Aalto University
School of Science
Degree Program of Computer Science and Engineering

Interactivity for Big Data: Preprocessing genomic data with MapReduce

Bachelor's Thesis

May 4, 2011

Matti Niemenmaa

Aalto University
School of Science
Degree Program of Computer Science and Engineering

ABSTRACT OF
BACHELOR'S THESIS

Author:	Matti Niemenmaa
Title of thesis:	Interactivity for Big Data: Preprocessing genomic data with MapReduce
Date:	May 4, 2011
Pages:	17
Major:	Theoretical computer science
Code:	IL3010
Supervisor:	Professor (pro tem) Tomi Janhunen
Instructor:	Professor Keijo Heljanko (Department of Information and Computer Science)

Next-generation sequencing projects are generating vast amounts of genomic data. It is impractical to analyse these several-terabyte datasets without leveraging *cloud computing*. Interactive applications such as interactive visualization, in which latency needs to be minimized, are particularly affected by the dataset size. A cloud-hosted backend, though providing the computational power necessary, brings latency issues of its own.

This Thesis explains how the interactive zooming feature of the Chipster data analysis and visualization platform can be made performant on large datasets by using genome data *preprocessing* in the cloud. The implementation of a summarizing tool and its supporting library, `hadoop-bam`, is described. The programming model used, MapReduce, is explained, as well as some details concerning the Hadoop framework on which the tools are built. In particular, a heuristic approach to splitting the genomic data files for distributed processing is presented and compared to an indexing-based strategy.

Finally, experimental timings are shown: notably, a 50 gigabyte dataset can be summarized in well under an hour using only eight worker nodes. In addition, the heuristic splitting method is found to perform comparably to indexing without incurring the additional cost of computing the index.

Keywords:	BAM, Chipster, cloud, Hadoop, interactive
Language:	English

Contents

Abbreviations

BAM	Binary Alignment/Map
BGZF	`gzip`-compatible binary compression format used for BAM
DNA	deoxyribonucleic acid
ID	identifier
I/O	input/output
HDFS	Hadoop Distributed File System
GATK	Genome Analysis Toolkit
NFS	network file system
NGS	next-generation sequencing
RNA	ribonucleic acid
SAM	Sequence Alignment/Map

1 Introduction

In recent years, with the advent of the genome sequencing technology known as *next-generation sequencing* (NGS), the rate at which genomic information is being generated has begun to grow too quickly for electronic storage to keep up. Given that the datasets' sizes are often measured in terabytes, it is too inefficient to just store them somewhere on the Internet for users to download: there is simply too much data for this to be feasible. [Ste10]

This is a so-called *Big Data* problem. In some cases, such as NGS, storage capacity has trouble keeping up with data generation. But even when this is not the case, hard disk access speeds are insufficient when compared to the storage capacity. Nowadays, one must wait hours just to read all of the data from a single hard disk [Whi09, p. 2–3]. This means that performing any kind of analysis on Big Data is a nontrivial task.

A solution to this issue is the model known as *cloud computing*: the computation is brought to the data instead of the other way around. More generally, the definition of cloud computing in this Thesis refers to server-side computing and *scaling out* as opposed to scaling up: using more computers instead of more powerful computers [BH09]. The practicalities of how this is arranged, i.e. renting of virtual machines, web-based user interfaces, etc., are outside the scope of this Thesis, but the idea is the same in all of them. Huge amounts of data no longer have to be redundantly transported between locations; one can instead perform the analyses one requires remotely.

The other benefit of cloud computing is the ability to access computer clusters, which can provide computational power far exceeding that of desktops. Given that the datasets produced by NGS, like Big Data in general, can easily be too large to fit on a single modern hard disk[1], it is clear that performing any kind of analysis on them is a time-consuming process: recall that scanning through the contents of just one disk can take hours. Because cloud computing clusters can access the dataset from multiple disks at once, the process is much more efficient.

How much interactive applications stand to benefit from this kind of architecture is not so clear. Certainly, avoiding local storage of large datasets is important, but remote access, in addition to the typically bandwidth-oriented designs of distributed computing frameworks, can increase the latency of operations greatly. And, especially as latency is improving much slower than bandwidth [Pat04], latency is a highly relevant concern in interactivity.

The Chipster[2] data analysis and visualization platform is one such interactive application. Among its various features, it provides a graphical user interface that can be used to visualize genomic data.

In particular, this Thesis concerns the zooming feature of Chipster. With it, one can transition from viewing an entire genome at once all the way down to the nucleotide level. Certain areas of the genome have typically been sequenced more than once: a higher concentration of sequences implies that the area is of relatively greater interest. At outer zoom levels one can, and need, only see precisely the concentration, so that one can focus one's attention to these areas. This kind of zooming always worked well for small datasets,

[1] Two-terabyte hard disks are the current high-end norm.

[2] `http://chipster.csc.fi/`, last fetched May 4, 2011.

but prior to the solutions presented here it tended to rapidly become unresponsive when confronted with large amounts of NGS data. This is a situation in which cloud computing is not of direct help: even if the calculations are performed in the cloud, the round-trip latency is prohibitively high.

The problem can be solved by *preprocessing* the data: simplifying it in such a way that the result is as visually indistinguishable from the original as possible, but takes up much less storage space. This separates the visualizing frontend and user interface from the backend that performs the heavy computation required for visualization. Now the best of both worlds is achieved: the frontend can be highly interactive while still being relatively lightweight, with reasonable system requirements, while delivering sufficient performance for handling huge datasets.

