A Case Study on Performance Optimization Techniques in Java Programming

Ciprian Khlud[1][a] and Cristian Frăsinaru[1][b]
[1] *"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/0000-0001-6211-3199
[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.

2.2 THREADS, LOCKS AND THE FILE SYSTEM

Java platform supports concurrent programming by using threads (Gosling et al., 2014). Multiple threads can execute at the same time, taking advantage of computing units that have more than one processor and of processors that have more than one core. Threads performing operations that are not atomic will interleave when they access shared data. A *synchronized* statement acquires a mutual-exclusion lock when entering a critical section, executes the block that references the shared data, then releases the lock. While one thread owns the lock for that data, no other thread may access it. The proper use of this mechanism is crucial for the concurrent implementation of an algorithm. If threads are waiting to much on locked resources, the overall performance of the application will suffer.

In our case study, multiple threads are performing operations on the file system, reading and writing large amounts of strings from and into text files. Regardless of the operating system and programming language, the underlying hardware is optimized to work with streams of bytes. In an atomic operation, data is read into a buffer of bytes, in a contiguous manner. Similarly, data is written into a buffer of bytes that is flushed afterwards to the file. Both these operations are single threaded by design and historical reasons, so Java libraries have locks to make the access to streams single-threaded. A `write` method usually looks like this:

```
public void write(String s, int off, int len){
  synchronized (lock) { ... }
}
```

We will show that creating a large number of short lived strings and writing them to a file in a multi-threaded manner will generate a behavior similar to using a single thread, plus the overhead of acquiring and releasing the synchronization lock.

2.3 MEMORY USAGE

The Java Virtual Machine allocates memory either on stack or on heap (Lindholm et al., 2014). The *heap* is the place where all class instances and arrays are allocated and it is shared among all threads. Each JVM thread has a private *stack* which holds local variables and partial results during successive method invocations and returns. When working with large amounts of objects, it is quite important to assess the memory consumption of a data structure, in similar way as the *sizeof* construct in C or C++. An object allocated on the heap has a *header* which contains information used for locking, garbage collection or the identity of that object. The size of the header depends on the operating system, and it may be 8 bytes on 32 bit architectures or 16 bytes on 64 bit architectures. Also, for performance reasons and in order to conform with most of the hardware architectures, JVM will *align* data. That means that if we have an object that wraps just one byte, it will not

use: 8(object header) + 1(content) = 9 bytes of memory on the heap, but it will use 16 bytes as it needs to be aligned to the next 8 byte boundary.

In Java, strings are objects and they are allocated on the heap. That fact that string literals are stored in a shared object pool, in order to reduce memory consumption, is of no relevance in our context. Inspecting the source code of the `String` class, one can observe the following instance fields:

```
private final byte[] value;
private final byte coder;
private int hash; // Default to 0
```

As expected, a `String` object keeps a reference to an internal byte array. However, the other two fields will make the size of the object equal to: 8 (header) + 4 `value` reference + 1 `coder` value + 4 `hash` value = 17 bytes. Being aligned to 8 bytes, it will actually use 24 bytes. When creating many `String` instances (like millions of them, as in our case study), the extra information included in this class will add up, consuming memory and triggering the garbage collector more often than necessary.

We will show that replacing the `String` usage to the underlying `value` byte array will improve the performance of the application, and this approach should be implemented in every scenario that involves processing large amounts of text data.

2.4 MEMORY COMPACTATION

Another important part of working with large data sets that have to be accessible in memory regards the format in which they are represented. Choosing the right format will not only reduce the amount of consumed memory but it will also reduce the GC cost to copy the objects between regions and the cost of visiting and marking them.

The most common approach of representing information is in row based form, where a *row* is a record of some kind and a *column* is a certain property of that row. This type of representation is used in most relational databases management systems, where sets of rows of the same type form *tables*. Despite having many advantages, this format is not necessarily optimum when it comes to data representation in memory.

