High performance identification of structural variation in human genomes

Reda Affane
Uppsala University
Reda.Affane.6180@student.uu.se

Nils Anlind
Uppsala University
nils.anlind.2483@student.uu.se

Tugce Dilen
Uppsala University
Tugce.Dilen.5143@student.uu.se

Maria Svensson
Uppsala University
masv6227@student.uu.se

ABSTRACT

The 1000 genomes project has generated alot of data that could be used to study structural variations in the human genome. Data sizes in terabytes requires distributed software to enable a reasonable computation time. In this article we generate k-mers and density map over unmapped reads with a mapped mate pair using a Spark standalone cluster. We prove that our solution is scalable in terms of BAM files and the number of nodes in the cluster. Future suggestions are given to improve fault tolerance and file size scalability.

Keywords

Genome sequencing, Big Data, Spark, Hadoop

1. INTRODUCTION

Structural variation are the cause of evolutionary events or insertion of foreign DNA by viruses. By studying the structural variation of human genomes across the globe we can gain a better understanding of human evolution, regional genomes and virus infections. By looking at inserts origin we can gain insights into how the co-existence with viruses has shaped the human genomes we see today.

To study genome data is computationally expensive due to the sheer size of the genome. The human genome is 3234 million bases(Mb) long and sequencing data includes a quality score for each letter resulting in a duplication of file sizes. Low coverage data reads from chromosome 20 from one human genome results in a file sizes of about 300Mb [10]. The fast decreasing prices of sequencing genomes enables population wide studies. Processing tens of gigabytes of data would lead to latency on a single node. In order to get a high performance it is needed to scale up and parallelize the software to several nodes.

1.1 Problem statement

The purpose of this study is to in construct a computationally effective manner to study structural variation of large genomic data. This will be done by using a dataset consisting of chromosome 20 in over 2500 human genomes from the 1000 genomes project low-coverage data [9].

1.2 Related Work

The 1000 genomes project is one of the largest distributed data collection and analysis projects [9]. The availability of the data from the project makes it possible for researchers around the world to easily access the data and conduct research on it.

1.2.1 ADAM. Massie et al. introduced ADAM which is a data storage format and processing pipeline for genomic data [1]. They argue that ADAM provides better scalability than for instance Hadoop-BAM. They have run 250 GB human genome data on their system and they found that the system achieves 50 times speedup on a 100 node cluster. Apart from the scalability of ADAM, the ADAM files take up 25% less space on the disc compared to compressed BAM files [1]. In their implementation, they store the intermediate data for efficient reuse which does not require the rerun of the previous steps. They take into account that storing the data would cost space on disc and argue that it is preferable to keep the data on disc if the cost is lower than for the recomputation of the data.

2. METHODS

To improve the handling of such a large dataset we decided to use the cluster computing framework called Spark. On top of the Spark compute engine, the data is distributed using Sparks standalone cluster. The sequence files are handled using the Pysam module in a custom built Spark script that computes structural variation. The script extracts unmapped reads, positions and k-mers in the size of 10 that occur 10-200 times in the full dataset. Finally the data is trimmed by custom scripts. The motivation of software choices and further explanations are stated below.

2.1 Compute engine: Spark vs. Hadoop

There are several choices of different frameworks to use to distribute data in a fault tolerant manner. Perhaps the two most well known are Spark and Hadoop. Hadoop and Spark share a great number of similarities, even more than that, they can complete

each other. Both of these frameworks are big data solutions that deal with computationally demanding tasks by parallelizing them and splitting the workload among a large number of commodity hardwares [3][4]. The Hadoop framework comes with both a compute engine (MapReduce) as well as a storage engine (HDFS) [3]. Spark on the other hand is purely a compute engine [4]. Therefore, these two frameworks can complete each other by running Spark on top of the HDFS system. Spark can also use, as one solution among others, Hadoop's cluster manager, called YARN, or a standalone cluster manager[4].

With that said, there are however few critical features where both frameworks differ, and where, as we will see later, Spark usually emerge as the winner. The biggest downside that is reproached to Hadoop's MapReduce is the enormous added overhead and high latency due to I/O. To run two consecutive MapReduce jobs, you need to store the data into HDFS after the first job and then fetch it back for the second job [4]. This behavior affect the performance very badly when it comes to applications that need to run over the same data several times, like machine learning applications. Spark overcomes this issue by allowing in memory operations: caching the intermediate data instead of writing it on an external storage [4]. The second consideration that makes Spark more and more used among the data management community is the fact that it provides a much more flexible programming model than MapReduce. The MapReduce model: mapping and aggregating data in this particular order, can be used in a wild range of applications. However, this model cannot fit all of the applications, and not all of the applications can decompose their workload into a series of MapReduce tasks. And even if you succeed to split the work into a complicated succession of MapReduce jobs, the high latency will degrade the performance of your application [4].