This work details the implementation of a summarizing tool enabling interactive zooming of large NGS datasets, as well as the supporting library of the tool, `hadoop-bam`[3]. The distributed programming model applied, MapReduce, is also explained.

2 Related work

In recent years the MapReduce programming model, and in particular the Hadoop[4] MapReduce framework which is also used by `hadoop-bam`, has been harnessed for analysing NGS data in many ways. In this section, some of the approaches that are relatively closely related to `hadoop-bam` and the summarizing tool are detailed.

The Genome Analysis Toolkit (GATK) [MHB+10] is a programming framework for NGS analysis tools using MapReduce. It offers a rich set of methods for crafting high-level solutions to analysis problems. In contrast, `hadoop-bam` is a more low-level approach, providing only the most basic support for accessing NGS data. The GATK's options for distribution have various limitations and full distribution is currently in an experimental stage[5]. `hadoop-bam` can split data up more freely, without placing additional restrictions on the program's behaviour.

The SeqWare Query Engine [OMN10] is a database system built on top of HBase[6]. Accessing the database with MapReduce is a promising and performant way of analysing NGS data. This method was deemed unnecessarily complicated for the summarizing tool, which has no need for the features of the database.

Several other NGS-related applications of MapReduce and Hadoop exist; there are too many to even list here. The GATK and SeqWare and other similar approaches were two primary candidates for the summarizing tool before it was decided that making a new library, `hadoop-bam`, is appropriate. For a far more comprehensive overview of Hadoop usage in bioinformatics, see e.g. [Tay10].

[3] `http://sourceforge.net/projects/hadoop-bam/`, last fetched May 4, 2011.

[4] `http://hadoop.apache.org/`, last fetched May 4, 2011.

[5] See e.g. `http://www.broadinstitute.org/gsa/wiki/index.php/Parallelism_and_the_GATK`, last fetched May 4, 2011.

[6] `http://hbase.apache.org/`, last fetched May 4, 2011.

3 Problem description

The following section describes the constraints on the inputs and outputs of `hadoop-bam` and the summarizing tool. Thereafter the method used to produce the desired output will be described, without delving into implementation details.

3.1 Goal

The summarizing tool and `hadoop-bam` work on BAM (Binary Alignment/Map) files. BAM is the binary version of the textual SAM (Sequence Alignment/Map) file format. Both formats encode, along with some metadata, a number of *reads* a.k.a. genetic *sequences*: the compositions of nucleic acid molecules such as DNA (deoxyribonucleic acid) or RNA (ribonucleic acid). Often they are also referred to as *alignments*, as they are typically utilized in a procedure known as sequence alignment: comparing two or more sequences in an effort to find similar regions. [Mou01, LHW+09, SAM11] BAM files are stored in the BGZF compression format: a BGZF archive is composed of `gzip`-compatible blocks, providing good compression and efficient random access [SAM11].

The goal of the summarizing tool is to allow rapid computation of a zoomed-out view of the reads in a BAM file. To that end it preprocesses the file, creating "summary files" which hold sufficient information to describe the zoomed-out view of the original. Since, at these outer zoom levels, the only visible information is the number of sequences encompassing a given area, this is exactly the data that needs to be stored.

Summary files are in a BGZF-compressed line-based textual format, with each line consisting of four integers separated by horizontal tab characters[7]. The most important three fields are the last three: the leftmost and rightmost coordinates of an area and the number of reads that are in that area.

The first datum is the reference sequence identifier (ID). Each BAM file can contain, for example, reads from different species. For differentiation, they are tagged with the reference sequence, which is typically a complete genome sequence of the species in question [SAM11]. The ID needs to be carried through to the summary file to make sure that the visualization can also differentiate between them.

3.2 Method

A simple way of producing the summarized areas is to traverse reads tagged with the same reference sequence ID, grouping together reads at the same location, and outputting the location and the number of reads found there. Thus, for example, two reads with ID 0 at coordinates $[5, 15]$ would result in `0␣5␣15␣2` (where "␣" represents the horizontal tab character).

In practice, reads at the same general location are very rarely in exactly the same place. Instead of two reads at $[5, 15]$ it's more likely to have e.g. $[4, 15]$ and $[7, 16]$. To deal with this, the summarizer computes the mean range encompassed by the reads and outputs that as the summarized area. For the example, $[\lfloor (4+7)/2 \rfloor, \lfloor (15+16)/2 \rfloor] = [\lfloor 5.5 \rfloor, \lfloor 15.5 \rfloor] =$

[7] This precise format, including the order of the fields, was chosen because it is the format supported by the `tabix` tool (`http://samtools.sourceforge.net/tabix.shtml`, last fetched May 4, 2011).

$[5, 15]$. This introduces some lossiness into the summarizing process: one cannot know where the $[5, 15]$ came from, only that there were two reads at approximately that location.

Computing means also introduces a question: what reads should be grouped together? For example, $[0, 10]$ and $[1000000, 1000020]$ result in $[500000, 500015]$. In practice, BAM files are very dense: consecutive reads are not separated by such a long distance. Therefore, if the reads are first sorted by their position (rather, first by the reference sequence ID and then the position), this kind of pathological situation can be avoided.

