A Case Study on Performance Optimization Techniques in Java Programming

Ciprian Khlud[1][a] and Cristian Frăsinaru[1][b]

[1] *Institute of Problem Solving, XYZ University, My Street, MyTown, MyCountry*

[2] *"Alexandru Ioan Cuza University", Iaşi, Romania*

ciprian.mustiata@gmail.com, acf@info.uaic.ro

Keywords: Java, Runtime performance, Memory usage, Garbage collection, Sequence analysis, SAM/BAM files

Abstract: Choosing the right programming platform for processor or memory intensive applications is a subject that is debated in all types of contexts. When analyzing the performance of a specific platform, equally important is the usage of appropriate language specific constructions and programming interfaces (APIs). In this paper we investigate how a state-of-the art implementation, part of a multi-threaded framework for sequence analysis (elPrep) could benefit from various optimization techniques dedicated to improving the runtime performance of Java applications. ElPrep is an established tool for processing SAM and BAM files in sequencing pipelines. In order to migrate from its original implementation to a different software platform, more suitable for memory intensive tasks, the authors have re-implemented elPrep in Java, Go and C++ and benchmarked their performance. Overall, the Go implementation won by a good margin, considering a metric that involved both the RAM usage and the runtime performance. We show that, without changing the semantics of the algorithm, by using appropriate programming techniques we are able to significantly improve the behavior of the Java implementation, to a point that may even alter the conclusions of the original study.

1 INTRODUCTION

In the field of bioinformatics, DNA sequence analysis generally consists of processing large amounts of data and performing various operations on it, such as sequence alignment, variant detection, searches against biological databases, etc. A large variety of software tools exists for these operations, most of them having specific uses cases but with a common denominator regarding the fact they need to perform processor and memory intensive tasks: I/O operations on large file, compression/decompression, text processing, etc. (**?**)

Choosing a programming platform that offers all the required instruments to handle the specific challenges in bioinformatics is important, as pointed out in a recent study dedicated to migrating an existing Common Lisp application, called elPrep, to another platform with better support for memory management and concurrency (Costanza et al., 2019). ElPrep (Herzeel et al., 2019) is a a multithreaded tool for preparing sequence alignment/map files (SAM/BAM) for variant calling in DNA sequencing pipelines. A key feature of elPrep is the ability to avoid the standard practice of creating a pipeline consisting of multiple command line tools invocations, executing a single pass through a SAM/BAM file and keeping data as much as possible in main memory. In (Costanza et al., 2019) the authors investigated Go, Java and C++ programming platforms, as an alternative to Common Lisp. Te result of their study concluded that the Go implementation performed best, using a metric that involved both the RAM usage and the runtime performance. The benchmarks of the study showed that Java had a faster runtime, but a significantly higher memory usage, while Go offered a better balance between the two.

As the Java source code for elPrep is available at `https://github.com/exascience/elprep-bench`, we have analyzed key aspects regarding the memory management and thread synchronization, and propose a series of improvements that could increase significantly the performance of the Java implementation.

[a] https://orcid.org/0000-0000-0000-0000

[b] https://orcid.org/0000-0002-5246-7396

2 BACKGROUND

2.1 GARBAGE COLLECTION

In order to analyze the behavior of memory intensive applications, it is important to understand how garbage collection works and especially how Java (Java Platform, Standard Edition, 2019) implements its garbage collectors.

The Java Virtual Machine (JVM) (Lindholm et al., 2014) offers an automatic storage management system, called *garbage collector (GC)* which reclaims heap storage occupied by objects which are no longer used. The garbage collection process (Oracle, 2019) works typically by splitting the heap into two regions: a *young generation* region and an *old generation*. All new objects are allocated in the young region, in a very fast manner, using typically a ”bump-pointer” strategy. When this region becomes full a *minor* garbage collection occurs and all dead objects are deleted very quickly. The objects which are still referenced survive and they are moved to the old generation. This minor collection is always a ”stop the world” event, meaning that all of the application threads will be paused until the GC is finished. In the old generation, objects are expected to live longer and they are collected more seldom but with a more expensive algorithm, called *major* garbage collection.

The algorithm used by GC has two steps. The first one is to *mark* the objects that are still used from the heap. In the second step, it *sweeps (deletes)* the objects which have not been marked (dead), leaving only referenced objects and pointers to free space. Therefore, the speed of GC depends on two factors: the number of objects it has to analyze and the complexity of the relationships between them.

Considering the behavior we have described so far, we will analyze the impact of some simple tweaks meant to reduce the impact of GC over the application performance, such as: reducing the unnecessary small allocations in young region, controlling the scope in which objects are referenced in order to minimize the number of times when expensive collection of old region is triggered, simplifying the object graph and controlling the amount of memory JVM is allowed to use.

REFERENCES

Costanza, P., Herzeel, C., and Verachtert, W. (2019). Comparing ease of programming in C++, Go, and Java for implementing a next-generation sequencing tool. *Evolutionary Bioinformatics*, 15:1176934319869015.

Herzeel, C., Costanza, P., Decap, D., Fostier, J., and Verachtert, W. (2019). elprep 4: A multithreaded framework for sequence analysis. *PLOS ONE*, 14(2):1–16.

Java Platform, Standard Edition (2019). Java Development Kit Version 11 API Specification. https://docs.oracle.com/en/java/javase/11/docs/api. Accessed: 2019-06-01.

Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. (2014). *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition.

Oracle (2019). Java garbage collection basics - oracle. https://www.oracle.com/webfolder/technetwork/ //tutorials/obe/java/gc01/index.html. Accessed: 2019-06-01.