VariantSpark: applying Spark-based machine learning methods to genomic information

Aidan R. O'BRIEN[a] and Denis C. BAUER[a,1]
[a] *CSIRO, Health and Biosecurity Flagship*

Abstract. Genomic information is increasingly being used for medical research, giving rise to the need for efficient analysis methodology able to cope with thousands of individuals and millions of variants. Catering for this need, we developed VariantSpark, a framework for applying machine learning algorithms in MLlib to genomic variant data using the efficient in-memory Spark compute engine. We demonstrate a speedup compared to our earlier Hadoop-based implementation as well as a published Spark-based approach using the ADAM framework. Adapting emerging methodologies for fast efficient analysis of large diverse data volumes to process genomic information, will be the cornerstone for precision genome medicine, which promises tailored healthcare for diagnosis and treatment.

Keywords. Spark, genomics, clustering,

Introduction

We apply Apache Spark to publicly available genomic data. We demonstrate the potential Spark has to process huge amounts of data, as well as its scalability and flexibility. We also compare it to our previous implementation of similar techniques in Apache Hadoop, in regards to speed and performance.

1. Background

With the decreasing costs of DNA sequencing [1], coupled with large cohorts, the amount of data available for processing can quickly become overwhelming. The 1000 Genomes Project [2], for example, has made publicly available 100s of gigabytes of VCF (Variant Call Format) files containing the genomic variants of over 1000 individuals. To cope with these huge amounts of data, challenges such as memory requirements need to be taken into consideration, which are not addressed by other parallelization strategies like OpenMPI or hardware accelerators (GPGPU). Projects such as 'Apache Hadoop MapReduce' [3] can transform data into 'key-value pairs' that can then be distributed between multiple nodes across a compute-cluster, effectively allowing the abstraction of one large job into many smaller, independent jobs. Unfortunately, the MapReduce paradigm that Hadoop is based on is not always the

optimal solution, and can prove to be an obstacle when designing jobs. In addition, Hadoop is disk IO intensive, and this can prove to be a bottleneck in processing-speed.

'Apache Spark' [4] is a more recent compute engine, which overcomes many of Hadoop's limitations. One of the main benefits is that it allows programs to cache data in memory; potentially eliminating, or at least reducing, the bottleneck of disk IO. When utilizing caching, Apache claim Spark to be 100x faster than Hadoop. Although Spark allows MapReduce-like programs, it does not require programs to exactly model the MapReduce paradigm, which in turn allows more flexibility when designing programs. Coupled with MLlib (Spark's machine-learning library), the possibilities to apply Spark to genomic data are endless.

Recognizing this, the Big Data Genomics (BDG) group has recently demonstrated the strength of Spark in a genomic clustering application using ADAM, a set of formats and APIs as well as processing stage implementations for genomic data [5]. While the speedup over traditional methods was impressive, being limited by constraints within this general genomics framework hampered performance. We hence developed a more focused purpose-built application in Spark to perform k-means clustering of individuals. Using this we are able to compare various processing, software and clustering techniques; i.e., a similar implementation in Hadoop and Mahout.

2. Methods

We obtain the genomic data from the 1000 genome project as VCF file from http://www.1000genomes.org/data. It represents a 1092 (individuals) x 11,815,007 (variants) dimensional dataset of individual genotypes. Furthermore, we also download the metadata containing the population association for each individual.

We implemented VariantSpark in Scala, leveraging Apache Spark and MLLib. The aim of the first step is to read in VCF files and effectively transpose them to vectors of each individual's variants. Using tuples to store the data in key-value pairs allows us to achieve this task in a distributed fashion that should scale to allow for an input of hundreds of gigabytes. Effectively, we split each line from the VCF file(s) into an array. We then zip each item in each array with its heading from the VCF file, resulting in an array of tuples (See Figure 1 Spark "Zip with header"). Each of these arrays now stores a single variant for every individual in the dataset and can be evenly distributed across the cluster in Spark's Resilient Distributed Dataset (RDD) format [6]. However, further processing is required, as we need arrays of individuals, not arrays of variants.

