Fast high-level programming languages

17 May 2020

Background

Python and R are slow when they can't rely on functionality or libraries backed by C/C++. They are inefficient not only for certain algorithm development but also for common tasks such as FASTQ parsing. Using these languages limits the reach of biologists. Sometimes you may have a brilliant idea but can't deliver a fast implementation only because of the language in use. This can be frustrating. I have always been searching for a high-level language (https://en.wikipedia.org/wiki/High-level_programming_language) that is fast and easy to use by biologists. This blog post reports some of my exploration. It is inconclusive but might still interest you.

Design

Here I am implementing two tasks, FASTQ parsing and interval overlap query, in several languages including C, Python, Javascript, LuaJIT (http://luajit.org/), Julia (https://en.wikipedia.org/wiki/Julia_(programming_language)), Nim (https://en.wikipedia.org/wiki/Nim_(programming_language)), and Crystal (https://en.wikipedia.org/wiki/Crystal_(programming_language)). I am comparing their performance. I am proficient in C and know Python a little. I have used LuaJIT and Javascript for a few years. I am equally new to Julia, Nim and Crystal. My implementations in these languages may not be optimal. Please keep this important note in mind when reading the results.

Results

The source code and the full table are available at my lh3/biofast (https://github.com/lh3/biofast) github repo. You can also found the machine setup, versions of libraries in use and some technical notes. I will only show part of the results here.

FASTQ parsing

The following table shows the CPU time in seconds for parsing a gzip'd FASTQ (t_{gzip}) or a plain FASTQ (t_{plain}). We only count the number of sequences and compute the sum of lengths. Those seamlessly parsing multi-line FASTA/FASTQ all use an algorithm similar to my kseq.h (https://github.com/lh3/biofast/blob/master/lib/kseq.h) parser in C.

Language	Ext. Library	t_{gzip} (s)	t_{plain} (s)	Comments
Rust	needletail	9.3	0.8	multi-line fasta/mostly 4-line fastq
C		9.7	1.4	multi-line fasta/fastq
Crystal		9.7	1.5	multi-line fasta/fastq
Nim		10.5	2.3	multi-line fasta/fastq
Julia		11.2	2.9	multi-line fasta/fastq
Python	PyFastx	15.8	7.3	C binding
Javascript		17.5	9.4	multi-line fasta/fastq; k8 dialect
Go		19.1	2.8	4-line fastq only

LuaJIT		28.6	27.2	multi-line fasta/fastq
PyPy		28.9	14.6	multi-line fasta/fastq
Python	BioPython	37.9	18.1	multi-line fastq; FastqGeneralIterator
Python		42.7	19.1	multi-line fasta/fastq
Python	BioPython	135.8	107.1	multi-line fastq; SeqIO.parse

This benchmark stresses on I/O and string processing. I replaced the low-level I/O of several languages to achieve good performance. The code looks more like C than a high-level language, but at least these langauges give me the power without resorting to C.

It is worth mentioning the default BioPython FASTQ parser is over 70 times slower on plain FASTQ and over 10 times slower on gzip'd FASTQ. Running the C implementation on a human 30X gzip'd FASTQ takes 20 minutes. The default BioPython parser will take four and half hours, comparable to bwa-mem2 multi-thread mapping. If you want to parse FASTQ but doesn't need other BioPython functionality, choose PyFastx (https://github.com/lmdu/pyfastx) or mappy.

Interval overlap query

The following table shows the CPU time in seconds for computing the breadth of coverage of a list intervals compared against another interval list. There are two columns for timing and memory footprint, depending on which list is loaded into memory.

Language	t_{g2r} (s)	M_{g2r} (Mb)	t_{r2g} (s)	M_{r2g} (Mb)
C	5.2	138.4	10.7	19.1
Crystal	8.8	319.6	14.8	40.1
Nim	16.6	248.4	26.0	34.1
Julia	25.9	428.1	63.0	257.0
Go	34.0	318.9	21.8	47.3
Javascript	76.4	2219.9	80.0	316.8
LuaJIT	174.1	2668.0	217.6	364.6
PyPy	17332.9	1594.3	5481.2	256.8
Python	>33770.4	2317.6	>20722.0	313.7

The implementation of this algorithm is straightforward. It is mostly about random access to large arrays. Javascript and LuaJIT are much slower here because I can't put objects in an array; I can only put references to objects in an array.

My take on fast high-level languages

The following is subjective and can be controversial, but I need to speak it out. Performance is not everything. Some subtle but important details are only apparent to those who write these programs.

Javascript and LuaJIT

These are two similar languages. They are old and were not designed with Just-In-Time (https://en.wikipedia.org/wiki/Just-in-time_compilation) (JIT) compilation in mind. People later developed JIT compilers and made them much faster. I like the two languages. They are easy to use, have few performance pitfalls and are pretty fast. Nonetheless, they are not the right languages for bioinformatics. If they were, they would have prevailed years ago.

