Comparing Ease of Programming in C++, Go, and Java for Implementing a Next-Generation Sequencing Tool

Evolutionary Bioinformatics
Volume 15: 1–5
© The Author(s) 2019
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/1176934319869015
SAGE

Pascal Costanza*, Charlotte Herzeel* and Wilfried Verachtert

ExaScience Lab, IMEC vzw, Leuven, Belgium.

ABSTRACT: elPrep is an extensible multithreaded software framework for efficiently processing Sequence Alignment/Map (SAM)/Binary Alignment/Map (BAM) files in next-generation sequencing pipelines. Similar to other SAM/BAM tools, a key challenge in elPrep is memory management, as such programs need to manipulate large amounts of data. We therefore investigated 3 programming languages with support for assisted or automated memory management for implementing elPrep, namely C++, Go, and Java. We implemented a nontrivial subset of elPrep in all 3 programming languages and compared them by benchmarking their runtime performance and memory use to determine the best language in terms of computational performance. In a previous article, we motivated why, based on these results, we eventually selected Go as our implementation language. In this article, we discuss the difficulty of achieving the best performance in each language in terms of programming language constructs and standard library support. While benchmarks are easy to objectively measure and evaluate, this is less obvious for assessing ease of programming. However, because we expect elPrep to be regularly modified and extended, this is an equally important aspect. We illustrate representative examples of challenges in all 3 languages, and give our opinion why we think that Go is a reasonable choice also in this light.

KEYWORDS: Next-generation sequencing, sequence analysis, SAM/BAM files, C++, Go, Java

RECEIVED: June 27, 2019. **ACCEPTED:** July 3, 2019.

TYPE: Commentary

FUNDING: The author(s) received no financial support for the research, authorship, and/or publication of this article.

DECLARATION OF CONFLICTING INTERESTS: The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

CORRESPONDING AUTHOR: Pascal Costanza, ExaScience Lab, IMEC vzw, Kapeldreef 75, 3001 Leuven, Belgium. Email: pascal.costanza@imec.be

COMMENT ON: Costanza P, Herzeel C, Verachtert W. A comparison of three programming languages for a full-fledged next generation sequencing tool. *BMC Bioinformatics*. 2019;20(1):301. doi:10.1186/s12859-019-2903-5. PubMed PMID: 31159721. PubMed Central PMCID: PMC6547519. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6547519/.

Introduction

elPrep is an open-source, multithreaded software tool for processing Sequence Alignment/Map (SAM)/Binary Alignment/Map (BAM) files to efficiently execute the time-consuming phases of typical next-generation sequencing pipelines.[1] It can be used as a drop-in replacement for many of the tools provided by GATK, Picard, and SAMtools, producing identical results, albeit considerably faster. elPrep has been designed with performance and extensibility in mind. It has a unique software architecture which allows combining the processing of multiple pipeline steps into a single program run, whereas the standard approach is to spread out the steps over several tool invocations. Our approach allows us to merge and parallelize the resulting computations, which we have shown to be significantly more efficient at reducing the overall runtime of a pipeline compared with optimizing individual steps in isolation.[2]

For example, elPrep executes a 4-step pipeline from the Broad Best Practices (sorting, marking duplicates, base quality score recalibration and application) up to 7.4× faster for whole-genome data, and up to 13× faster for whole-exome data while needing fewer compute resources compared with using GATK4.[1]

To achieve this level of efficiency, we faced multiple performance challenges when developing elPrep. For example, a major effort went into developing more efficient algorithms for particular pipeline steps (duplicate marking, base quality score recalibration, etc) that still produce identical results compared with their reference implementations in GATK, Picard, and SAMtools.[1,2] This included introducing parallelization into these computations in such a way that their calculations can be combined to maximize CPU usage.