However, sorting by position can be done in more than one way. Reads can be of varying length: while unlikely, it is not unheard of to have, for instance, $[0, 10]$ and $[0, 1000]$ in the same BAM file for the same reference sequence. Thus sorting by the leftmost position may result in poor approximations of the underlying data. A significant improvement is achieved by sorting by the centre of mass i.e. the mean of the start and end coordinates: this brings reads like $[0, 10]$ and $[1, 11]$ close together, while $[0, 1000]$ may get grouped with $[500, 510]$.

The remaining issue is that of the group size: how many reads should be summarized together? In the visualizer, as one zooms further and further inwards, one needs more detailed information i.e. a smaller group size. The tool lets the user decide which sizes they want: in practice, increasing powers of two until the summary file is "very small" (a few kilobytes) gives a sufficient spread of sizes for the visualizer to choose from so that it can display the sequences sufficiently accurately and quickly.

In summary, the task is to:

1. Extract the coordinates and reference sequence IDs of each read from the given BAM file.
2. Sort the resulting records first by their ID and second by their centre of mass.
3. For each consecutive group of records of size at most N with the same ID, output their ID, mean position, and the group size. N here is the user-requested group size. Some groups may have size less than N: for example, if $N = 4$ and there are 5 reads with ID 0 in the BAM file, the second group will have size 1.

4 Solution

Knowing the required tasks established in the previous section, it is now necessary to determine how they can be performed quickly in a distributed fashion. This section details how the summarizing tool and `hadoop-bam` achieve the requirements.

The foundation used to facilitate distributed computation in all stages of the solution is the Hadoop MapReduce framework. MapReduce is fully described in [DG04, DG08]. The following section summarizes the model and explains how it works without going into excessive detail.

4.1 MapReduce

MapReduce is a programming model based on two functions specified by the user: a *map* function, which takes a key-value pair and transforms it into a list of intermediate

key-value pairs, and a *reduce* function whose job it is to merge intermediate values that are associated with the same key. These functions can be typed as:

$$\begin{aligned} \text{map} &: (k_1, v_1) \rightarrow (k_2, v_2)^k \\ \text{reduce} &: (k_2, v_2^m) \rightarrow v_3^n \end{aligned}$$

In the above, k is short for "key" and v for "value", and the subscripts serve to differentiate the types beyond that division. The superscripts k, m, and n denote differing list lengths. v_3^n is thus the end result which gets written to the output files.

The execution model of MapReduce is a simple sequence. The following summarizes the most important parts, common to all MapReduce jobs: [Ven09, p. 178–182]

1. The input files are divided into *splits*: large files are separated into blocks that can be mapped over. This splitting enables parallelism within a file as well as across files.

2. A number of map tasks, or mappers, begin executing. The number of tasks is determined by various factors and can be anywhere from just one to several thousands. Each input split is sent to a task as soon as the mapper becomes available. Note that the file data is not sent, only a file path and beginning and ending offsets: reading the actual data can be done without any network I/O (input/output) if the mapper has a local copy of it. The map function is run across the split.

3. The output of the map function is partitioned according to which reduce task it should be sent to, and each partition is sorted. This is known also as the *shuffle* step.

4. A job-defined number of reduce tasks are started. Each reducer fetches, from all the mappers, the map outputs that are assigned to its partition, and merge-sorts them together.

5. At each reducer, the reduce function is run on the resulting set of key-value groups. When all are complete, the job is done, having produced (typically) one output file for each reduce task.

A graphical representation of the MapReduce model using the canonical example, word counting, is shown in Figure 1.

Other notable features of the MapReduce pipeline not detailed here include the system's fault tolerance and failure semantics as well as the combiner function [DG04, DG08, Ven09, Whi09].

4.2 Hadoop Distributed File System

Hadoop includes the Hadoop Distributed File System (HDFS) [SKRC10]. It has been designed with the application of MapReduce in mind, and as such they are almost always used together. `hadoop-bam` and the summarizing tool are also based around it.

Originally based on the design of the Google File System [GGL03], HDFS is a scalable distributed file system with a few key features:

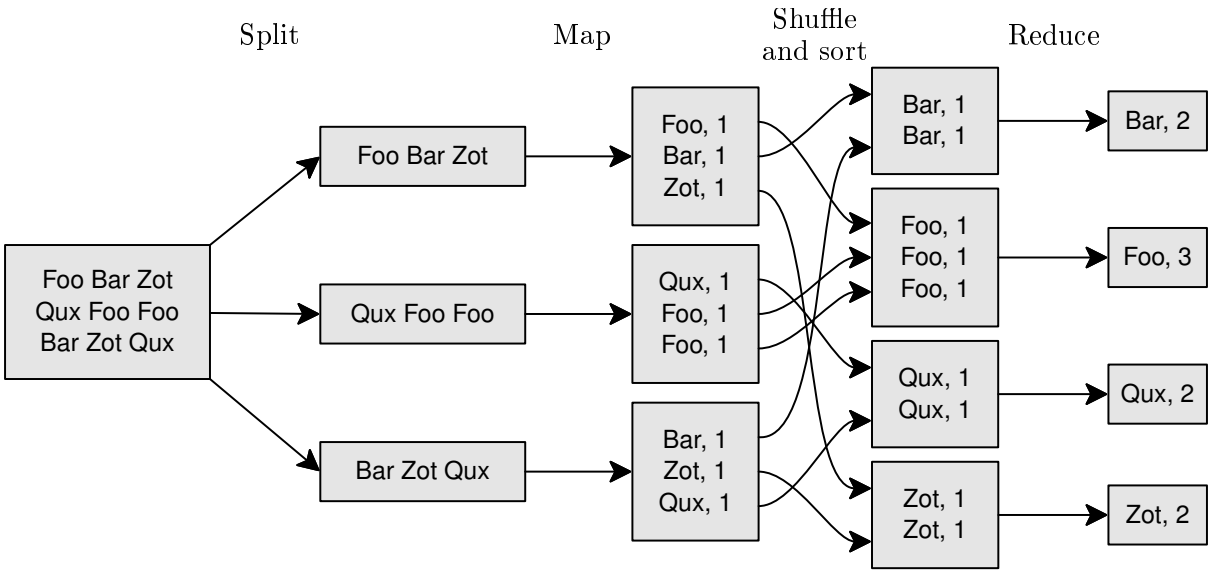

Figure 1: The MapReduce process performing a word count.