We therefore zip each array with a unique index representing a variant (See Figure 1 Spark "Zip with index"). We now have a tuple containing an integer paired with an array of yet more tuples. We can now use the FLATMAP function to effectively flatten the data structure and get closer to the structure we require. The FLATMAP, with a custom inner function, results in a tuple for every variant across every individual. This tuple contains the individual's ID as the 'key', and as the 'value', a secondary tuple of the variant index and the variant (See Figure 1 Spark "Flatmap").

The value of each variant is the distance from the wild-type allele. For example, a homozygous variant is 2, a heterozygous variant is 1 and no variant is 0. We calculate this value from the variant string during the aforementioned FLATMAP stage. In addition, to keep data-sizes to a minimum, we remove any variants with a value of zero. Therefore, we can store the data as a sparse vector, rather than a dense vector. The

sparse vectors store non-zero variants with their index, rather than the entire array of variants, as a dense-vector would do.

We need to group the collection of tuples by the individual using Spark's GROUPBYKEY. Following this function, we have an array for each individual, containing the variant-tuples for said individual (See Figure 1 Spark "Group by individual"). This is essentially the structure of a sparse vector, and we can map each array to the sparse vector structure required by MLlib's machine-learning algorithms.

We proceed to create an MLlib k-means model and train the model on this data. We report the accuracy based on the resulting cluster-set compared to the expected result, according to each individual's population and super population. We run a maximum of ten k-means iterations and use the Manhattan distance measure.

3. Results and discussion

3.1. Comparison to Hadoop

We previously implemented a similar program in Hadoop using the Apache Mahout machine-learning library. Although falling shy of Apache's claim, we observe a speedup of more than 50% with our Spark implementation. In Hadoop, clustering under 1 million variants took close to 16 minutes. Processing and clustering over 1 million variants using Spark, on the other hand, took less than 7 minutes. One contributing factor to the speedup is in-memory caching. Unlike Hadoop, which writes the output of its data transformations to disk, Spark can cache intermediate data-structures in memory. This minimizes disk IO and removes a huge bottleneck. Another reason for the better performance is the different programming model. Hadoop relies on the MapReduce paradigm, which, although easy to use with distributed systems [7], can be restrictive in job design. A 'Map' task must generally precede a 'Reduce' task. Spark implements Map and Reduce functions, however a Spark job can call these functions in any order. This allowed us to implement the VCF transformations in a more optimized way (See Figure 1 for a comparison between the Spark and Hadoop implementation).

3.2. Comparison to Big Data Genomics

Unlike the Big Data Genomics (BDG) implementation, we perform transformations on the variant data directly from VCF files. Although the ADAM columnar data-structure may have benefits for certain operations, for k-means clustering, we observed better performance forgoing the ADAM format (Table 2).

We processed and clustered the human chromosome 22, which includes close to 500,000 variants, using the same options as BDG. The process took approximately 1.3 minutes to complete. This includes training and predicting, and is faster than the 2 minutes runtime reported by BDG. Our method also eliminates the time required for converting VCF files to the ADAM format. However, a trade-off for our faster approach is disk-space. Because the ADAM format uses compressed files, as opposed to uncompressed VCF files, their file-sizes would be smaller.

3.3. Scalability

We demonstrate the potential of Spark to scale from a subset of chromosome 1 containing less than one hundred thousand variants, to multiple chromosomes containing millions of variants. The time required scales in a linear-fashion, based on the number of variants present in the data. However, the overhead of Spark becomes apparent with relatively small datasets, as can be seen in Table 2, where the times required for completing the two smallest jobs only differ by one second.
For the smaller jobs, Spark requires only 1GB memory per executor (or core). Increasing the number of variants beyond 2 million requires a greater memory allocation for each executor. Although this is not a problem with high-memory clusters, on smaller clusters, such as our four-node Azure cluster, increasing the memory allocation beyond 1GB will limit the number of executors that can run simultaneously, due to memory-constraints. Although of no advantage to clustering quality, as we explain in the next section, we further demonstrate the scalability by increasing the dataset to include multiple chromosomes. These jobs complete successfully, albeit after more time and with higher memory requirements (Table 1).

