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. We also show that, by changing the manner in which data is represented, to better fit the particulars of the Java memory management, we are able to improve the original scoring (based on computing time and memory consumption) to around one order of magnitude better on the most expensive component (read/write).

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 (GC, 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. By moving referenced object together, this makes new memory allocation much easier and faster. 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: $8(\text{object header}) + 1(\text{content}) = 9$ 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;
// Defaults 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 COMPACTION

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.

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;
```

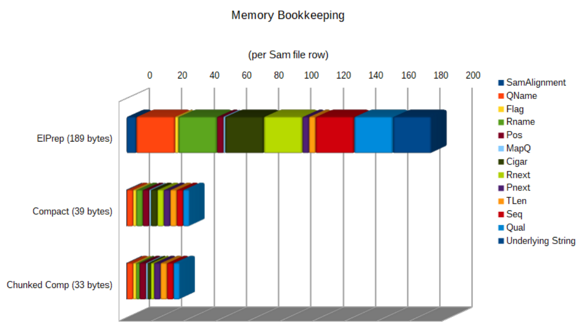
So, instead of having a large number of `SamAlignment` instances, we will have a single object of type `SamBatch` which contains references to the "columns", i.e. our data structures holding all the information of a specific type.

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.

We will later explain how few more saves can be achieved in `DeduplicatedDictionary` to save another 2 bytes per instance in the future "Batching" section, but as a visual description, this is the data between algorithms per row just to do bookkeeping and keeping primitive values:



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:
`00001001001001000110110110110100`
`10010010001001001001000000000000`

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.

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.

These 20 references and the extra two ArrayList instances mean that there is an extra fixed overhead of: 20 (references) * 8 bytes + 2 (ArrayList) * 64 = 288 bytes which are on-top of original ElPrep's row store memory usage. In our calculation just for bookkeeping the row without tags (189 bytes) and the tags/temp costs (another 288 bytes) it translates in 477 bytes (and some rounding-up done by JVM: so we talk about 480 bytes). For smallest file (144 MB

BAM file) as it has 2.1 million rows, just for book-keeping the JVM will use 1 001 700 000 bytes. Our Compact algorithm will use much less extra, 4 bytes for length, and the indices into a table, every index which would be 2 bytes long. On average we found around 10 tags per row, and the indices would in fact store the equivalent data with the pointer: so book keeping would be on average: (length*1 + index * 10) : 4 + 2*10 = 24 bytes, a 10x saving.

For 2.1 million entries, for tags the usage is just 24 x 2.1 million = 50 400 000 bytes for tags and a combined value for `Compact` representation of: (39+24)x2.1 million = 132 300 000 bytes for book keeping.

For `Compact Chunked` the saves are noticeable as it would use 6 bytes less per row: (33 + 24) x 2.1 million = 119 700 000, around 10 percent less than `Compact`

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];
```

It may be noticed is that tag sequence is not warrantied to fit it in two bytes, but if using `Chunking/Batching`, the experimental possibility to not fit a tag into 2 bytes unique value is very unlikely: for around every 20K rows there are around 2400 unique tags and around 10 tags per row (or 200K tags in total for every 20K lines).

4 OPTIMIZING I/O OPERATIONS

When ElPrep algorithm finishes reading and creating a large data set in memory, the algorithm goes to write down the data. ElPrep will create parallel tasks, which in turn will take all SamAlignment instances and serialize them into the string format of SAM files.

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

Especially when the full data set is loaded into memory, these tasks create many small objects and many of them can be removed just by preallocating the internal buffer of StringWriter as follows: -+

```
var outputStream = alnStream
.parallel()
.map((aln) -> {
var sw = new StringWriter(350);
try (var swout = new PrintWriter(sw)) {
aln.format(swout);
}
return sw.toString();
});
```

StringWriter delegates writing to an internal StringBuilder, which in turn has an internal buffer that is resized every time when the code reaches the end of this buffer. As the memory algorithm is concerned, the buffer is doubled. Given the default size for StringBuilder is 16 bytes, when wanting to write a text of 400 characters, it would require resizes from 16-512, so 5 resizes.

