems from its dynamic runtime capabilities that are rarely, if at all, used in genomics pipelines

• Solution: strongly typed language with Python syntax that does everything at compile time
Implementation: a strongly-typed Python

CPython

LOAD_CONST 3.14
STORE_FAST x

x = 3.14

Seq

x: float = 3.14

%x = alloca double, align 8
...
store double 3.140000e+00, double* %x

typedef struct {
    struct _object *_ob_next;
    struct _object *_ob_prev;
    Py_ssize_t ob_refcnt;
    struct _typeobject *ob_type;
} PyObject;
...

typedef struct {
    PyObject ob_base;
    double ob_fval;
} PyFloatObject;
...

x = (PyFloatObject){.ob_fval = 3.14, ...};
Implementation: a strongly-typed Python

```
def f(x):
    return 3*x + 1
```

```
def f[T](x: T):
    return 3*x + 1
```

```
def f(x: int) -> int:
    return 3*x + 1
```

```
def f(x: float) -> float:
    return 3*x + 1
```

```
def f(x: int) -> int:
    return int.__add__(
        int.__mul__(3, x), 1)
```

```
def f(x: float) -> float:
    return float.__add__(
        float.__mul__(3, x), 1)
```

Fig. 5. Seq's implicit generic type parameters. The function `f` is declared to take a parameter `x` of unspecified type; the Seq compiler treats the type of `x` as generic and clones `f` on demand for each new input type, and subsequently deduces return types.

Duck typing reasonably well. Explicit type annotations enforce an extra layer of typing discipline (à la `mypy`), and as such coexist peacefully with it.

Type Inference. Any strongly typed language needs a way to infer the type of each variable present in a given program. Languages such as C or Pascal require end users to manually annotate each variable with a type. Other languages, such as C++ or newer versions of Java, support uni-directional type inference by automatically deducing types of left-hand side terms based on right-hand side types. Initial versions of Seq also used uni-directional type inference, allowing users to say, for instance, `x=5` instead of `x: int = 5`. However, uni-directional type inference is unable to handle a few common constructs in the Python language, including empty lists (e.g. `a = []`), nullables (e.g. `a = None`) and lambda functions (e.g. `lambda x: x+1`). With uni-directional inference, each of these constructs requires the user to provide manual type annotations (e.g. `a: list[int] = []`) even if the type can be inferred later.

Because of this, Seq uses bi-directional type inference, implemented on top of the Hindley-Milner inference algorithm, to automatically annotate such types. We slightly modified the standard Hindley-Milner algorithm to support generic classes, functions and instantiations on demand. We also enforce an invariant where all types within a scope (be it a function scope, class scope or the top-level scope) must be fully deduced by the end of that scope. This implies that a function cannot return a non-instantiated generic type: `def f(): return []` will cause a compilation error, but `def f[T]() -> list[T]: return []` will compile successfully. Any weakly typed variable or lambda is instantiated as soon as possible (note that Seq treats lambdas as weakly typed constructs and does not generalize them—generalizations are only applied to generic functions defined with `def` and generic classes).

Limitations. The strongly-typed nature of Seq does come with some limitations compared to conventional Python. Since all types must be fixed at compile time, a Seq program cannot (for example) create a collection of elements (e.g. `list`) with varying types. Seq's tuples are also less versatile than Python's: they cannot be iterated over if they contain different types, and a list cannot be cast to a tuple easily, as tuple sizes must be known at compile time. Seq also does not support method or class monkey-patching at runtime (but it does support this at compile time—see 2 This is a recent addition to Seq, and is currently still in the testing stage. At the time of writing, Seq's master branch uses uni-directional type deduction with added support for generics and type-less nullables. Note that this uni-directional version does not support lambdas. }


Implementation: a strongly-typed Python
Implementation: a strongly-typed Python

```python
class Node[T]:
    next: Node[T]
    data: T

def item[T,U](n: Node[T], f: function[T,U]) -> list[U]:
    return [f(n.data)]

n = Node(None, 5)
def foo(x: int) -> str:
    return str(x)
i = item(n, foo)  # type parameters deduced as int and str
i = item[int,str](n, foo)  # explicit specification also OK
```

- Explicit (but optional) generics for easier type inference
- Hindley-Milner inference is being merged into Seq
Implementation: bootstrapping

- The standard library and most Pythonic constructs are bootstrapped directly in `Seq`
- `list[T]` or `dict[K,V]`: all implemented in `Seq`
- all functions (map, zip, sort etc.) are implemented in `Seq`
Differences with Python

- Functions can only return objects of a single type

- `[42, 3.14, "hello"]`
  Collections cannot contain objects of different types

- `obj.method = new_method`
  Methods of an object cannot be modified at runtime (possible at compile time though!)