Spark not only scales well with data size, but also with cluster size. To demonstrate this we run identical jobs on our in-house cluster. Clustering every chromosome 1 variant from VCF files takes only six minutes when using 80 executers. As is expected, this is much faster than the small cluster which took over 17 minutes for the same job. Although these two clusters are different in OS, size and hardware configuration, a Spark jobs, being implemented in Scala (or Java), can run on any cluster. The only change we make is to the launch script where we specify how many executors to use.

3.4. Clustering quality

When we cluster the entire group of 1092 individuals, the majority are clustered with individuals from the same super-population. This is portrayed in the Adjusted Rand Index (ARI) [8], which is 0.8 (where a value of '1.0' would represent a result where all like-individuals are grouped together). Note, we chose ARI as it is frequently used to compare the accuracy of two clusterings (here: our prediction compared to the true population associations).

We can substantially improve the accuracy to a perfect classification (ARI=1.0) by removing the fourth super-population, AMR (American). Including AMR individuals, we observe that the majority of these individuals are placed in the same group as Europeans, likely accurately reflecting their migrational backgrounds. Only a minority of AMR individuals form an independent group, likely comprising of genetic information otherwise not captured by the 26 sub-populations of the 1000 genomes project.

Interestingly, we achieved these results with as few as one million variants, i.e. less than half the variants available in chromosome 1, and increasing the size of the dataset further saw no improvement on clustering the the AMR individuals.

4. Conclusion

We demonstrate the potential applications of processing VCF files using Apache Spark. We successfully apply k-means clustering to over 1 thousand individuals with millions of variants. We explore performance on different architectures starting with a relatively small 24-node cluster and subsequently scale up to a cluster with hundreds of nodes. We demonstrate our method of transforming and clustering individuals from VCF files to be faster than clustering individuals stored in the ADAM format, albeit at the cost of disk space.

In future work we aim to improve the disk space footprint by using compressed VCF files. The 1000 genomes consortium distributes their VCF files with gzip compression, which is unsuitable for use in Spark as it is not possible to split gzip files. We therefore plan to investigate other compression methods, such as bz2, which Spark can read line by line. We expect this to not only minimize storage requirements of the actual data, but also further reduce runtime, making the approach attractive for applying the more complicated machine learning approaches to genomic data.

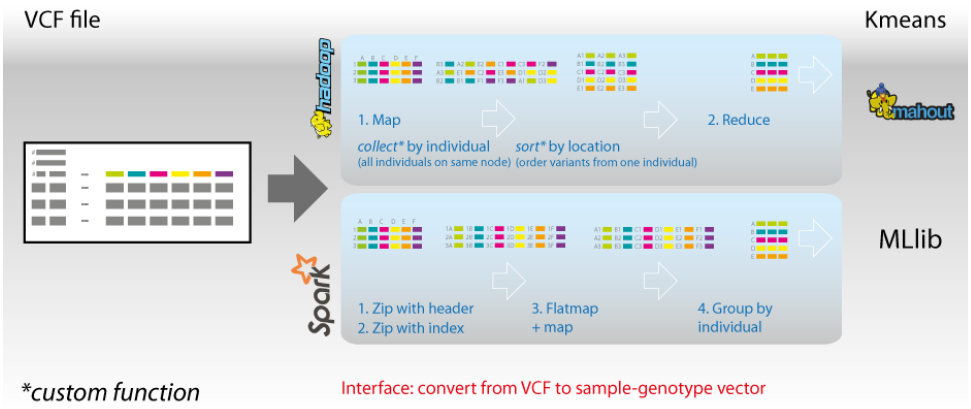

Figure 1. Schematic visualizing VCF processing using Hadoop and Spark respectively.