- Reliability. This is achieved primarily by *replication*: each block of data is stored on multiple hosts. By default, all data is stored on three different machines.
- Being tuned for batch processing of large files. In accordance with this assumption, files can be written only once: when closed after creation, they are immutable. File readers typically use linear instead of random access, and files tend to be very large, so the HDFS default block size is 64 megabytes: this aids in providing high bandwidth streaming of the data.
- Interaction with the Hadoop MapReduce system. Hadoop attempts to schedule both map and reduce tasks in such a way that network I/O is minimized. HDFS can inform Hadoop of the physical locations of the blocks that the task needs to access [DG04, Ven09]. Note that as a side effect of replication for reliability, it is more likely that a task can be scheduled where a copy of the data resides.

4.3 `hadoop-bam`

MapReduce is a good fit for the summarizing task because it is naturally expressed as a map followed by a reduce on sorted key-value pairs. Clearly, extracting the coordinates and reference sequence IDs of each read is a map function and grouping consecutive ranges together is a reduce function. Sorting is also provided "for free" by the MapReduce execution process, and thus all the needed operations are cleanly supported.

Practical considerations apply as well, of course: Hadoop is a mature framework, known to perform and function well. It has been used by large companies such as Yahoo! and Facebook with clusters composed of thousands of servers and accessing petabytes of data [SKRC10, TSA+10, Whi09]. Therefore one does not need to worry about running into show-stopping issues, although a great deal of configuration can be necessary to achieve desired performance levels [SRC10, Ven09, ZKJ+08].

Before going into details of the summarizer, the following section explains `hadoop-bam`'s primary contribution to the solution of the summarizing task: the custom splitting

function. Basic input and output of BAM files is provided by the Picard library[8] and is thus not a concern here.

4.3.1 Splitting

Hadoop's default file splitting simply divides the input evenly into parts, each part having approximately the same byte length. Due to the nature of the input format, this cannot be relied upon: having a record-oriented file be split along the middle of a record is problematic, since then that record cannot be handled on either side of the split. Typically, it is possible to work around the issue using a simple technique shown in Algorithm 1.

Algorithm 1 Typical way of reading records from a part of a split file.

```
1: pos ← 0
2: if this is not the first split then
3:     skip input until the beginning of a record
4:     pos ← pos + amount of data skipped
5: end if
6: while pos < end of this split do
7:     r ← record at pos
8:     handle r
9:     pos ← pos + length(r)
10: end while
```

Unfortunately, for BAM files the implementation of line 3 is somewhat complex due to the binary format and the BGZF compression applied on top of it. Two stages of heuristic guesswork are required: one must find, first, the BGZF block containing the position where the split begins; and second, the beginning of the next alignment.

The first task is easier: BGZF does have, at the start of each block, four bytes with guaranteed values as well as more later on, as can be seen from Table 1. Note that the two magic numbers are composed of multiple shorter fields, but they can be considered as units for the purposes of `hadoop-bam`. Recognizing a BGZF block using solely these numbers would unfortunately not work, since nothing prevents a sequence of bytes conforming to these requirements from showing up within the compressed data as well: there is a low probability of treating unrelated data as a BGZF block. Practically speaking, the likelihood of just finding the identifier bits is very low, let alone an otherwise valid-looking block with a correct CRC-32 hash of the uncompressed contents! Even in this ridiculously unlikely situation, the probability of treating the input incorrectly can be further reduced: when the "block" eventually terminates, it is most likely not followed by data that can be again interpreted as a valid BGZF block. Upon noticing this, one can backtrack past the misleading data and search for the next BGZF block.

The method of determining whether an arbitrary byte sequence appears to be a valid BGZF block, based on the information in Table 1, is presented in Algorithm 2. The CRC-32 hash is not checked at this guessing stage, since that would involve unpacking the data and thus is a relatively expensive operation. Instead, the check can be performed later, when the data is actually used.

[8] `http://picard.sourceforge.net/`, last fetched May 4, 2011.

Description	Type	Value
BGZF block magic number	`uint32`	0x04088b1f
Modification time	`uint32`	
Extra flags	`uint8`	bit 2 is set
Operating system identifier	`uint8`	
Total length of extra subfields (XLEN)	`uint16`	at least 6
Extra subfields		
Other extra subfields		
BGZF extra field magic number	`uint16`	0x4342
BGZF extra field length	`uint16`	2
Total block size minus 1 (BSIZE)	`uint16`	
Other extra subfields		
Compressed data	`uint8[`BSIZE−XLEN−`19]`	
CRC-32 hash of the uncompressed data	`uint32`	
Length of the uncompressed data	`uint32`	

Table 1: The format of one block in the BGZF format. All integers are little-endian. [SAM11]

Algorithm 2 Guessing whether a BGZF block starts at the given position.