In our particular case, we do not need to iterate over the data more than one time, or use the powerful streaming API that comes with Spark. Nevertheless, we decided to use Spark for this project, for the simple reason that Spark is faster than Hadoop. Even if the application only uses batch processing, Spark can demonstrate better speed than Hadoop [5] [6].

2.2 File system and Cluster manager

The cluster manager is responsible for talking to the file system. The manager must hence be compatible with the file system. It is possible to use Spark together with HDFS, YARN and Mesos. As stated above HDFS is believed to be one of the fastest choices of file system together with Spark. Therefore we decided to use HDFS using Sparks standalone cluster manager. We launched a standalone cluster manually, by staring one master and four workers as instances in our cloud. By using the Spark web UI, one could easily observe the behavior and performance of the cluster, Figure 2.2 [8].

2.3 BAM reader

Our next challenge was to find a suitable module for handling BAM files. There are several options available such as Pysam, ADAM and Hadoop-BAM. Since we had already decided against using Hadoop the choice was between Pysam and ADAM. The ADAM package have impressive features such as a smaller storage space than BAM files and a better scalability than Hadoop-BAM [1]. However ADAM is written in Scala and is more complex than Pysam. Pysam is a Python module that enables access to the command line functionality of the Samtools packages. One feature is to collect BAM files through the URL, which is used in this solution

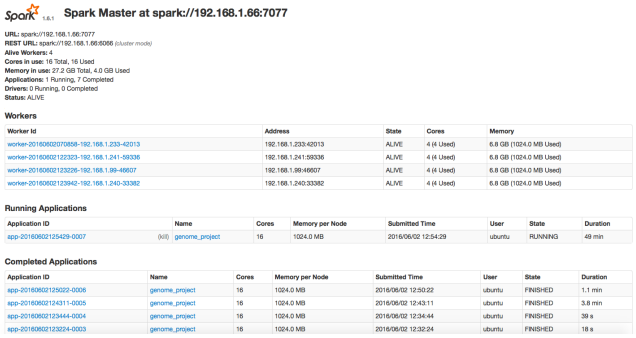

Fig. 1. Spark web UI when running our application.

for the project. The advantage of using ADAM is further discussed in "Future work".

2.4 K-mers and density of unmapped reads

We will be using two methods to identify structural variation counting k-mers and density of reads over the genome for unmapped reads with a mapped pair. An unmapped read with a mapped pair represents a structural variation close to the mapped read pair. Since paired reads are two ends of one long fragment they always occur together. If one read is mapped and the other one is not we know that either the mapping is wrong or there have been some alteration in the genome. A density map of the unmapped reads with a mapped pair hence gives us an estimation of the level of structural variation at different locations in the genome. However this does not tell us anything about how the altered regions sequences look. To look at how the sequences we can extract k-mers from the unmapped reads. K-mers are short sliding windows over the sequence, by counting them we can get an idea if a sequence is rare or common. A sequence that is rare but higher than the noise will probably be an interesting structural variation that only occurs in parts of the population. To extract the k-mers and density map we decided to write our own software.

2.5 Custom script

In general our script parallelize the BAM-files via Spark, runs either k-mer count or reads position counts, maps the k-mer or position as a key/value pair and reduce by the key, see Figure 2.5. The parallelization is done by creating a task for each BAM file name and distributing them among the nodes. The code is programmed to be able to choose either counting k-mers or getting the count of positions. This was a design choice to reduce the running time of the script if you only need one of the results. The actual algorithm extracting k-mers and positions are both written in a map/reduce fashion as this is an effective way of solving the problem . The mapper in the k-mer algorithm will load the BAM files into Pysam, for each read that is unmapped it will extract all k-mers to a list using a sliding window. The key/value pair is then created by taking each element in the list and outputs $< kmer, 1 >$ as key/value pair. The process for the position is similar with the BAM file being read into Pysam, for each read, if it is unmapped take the mate position and then appends it to a list. Then it is outputted as a $< position, 1 >$ with the same procedure as in k-mers. The final results are then written to a file.