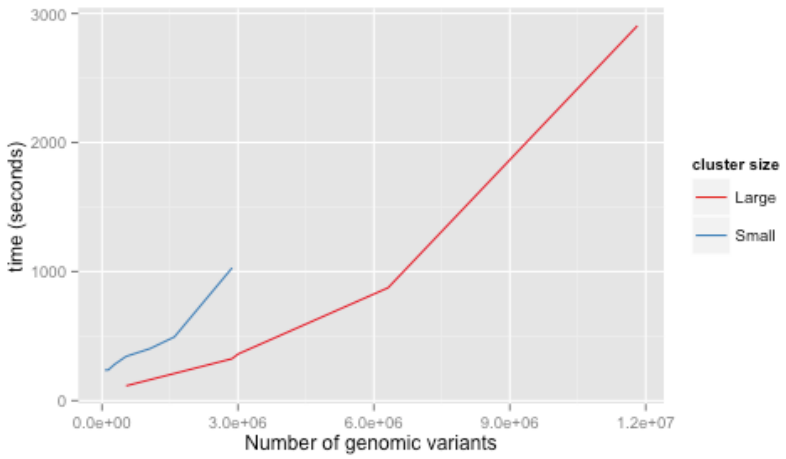

Figure 2. Time scaling with increasing number of clustered variants.

Table 1. Benchmarks from VCF clustering jobs from different variants and configurations. The 'small' cluster has three nodes, with a total of 24 CPU cores and 36GB memory, running on Microsoft Azure. The 'large' cluster has fourteen nodes, with a total of 448 CPU cores and 1.31TB memory, running on an in-house Hadoop setup.

Cluster	Chr(s)	Size (Kbase)	Variants	Executers (memory)	Time
Small	1	5,000	65,572	24 (1GB)	3m 57s
	1	10,000	138,840	24 (1GB)	3m 58s
	1	20,000	272,364	24 (1GB)	4m 40s
	1	40,000	531,230	24 (1GB)	5m 43s
	1	80,000	1,058,736	24 (1GB)	6m 43s
	1	120,000	1,598,000	24 (1GB)	8m 14s
	1	240,000	2,869,355	12 (2GB)	17m 8s
Large	1	40,000	531,230	80 (1GB)	1m 55s
	1	240,000	2,869,355	80 (2GB)	5m 25s
	1	248,956	3,007,196	80 (2GB)	6m 02s
	1	248,956	3,007,196	400 (2GB)	4m 30s
	1-2	491,149	6,314,788	80 (3GB)	14m 35s
	1-4	879,658	11,815,007	80 (4GB)	48m 24s

Table 2. Benchmark from clustering individuals from VCF files and ADAM files. The time for ADAM clustering is reported by BDG. For both methods, the data was filtered to only include the populations listed. The time does not include the one-off task of extracting the compressed VCF file (required by both clustering methods) or the conversion of the VCF file to the ADAM format (only required for clustering the ADAM file).

Cluster	Populations	Chr	Variants	Executers	Time
ADAM	GBR, ASW, CHB	22	494,328	80	2m 01s
VCF	GBR, ASW, CHB	22	494,328	80	1m 20s

References

[1] L.D. Stein, The case for cloud computing in genome informatics, *Genome Biology* **11** (2010), 207.

[2] The 1000 Genomes Project Consortium, An integrated map of genetic variation from 1,092 human genomes, *Nature* **491** (2012), 56-65.

[3] Borthakur, Dhruba. "The hadoop distributed file system: Architecture and design." Hadoop Project Website 11.2007 (2007): 21.

[4] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

[5] Massie, Matt and Nothaft, Frank and Hartl, Christopher and Kozanitis, Christos and Schumacher, André and Joseph, Anthony D. and Patterson, David A, ADAM: Genomics formats and processing patterns for cloud scale computing, (2013) EECS Department, University of California, Berkeley http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-207.html

[6] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing Technical Report UCB/EECS-2011-82. July 2011.

[7] D. Jeffrey and S. Ghemawat, MapReduce: Simplified data processing on large clusters, *OSDI* (2004) Sixth Symposium on Operating System Design and Implementation, San Francisco

[8] L. Hubert and P. Arabie. Comparing partitions. Journal of Classification, 2(1):193–218, 1985.