Distributed analysis of expression quantitative trait loci in Apache Spark

Tammo Rukat, University of Oxford

***Abstract*—A major challenge in current genomic research is the development of computational and statistical tools that are capable of analysing the ever increasing amount of data provided by next generation sequencing methods. Here we investigate the potential of the open-source distributed computing framework Apache Spark, to facilitate fast, horizontally scalable genetic data analysis, exemplified by the search for expression quantitative trait loci (eQTL) in the 1000 Genomes dataset.**

An algorithm for trans eQTL analysis is proposed and enables the analysis of all available pairwise correlations of gene-SNP-pairs in the dataset ($> 3 \times 10^{11}$) in approximately 90 minutes on a cluster with 768 worker nodes. The algorithm scales linearly with the problem size, and benefits from larger cluster sizes up to 768 nodes. A second algorithm for cis eQTL analysis is introduced but shown to be inherently difficult to design within a distributed system and therefore only recommended for very targeted investigations. Though constantly enhanced and extended, Spark already offers a simple and powerful environment, suitable for distributed genetic data analysis.

Code is available under: https://github.com/TammoR/spark_eqtl

I. Introduction

STEADILY declining costs and improvements in the quality of DNA sequencing in the last 15 years have sparked research efforts to better understand genome function and variation. The cost for sequencing a human genome dropped from $2.7 billion in 2003 [1] to about $1000 in 2014. By the end of 2014 an estimate of 228,000 genomes have been sequenced [2] and further large-scale efforts, such as the 100,000 Genomes Project project in the UK [3], leave no doubt about the continuing growth of available data. While this development entails a significant increase in the statistical power of genome studies, it also poses severe computational and statistical challenges. The cost of data maintenance and processing easily surpasses that of sequence generation [4], while the growth of data will continue to outpace the growth in available computational power. Several strategies to address this challenge have been proposed:

The increased use of cloud computing allows for elastic scaling of resources and eliminates the maintenance effort of a high performance computing (HPC) cluster [5, 6]. The most recently announced partnership between Google and the Broad Institute of MIT and Harvard is a striking example, since one of the collaborations main formulated goals is to facilitate genomic data analysis to the cloud [7]. Furthermore, the development of algorithms for approximate inference empowers researchers to choose a tradeoff between accuracy and computation time [8, 9]. Eventually, the work that is presented here investigates the use of distributed computing across a network of computers. This approach provides a cheap, horizontally scalable framework that can be run on commodity hardware. However, it also requires algorithms that scale horizontally and eschew extensive communication between computing nodes [10, 11].

This work focuses on the analysis of genetic variants that explain variation in gene expression levels, so called expression quantitative trait loci (eQTL). They are commonly studied to characterise functional variation and to comprehend processes of gene regulation [12, 13]. The analysis is implemented within the Apache Spark open-source framework for distributed computing [14] and is put to proof by analysing the publicly available 1000 Genomes dataset [15, 16].

The remainder of this introduction presents essentials of eQTL analysis and gives a short introduction to Apache Spark. Thereupon section II describes the algorithms that were implemented, with a particular emphasis on how computational efficiency is achieved within Spark. Subsequently section III presents performance benchmarks and analysis results as a proof of concept. Eventually section IV concludes with a discussion of the performance, results and topics that remain to be investigated.

A. Expression Quantitative Trait Loci

The analysis aims to find single nucleotide polymorphisms (SNPs) in the genome that explain a fraction of the variance in gene expression across samples, e.g. patients or different tissues. Typically this is done by assuming a linear relationship between genotype and the gene expression level, which can be thought of as a linear regression problem:

$$Y = \beta_0 + \beta X + \epsilon, \tag{1}$$

- Y contains the gene expression levels for every gene and every sample.
- X contains the genotype variation for every SNP and every sample, coded as 0, 1, or 2 according to the number of minor alleles carried by the individual.
- β_0 is the mean expression level for the major allele homozygotes.
- β contains the regression coefficients for every SNP/gene pair

This work was jointly supervised by Satu Nahkuri (pRED Informatics, Roche Innovation Center Basel), Peter Humburg, and Christopher Yau (University of Oxford). It was carried out within the EPSRC funded Systems Approaches to Biomedical Sciences Centre for Doctoral Training, where TR is a doctoral student.