A *column store* model (Abadi et al., 2013) "reverses" the orientation of the tables. It stores data by columns and uses row identifiers in order to access a specific cell of the table. By storing each column separately, query performance is increased in certain contexts as they are able to read only the required attributes, rather than having to read entire rows from disk and discard unneeded attributes once they are in memory. Another benefit is that column stores are very efficient at data compression, since representing information of the same type inside of a column helps the *data alignment* process that we have previously mentioned.

Let's consider a simple example, using the class `Point`, defined as a pair of two integer fields `x` and `y`. The basic idea is that instead of having a row-based model consisting of an array `Point[]` of instances (each `Point` object is a row and its members `x` and `y` are the columns), to use two arrays of integers `x[]` and `y[]`, representing the two columns. This way we can store the same data, minus the object headers corresponding to all the `Point` instances. Not only the memory consumption will be lower (so the GC will be triggered less often), but the structure will also take shorter time to visit, since there are only two objects now (the two arrays).

Though a column store is a very good solution for size reduction, it has the downside of requiring more computational effort in order to work with multiple properties of the same object. However, when saving memory is the major concern, and especially when it comes to hundreds of GB per instance, the execution slowdown becomes far less important if we can achieve significant reductions in consumed memory.

3 REPRESENTING THE DATA STORE

3.1 THE ROW-BASED MODEL

The data structure which is used in original ElPrep algorithm is represented by the class `SamAlignment`, an object of this type storing one row of a SAM file. The class contains the following declaration of instance variables:

```
public Slice QNAME;
public char FLAG;
public Slice RNAME;
public int POS;
public byte MAPQ;
public Slice CIGAR;
public Slice RNEXT;
public int PNEXT;
public int TLEN;
public Slice SEQ;
public Slice QUAL;
public List<Field> TAGS = new ArrayList<>(16);
public List<Field> temps = new ArrayList<>(4);
```

For a small BAM file of 144 MB there will be created around 2.1 million `SamAlignment` instances and for a 1.27 GB BAM file there will be created around 17.6 million objects.

For simplicity, let's disregard `TAGS` and `temps` fields (which can have different lengths) as it makes

the calculation simpler and analyze the memory consumption in both cases. We suppose also that the JVM uses 32 bits for representing an object header.

One `SamAlignment` object contains: 8 bytes object header, 6 instances of `Slice` objects (`QNAME`, `RNAME`, `CIGAR`, `RNEXT`, `SEQ`, `QUAL`) of 4 bytes each, 3 integer fields (`POS`, `PNEXT`, `TLEN`) of 4 bytes each, 1 character (`FLAG`) of 2 bytes, and an additional byte (`MAPQ`). So, the total size of the object is: $8+6*4+3*4+1*2+1*1=47$ bytes, and as it is rounded up to a multiple of 8, the result is 48 bytes.

In order to save memory, the string representing a row scanned from the original file is shared between multiple objects. All 6 `Slice` instances contain a reference to the underlying string and two integers pointing to the start index and length. So, a `Slice` instance uses: 8 (object header) +4 (reference to the string) $+4+4=20$ bytes, being rounded to 24.

As `Slice` instances point to a `String` object, the `String` itself adds another 24 bytes, as we have already seen, and the byte array object referenced from the `String` adds another 24 bytes (not counting its content size).

Adding all these numbers up, we conclude that for representing a `SamAlignment` object, the JVM needs: 48 (the object itself) $+6*24$ (Slice) $+24+24=240$ bytes.

For a 144 MB file there are 2.1 million entries, so the memory requirement for storing the graph of objects and the integer fields is approximately $504,000,000$ bytes, which equals to more than 480 MB (not counting the 144 MB of content in byte array).

For the 1.27 GB BAM file, the numbers are much larger as there are a 17.6 million rows. The total is $4,224,000,000$ bytes, representing almost 4 GB.

At the point when GC executes, there are 9 objects per row (`SamAlignment` +6 `Slice` +1 `String` +1 byte array) on heap.

The ElPrep algorithm will duplicate all the `SamAlignments` once, as it looks for duplicate objects, so the required memory will double. On top of it, there are the underlying strings which keep the data. For 2.1 million entries the grand total is around $700,000,000$ bytes, and $6,050,000,000$ bytes for the 1 GB entry.