Julia

Among the three more modern languages Julia, Nim and Crystal, Julia reached 1.0 first. I think Julia could be a decent replacement of Matlab or R by the language itself. If you like the experience of Matlab or R, you may like Julia. It has builtin matrix support, 1-based coordinate system, friendly REPL (https://en.wikipedia.org/wiki/Read-eval-print_loop) and an emphasis on plotting as well. I heard its differential equation solver might be the best across all languages.

I don't see Julia a good replacement of Python. Julia has a long startup time. When you use a large package like Bio.jl, Julia may take 30 seconds to compile the code, longer than the actual running time of your scripts. You may not feel it is fast in practice. Actually in my benchmark, Julia is not really as fast as other languages, either. Probably my Julia implementations here will get most slaps. I have seen quite a few you-are-holding-the-phone-wrong type of responses from Julia supporters. Also importantly, the Julia developers do not value backward compatibility. There may be a python2-to-3 like transition in several years if they still hold their views by then. I wouldn't take the risk.

Nim

Nim reached its maturity in September 2019. Its syntax is similar to python on the surface, which is a plus. It is relatively easier to get descent performance out of Nim. I have probably spent least time on learning Nim but I can write programs faster than in Julia.

On the down side, writing Nim programs feels a little like writing Perl in that I need to pay extra attention to reference vs value. For the second task, my initial implementation was several times slower than the Javascript one, which is unexpected. Even in the current program, I still don't understand why the performance get much worse if I change by-reference to by-value in one instance. Nim supporters advised me to run a profiler. I am not sure biologists would enjoy that.

Crystal

Crystal is a pleasant surprise. On the second benchmark, I got a fast implementation on my first try. I did take a detour on FASTQ parsing when I initially tried to use Crystal's builtin buffered reader, but again I got C-like performance immediately after I started to manage buffers by myself.

Crystal resembles Ruby a lot. It has very similar syntax, including a class/mixin (https://en.wikipedia.org/wiki/Mixin) system familiar to modern programmers. Some elementary tutorials on Ruby are even applicable to Crystal. I think building on top of successful languages is the right way to design a new language. Julia on the other hand feels different from most mainstream languages like C++ and Python. Some of its key features haven't stood the test of time and may become frequent sources of bugs and performance traps.

To implement fast programs, we need to care about reference vs value. Crystal is no different. The good thing about Crystal is that reference and value are explicit with its class system. Among Julia, Nim and Crystal, I feel most comfortable with Crystal.

Crystal is not without problems. First, it is hard to install Crystal without the root permission. I am providing a portable installation binary package in lh3/PortableCrystal (https://github.com/lh3/PortableCrystal). It alleviates the issue for now. Second, Crystal is unstable. Each release introduces multiple breaking changes. Your code written today may not work later. Nonetheless, my program seems not affected by breaking changes in the past two years. This has given me some confidence. The Crystal devs also said 1.0 is coming "in the near future (https://crystal-lang.org/2020/03/03/towards-crystal-1.0.html)". I will look forward to that.

Conclusions

A good high-level high-performance programming language would be a blessing to the field of bioinformatics. It could extend the reach of biologists, shorten the development time for experienced programmers and save the running time of numerous python scripts by many folds. However, no languages are good enough in my opinion. I will see how Crystal turns out. It has potentials.

Anecdote

Someone posted this blog post to Hacker New (https://news.ycombinator.com/item?id=23229657), Crystal subreddit (https://www.reddit.com/r/crystal_programming/comments/gm2dps/crystal_in_bioinformatics_comparison_fast/), and Julia discourse (https://discourse.julialang.org/t/lhe-biofast-benchmark-fastq-parsing-julia-nim-crystal-python/39747). The reaction from many Julia supporters is just as I expected. That said, I owe a debt of gratitude to Kenta Sato (https://github.com/bicycle1885) for improving my Julia implementation. I geniunely appreciate.

Update on 2020-05-19: Added contributed Go implementations. More accurate timing for fast implementations, measured by hyperfine (https://github.com/sharkdp/hyperfine).

Update on 2020-05-20: Added a contributed Rust implementation. Added PyPy.

Update on 2020-05-21: Faster Nim and Julia with memchr (http://man7.org/linux/man-pages/man3/memchr.3.html). Faster Julia by adjusting three additional lines (https://github.com/lh3/biofast/pull/7). For gzip'd input, Julia-1.4.1 is slow due to a misconfiguration (https://github.com/JuliaPackaging/Yggdrasil/pull/1051) on the Julia end. The numbers shown in the table are acquired by forcing Julia to use the system zlib on CentOS7. Added Python bedcov implementation. It is slow.

Update on 2020-05-23: Added a faster contributed Rust implementation.