For correspondence please visit: https://tammor.github.io.

Handed in for assessment July 2, 2015.

- ϵ is a matrix of residuals, assuming: $\epsilon \sim$ i.i.d. $N(0, \sigma^2)$.

Eq. (1) can be minimised with respect to ϵ and solved for β, yielding a large matrix multiplication problem, a formulation that is employed by the *R* package Matrix eQTL [18], which allows for fast eQTL analysis on a desktop computer.

After scaling genotype and expression data to unit variance and zero mean, the matrix multiplication is equivalent to calculating all pairwise SNP-gene Pearson correlations. The 1000 Genomes data features approximately 4×10^7 SNPs and expression levels for 2×10^4 genes, both available for 421 common samples. Hence, the analysis requires multiplying an approximately 2×10^4 by 4×10^2 expression level matrix with a 4×10^2 by 4×10^7 genotype matrix, resulting in a matrix with 2×10^4 by 4×10^7 correlation coefficients. The algorithms that are discussed in the following sections implement this procedure and, where appropriate, investigate the significance of the gene-SNP-correlation. Any further analysis that would be needed to obtain reliable results of the functional sequence variation, such as the adjustment for confounding factors, e.g. based on sample meta-information or principal components of the data [19, 20] exceeds the scope of this investigation.

Practical parallel data preparation in Spark

Genotype data is usually provided as tab-separated plain text in the Variant Call Format (vcf) [a]. The following snippet shows data for two SNP locations and four subjects. The first six columns contain meta information about each SNP, while each of the following columns refers to a specific tissue sample.

```
## fileformat=VCFv4.1
## demonstration example of a vcf genotype file
# CHROM  POS       ID          REF ALT  QUAL  HG00096 HG00097 HG00099 HG00100 ...
22       16051432  rs587672056 A   G    PASS  1|0     0|0     0|0     0|0     ...
22       16051453  rs62224611  A   C,G  PASS  0|0     2|2     0|0     1|0     ...
```

Once the Spark context is defined (here: `sc`), the text file can be loaded into an RDD (here: `genotype_input`), where each RDD element represents a line of the text file (`In[1]`). If materialised, this RDD resides in memory, distributed into partitions across the cluster. Subsequently, the filter method operates on the RDD, removing every element that starts with a `#` (`In[2]`). Eventually another anonymous function is 'mapped' onto the data, splitting it at every tab (`In[3]`). Python uses the lambda syntax to create anonymous functions, but any function could be applied to the data in this manner.

```
In[1]: genotype_input      = sc.textFile(source)
In[2]: genotype_uncomment = gt_mat.filter(lambda line: line[0] != '#')
In[3]: genotype_lists      = gt_mat.map(lambda line: line.split('\t'))
```

Notice, that due to Spark's lazy evaluation the three commands above would not induce any actual computation. They represent *transformations* and merely create a set of rules which describe how to derive the data. Only if an *action* such as `.collect()` is performed these rules are executed:

```
In[4]: genotype_lists.collect()
Out: [['22', '16051432', 'rs587672056', 'A', 'G',   'PASS', '1|0', '0|0', '0|0', '0|0', ...],
      ['22', '16051453', 'rs6222461',    'A', 'C,G' 'PASS', '0|0', '2|2', '0|0', '1|0', ...]]
```