- `(1, 2.2)[idx]`
  Tuple indices must be constants, and iteration over a tuple is allowed only if its elements all have the same type

- No inheritance nor polymorphism (partially alleviated by instantiation)

- `if cond():
  x = 1
else:
  x = 2
print x`
  Stricter scoping rules than Python

- Other temporary restrictions (lambda, for-else, empty literals etc.)
Coroutines and generators

- Generators: extremely important feature of Python
  - for a in range(3) iterates over the generator range(3)
- LLVM supports coroutines!
  - Allows us to implement Python generators with virtually no overhead thanks to coroutine passes and inlining
  - Enables efficient pipelining and laziness

C++

```cpp
for (i = 0; i < 3; i++)
print(i);
```

Seq

```python
for i in range(3):
print i
```

```
entry:
%g = call i8* @range(i64 3)
br label %for
for:
call void @llvm.coro.resume(i8* %g)
%done = call i1 @llvm.coro.done(i8* %g)
br i1 %done, label %exit, label %body
body:
%p0 = call i8* @llvm.coro.promise(i8* %g, i32 8, i1 false)
%p1 = bitcast i8* %p0 to i64*
%i = load i64, i64* %p1
call void @print(i32 %i)
br label %for
exit:
call void @llvm.coro.destroy(i8* %g)
```

Fig. 6. Compilation of Seq generators. Two semantically identical loops in C++ and Seq are shown in the uppermost boxes. Seq generators are implemented as LLVM coroutines, iteration over which in LLVM IR is shown in the middle box. The LLVM coroutine passes subsequently deduce that the "range" coroutine is created and destroyed in the same function without escaping, and inline/unroll the coroutine to produce code identical to the C++ example’s.
New features

- Pipelines
- Genomics types
- Pattern matching
- C/C++ interop
- Type extensions
- Prefetching
New features: pipelines

Pipelines

Genomics types
Pattern matching
C/C++ interop
Type extensions
Prefetching

```
 dna = s'ACGTACGTACGT'  # sequence literal

# (a) split into subsequences of length 3
#    with a stride of 2
 dna |> split(...., 3, 2) |> echo

# (b) split into 5-mers with stride 1
 def f(kmer):
   print kmer
   print ~kmer

dna |> kmers[Kmer[5]](1) |> f
```

```
 dna = s'ACGTACGTACGT'  # sequence literal

# (a) split into subsequences of length 3
#    with a stride of 2
 dna |> split(...., 3, 2) ||> echo

# (b) split into 5-mers with stride 1
 def f(kmer):
   print kmer
   print ~kmer

dna |> kmers[Kmer[5]](1) ||> f
```

A single character difference!
New features: genomics types

- Sequence and k-mer types
- Various compile-time optimizations:
  - 2-bit sequence encoding
  - Low-level lookup table for fast reverse complementation
  - Avoid copying and allocations unless really necessary

```python
import std

# (a) split into subsequences of length 3 with a stride of 2
for sub in dna.split(3, 2):
    print sub

# (b) split into 5-mers with stride 1
for kmer in dna.kmers[Kmer[5]](1):
    print kmer
    print ~kmer  # reverse complement

# (c) convert entire sequence to 12-mer
kmer = Kmer[12](dna)
```
New features: pattern matching

def describe(n: int):
    match n:
        case m if m < 0:
            print 'negative'
        case 0:
            print 'zero'
        case m if 0 < m < 10:
            print 'small'
        default:
            print 'large'
New features: pattern matching

# (a)
def has_spaced_acgt(s: seq) -> bool:
m = len(s)
match s:
case s'A...T' or s'T...A' or s'C...G' or s'G...C':
    return has_spaced_acgt(s[1:-1])
case s'':
    return True
def is_own_revcomp(s: seq) -> bool:
    match s:
case s'A...T' or s'T...A' or s'C...G' or s'G...C':
    return is own_revcomp(s[1:-1])
case s'':
    return True
default:
    return False

def count_bases(s: seq) -> BaseCount:
m = len(s)
match s:
case s'A...':
    return count_bases(s[1:]) + (1, 0, 0, 0)
case s'C...':
    return count_bases(s[1:]) + (0, 1, 0, 0)
case s'G...':
    return count_bases(s[1:]) + (0, 0, 1, 0)
case s'T...':
    return count_bases(s[1:]) + (0, 0, 0, 1)
def __add__(self, other: BaseCount):
a1, c1, g1, t1 = self
a2, c2, g2, t2 = other
return (a1 + a2, c1 + c2, g1 + g2, t1 + t2)
New features: C interop