In a recent article, we showed that memory management is another major performance bottleneck when implementing a sequencing tool such as elPrep.[3] In general, sequencing software needs to manipulate large amounts of data as SAM/BAM files are in the range of hundreds of gigabytes of data. elPrep additionally tries to keep as much data as possible in main memory while processing multiple pipeline steps. This allows elPrep to avoid unnecessary file I/O and eliminates synchronization bottlenecks for parallelization, which are both key reasons for elPrep's efficiency.[2]

Manual memory management is too complex when designing an open-ended software framework that has to be both efficient and extensible at the same time. We therefore decided to evaluate several programming languages in terms of their support for assisted or automated memory management.[3] We narrowed down the candidates to 3 languages: C++ because of its support for safe reference counting, and Go and Java, because of their support for concurrent, parallel garbage collection. Other languages were discarded early on because they were missing other features we needed to implement elPrep, for example, specific support for synchronization between threads.[3]

We implemented a nontrivial subset of elPrep in all 3 languages, and careful benchmarking reveals that Go yields the

* Pascal Costanza and Charlotte Herzeel contributed equally.

best balance between runtime performance and peak memory use.[3] Briefly, our benchmarks show that the Java version runs slightly faster than the Go version, but uses significantly more memory, and that the C++ version runs significantly slower than both the Go and Java versions. Because of these objectively quantifiable performance measurements, we decided to base elPrep since version 3 on Go.[1]

However, runtime performance and peak memory use were not the only important aspects for choosing a programming language for a project such as elPrep. elPrep is designed as an open-ended framework which we expect to be regularly modified and extended.[2] Therefore, an essential question for us when choosing a programming language is to determine how easy it is to express effective solutions to particular programming tasks, whether related to performance, or other programming problems. Even if a particular programming language would allow us to achieve the best runtime performance and the smallest memory footprint, it may simply be too much effort to reach that goal.

In the rest of this article, we present some of the challenges we encountered while implementing elPrep and illustrate how easy or difficult it was to solve them by focusing on how to design data structures to represent the contents of the SAM/BAM file format in each language. The SAM file format has a number of characteristics, like character encoding, size of particular entries, and representation of optional information, which requires careful data structure design to efficiently deal with them. Different programming languages have different strengths and weaknesses to support such data structures. Although ease of programming is much harder to objectively assess, we argue that Go, which we have shown to yield the best performance,[3] also has excellent language support to implement a sequencing tool, and is therefore a good choice as a basis for elPrep also in this light.

Implications of the SAM File Format

The sequence alignment map (SAM) format is the de facto standard for representing aligned sequencing data.[4] The format is specified in a community-maintained reference.[5]

The text format consists of a (relatively small) number of header lines, followed by a (typically large) number of lines each representing one read and its alignment to a reference sequence. The header lines contain meta-information, like format version number, sorting/grouping order, program information, and comments; and common data that can be referenced in alignments, like reference sequence and read group information.

The text format is based on ASCII encoding (with a few places allowing for UTF-8 representation for purely descriptive purposes). When elPrep reads a SAM file into main memory, it has to find a good internal representation for its contents. Many entries in a SAM file can be converted to primitive data types (like integer and floating point) which are well supported in most programming languages, including C++, Go, and Java. Some entries remain as text strings, including among others the sequence read itself and the associated qualities per base pair.

As the header of a SAM file is typically only on the order of a couple of dozen text lines, it is not important to find particularly efficient representations for their contents. On the contrary, the number of alignment lines is significantly larger, so representing their contents efficiently is very important. Each alignment line consists of 11 required entries separated by tabulators, followed by an arbitary number of additional optional entries also separated by tabulators. The required entries are as follows:

1. The unique name for the read (string).
2. A flag field indicating some characteristics (integer).
3. The name of a reference sequence (string).
4. The mapping position (integer).
5. The mapping quality (integer).
6. The CIGAR string.
7. The mate's reference sequence (string).
8. The mate's mapping position (integer).
9. The length of the alignment (integer).
10. The sequence read (string).
11. The sequence qualities (string).