3.2 THE COLUMN-BASED MODEL

Let us analyze how much memory can be saved by switching to a column-based approach. We have defined the following data structures: `StringSequence` for representing in a compact manner a collection of strings, `DeduplicatedDictionary` for eliminating duplicate copies of repeating strings, `DnaEncodingSequence` for storing `A,C,G,T,N` sequences using an encoding of 21 letters per `long` and `TagSequence` for representing tags encoded in an array of `short` values. We have also used the classes `CharArrayList`, `IntArrayList` and `ByteArrayList` from FastUtil library (Vigna, 2019), which offers implementations with a small memory footprint and fast access and insertion.

The new definition of the data store is described by the class `SamBatch`, containing the following members:

```
StringSequence QNAME;
CharArrayList FLAG;
DeduplicatedDictionary RNAME;
IntArrayList POS;
ByteArrayList MAPQ;
DeduplicatedDictionary CIGAR;
DeduplicatedDictionary RNEXT;
IntArrayList PNEXT;
IntArrayList TLEN;

DnaEncodingSequence SeqPacked;
StringSequence QUAL;
```

So, instead of having a large number of `SamAlignment` instances, we will have a single object of type `SamBatch` which contains references to the ”columns”, i.e. our data structures holding all the information of a specific type.

Regardless of VM bitness, the memory consumption for representing one row of the input file is:

Data Type	**Count**	**Bytes**
`StringSequence`	2	4
`DeduplicatedDictionary`	3	4
`IntArraySequence`	3	4
`CharArraySequence`	1	2
`ByteArrayList`	1	1
`DnaEncodingSequence`	1	4
	Total	**39**

Considering that no rounding up is necessary, for 2.1 million rows this sums up to $81,900,000$ bytes, equivalent to 78 MB. The header sizes of the column objects ($11*8$ bytes) become negligible in this context.

of just 60 MB of indices and after that we have the joined data.

As this is a major reduction in memory consumption, let us analyze the technique used to achieve this result.

The basic idea is that instead of storing an array of `String` objects, for example:

```
String items[] = { "abc", "def" };
```

each consuming memory due to their headers, we can use a single object of type `String`, storing all the characters, and an additional array for their lengths.

```
String dataPool = "abcdef";
int[] endLengths = {3, 6};
```

For such a small array, the save is minor, but for a large number of items (millions), the memory reduction becomes significant. Even more important, the GC work is also reduced, since no matter how many items are in the `dataPool` and `endLengths` fields, there are only two objects to visit. The technique described above was implemented in the class `StringSequence`.

If the strings that are to be stored are repeated frequently, we can apply another optimization: instead of keeping them joined, we will use an indexed collection containing all the distinct strings and an array holding one index for each string. For example, {"abc", "def", "abc", "xyz", "abc"} becomes:

```
table   : {abc=>0, def=>1, xyz=>2}
items   : [0, 1, 0, 2, 0]
```

The `table` structure is based on the class `Object2IntOpenHashMap<String>` from FastUtil library, instead of the standard `HashMap<Integer, String>`, since it uses the primitive data type `int` for representing the keys, which also saves some memory. This data deduplication technique (He et al., 2010), (Manogar and Abirami, 2014) was implemented in the class `DeduplicatedDictionary`.

When storing strings containing characters from a restricted alphabet, one optimization that can be performed is using an array of primitive values, for example a `long[]`, and encoding each character into a block of bits. The number of bits required for a character depends on the size of the alphabet. DNA sequences use four letters `A,C,G,T`, but it is possible for a machine to read incorrectly a symbol and to return `N`. In order to represent 5 possible characters we need at least 3 bits, which means that a `long` can store in its 64 bits 21 DNA letters. The class `DnaEncodingSequence` which implements this string encoding technique contains the following members:

```
LongArrayList content;
ShortArrayList lengths;
IntArrayList positions;
```

For example, encoding the 21 letters string "AAAACCCCGGGGTTTTNNNNA" would produce a single long value, containing the bits:
`0000100100100100011011011011010010010001001001001000000000000`

From right to left, 000 represents A, 001 represents C, and so on.