- Python and R interop capabilities (pydef and rdef) are coming soon as well.
New features: type extensions

```python
extend int:
    def to(self: int, other: int):
        for i in range(self, other + 1):
            yield i

    def __mul__(self: int, other: int):
        print 'caught int mul!'
        return 42

for i in (5).to(10):
    print i  # 5, 6, ..., 10

# prints 'caught int mul!' then '42'
print 2 * 3
```
New features: prefetching

- In many popular genomics tools, more than 50% of time is wasted on stalling
New features: prefetching

- Seq supports dynamic prefetching for faster index queries (\textit{a la} Cimple\textsuperscript{1})
  1. Just add a single \texttt{prefetch} statement to indicate prefetching
  2. Make sure that index exposes \texttt{__prefetch__} magic

\begin{verbatim}
class MyIndex:  # abstract k-mer index
    ...  
def __getitem__(self: MyIndex, kmer: Kmer[20]):  # standard __getitem__
def __prefetch__(self: MyIndex, kmer: Kmer[20]):  # similar to __getitem__, but performs prefetch

type k20 = Kmer[20]
def process(read: seq, index: MyIndex):
    ...
    for kmer in read.kmers[k20](step):
        prefetch index[kmer], index[~kmer]
hits = index[kmer]
hits_rev = index[~kmer]
    ...
    return x
\end{verbatim}

[1] Kiriansky et al. ACM PACT 2018
New features: prefetching

The function surrounding prefetch is automatically converted to a generator

def process(read: seq, index: MyIndex):
    ...
    for kmer in read.kmers[k20](step):
        prefetch index[kmer], index[^kmer]
        hits = index[kmer]
        hits_rev = index[^kmer]
        ...
    return x
New features: prefetching

- The pipeline that uses a prefetch function is also transformed to allow suspension
- Only a single line modification
- Results in up to 50% runtime improvements over the baseline
Benchmarks

1. Computer Language Benchmarks Game
   i. fasta (60 LOC, 2 min)
   ii. revcomp (35 LOC, 2 min)
   iii. knucleotide (40 LOC, 4 min)

2. In-house suite (100 million reads)
   i. rc (25 LOC, 35 min)
   ii. 16mer (35 LOC, 4+ hrs)
   iii. cpg (40 LOC, 1 hr)

3. Genome index suite
   i. snap (70 LOC, 8 min):
      query k-mer-based genome index
   ii. sga (100 LOC, 9 min):
      query FM-based genome index
Benchmarks

- **Python** is the reference implementation
Benchmarks

- **Python** is the reference implementation
- Compiled and JIT Pythons, such as **Nuitka**, **Shedskin** and **PyPy** help a bit

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Speed-up (times, ×) over Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc</td>
<td>2.36, 2.37, 9.34</td>
</tr>
<tr>
<td>cpg</td>
<td>2.25, 2.69, 17.63</td>
</tr>
<tr>
<td>16mer</td>
<td>2.56, 7.22, 33.32</td>
</tr>
<tr>
<td>fasta</td>
<td>1.1, 7.13, 1.62</td>
</tr>
<tr>
<td>revcomp</td>
<td>1.8, 6.09, 6.31</td>
</tr>
<tr>
<td>knucleotide</td>
<td>1.47, 1.2, 4.32</td>
</tr>
</tbody>
</table>
Benchmarks

- **Python** is the reference implementation
- Compiled and JIT Pythons, such as **Nuitka**, **Shedskin** and **PyPy** help a bit
- **Julia** is similar…
Benchmarks

• **Python** is the reference implementation

• Compiled and JIT Pythons, such as **Nuitka**, **Shedskin** and **PyPy** help a bit

• **Julia** is similar…
Benchmarks

- **Python** is the reference implementation
- Compiled and JIT Pythons, such as
  - Nuitka, Shedskin and
  - PyPy help a bit
- **Julia** is similar…
- … but none come close to
  - **Seq**
- up to **160x speed-ups** over
  - **Python**
- or 1m 30s vs 4 hours and counting
Benchmarks

- When compared to the reference ▶ Clang++ implementation…
- g++ is similar to Clang++
- (this is ▶ Python)
Benchmarks

• When compared to the reference \underline{Clang++} implementation…

• \underline{g++} is similar to \underline{Clang++}

• (this is \underline{Python})
Benchmarks

- When compared to the reference Clang++ implementation…
- g++ is similar to Clang++
- (this is Python)
- and Seq outperforms even C++ in most experiments
- up to 7× speed-ups over C++
Benchmarks

- Finally, on genome index query benchmarks...
- reference **Clang++** implementation and **g++** are similar...