Further processing is performed by mapping a more complex function to the data (not shown). It expands loci that are subject to different polymorphisms, such as the second locus in this example (using the `flatMap` instead of `map` method to allow for a one-to-many mapping of RDD elements). Thereupon, the mutation representation of the form '$i|j$' is transformed to an integer number (0: no mutation, 1: heterozygous mutation, 2: homozygous mutation) and each SNP is indexed by an integer number to facilitate identification with the metadata that is written into a separate lookup array (not shown).

```
In[5]: genotype_vectors_labeled.collect()
Out:   [([1,0,0,0,...],0), ([0,0,0,1,...],1), ([0,2,0,0,...],2), ...]
```

Eventually, the genotype data for each location is transformed into a sparse row matrix object (using the Python package for sparse matrices, scipy.sparse [17]). In the case of the 1000 Genomes data, this allows for a compression by more than an order of magnitude. Now the genotype data is prepared for eQTL analysis, distributed across nodes as an RDD of indexed sparse (1xN) matrices, where N is the number of samples.

[a] Format specifications: https://github.com/samtools/hts-specs

Box 1: **Processing a genotype input file in Spark** – Data cleaning and preparation serves as an example of how declarative programming facilitates parallel processing, here within Spark's Python API.

B. Apache Spark

Apache Spark is an open-source framework for distributed computing. It was created by Matei Zaharia at UC Berkeley in 2009 [14] and donated to the Apache Foundation in 2013. As of June 2015, Spark is the most active big data open source project with 570 contributors over the last 12 months [21]. Introductory, as well as detailed textbooks on the practical usage of Spark have been published by Karau et al. [22] and Ryza et al. [23] respectively, while the official Spark documentation [24] provides the most up-to-date comprehensive coverage. The following paragraph gives a short overview about Spark's high-level concepts, sufficient to comprehend the implementation of eQTL analysis. For a more technical and detailed discussion please refer to Box 1 and to the above-mentioned references.

The Spark execution engine can be regarded as a more versatile and faster alternative to the MapReduce paradigm which is most prominently deployed by Apache Hadoop [25]. A major advance that both Spark and Hadoop offer over traditional high performance computing (HPC) is *data locality*. The data is stored in a system of distributed computing nodes and instructions for computation are shipped to and executed at the nodes. Ideally, this minimises expensive communication of data between nodes. Suitable clusters can be built from commodity hardware at much lower cost than traditional HPC systems, and allow for simple scalability by adding additional resources at a later time. In Hadoop MapReduce the data is written to disk after every MapReduce job, inducing an overhead that easily dominates execution time. In contrast to that Spark enables repeated queries and interactive processing by keeping the data in memory. Spark's main abstraction that serves to this end is called *Resilient Distributed Dataset (RDD)* [26]. Spark's data processing paradigm is centred around the notion of an RDD as explained below. For an example of the following concepts see Box 1.

1) At its core, an RDD is a **read-only collection of objects** (e.g. lists, array, strings, ...), distributed in partitions across the cluster.
2) RDDs provide methods for *transformations*, such as `.map()` or `.filter()`, that apply the same **operations in parallel** to all items inside the RDD.
3) RDDs reside **in memory**, if not specified otherwise and are only spilled to disk if insufficient memory is available.
4) RDDs are evaluated in a **lazy** fashion. A transformation that is applied to an RDD does not trigger any computation until the user (or another part of the programme) ex-

Data preprocessing

1) Load the following data files into Spark RDDs (distributed across the cluster):
   - Genotype data (.vcf file)
   - Expression data (.gct file)
   - genes' start/end position
2) Filter genotype and expression data, to contain identical samples in identical order.
3) Preprocess expression and genotype data (see also Box 1 for a more detailed treatment), in particular:
   - Assign IDs to all genes and SNPs, to allow for its identification in a lookup table containing meta information.
   - Filter genes and SNPs that are not expressed in at least 5 samples or 5% of the samples (whichever is larger).
   - Convert the expression levels to floating point numbers and scale them to zero mean and unit variance.
   - Expand SNPs with different mutations at the same position in the genome to be distinguishable.
   - Translate the genotype codes to integer numbers (e.g. `0|1` $\rightarrow$ `1` or `1|1` $\rightarrow$ `2`).
   - Cast the integer vector containing the genotype information for every SNP to a sparse vector, using the scipy.sparse Python package [17].
   - Materialise the gene expression data and broadcast a copy to every worker node.
4) Depending on whether a cis or full trans analysis is required, apply one of the following algorithms:

Cis Algorithm

5) Broadcast a list of genes that will be analysed and their position in the genome to every worker node
6) Iterate for every gene in that list
   a) Filter out all SNPs that are not cis for the current gene.
   b) Map the following function onto the RDD of remaining SNPs:

Trans algorithm

5) Broadcast the expression data of all genes of interest to all nodes.
6) Map the following function onto the genotype RDD, taking the expression data as argument:

Inner correlation function

i) Expands the sparse genotype data, demean and scales it to unit variance.
ii) Calculate Spearman correlation and corresponding p-value.
iii) Return at tuple: (p-value, SNP-index, gene-index).

Box 2: **Algorithms for correlation analysis** – After data preparation, the cis or trans algorithm can be applied to calculate correlations and measures of their statistical significance.

plicitly asks for its results. In contrast to *transformations*, operations that trigger a computation are called *actions*. The simplest Spark action is `collect()`, which merely materialises the RDD, i.e. carries out all transformations that have been commissioned before it returns the RDD's content.

5) RDDs are **ephemeral**. Content will be removed from memory, immediately after the execution of the action that triggered its materialisation. To circumvent this behaviour RDDs can be explicitly cached.
6) Fault tolerance is achieved through the notion of **lineage**. While data resides in memory as little as possible (through lazy evaluation and ephemerality), the Spark engine saves all transformations used to derive the RDD. Thus partitions of data can be reconstructed on the fly.

Spark is implemented in Scala, but provides APIs in Java, Python, and R. Most recent features, however, are usually available in Scala first. Spark also offers several higher level libraries that are under highly active development. *Mllib* provides a variety of machine learning functionality. *Spark Streaming* enables streaming applications that should prove useful for data that exceeds the memory size. *Spark SQL* facilitates processing of structured data using SQL queries and introduces the DataFrame API [27], which resembles data frames in R or in Python's pandas library. Spark's capability of caching data in memory is also of great practical importance for postprocessing and enables interactive queries to data that would be impossible to analyse on a local computer.

Spark can run on an existing Hadoop or Mesos cluster, in the cloud (e.g. Amazon EC2), or in standalone mode. The latter allows for for testing and debugging on a local computer before moving applications to a cluster. Spark also seamlessly interacts with a variety of data sources, such as the Hadoop Distributed File System (HDFS) or Amazon S3.

II. Methods and Algorithms

As introduced in section I-A, the basic task for eQTL analysis is determining correlation coefficients and their statistical significance, in order to quantify the variation in gene expression that is explained by the variation in a single nucleotide. Instead of a simple vector multiplication for every gene-SNP-pair (equivalent to Pearson correlation if data is demeaned and unit variance), we choose to calculate Spearman correlation coefficients which relax the assumption of normally distributed data.

Two algorithms for eQTL analysis are developed using Spark's Python API. Their general properties are discussed in the following while a thorough outline of both algorithms is provided in Box 2.

The *cis-algorithm* calculates all pairwise correlations for SNP-gene-pairs that lie within a given cis range of base pairs. Conversely, the *trans-algorithm* operates on all available SNP-gene-pairs. Their main difference is that the number of correlations is about 5,000 times higher for the trans-algorithm (Based on the 1000 Genomes dataset and a typical cis range of 5e5 base pairs), while the cis-algorithm needs to select the nearby SNPs for every gene, which requires a search through all SNP loci. Ideally the data should initially reside at the node where it will be used, in order to avoid expensive shuffling across the cluster. Since all SNPs have to be compared with all genes, only one of the two datasets should be distributed across the cluster, while a copy of the other is shipped (*broadcasted*) to every worker node. This constraint is reflected by the fact that a transformation of a Spark RDD (always operating on all its items) can not take another RDD as argument. The size of the datasets (4×10^7 SNPs, 2×10^4 genes) makes the choice apparent to distribute the genotype data and broadcast the expression data.