350 is around 10% larger than average line (which is around 325-330 characters per row) which based on the regular statistical distribution, will mean that most lines would have no extra resizes of buffers, which creates no extra garbage which in turn makes calling the GC less often.

Changing just this pre-sized internal buffer would give a note-worthy savings and we will name this technique as the `PresizedBuffers`.

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`.

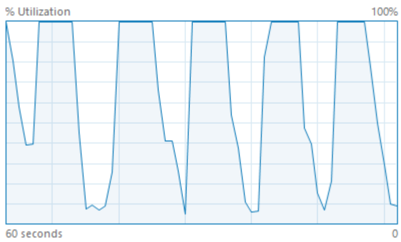
```
var out = new PrintWriter(
  new BufferedWriter(
    new OutputStreamWriter(output)));
...
outputStream.forEachOrdered(
  (s) -> out.println(s));
```

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.

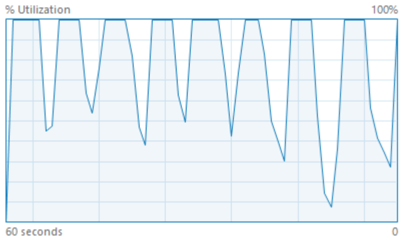


The graph looks like a chainsaw, which is a sign that the code is bottle-necked on a single thread. This is no surprise, since the `outputStream` is written in a sequential fashion. On top of that, for every row there has to be a mutex check whenever a synchronized block of code is invoked. Since there 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:



Here is the execution profile when the Original algorithm has plenty of memory and the "Chainsaw" in execution is in fact that we ran for 50 times the algorithm, and every time the CPU jumps from original 12 core usage when reading to back to 1 Core used and back. This happens only when executing on "big machines" (when the RAM available is much more than the live set, like 3-4 times at least the size)

4.1 CHUNKING-BATCHING AND EXTRACTING PARALLELISM

In the elPrep original algorithm, a `SamAlignment` object is created for each row in the input file. In order to extract parallelism, there is an explicit `.parallel` () call, a construct that would create a task for each row that must be processed. These tasks are queued and executed in a concurrent fashion using threads created transparently by the Java Streams Framework.

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

In the column-based model, described in section 3.2, there is a single `SamBatch` instance storing all the data. Because we perform data compaction, the *read* operation must take into account the dependencies between rows. Just like in the case of removing duplicates, in order to process a new row we have to inspect the values of the previously read rows. This means that the reading process cannot be fully parallel per-row. In order to obtain a better performance than reading using a single thread, we propose a technique that splits the `SamBatch` structure in several ”chunks”. Instead of having a single large object, we will represent the data using an array of smaller `SamBatch` objects. A sketch of our *Compact* algorithm is described by the following pseudo code:

```
int chunkSize = 20000;
int nbOfThreads = cpuCoreCount * 4;
var samBatches = new ArrayList<SamBatch>();
while (!endOfFile) {
  var rowsRead = parallelReadRows(
    batchSize, nbOfThreads);
  parallelTransformRowsIntoBatches(
    samBatches, rowsRead);
}
```

The `readRows` method reads *chunkSize* $*$ $nbOfThreads$ rows out of the SAM file for the next processing step. The actual reading is done using an appropriate number of threads (based on the CPU configuration), each thread reading sequentially a fixed number of rows, calculated taking into account the nature of the information being encoded. Having the data split into chunks, we can process in parallel the mapping between the associated text and the `SamBatch` data structure, where we perform data compaction and deduplication. The techinque of grouping similar tasks requiring the same resources in order to streamline their completion is called *batching* .