In the sample files, one DNA sequence is typically around 100 letters, so the memory needed in order to represent it would be 1 `int` (encoding length) and 5 `long`s (the content), that is 44 bytes. This reduces the memory consumption by a factor of two.

Another advantage of using such an encoding is that when checking if two sequences are exactly the same, we can compare first their lengths and, if they are equal, comparing `long` values means comparing 21 characters at once. As before, the GC will also benefit from the reduced number of objects that must be visited.

Running the smaller input file (144 MB), we have estimated that the original ElPrep algorithm would use around three times more memory than the size of the input SAM file. However, when trying to process the larger input file (1.2 GB), on a 32 GB machine, we have obtained an `OutOfMemoryError`, meaning that the penalty of using too many objects in order to represent the information was preventing us in loading the entire data set into memory.

JVM using how much memory?

1 GB input: Original algorithm: 32 024 799 640 Bytes measured in VisualVM after GC was called and program suspended right before writing to disk. This is the lowest maximum memory consumption for 1.2 GB BAM file Optimized algo: 31 463 351 904 Compact algo: 4 703 674 464 bytes Live data memory usage: 6.8x less. 144 MB input: Original algorithm: 2 234 575 120 Bytes measured in VisualVM after GC was called and program suspended right before writing to disk. This is the lowest maximum memory consumption for 1.2 GB BAM file Optimized algo: 2 273 444 096 Compact algo: 607 589 728 bytes Live data memory usage: 3.67x less. 8 GB input: 20 334 009 424 B. Obviously we had a too small machine to process a live data set that is using as minimum 3 12 GB input: 35 401 115 712 B.

Time is 300 seconds (on 8core machine) as the machine touches Swap file, but being able to process 12 GB BAM file on a 48 GB machine is a feature in itself.

With memory compaction, the memory needed to represent all the objects was estimated to 17.6 million rows * 39 bytes per row = 686,400,000 bytes (655 MB), plus the DNA sequences 17.6 millions $*44 = 774MB$.

As the string data was estimated to at most 3 GB, in 4.4 GB we have stored both the entire SAM file and the fields associated with the alignments.

Duplicating all this (as elPrep does it) will make the entire live set go to 9 GB, which is still around half of the original live set (what is the size of the original live set? definition of "the live set".

On top of this, there is the cost of tags. In the original implementation, every `SamAlignment` object has references to `temps` and `TAGS` arrays. These arrays

have an initial size of 16, respectively 4, and contain references to `Field` objects, which in turn contain a reference to a `Slice` object. Using the row-based model, for every row the memory consumption is: 20 references * 4 bytes + 2 * 64 (an empty `ArrayList` costs 64 bytes)= 256 + 80 bytes. The memory usage for tags/temps is 5.9 GB. Tags only will be 80% of usage = 4.2 GB

In order to address `temps` and `TAGS` we have implemented the `TagSequence` class, which is a combination between `StringSequence` and `DeduplicatedDictionary`. Since the `tags` are repeating frequently, we save them using one `short` value per tag, but instead of using a list of tags, we define a sequence of indices.

TagSequence does memory compaction by using a mix of short per-tag encoding and a full sequence of tags is joined together.

For example, for the input `"tag0 tag1 tag2"`, `"tag1 tag2 tag3"`, the representation would be:

```
table = {"tag0":0, "tag1":1,
         "tag2":2, "tag3":3},
lengths: [0,3],
tagSequence = [0,1,2,1,2,3];
```

For a 20K rows, we found following numbers: Unique tag values: approx 2400 Total number of tags: approx 200000 (10 tags per row on average)

Numbers were correct, it was simply looking into some random batches when debugging.

4 OPTIMIZING I/O OPERATIONS

The first and most simple technique that could improve performance when working with large files is I/O `buffering` (Oaks, 2014). Invoking a Java *read* method, for example, will eventually trigger an invocation of the operating system method responsible with reading data from the disk. Although most operating systems use I/O buffering by default, an invocation of a method outside JVM is nevertheless expensive. Another reason to minimize the number of I/O operations is that, for each one of them, temporary objects needed to store the data must be created, meaning that GC will have more work to do when recovering the memory.