Input: *bpos*, the position to examine

1: **if** read($bpos, 4$) $\neq$ 0x04088b1f **then** {Incorrect magic number: not a BGZF block.}
2: **return false**
3: **end if**
4: $subpos \leftarrow bpos + 12$ {The offset where the extra subfields begin.}
5: $subend \leftarrow subpos +$ read$(bpos + 10, 2)$ {Add the value of the XLEN field.}
6: **while** $subpos < subend$ **do**
7: $magic \leftarrow$ read$(subpos, 2)$
8: $slen \leftarrow$ read$(subpos + 2, 2)$
9: $subpos \leftarrow subpos + 4 + slen$
10: **if** $magic \neq$ 0x4342 $\vee$ $slen \neq 2$ **then** {This is not the BGZF extra field.}
11: **continue**
12: **end if**
13: **while** $subpos < subend$ **do** {Skip over the rest of the extra subfields.}
14: $slen \leftarrow$ read$(subpos + 2, 2)$
15: $subpos \leftarrow subpos + slen + 4$
16: **end while**
17: **return** $subpos = subend$ {XLEN must be exact for this to be a valid gzip block.}
18: **end while** {No BGZF extra field found.}
19: **return false**

The second issue, that of finding the next alignment, is somewhat more problematic since BAM records have no clear identifying features. Fortunately, various fields cross-reference each other enough that in practice, some guesswork succeeds.

Field name	Description	Type
`block_size`	Length of the rest of the record	`int32`
`refID`	Reference sequence ID	`int32`
`pos`	0-based coordinate	`int32`
`l_read_name`	Length of `read_name`	`uint8`
`mapq`	Mapping quality (*ignored*)	`uint8`
`bin`	Bin number (*ignored*)	`uint16`
`n_cigar_op`	Length of `cigar`	`uint16`
`flag`	Flags bit field (*ignored*)	`uint16`
`l_seq`	Length of uncompressed `seq`	`int32`
`next_refID`	Reference sequence ID of next fragment	`int32`
`next_pos`	0-based coordinate of next fragment	`int32`
`tlen`	Template length (*ignored*)	`int32`
`read_name`	Name, null-terminated	`uint8[l_read_name]`
`cigar`	CIGAR string (*ignored*)	`uint32[n_cigar_op]`
`seq`	Fragment sequence (*ignored*)	`uint8[(l_seq+1)/2]`
`qual`	Phred base probability (*ignored*)	`uint8[l_seq]`
Auxiliary data until `block_size` is filled (all *ignored*)		
`tag`	Identifier (*ignored*)	`uint8[2]`
`val_type`	Type specifier (*ignored*)	`uint8`
`value`	Value (*ignored*)	depends on `val_type`

Table 2: The format of the fields of one alignment in the BAM format [SAM11]. All integers are little-endian. Fields which are not used by the algorithms presented here are marked as *ignored*.

The following constraints hold on the fields of the BAM record format, displayed in Table 2. `n_ref` is not a field in each alignment; it is the number of reference sequences and can be found at the beginning of the BAM file.

1. $\texttt{block\_size} \geq 32 + \texttt{l\_read\_name} + 4 \cdot \texttt{n\_cigar\_op} + (3 \cdot \texttt{l\_seq} + 1)/2$
2. The reference IDs are -1 or in the range $[0, \texttt{n\_ref})$:
   $-1 \leq \texttt{refID} < \texttt{n\_ref} \ \wedge \ -1 \leq \texttt{next\_refID} < \texttt{n\_ref}$
3. The positions are -1 or non-negative: $\texttt{pos} \geq -1 \ \wedge \ \texttt{next\_pos} \geq -1$
4. Null-termination of `read_name`: $\texttt{read\_name}[\texttt{l\_read\_name} - 1] = 0$

By using all of these constraints together, one can detect BAM alignments with sufficient accuracy. Pseudocode for this is not given explicitly here, as it is a simple matter of reading integers at constant offsets from each other and performing the comparisons listed above.

Algorithm 3 gives a more detailed account of how the splitting can be made to work in

all its complexity, with the help of Algorithm 2 and an equivalent algorithm for BAM records based on the above constraints.

Algorithm 3 Reading BAM records from a part of a split file.

```
 1: pos ← cpos ← 0
 2: if this is not the first split then
 3:    for all pos ∈ apparent BGZF block positions in the split do
 4:       pos_0 ← pos
 5:       for all cpos ∈ apparent BAM record positions in the block at pos do
 6:          cpos_0 ← cpos
 7:          b ← 0
 8:          while pos < end of this split and b < 2 do
 9:             if the data at (pos, cpos) does not form a valid BAM record then
10:                continue at line 5 with pos_0 and next cpos
11:             end if
12:             cpos ← cpos + length(r)
13:             if cpos ≥ block size then
14:                pos ← position of next block after the one at pos
15:                cpos ← 0
16:                if the data at pos does not form a valid BGZF block then
17:                   if pos ≥ 2^16 then
18:                      input file is invalid or data corruption occurred
19:                   end if
20:                   continue at line 3 with next pos
21:                end if
22:                b ← b + 1
23:             end if
24:          end while
25:          pos ← pos_0
26:          cpos ← cpos_0
27:          goto 31
28:       end for
29:    end for
30: end if
31: while pos < end of this split do
32:    r ← BAM record at (pos, cpos)
33:    handle r
34:    advance (pos, cpos) by length(r)
35: end while
```

The bulk of the algorithm is the **while** loop on lines 8–24. Having found a partially validated BAM record, it is fed to a fully featured BAM decoder in order to verify its validity fully (lines 9–11). One can then continue looping through BAM records without any further guessing. The **if** on lines 13–23 handles advancing to the next BGZF block. Note the increment of b: b is the number of BGZF blocks that have been traversed from start to finish. The two on line 8 is the number of BGZF blocks that should be fully deciphered before accepting the appropriate location to start reading from has indeed been found. When that occurs, the **while** loop ends and the code proceeds to read records as usual, now that it knows where to start from.