In our case, the advantages of the chunking-batching approach are multiple:

- Transforming data, deduplicating and DNA encoding are all executed in a single-threaded manner, which is much easier to debug and understand than a multithreaded equivalent.
- Since a `DeduplicatedDictionary` will now have less than 20000 unique strings, the values needed to encode the strings could be represented on 2 bytes, instead of 4. Similarly, we can reduce the tag size from 4 to 2 bytes.
- After the algorithm is fully executed, we will have instead of 2.1 million `SamAlignment` objects, about 105 `SamBatch` instances, each of them having around 4 orders of magnitude less objects overall. This translates into 2 orders of magnitude less objects.
- When creating the output file, the `SamBatch` array could be processed in parallel, with no blocking except the actual operation of writing to disk.

We have seen that the column-based model saves memory at the expense of the running time. This optimization, however, reduces the overall execution time of the read operation, which is now on par with the original implementation.

When it comes to writing, compared to our `StreamByteWriter` class, which is anyway much faster than the original implementation, the execution time is drastically reduced from 28 seconds (for the 12 GB BAM file) to 12 seconds. Preparing the strings that are to be written in the file can be done in almost perfect parallelism, using the available cores. The only limitation remains the speed of the output device, which can vary depending on its type: in memory virtual partition, SSD, HD, etc.

In order to make sure there are no dead times when using the external device, we have also implemented the *async/await* pattern. This allows the program to perform in advance reading operations, using a dedicated thread, while waiting for the data processing threads to complete their execution. This new algorithm, called *Compact/Par* , offers a small improvement in the running time, as we will see in the next section, but with the disadvantage of a significant increase in code complexity.

5 EXPERIMENTAL RESULTS

5.1 OVERVIEW

We have created five implementations that address the most expensive parts of the elPrep algorithm, which are reading, writing and storing all data in the memory. Except for the original version, which was taken from elPrep public repository, all other algorithm implementations contain various types of optimization that are meant to improve runtime performance and to lower the memory usage, especially on large files where GC becomes a limiting factor. We recap the names of the algorithms as they are used in the following sections:

- *Original*, the original algorithm described in (Costanza et al., 2019);
- *PresizedBuffers* reduces the number of memory allocations by carefully constructing the data structures that hold the data to be written, taking into consideration the specific size of the file rows;
- *StreamByteArray* transforms string into bytes and writes them directly into an `OutputStream`, reducing to almost zero the number of memory allocations related to writing.

- *Compact* employs a more elaborate column-based model for storing the data set in a "compact" form, instead of the simple row-based approach used in the previous two algorithms; this reduces drastically the memory usage at the expense of running time.
- *Compact/Par* represents a modified variant of the *Compact* algorithm aimed at reducing to a minimum the dead times regarding I/O operations.

The computer we have used in order to perform the experiments is a Ryzen 9-3900X, having 12 cores and using 48 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 processor 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 show only the *Compact* algorithm but we will make some inferences over the scaling of the original algorithm across file sizes and cores.

Since the elPrep algorithm is designed to run everything in memory, we tuned the JVM heap size (using the `-Xmx` flag) to the maximum value allowed by the operating system.

In order to analyze the performance of read/write operations, we made sure that no background OS services are running during our tests, by manually stopping them.

5.2 RUNTIME PERFORMANCE

The following table shows a brief comparison of the running times obtained by our algorithms, in three configurations: 144 MB file using 4 cores and 12 cores, and 1.2 GB file using 12 cores.

Running time per algorithm in seconds

	144 MB (4c)	144 MB (12c)	1.2 GB (12c)
Original	9.13	3.938	123.91
PresizedBuffers	8.98	4.19	75.1
StreamByteArray	8.09	3.43	64.62
Compact	5.64	3.4	34.5
Compact/Par	5.42	4.68	26.7

Comparing 4 cores to 12 cores, we notice that the original algorithm scales with a factor of 2.3, *PresizedBuffers* by a factor of 2.14, *StreamByteArray* scales by 2.36 and *Compact* would scale by 1.69. So, at least for the small file, it seems that using larger machines will offer a better performance. It is important to notice that, in its current implementation, the *Compact* algorithm has an explicit sequential part that reduces its scalability. Some potential fixes are described in section named "Future Work" where are described some techniques to make the sequential part more lightweight.