An optional entry consists of a mnemonic identifier (2 characters), a type (1 character), and the entry value, which can be a character, an integer, a floating point number, a string, a byte array, or a numeric array, depending on the previous type character.

As the last 2 required entries, ie, the sequence reads and qualities, make up a large part of a SAM file, it is important that strings have a memory-efficient representation in main memory. With C++ and Go, and with current versions of Java, this is not a issue, because ASCII strings are represented in all 3 languages in a way that uses 1 byte per character in a string. (Before Java JDK 9, all strings in Java were represented using 2 bytes per character, to allow for representing extended character sets like Unicode. However, since Java JDK 9, the Java runtime dynamically recognizes *compact* strings that can be represented more efficiently with just 1 byte per character. Java JDK 9 was released in September 2017.)

However, we observed that for efficiency, it is not sufficient on its own to have a straightforward ASCII string representation, but it is important to also have an efficient representation for substrings, which is less obvious to achieve. We discuss in the next subsection the implications for each of the 3 programming languages.

A second issue that we encountered is the representation of optional fields. An obvious implementation choice would be to use a hashtable per alignment that maps mnemonic identifiers to values, which are commonly available in modern programming languages. However, it is known that for small maps consisting of a few dozen entries, hashtables have an unnecessary overhead both in terms of memory use and access times. A simpler search list is usually not only more compact but also faster in such a case.[6] We discuss the implications for the 3 programming languages in the second subsection below.

Representation of substrings

Sequence reads and quality score strings in SAM files are relatively large in size. For example, for reads of length 150 base pairs, the sequence reads and quality strings are 150 characters each.

When a SAM file is parsed by elPrep, it is first split up into separate text lines. It is difficult to avoid the splitting into text lines before parsing the required and optional entries in an alignment line. This is due to the fact that in elPrep, the parsing of alignment lines is parallelized over several CPUs. However, splitting up alignment lines into their tabulator-separated entries before the parallel phase wastes a relevant opportunity for efficient parallel execution.

When each text line is then parsed into a data structure representation for the corresponding alignment, a naive approach would create another copy of both the read data and the quality scores. This additional copying turns out to be a performance bottleneck and leads to unnecessary additional memory allocations.

It is therefore better to keep the alignment text line unchanged and instead refer to the positions and lengths in the text line where the corresponding information resides from within the alignment data structure.

Go. This kind of representation has first-class support in Go in the form of *slices*. For example, if s is a variable containing a string, and `pos` and `length` are variables containing a starting position and a length describing a desired substring of s, then the *slice expression* `s[pos:pos+length]` yields that substring. Instead of creating a freshly allocated string of size `length` and copying the relevant portion from `s` into that new string, the slice expression returns a data structure internally containing 3 entries: the reference to `s`, and `pos` and `length`. This data structure otherwise behaves like a regalur string, with index accesses being automatically mapped to the original string after index adjustment.

Java. In Java, the `String` class of the standard library supports a method `substring` which, similar to slice expressions in Go, expects a start and end position. Up until Java JDK 7 update 5, this method behaved similar to Go slice expressions, returning an object that internally only referred to a view of the original string. However, since Java JDK 7 update 6 (released in August 2012), `substring` actually creates a freshly allocated string and copies the relevant portion from the original string.

This means that for elPrep, we had to implement our own custom string class to emulate the previous behavior of `substring`, which adds to the complexity of the Java implementation of elPrep.

C++. In principle, C++ implements the desired behavior in the form of the std::string_view class in its standard library. It has a constructor that can be passed a string address and a length, and yields an object that refers to that string address and length rather than copying a string. The string address can point to a position in the middle of a string and can therefore express an offset into a string as well. Unfortunately, the `std::string_view` class is not compatible with reference counting as implemented in the C++`std::shared_ptr` template, which we need for managing memory in the open-ended elPrep software framework.