Hence the central object of the trans algorithm is the genotype RDD (distributed across the cluster) that undergoes a transformation taking the broadcasted gene expression data as argument and enabling a fully parallel analysis without data shuffling. As opposed to this, the same argument poses a challenge for the design of the cis algorithm. Selecting the cis SNPs for every gene necessitates as many searches through the genotype data as there are number of genes (here 2×10^4), while the converse requires as many iterations through the expression data as there are SNPs (here 4×10^7). We therefore decided to render the gene expression data the central object of the analysis and proceed gene by gene. This choice is also in line with the typical target of inference, that is finding (cis) eQTL for a particular gene, rather than genes that are influenced by a particular SNP. However, genotype data is distributed and expression data is broadcasted not only for the convenience of the trans algorithm but also as a requirement

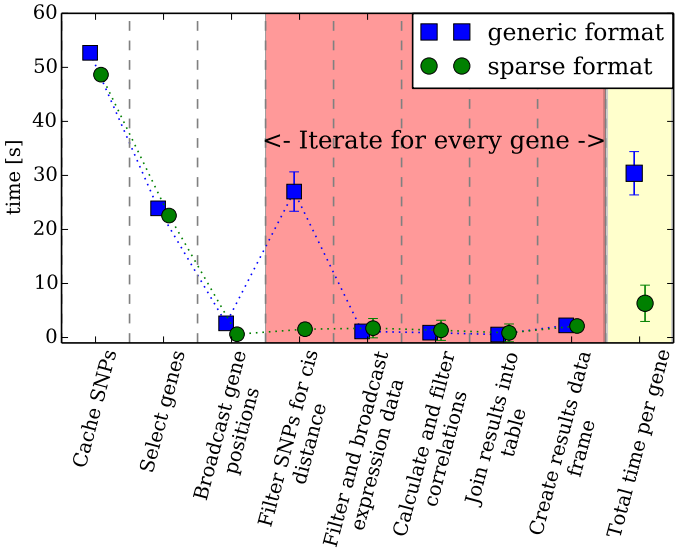

Fig. 1: **Individual steps of the cis algorithm** – Cis analysis was performed for 100 randomly selected genes and all SNPs on chromosome 22 with a cis distance of 5×10^5 base pairs. Calculations were carried out on 256 worker nodes. Shown are mean and standard deviation from the 100 repetitions. The red shaded steps of the procedure are repeated for every gene and therefore most critical for performance. Using a sparse representation of genotype data (green circles) accelerates the search for cis SNPs, which previously was the crucial bottleneck for the dense representation (blue squares).

by virtue of the respective data sizes. Therefore it is necessary to introduce another layer of iteration, treating every gene consecutively (see Box 2, step 6)).

For a moderate number of genes (around 200 iterations), the cis algorithm suffers from insufficient memory on the head node, because the RDD lineage grows with every iteration and eventually becomes too long to be stored. This can be circumvented by regularly checkpointing the RDD (using *Spark Streaming*), i.e. saving its current state to disk and forgetting about its lineage. Benchmarks of the individual steps of the cis-algorithm are shown in Fig. 1 and demonstrate the great improvements that are achieved by choosing a sparse representation for the genotype data. Another shortcoming of the cis algorithm is the filtering of SNPs that initially are distributed uniformly across the cluster. Subsequently, the remaining SNPs are likely to be distributed unevenly, introducing a higher workload on a few nodes. Remaining nodes will be idle until the last worker node has finished its jobs for the current gene.