On line 17, the 2^{16} is the maximum allowed compressed size of a BGZF block. This limitation can be clearly seen in Table 1: it arises due to the fact that the BSIZE field is a 16-bit unsigned integer. Since the input is fully composed only of such blocks, if the algorithm travels past that much space without finding a satisfactory block, something has clearly gone wrong.

An alternative solution to the whole issue is to use an *index*: precompute positions of reads and BGZF blocks in the BAM file. The MapReduce job would then build a search structure such as a binary search tree from it, which could be used to find the appropriate position to read from. This also works and is supported by `hadoop-bam`, but if the summarizing task only needs to be performed once, computing the index can be a waste of time.

4.4 The summarizing tool

With the BAM file split into usable chunks, one must move from the realm of `hadoop-bam` to that of the summarizing tool. The following sections discuss the remaining phases in the MapReduce execution sequence: the map function, the partitioning and sort, and the reduce function.

4.4.1 Map

The map function of the summarizer extracts the coordinates and reference sequence IDs of a read. Thus it produces, for each read, the pair of coordinates, keyed on the reference sequence ID and the centre of mass in order to be sorted properly. The resulting key is (conceptually[9]) a pair of the form `(ID,centre)`, ordered lexicographically, and the value is a `(beg,end)` pair.

This is a drastic reduction in the amount of data compared to the original read as seen in Table 2. Such a reduction means that the performance of the sort stage is greatly improved, as there is far less network traffic. As much as 90% of the data may be discarded by the mappers.

4.4.2 Partition and sort

As the mappers complete their runs over input splits, they need to partition their output i.e. assign a reduce task for each output key-value pair. Ideally, this results in a perfectly even distribution, with each reducer getting exactly $1/R$ of the map tasks' output, where R is the number of reduce tasks.

Hadoop's default partitioner for data simply uses hashing, which typically does rather well: for each key k, the partition is $\text{hash}(k) \bmod R$. This is a fast way of getting a good distribution, but it has one unfortunate side effect: the input will not be totally ordered. There is no guarantee that all the input of reducer r is less than that of reducer $r + 1$. For the purpose of summarizing, this is inadequate: the reducers should always consider reads that are globally consecutive.

[9]For performance reasons, the implementation packs this pair of 32-bit integers into one 64-bit integer.

Fortunately Hadoop provides a solution in the form of a partitioner that associates each reduce task with a range of keys that should be sent to it. For example, with three reducers and integer keys, the three ranges might be $(-\infty, 1000)$, $[1000, 5000)$, and $[5000, \infty)$. This ensures a global total ordering.

However, a new issue is introduced: how should the ranges be selected in order to get an even distribution? The contents of BAM files can vary to a great extent, so presupposing certain values is out of the question. Hadoop's answer is *sampling*: examine some records of the original input before even starting any map tasks, then base the distribution on that. For example, if $R = 2$, the two partitions would be split around the median of the sampled keys.

The choice of sampling strategy can have a noticeable effect on the performance of the MapReduce job [Ven09]. For now, the summarizer uses a sampler intended for sorted data, which examines the input at regular intervals. Since providers of BAM data tend to sort the files before publication, this seems to be a sensible choice. In the future it would be prudent to perform a comparison of the different sampling strategies provided by Hadoop, but this has not been done yet.

4.4.3 Reduce

With the mappers' output at the reducers, all that is left is to run the reduce function over the data. That is, a certain predetermined amount of consecutive ranges need to be grouped together.

The algorithm is simple: keep two sums, one for the beginning and one for the ending coordinate, a count of how many pairs have been added into the sums, and the current reference sequence ID. When the count reaches the requested amount, the input split ends, or the ID changes, divide the sums by the count to get the arithmetic means and output a summary record.

Note that this process can be done in parallel for any number of requested summary levels: an arbitrary amount of summaries can be computed in only one pass over the BAM file. This is a clear improvement over doing a separate MapReduce job for each summary.

5 Experiments

With `hadoop-bam` and the summarizer implemented, experiments were run to determine how well the task performs and, in particular, scales when distributed across several computers.

5.1 Environment

The Triton computing cluster was used as the test environment. Triton is a cluster of 112 computers or *nodes*, each of which has two six-core AMD Opteron 2435 CPUs with a clock speed of 2.6 GHz. For RAM, 32 of the nodes have 64 gigabytes of 800 MHz DDR2 SDRAM each, while the remaining 80 have 32 gigabytes each. Each node also has 250 gigabytes of local disk space available for use.

The nodes of Triton are physically split into seven enclosures holding 16 nodes each. This has some implications for inter-node bandwidth, but in both the intra-enclosure and inter-enclosure cases the InfiniBand link used has been tested to provide a TCP/IP latency of about 23 μs and a bandwidth of about 700 megabytes per second.