Therefore, like in Java, we had to implement our own custom string class that supports the desired behavior for reference-counted strings. On top of that, we also had to implement a wrapper around file input routines, to be able to use our own custom string class as a result of reading strings from a file, due to other restrictions resulting from the design of the C++ standard library. Defining new classes requires substantially more care and effort in C++ than in Java, due to the need to specify several special cases for default constructors, copy constructors, and destructors. This is an arduous and error-prone task which added substantially to the development time of the C++ implementation of elPrep.

Representation of search lists

A search list is a simple implementation of a mapping from a set of keys (typically strings) to values. It can be implemented either as a linked list data structure or as a growable array, with entries mapping each key to its associated value. Retrieving the value for a particular key is implemented by searching the list from front to back for an entry that matches this key; setting a new value for an existing key is also implemented this way, followed by a modification of the value in the found entry; and adding a brand new key/value mapping is implemented by appending it to the end of the search list. For a few dozen key/value mappings, such an implementation is more efficient than a general-purpose hashtable, because it avoids the memory overhead of having more buckets for storing key/value lists than there are actual key/value mappings, and because it avoids computing hash values for keys, which only pays off if it helps to avoid searching through a very large search list.[6]

None of the 3 programming languages have direct support for such a search list, which is why we had to implement it ourselves in each case. The implementations differ in several regards, as discussed next.

Go. The Go implementation of a search list was the most straightforward. A search list is represented as a slice of key/value entries. Each key/value entry is a data structure that contains a key, represented as a string, and a value represented by the Go type `interface{}`. This Go type is called an empty interface and is treated specially in Go in that it can store values of any type that is supported by the Go programming language. The value that is currently stored in a variable of type `interface{}` can be retrieved, along with the type of that value. That makes this type ideal for storing the different optional field types of the SAM file format. An advantage of the empty interface type in Go is that values of primitive types (like small integer types and floating point numbers) may be represented as immediate values rather than stored separately in heap memory.[7] This means that empty interfaces in Go do not incur an unnecessary overhead with regard to memory use.

Adding new key/value mappings to a slice is also straightforward in Go, because slices support an `append` function for this purpose, which silently grows the slice if necessary.

Java. The Java implementation of a search list was slightly more complex than the Go version. The main reason for this is that Java has no type for storing any type of value. Java has the type `java.lang.Object` which can be used for storing any kind of class instance, but it does not support storing immediate primitive types (like integers or floating point numbers). Instead, primitive types would be silently converted into *boxed* objects, which are instances of corresponding classes. This would require additional storage on the heap in the general case. As primitive types for optional fields in SAM files are very common, this would be a too large price to pay.

We therefore opted instead to design a flat class hierarchy, with the root of the class hierarchy containing only the key for each key/value entry, and each subclass additionally containing the corresponding value type. This value type can then be a primitive type, depending on the supported optional field type, which avoids the additional heap storage.

Contrary to the Go and C++ implementations, though, each key/value entry is still itself an object stored separately on the heap, whereas in Go and C++, key/value entries can be stored immediately in the corresponding container types (slices for Go, and vectors for C++).

The search list functionality itself is straightforwardly implemented using the `java.util.ArrayList` class which supports all needed operations, including silent growth if necessary when appending new key/value entries.

C++. The C ++ implementation of a search list is very similar to the Go implementation. A search list is represented as a `std::vector` of key/value entries, which supports all necessary operations, including silent growth if necessary. Each key/value entry is a data structure that contains a key represented as a string, and a value represented by the C++ type `std::any`, which can store values of any type that is supported in the C++ programming language.