In contrast, the trans algorithm is much simpler and fully exploits the strengths of the Spark engine. To avoid insufficient memory issues, full trans analyses are performed chromosome-wise, with a few million SNPs per chromosome combined with all 20,000 genes.

Careful tuning of the Spark environment may yield substantial performance increases. An easy and effective approach is to increase the number of partitions of an RDD and thereby the degree of parallelism. We observed improvements for up to eight times as many partitions as nodes. Additionally the broadcasting of local variables that are used more than once can greatly reduce communication cost. A wide variety of approaches to improve Spark performance exists and is mostly dependent on the cluster and the type of algorithm. They often require deeper insights into Spark's internals and exceed the scope of this article [22, 24, 28, 29].

III. Benchmarks and Results

As discussed in the previous section, the trans algorithm is simpler and promises superior performance, if more than a few specific genes are interrogated. Here, its performance and scaling behaviour is explored. Fig. 2 shows that the trans algorithm scales approximately linear with variation of the number of SNPs. Furthermore the runtime steadily declines with increasing cluster size is shown in Fig. 3 but reveals diminishing returns for larger clusters. This is partly due to the data preprocessing steps, as shown in the same figure. Preprocessing involves data shuffles that are not expected to scale well. For large clusters it accounts for close to half the processing time. However, preprocessed data can easily be saved to disk, locally or in a distributed file system, in order to avoid this overhead for repeated analyses.

By extrapolating and comparing results for the cis algorithm from Fig. 2, it becomes apparent that the trans algorithm performs 3 to 4 orders of magnitudes faster. Therefore we advise against the usage of the cis-algorithms in its current form, if more than a few specific genes are analysed (see section IV for possible improvements).

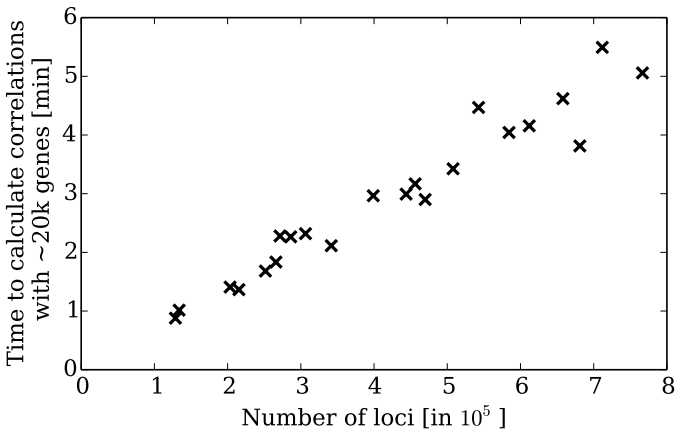

Fig. 2: **Scaling with the number of loci** – Each data point corresponds to a chromosome. The time to calculate and filter all pairwise correlations of SNPs in that chromosome with approximately 20.000 available genes is depicted as a function of the number of available SNPs. Calculations were performed on 768 worker nodes with 16GB memory for each node.