The original code that we are analyzing uses correctly the `BufferedReader`, respectively the `BufferedWriter` classes in order to perform I/O operations on the SAM file. Checking the definition of the `BufferedReader` class we notice that the default buffer size is set to 8192 characters, which is reasonable for most scenarios. The *read* operation is implemented in a concurrent fashion, using Java parallel streams:

```
var alnStream = inputStream.parallel()
  .map((s) -> new SamAlignment(s));
```

In this context, for larger files (as in our case), and in the presence of multiple cores, increasing the size of the character buffer may offer a better performance, estimated by our experiments at around 5%. Este corect? Pe laptop-ul meu nu obtin nicio imbunatatire

The same simple optimization can be made for writing, by presizing the `BufferedWriter` object to a more suitable value than the default one.

When it comes to writing, we notice another possible improvements. The following code sequence iterates through the `SamAlignment` objects found in the input stream and, for each one of them, it creates a string representation containing the fields of the `SamAlignment` class and tabs as separators. This is implemented using a `StringWriter`, which is a character stream collecting its output in a memory buffer.

```
var outputStream =
 alnStream.parallel().map((aln) -> {
   var sw = new StringWriter();
   try (var swout = new PrintWriter(sw)) {
     aln.format(swout);
   }
   return sw.toString();
});
```

The initial size of the `StringWriter` internal buffer is by default 16 characters. A line in the output file is more than 200 characters. By presizing the buffer to a value of 256, we have obtained a speed-up of the writing process estimated by our experiments to more than 10%.

If we take a closer look at the `StringWriter` class we notice that it uses an internal `StringBuffer` object. To quote from its documentation, a `StringBuffer` is a thread-safe, mutable sequence of characters. That means that most of its methods are declared as *synchronized* in order to control the access to the buffer of characters in a multithreaded environment. However, in our case, each writing thread spawned by the parallel stream implementation gets his own copy of a `StringWriter` so there is no resource contention that would require synchronization. So, instead of using a `StringWriter`, it would be better to simply use a common buffer, with no synchronization. To maximize the performance gain, the buffer could be implemented as a byte array. Using bytes instead of characters is quite trivial as Java `String` class has the method `getBytes()` and recreating a `String` object can be done by simply using one of its constructors: `new String(bytes)`.

Writing to the file is implemented by simply invoking the `println` method of a `PrintWriter` output stream, decorating a `BufferedWriter`, which in turns decorates a `FileWriter`.

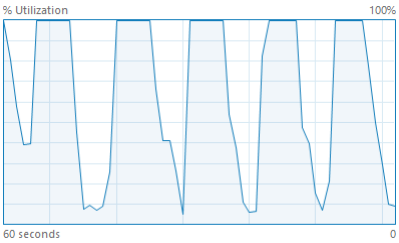
```
var out = new PrintWriter(
  new BufferedWriter(
    new OutputStreamWriter(output)));
...
outputStream.forEachOrdered(
  (s) -> out.println(s));
```

Again, the `PrintWriter.println` method is using a `synchronized` block in order to fulfill its task, which is eventually an invocation to `BufferedWriter.write` method.

```
public void println(String x) {
  synchronized (lock) {
    print(x); println();
   }
}
```

The `BuffereWriter.write` method is also synchronized and so is `FileWriter.write`, which is the last one invoked in this chain.

Since more than half of the I/O time is spent writing, we have analyzed the CPU utilization, using the operating system profiler.

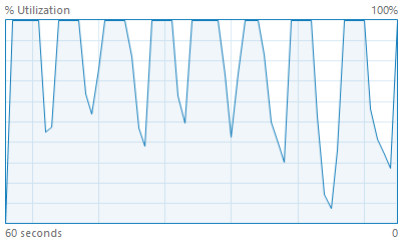


The graph looks like a chainsaw, which is a sign that the code is bottle-necked on a single thread. This is no surprise, since the `outputStream` is written in a sequential fashion. On top of that, for every row there has to be a mutex check whenever a synchronized block of code is invoked. Since there is a large number of rows, all these checks will add up and produce a significant slowdown of the overall writing process.