A seemingly obvious alternative for representing the value in each key/value entry would be to use the C++ template `std::variant` and enumerate the possible types for that template. A `std::variant` object imposes the restriction on the stored values that they can be only any of the explicitly enumerated types. As the number of optional field types in the SAM file format is finite, this would not only be possible, but would also have the advantage of increasing the static type safety of the representation, which is significantly harder to achieve in Go and Java. However, the downside of `std::variant` is that its implementation is not allowed to represent primitive types as immediate values, but has to allocate space on the heap for any possible value type. The `std::any` type does not have this disadvantage, because it allows primitive types to be represented immediately inside the storage already allocated for the `std::any` type, which avoids the additional heap storage.

Conclusions

The issues we discussed above are by far not the only challenges we encountered during our evaluation, but we think these are good examples to give an overall impression of the different programming languages when implementing a software tool for SAM/BAM processing such as elPrep.

Overall, the C++ implementation incurred the most development effort, significantly more than both the Go and Java implementations. This is due to having to explore significantly more design choices (like the choice between `std::any` and `std::variant` discussed above, for example), the permeation of low-level design choices (like the choice between stack allocation and heap allocation, which is explicit in C++ and typically affects large parts of the source code), the choice between different memory managers (as discussed in our other publication on this topic),[3] and so on.

The complexity of the Go and Java implementations, on the contrary, are roughly comparable. For example, the parallelization framework in elPrep was easiest to implement in Java, due to its excellent support for functional-style operations on streams of elements, including parallel operations, in the `java.util.stream` package introduced in JDK 8. In fact, this inspired us to add a number of similar operations to a library for parallel programming in Go, called Pargo,[8] that we develop and also use in elPrep. On the contrary, Go has direct language support for slices, for example, which makes the Go implementation significantly easier in this regard, as discussed above.

Our other article shows that the Go implementation has the best balance of runtime and memory use compared with the Java and C++ implementations.[3] This article shows that Go fares extremely well with regard to ease of programming and is therefore a good choice for the official elPrep implementation also in this light. Had the Java implementation shown better performance than it did, it would have been a defensible implementation language for elPrep with regard to ease of programming as well.

Acknowledgements

The authors are grateful to the imec.icon GAP project members, and especially Western Digital for providing the compute infrastructure for performing benchmarks. The authors also thank Thomas J. Ashby and Tom Haber for in-depth

discussions about memory management techniques in various programming languages.

Author Contributions

PC designed and performed the study, participated in the Go implementation of elPrep, implemented the C++ and Java versions of elPrep, and drafted the manuscript. CH designed the elPrep software architecture, participated in the Go implementation of elPrep, and drafted the manuscript. PC, CH, and WV contributed to the final manuscript. All authors read and approved the final manuscript.

ORCID iD

Pascal Costanza https://orcid.org/0000-0001-8894-3238

REFERENCES

1. Herzeel C, Costanza P, Decap D, Fostier J, Verachtert W. elPrep 4: a multithreaded framework for sequence analysis. *PLoS ONE*. 2019;14:e0209523. doi:10.1371/journal.pone.0209523.
2. Herzeel C, Costanza P, Decap D, et al. elPrep: high-performance preparation of sequence alignment/map files for variant calling. *PLoS ONE*. 2015;10:e0132868. doi:10.1371/journal.pone.0132868
3. Costanza P, Herzeel C, Verachtert W. A comparison of three programming languages for a full-fledged next-generation sequencing tool. *BMC Bioinformatics*. 2019;20:301. doi:10.1186/s12859-019-2903-5.
4. Li H, Handsaker B, Wysoker A, et al. The Sequence Alignment/Map format and SAMtools. *Bioinformatics*. 2009;25:2078-2079. doi:10.1093/bioinformatics/btp352.
5. SAMtools organization. SAM/BAM and related specifications. http://samtools.github.io/hts-specs/.
6. Seibel P. *Practical Common Lisp*. New York, NY: Apress; 2005.
7. Cox R. Go data structures: interfaces. https://research.swtch.com/interfaces.
8. Costanza P. pargo—a library for parallel programming in Go. https://github.com/ExaScience/pargo.