Hadoop version 0.20.2 and `hadoop-bam` revision `1.0-6-g03b0ae1` (a development version after 1.0) were used for the experiments. In addition, hadoop-lzo[10] 0.4.4 was used to compress the data during the sort phase. This dramatically reduces network I/O between the mappers and reducers.

5.2 Results

A 50 gigabyte BAM file[11] was summarized with 16 different group sizes: the powers of two from 2 to 65536. In the shown results, the heuristic approach for BAM file splitting was used. Using a precomputed index for splitting gives practically identical MapReduce performance, so those results are not shown. Indexing the file took about half an hour, so indexing was simply half an hour slower than heuristic splitting.

Figure 2 displays, for five different Hadoop cluster sizes, mean times to complete four tasks:

1. Importing the input BAM file from Triton's network file system (NFS) to HDFS.
2. Running the actual summarizing MapReduce job.
3. Exporting the resulting summary files from HDFS to the NFS.
4. All of the above together.

These tasks were performed four times for each cluster size: the mean and the minimum and maximum are shown in the figure.

The total time is well under an hour already with eight worker nodes. This is very reasonable for a 50 gigabyte dataset. Extrapolating linearly, a five terabyte BAM file would take around half a week to be summarized, which is still well within acceptable ranges on so few machines.

As can be seen from Figure 3, the MapReduce job scales well up to about eight worker nodes, after which scaling is minimal. This also has a significant effect on the total time: starting at the four worker mark, the MapReduce job actually takes less time than the file system transfers.

Unfortunately, the HDFS operations show zero scaling. One likely cause for this, other than bottlenecks on the NFS side of things or the Triton network, is hardware limitations on the Triton nodes: one 250 gigabyte disk is not considered sufficient even for CPU-bound MapReduce tasks run under Hadoop[12]. Thus it is to be expected that disk I/O would be a bottleneck.

[10] `https://github.com/kevinweil/hadoop-lzo`, last fetched May 4, 2011.

[11] `NA19240.chrom6.SOLID.bfast.YRI.high_coverage.20100311.bam` from the 1000 Genomes Project (`http://www.1000genomes.org/`, last fetched May 4, 2011).

[12] See e.g. `http://www.cloudera.com/blog/2010/03/clouderas-support-team-shares-some-basic-hardware-recommendations/`, last fetched May 4, 2011.

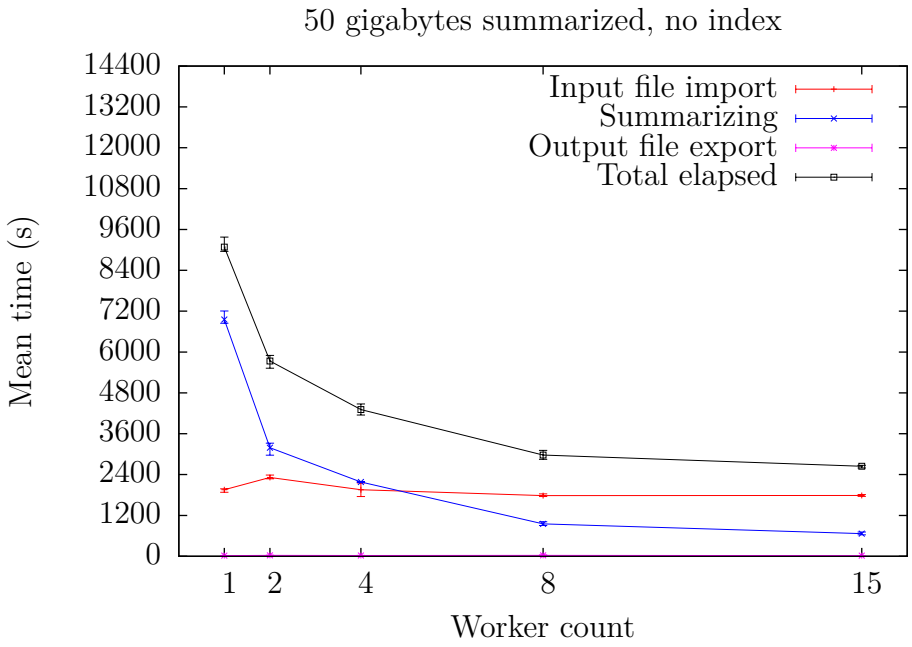

Figure 2: Mean times for the parts of the process of summarizing a 50 gigabyte BAM file with Hadoop, using heuristic splitting.