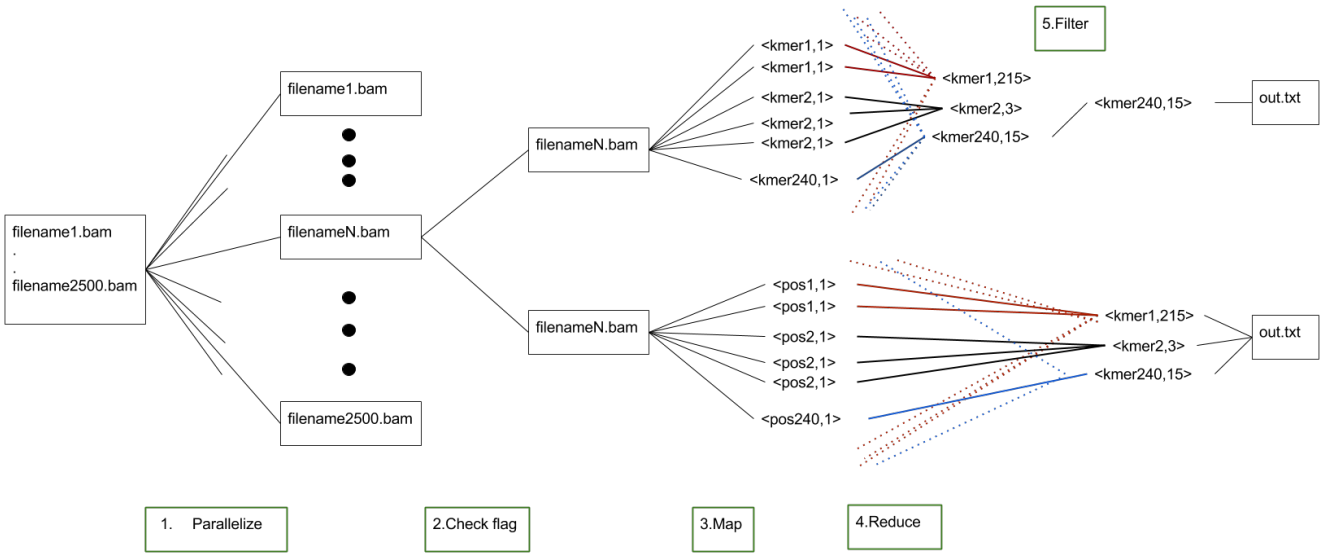

Fig. 2. Schematic picture of how the script handles the data. The main steps are 1.Parallization, 2.Check flag, 3.Map, 4.Reduce, 5.Filter. The result is written to a txt file.

3. RESULTS AND DISCUSSION

3.1 K-mer count

The number of k-mers with a count between 10-200 with sequences containing 'N' was 70269. However 'N' is a letter meaning any nucleotide. This means that the data is of low quality and we do not want this data in our list of k-mers. By sorting out all k-mers with an 'N' in the sequence we get 28843 remaining k-mers. This is still a large amount that would be very hard to check individual cases for a origin. This could be due to a bad selected range that does not exclude noise good enough. When consulting our expert Carl Nettelblad at Uppsala University we found out that the suggested range was misinterpreted. The k-mers where supposed to have a max count of 1 per individual which was not included in our script. This means that all the k-mers that where supposed to have a value bellow 10 have been increased if they occur more then once in each individual. This means that the noise has entered our range of k-mers and probably greatly increased the results size.

3.2 Density plot

The density plot of unmapped reads was plotted as a dot plot for easier comparison, see Figure 3.2. However this data suffers from the same error as the kmer count. By looking at patterns and relative sizes we can however draw some conclusions. In the region upstreams of 30Mb position in the genome we can see a region of almost no unmapped reads. This would mean that there would be no structural variation at this site. It could be the region in the middle of the short and long arm of the chromosome since the short arm is 26Mb long [7].

3.3 High performance

To run our Spark script over all the 2500 chromosome 20 data took 198 minutes for the kmer and 180 minutes for the density of positions running on a Spark Cluster with 4 nodes. The kmer script

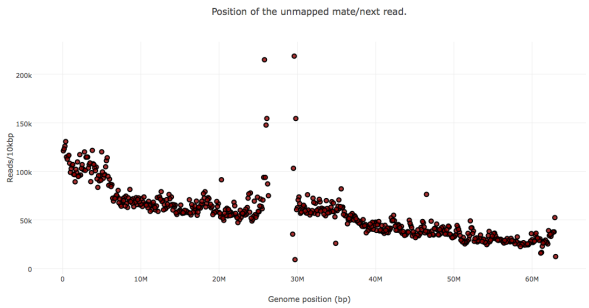

Fig. 3. A dotplot over the density of reads at different positions. The Y-axis is number of counts/10kb and x-axis is the position(bp) along chromosome 20. Just below 30Mb we see a region with no unmapped reads flanked by a region with high unmapped read density.

needs to keep more data in memory due to generating between 35-100 key/value pairs per read while the position script only needed to generate one key/value pair per read. Hence our differences in computational time seems reasonable. The timestamp of about 200min is a significant decrease from the 600min it took to filter out unmapped reads with Samtools on one node over the full dataset. This comparision does not tell us of the scaling of our software.

To test the scaling of our software we tried measuring the time running on different number of BAM files on different number of nodes, Figure 3.3. The computation time can be seen to scale linearly with the number of BAM files. When increasing the number of nodes we can see a significant decrease in time at 1000 BAM files. Below 1000 BAM files the gain of using more nodes is insignificant and should be avoided to keep costs down.