Considering the file size, the ElPrep algorithm uses a lot of memory and, since the JVM has to call more often the Garbage Collector, the cost of GC visibly affects the executing time, affecting its scalability. Our techniques of reducing object allocations presented in *PresizedBuffers* and *StreamByteArray* algorithms pay off now, their runtime profiles are much better than the original. Since the *Compact* algorithm reduces drastically the memory consumption, it is not affected too much by GC and the overall running time is significantly superior. For an increase in file size 8 times, the original algorithm would slow down 31.46 times, the *PresizedBuffers* will slow down 17.92 times, *StreamByteArray* will slow down 18.83 times, and the *Compact* algorithm will have an almost perfect 8.33 times slow down.

For the 8 GB BAM file, we could only execute the *Compact* algorithm, as the others ElPrep derived algorithms would produce an `OutOfMemoryError`, even when setting the heap size up-to 42 GB. The execution time was around 215 seconds, so as we are reaching the limits of the machine, for a file that is approximately 5 times larger, the algorithm slows down by a factor of 12. When moving the input file to an SSD, which offers a faster I/O, the running time drops to 139.8 seconds, which means a 10 times slow down.

5.3 MEMORY USAGE

To measure the live data set, we have used the standard instrumentation tools of the JVM and Java VisualVM (VisualVM, 2019), which provides a visual interface for profiling a running application. Using VisualVM, we have analyzed the memory consumption in each scenario and we have estimated the minimum amount of memory that JVM requires in order to load a specific data set.

Unlike the original paper, which measures process memory size, we have measured live data size. This is possible due to VisualVM which offers very precise information regarding the objects consuming memory. It is also important to understand that the more live memory is used, and more complex the graph of objects is and the less free memory exists on a particular machine, the GC impact will be higher. This is because the GC algorithm is triggered when a critical amount of the heap is occupied. Therefore, the more memory JVM has access to, the more seldom GC will execute. The default *Garbage First Garbage Collector (G1 GC)* is automatically triggered at 45% memory occupancy (this value being configurable using the `InitiatingHeapOccupancyPercent` parameter). When the heap is arriving at this level, JVM will spawn background threads in order to do the marking phase, and this affects the running time. Having a larger machine with double memory will trigger the GC at least 2 times more seldom and although the number of objects to be removed is the same, the overall time necessary for this operation is improved.

Memory usage per algorithm in MB

	144 MB file	1.2 GB file
Original	2326 MB	32025 MB
PreSizedBuffers	2326 MB	32025 MB
StreamByteArray	2275 MB	31463 MB
Compact/Par	606 MB	4689 MB

The peak usage was measured by suspending the program at the moment when the whole file was read. We notice that for the 1.2 GB BAM file, the *Original* and *Presized Buffers* algorithms are using around 32 GB of memory. In order to offer this amount of memory to JVM we used a machine with 48 GB of RAM. To further reduce the overhead of GC, 64 GB would certainly have been better.

On a 48 GB machine, *Compact* algorithms can process larger files: the 8 GB BAM file uses 24283 MB of live data and the 12 GB BAM input uses 35265 MB.

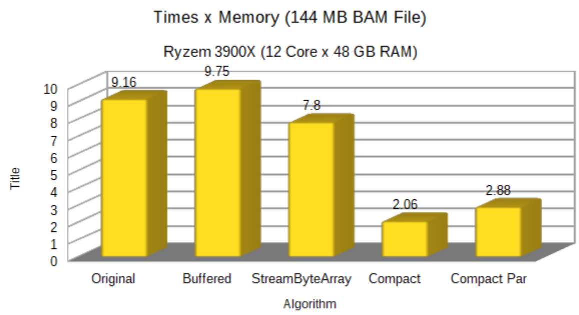
We also have to note that not all of the used memory represents data related to the input file. For example, in case of the original elPrep algorithm, as it creates millions of tasks even for the smallest 144 MB BAM file, Java Streams library will create a queue of objects that will remain in memory very likely until the end. This might add maybe around 100 MB of memory, as it is quite hard (if not impossible) to measure it.