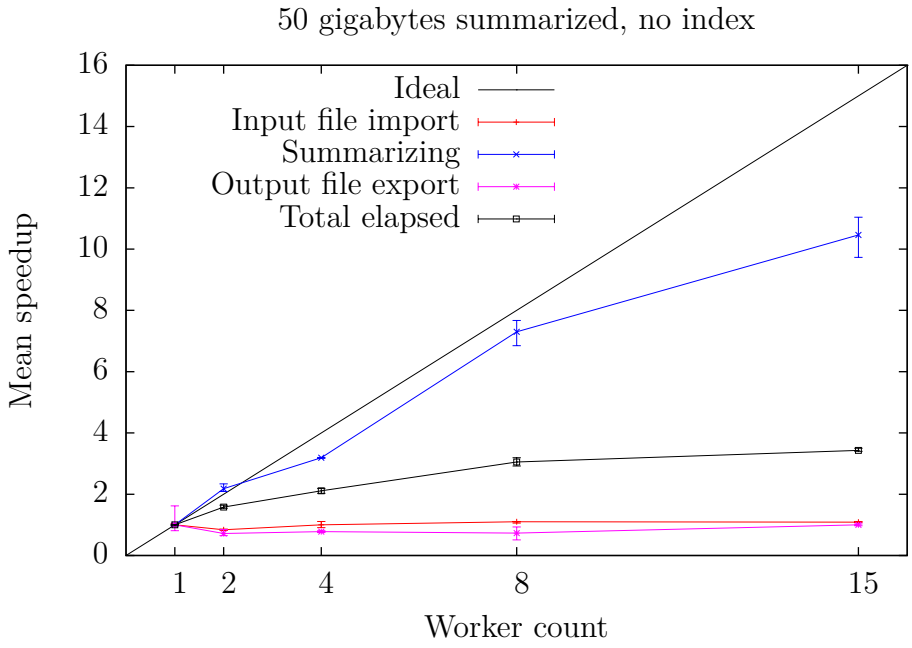

Figure 3: Mean speedups for the parts of the process of summarizing a 50 gigabyte BAM file with Hadoop, using heuristic splitting.

6 Summary and conclusions

This Thesis has presented the implementation of core features of the `hadoop-bam` library for cloud computing and a visualization-aiding tool for summarizing BAM files. The tool has been benchmarked and found to perform sufficiently well: even with a relatively cheap cluster of commodity hardware, one can expect to be able to visualize a several-hundred-gigabyte BAM file within a day or two.

Two different approaches to the subproblem of BAM file splitting were compared: using a precomputed index of sequence locations and heuristic on-the-fly calculation. The heuristic approach won out, with MapReduce performance being practically identical, but indexing incurring a noticeable additional cost.

The expensiveness of transferring large BAM files from traditional storage systems to HDFS for MapReduce processing should be avoided in some way. Two ways of achieving this are doing long-term data storage in HDFS and running Hadoop with a non-HDFS distributed file system such as Lustre or a traditional NFS. Both have various pros and cons whose investigation is outside the scope of this Thesis.

Clearly, preprocessing in the cloud is a viable way of visualizing BAM files. Cloud computing provides performance: even with a relatively low speedup factor, one can simply "throw more hardware at the problem" to reduce the time taken. Preprocessing provides interactivity: latency between the user and the computing platform is made irrelevant. By combining the two, robust methods for solving interactivity-related Big Data problems can be created.

Using `hadoop-bam` currently requires writing the specific tool such as the summarizer by hand, in a relatively low-level language such as Java. To make it more accessible, a future direction is to evaluate using simpler and higher-level Hadoop-using platforms for working with BAM files. Examples of such include Apache Pig [ORS+08] and Hive [TSJ+10].

References

[BH09] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009. doi:10.2200/S00193ED1V01Y200905CAC006.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150, 2004.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM (CACM)*, 51(1):107–113, 2008. doi:10.1145/1327452.1327492.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In Michael L. Scott and Larry L. Peterson, editors, *SOSP*, pages 29–43. ACM, 2003. doi:10.1145/945445.945450.

[LHW+09] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, August 2009. doi:10.1093/bioinformatics/btp352.

[MHB+10] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark A. DePristo. The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20(9):1297–1303, 2010. doi:10.1101/gr.107524.110.

[Mou01] David W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, first edition, 2001. ISBN 0-87969-608-7.

[OMN10] Brian O'Connor, Barry Merriman, and Stanley Nelson. SeqWare Query Engine: storing and searching sequence data in the cloud. *BMC Bioinformatics*, 11(Suppl 12):S2, 2010. doi:10.1186/1471-2105-11-S12-S2.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In Jason Tsong-Li Wang, editor, *SIGMOD Conference*, pages 1099–1110. ACM, 2008.

[Pat04] David A. Patterson. Latency lags bandwith. *Communications of the ACM (CACM)*, 47(10):71–75, 2004. doi:10.1145/1022594.1022596.

[SAM11] The SAM format specification (v1.4-r962). Technical report, The SAM Format Specification Working Group, 2011. Last fetched on May 4, 2011. Available from: http://samtools.sourceforge.net/SAM-1.4.pdf.

[SKRC10] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10, May 2010. doi:10.1109/MSST.2010.5496972.

[SRC10] J. Shafer, S. Rixner, and A.L. Cox. The Hadoop Distributed Filesystem: Balancing portability and performance. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 122–133, March 2010. doi:10.1109/ISPASS.2010.5452045.

[Ste10] Lincoln Stein. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010. doi:10.1186/gb-2010-11-5-207.

[Tay10] Ronald Taylor. An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. *BMC Bioinformatics*, 11(Suppl 12):S1, 2010. doi:10.1186/1471-2105-11-S12-S1.

[TSA+10] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at Facebook. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *SIGMOD Conference*, pages 1013–1020. ACM, 2010. doi:10.1145/1807167.1807278.

[TSJ+10] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang 0002, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive – a petabyte scale data warehouse using Hadoop. In Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras, editors, *ICDE*, pages 996–1005. IEEE, 2010.

[Ven09] Jason Venner. *Pro Hadoop*. Apress, 2009. ISBN 978-1-4302-1942-2.

[Whi09] Tom White. *Hadoop - The Definitive Guide: MapReduce for the Cloud*. O'Reilly, 2009. ISBN 978-0-596-52197-4.

[ZKJ+08] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In Richard Draves and Robbert van Renesse, editors, *8th Symposium on Operating System Design and Implementation (OSDI)*, pages 29–42. USENIX Association, 2008.