<table>
<thead>
<tr>
<th>qtp</th>
<th>Speed-up (times, x) over <strong>Clang++</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1.376</td>
<td></td>
</tr>
<tr>
<td>0.933</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sga</th>
<th>Speed-up (times, x) over <strong>Clang++</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0.933</td>
<td></td>
</tr>
</tbody>
</table>
Benchmarks

- Finally, on genome index query benchmarks...
- reference ■ Clang++ implementation and ■ g++ are similar...
- ■ Seq can improve runtime up to 25%
Benchmarks

- Finally, on genome index query benchmarks...

- reference ■ Clang++ implementation and ■ g++ are similar...

- ■ Seq can improve runtime up to 25%

- and ■ Seq with prefetch boosts the performance of ■ Seq up to 50%!
Seq: comparison with C++

- Re-implementation of homology table generator from CORA mapping software
- Ran on the whole human genome
- Highly optimized C++: LOC: 1,346 (27 screens) Runtime: 3+ hrs
Seq: comparison with C++

- Re-implementation of homology table generator from CORA mapping software
- Ran on the whole human genome
- Highly optimized C++:
  LOC: 1,346 (27 screens)
  Runtime: 3+ hrs

- Seq:
  LOC: 126 (2½ screens — > 10× smaller)
  Runtime: ~30 minutes (> 6× faster)
Seq: summary

• A Python-based language for computational genomics
  • Speed of C
  • Ease and expressiveness of Python
  • Compile-time type checking
  • Genomic-related optimizations
  • Natural pipeline syntax and many other enhancements
Acknowledgements

MIT:
- Ariya Shajii
- Bonnie Berger
- Riyadh Baghdadi
- Saman Amarasinghe
- Hyunghoon Cho
- Alexander Leighton
- Lorenzo Di Tucci

- William Leiserson
- Fredrik Kjolstad
- Tao Schardl

University of Victoria:
- Jordan Watson
- Jodie Weldon

MIT Computation and Biology Group
MIT Commit Compiler Group
University of Victoria Lab 0xTCG

Get Seq at
https://seq-lang.org

Check the development at Github

Read the paper at ACM DL
Thank you!

Questions, and (hopefully) some answers
Related work

Table 5. Comparison between Seq and other Python implementations. For “Domain”, “Bio.” means computational biology, “Sci.” means scientific computing and “Astro.” means astrophysical computing. “Unknown types” refer to types that cannot be statically determined. Also note that Pyston has several JIT tiers in addition to its LLVM JIT.

<table>
<thead>
<tr>
<th></th>
<th>Domain</th>
<th>Target</th>
<th>Compilation</th>
<th>Unknown types allowed?</th>
<th>Full Python?</th>
<th>CPython runtime?</th>
<th>Multi-threading?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPython</td>
<td>General</td>
<td>Bytecode</td>
<td>Interpreted</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Seq</td>
<td>Bio.</td>
<td>LLVM IR</td>
<td>AOT</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Cython</td>
<td>General</td>
<td>C</td>
<td>AOT</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PyPy</td>
<td>General</td>
<td>Bytecode</td>
<td>Interpreted</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Numba</td>
<td>Sci.</td>
<td>LLVM IR</td>
<td>JIT</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Nuitka</td>
<td>General</td>
<td>C++</td>
<td>AOT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Pythran</td>
<td>Sci.</td>
<td>C++</td>
<td>AOT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Pyston</td>
<td>General</td>
<td>LLVM IR</td>
<td>JIT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>HOPE</td>
<td>Astro.</td>
<td>C++</td>
<td>JIT</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Shed Skin</td>
<td>General</td>
<td>C++</td>
<td>AOT</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Grumpy</td>
<td>General</td>
<td>Go</td>
<td>AOT</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
</tbody>
</table>

Seq is the only Python-alike language that consistently matches the performance of C/C++