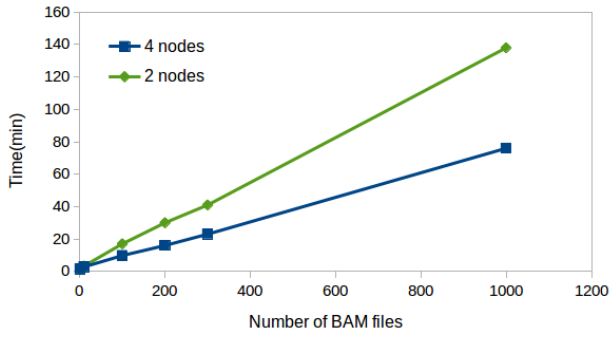

Fig. 4. The computation time scales linearly with the number of BAM files. 4 nodes results in a significant decrease of computation time for 1000 BAM files. For a lower amount of BAM files using many nodes is not critical for a good performance.

4. FUTURE WORK

4.1 Individual filter

As mentioned in Results and Discussion the counting was misinterpreted. Using a count that is instead based on how large fraction of the population that have a certain k-mer or unmapped read is more easily interpreted. This could be done by including making sure that each file only gives out unique keys with the the key/value pair ¡key,1¿. This would be used for both the k-mers and the position of unmapped reads.

4.2 Improved fault tolerance

For further development of the application, one suggestion of improvement is to distribute the BAM file names together with the size of the file. This make it possible to use check-sum to detect errors that has occurred during the transmission of the BAM files. By implementing this solution makes it possible to guarantee that the whole BAM file was downloaded correctly.

4.3 Origin of DNA insertions

Many of the structural variations that we see in our data are caused by insertion of DNA into the human genome. Insertion of DNA into the human genome could be caused by gene duplication or by foreign DNA integration. Foreign DNA can be located and identified by using databases of nucleotide sequences. One of the most well known tools is BLAST. To find out the origins of the insertions the kmers can be run into BLAST. Automation is desirable due to the many kmers generated, in our case about 30 000.

4.4 Scale-up using ADAM

Our solution right now only distributes full files into different nodes. Since the files have different contents and are of different sizes they take different time to complete. This is not a huge problem when using relatively small files as we are since each file is distributed as a separate task. In the worst case scenario the last file will run on one node while all other nodes are empty. However if we have a larger files such as full human genomes with high coverage data or simply all the data is saved in one file this becomes a large problem. In Related Work and Methods we briefly mention a package called ADAM. This software package enables distribution of one BAM files to several nodes. This would be a good solution if running the code with larger files.

5. CONCLUSION

Our software was able to extract k-mer and position data out of chromosome 20 data from 2500 individuals in a distributed fashion. The computation time was significantly reduced from over 600 minutes to only about 200 minutes. The code was shown to scale well both in terms of number of BAM files and the number of nodes used. Future work will be needed to increase fault tolerance and accommodate for even larger file sizes.

6. REFERENCES

[1] Massie, Matt and Nothaft, Frank and Hartl, Christopher and Kozanitis, Christos and Schumacher, Andr and Joseph, Anthony D. and Patterson, David A. ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing, Dec, 2013, EECS Department, University of California, Berkeley, http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-207.html.

[2] Andreas Heger, Kevin Jacobs et al. *"Pysam Documentation, Release 0.9.0"* [Online] Available: https://media.readthedocs.org/pdf/pysam/latest/pysam.pdf [Accessed: 20 may, 2016].

[3] "Welcome to Apache Hadoop!" 13 Feb. 2016. Web. $< http : //hadoop.apache.org/ >$.

[4] "Spark, Lightning-fast Cluster Computing." Web. $< http : //spark.apache.org/ >$.

[5] Reynold Xin. Apache Spark officially sets a new record in large-scale sorting. Databricks, Engineering Blog, November 5, 2014

[6] Juwei Shi,Yunjie Qiu,Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald and Fatma Ozcan. "Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics". Proceedings of the VLDB Endowment, Volume 8, Issue 13, September 2015. doi 10.14778/2831360.2831365

[7] P. Deloukas et al. The DNA sequence and comparative analysis of human chromosome 20. Nature. 2001 Dec 20-27;414(6866):865-71.

[8] *"Spark Standalone Mode"* [Online] Available: http://spark.apache.org/docs/latest/spark-standalone.html [Accessed: 4 june, 2016].

[9] *"1000 genomes webpage"* [Online] Available: http://www.1000genomes.org/about [Accessed: 5 june, 2016].

[10] *"Dataset"* [Online] Available: http://130.238.29.253:8080/swift/v1/1000-genomes-dataset/ [Accessed: 5 june, 2016].