5.4 CALCULATING PERFORMANCE × MEMORY

The goal of elPrep was to keep both the running time and the the memory consumption low. The evaluation function was defined as the multiplication of the average elapsed wall-clock time (in hours) with the average maximum memory use (in GB), with lower values (in GBh) being better (Costanza et al., 2019). We have used the same approach, changing only the measurement units to MB and seconds.

In order to analyze the impact of the hardware to the performance of algorithm, we have performed the tests on two distinct machines: the Ryzem 3900X (12 cores and 48 GB RAM) and a laptop with 4 cores and 16 GB RAM.

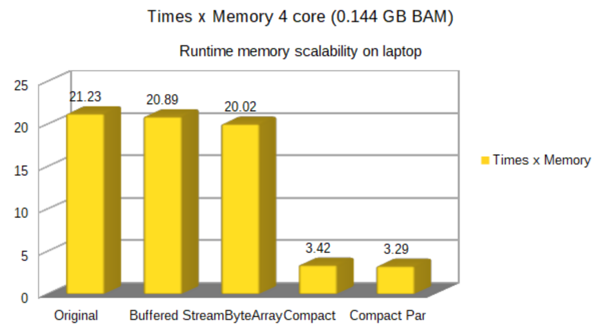
The results for the 144 MB file are presented below (lower bars are better):



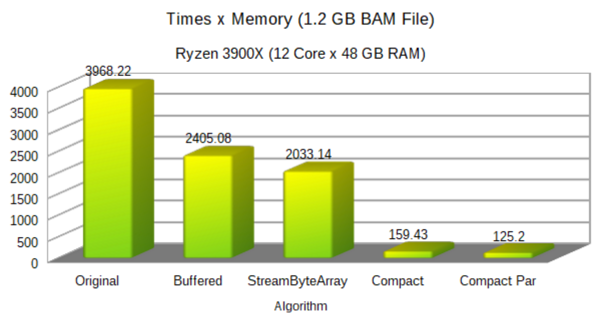
As the used memory is quite similar between the original algorithm, *PresizedBuffers* and *StreamByteArray*, their comparative performance is influenced only by the runtime savings. The *Compact* algorithms are on par from the point of view of the running time, but since they are using far less memory their performance is much better.

We observe the combination of small size of the file (meaning the algorithm will run just in seconds) and the high core count (12 cores) have a negative impact on the more elaborate algorithm *Compact/Par* which has a small slow-down compared to the more simple *Compact* implementation.

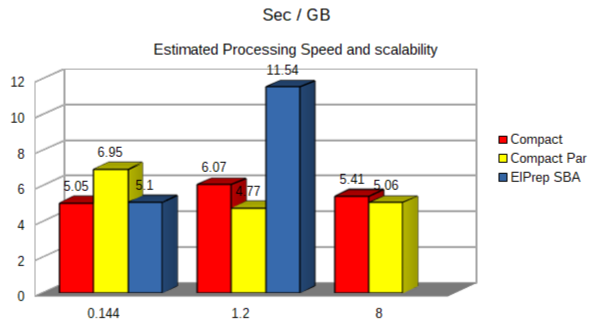
On the laptop (which has a better SSD), the *Compact* algorithm performs even better, since it could not fully exploit the high number of cores in the first test:



On the medium file (1.2 GB BAM) the values are more conclusive from the point of view of scaling the results. The improvements resulting from our optimization techniques are now clearly visible.



For the larger files, since we could only run the *Compact* and *Compact/Par* algorithms, and for ElPrep algorithm, we can't make a direct comparison between all algorithms, we can estimate scalability given the files we could run. To not penalize ElPrep, we used StreamByteArray version of the algorithm, which should scale better than the original elPrep. However, tests performed up-to the 8 GB file showed that they scale as expected, GC does not become a large impediment for both Compact algorithms and the overall performance seems limited only by other hardware components, such as disk drive read/write speeds. Both Compact algorithms keep a steady pace or processing at around 180 MB/second (or under 6 seconds to process every 1 GB of BAM). ElPrep (StreamByteArray version) has a sharp loss of speed as GC is triggered more often, processing data at the half of speed when increasing the BAM file size from 0.144 GB to 1.2 GB. The average values, including both read and write, are presented below:



6 FUTURE WORK

Batch reader scalability

As we have described in section 4.1, our *batch reader* works in two steps: initially, on the main thread, it extracts from the original file the rows for the number of the expected batches, and then it executes in parallel, using all cores, the data compaction step. Before chunking and batching is done, we split the full byte array read from file into distinct `List<byte[]>` instances. This separation may not be necessary, an alternate approach being to store inside a large `byte[]` structure all the information and to use an additional array of indices in order to retrieve the actual lines of text. This would reduce the number of allocations and eventually speed up the execution of the main thread.

A similar variation of speeding up the single threaded part is to not do the line splitting at all, but to read a block of text in advance, that would be around the expected chunk size, and then split it in lines in a multithreaded way.

Value Types

When Java specifications were elaborated, more than 25 years ago, the cost of a retrieving an object from the memory and executing an arithmetic operation was approximately the same. On modern hardware however, the memory fetch operations are up to 1000 times more expensive than arithmetic ones. This is why, the *Project Valhalla* (Valhalla, 2019), that is expected to be integrated in modern JDK releases, introduces new data structures and language constructs that improve various aspects regarding data manipulation. For example, *Value Types* provide the necessary infrastructure for working with immutable and reference-free objects. In our context, this would allow us to further reduce the memory used by the *Compact* algorithm by using an efficient by-value computation with non-primitive types.

7 CONCLUSION

This paper addresses the situation when one has to manipulate a large textual data set by reading it from a file, transforming it into objects, processing it and then writing it back to a file, and all these operations must be performed in a single in-memory sessions. We have analyzed a modern implementation of an algorithm for processing SAM and BAM files, elPrep (Herzeel et al., 2015), (Herzeel et al., 2019), which must handle input files up to 100 GB. The conclusion of the elPrep authors was that a Java implementation for this specific problem suffers from the memory management offered by JVM (Costanza et al., 2019). However, when using an object-oriented programming platform, one has to take into consideration all aspects regarding memory allocation offered by that specific platform and to adapt its model accordingly.

We have showed that major improvements can be obtained by using techniques that are aimed at reducing the number of created objects. This will not only save memory but it will also improve runtime performance by decreasing the overhead of the Garbage Collector. Using a column-based representation we have compacted the data set in a manner that boosted the overall score calculated as the multiplication between used memory and running time. The penalty incurred by the more elaborate data model was compensated by a multithreaded approach, called chunking-batching, that actually allows the algorithm to use all the available machine cores when processing the input file. In order to optimize the usage of the I/O device, we have also implemented the *async/await* pattern, which offered a small increases in performance.

Given the hardware differences between the machine used by the elPrep authors and ours, there are limits on the testing that could be done with the techniques used by this paper. Instead of a 384 GB of memory, dual-CPU server, we have used a machine with 8 times less RAM, half of CPU cores, a much slower disk drive (the SSD on the Ryzen machine can barely read at 300 MB/s, while a NVMe SSD can reach 32 GB/s) and so on. However, using input files ranging in size from 144 MB to 12 GB, we have proved that our algorithms are scalable and could perform as expected for files of any size, provided the machine has sufficient memory.

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.

GC, O. (2019). Java Garbage Collection Basics. https://www.oracle.com/webfolder/technetwork/ tutorials/obe/java/gc01/index.html. Accessed: 2019-06-01.

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 Reumers, J. (2015). elprep: High-performance preparation of sequence alignment/map files for variant calling. *PloS one*, 10:e0132868.

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.

Valhalla, P. (2019). OpenJDK Project Valhalla. https://openjdk.java.net/projects/valhalla/. Accessed: 2019-06-01.

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

VisualVM, O. (2019). Java VisualVM. https://docs.oracle.com/javase/8/docs/technotes/ guides/visualvm/index.html. Accessed: 2019-06-01.