The technique that should have been used in this context involves using a single byte array. Instead of creating and storing individual string representations of all the `SamAlignment` objects, and writing them one by one to disk, we add progressively information into the buffer. Only when the buffer becomes full, we write its content on disk. Obviously, all these steps will be performed in a single thread, but the bottleneck will be less obvious. The class implementing the byte array is called `StreamByteWriter`.

```
StreamByteWriter streamByteWriter
  = new StreamByteWriter(output);
for (SamAlignment aln: alignments) {
  aln.formatBuffer(streamByteWriter);
}
```

Even if we are using now a single core for all the writing, the necessary time for creating the output file was reduced by half. The following graph shows the CPU utilization for this approach:



Putem scapa si mai mult de portiunile albe?

5 EXPERIMENTAL RESULTS

The computer we have used in order to perform the experiments is a Ryzen 7-1700, having 8 cores and using 32 GB of RAM. Since we didn't have access to the hardware necessary to run all the tests in memory, as the original paper, we have used the smaller SAM files and ran the same processing repeatedly in order to obtain an accurate result of the running time. For example, running 10 times the algorithm on the smallest input SAM file, which is approximately 700 MB, will produce a total running time of around 70 seconds. Running the algorithm repeatedly will trigger the garbage collector and this will be the cause of variations in the collected results, ranging from around 2 to 3 seconds, when reading the smallest file, and 3 to 4 seconds for writing.

Before executing the timed invocations, we have *warmed-up* the JVM. This is usually achieved by simply running a couple of times an initial test (not timed) that uses all the classes involved in the algorithm. Such a warm-up is necessary because Java is using a lazy class loading mechanism and just-in-time compilation. After this step, all important classes are stored into the JVM cache (native code), making them available at runtime with no additional penalty.

The original elPrep benchmarks have been performed on a Supermicro SuperServer 1029U-TR4T node with two Intel Xeon Gold 6126 processors consisting of 12 pro-cessor cores each, clocked at 2.6 GHz, with 384 GB RAM (Costanza et al., 2019). The authors claim to do the processing of the 8 GB BAM file in 6 min:54sec to 7 min:31sec and memory usage is $330 - 340$ GB.

As we didn't have access to such a performing machine, we did most of testing with the smallest file, the 144 MB BAM file (673.3 MB SAM file). For the 8 GB BAM file (27.18 GB SAM) our results will be an extrapolation of the former ones. Sau nu?

5.1 RUNTIME PERFORMANCE

Aici trebuie sa punem niste teste comparative, care sa surpinda impactul fiecarei optimizari vs. original si la sfarsit castigul total

In terms of running time, the performance gain for reading the input file is negligible. Sau nu?

Writing

	144 MB BAM	8 GB BAM
Original	.... s	.... s
I/O Buffering	.... s	.... s
StreamByteArray	.... s	.... s
Overall	.... s	.... s

5.2 MEMORY USAGE

Aici trebuie sa punem cifre comparative, care sa surpinda castigul de memorie + pozele scalate

REFERENCES

Abadi, D., Boncz, P., and Harizopoulos, S. (2013). *The Design and Implementation of Modern Column-Oriented Database Systems*. Now Publishers Inc., Hanover, MA, USA.

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.

Gosling, J., Joy, B., Steele, G. L., Bracha, G., and Buckley, A. (2014). *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition.

He, Q., Li, Z., and Zhang, X. (2010). Data deduplication techniques. volume 21, pages 430 – 433.

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.

Manogar, E. and Abirami, S. (2014). A study on data deduplication techniques for optimized storage. pages 161–166.

Oaks, S. (2014). *Java Performance: The Definitive Guide*. O'Reilly Media, Inc., 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.

Vigna, S. (2019). Fastutil 8.1.0. http://fastutil.di.unimi.it/. Accessed: 2019-06-01.