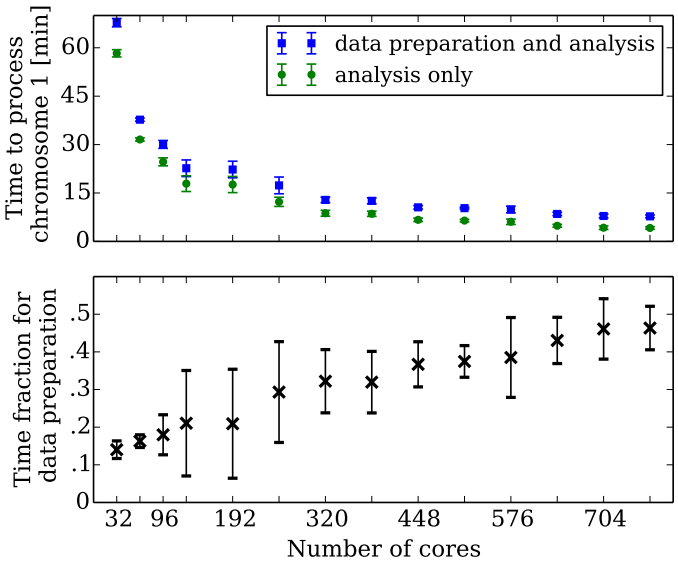

Fig. 3: **Scaling with cluster size** – Time for a full trans correlation analysis with all 20,000 genes and all SNPs on chromosome 1 as a function of cores in the cluster (top) and the fraction of that time that was used to load and clean the data. Shown are mean and standard deviation from 5 repetitions.

A full trans analysis of the 1000 Genomes data on 786 worker nodes, each with 16 GB memory, calculating more than 3×10^{11} correlations, takes approximately 90 minutes. Results of this analysis should be understood as a proof of concept. In particular, no adjustment for confounding variables has been performed and the majority of findings are presumably due to linkage disequilibrium. However, approximately $50,000$ eQTL above a Bonferroni corrected significance threshold of 10^{-4} are identified, 2500 of which act across chromosomes and are shown in Fig. 4. Additionally, Tab. I lists the 5 genes that are subject to the highest number of trans eQTL interactions. The

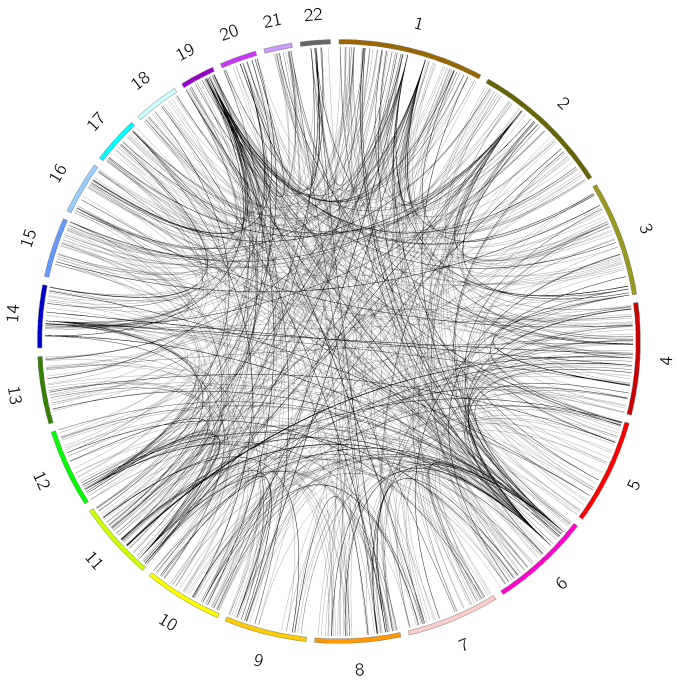

Fig. 4: **Most relevant transchromosomal eQTL** – Shown are 2500 interactions across chromosomes with a Bonferroni corrected p value threshold of 0.0001, i.e. the probability that a single one of the connections is a false positive is 1 in 10,000. The particularly active regions correspond to genes that are correlated with various SNPs as reported in Table I

results are restricted to autosomes, since allosomes have an additional gender bias.

IV. Discussion and Conclusion

A scalable algorithm for trans eQTL analysis in Apache Spark was developed, enabling fast comprehensive analyses of SNP-gene-interactions. It is suited to scale to larger datasets and larger clusters. For the latter, one of the main limitations is the data preprocessing. However, this task has do be performed only once, whereupon processed data can be stored in a distributed file system such as HDFS. Hence, the outlined approach is expected to be suitable for the growing datasets that recent and ongoing sequencing efforts are expected to provide.

Additionally, an algorithm for cis analysis has been proposed but suffers from performance issues, such that a trans analysis will almost always be faster to perform. Nevertheless, there is room for improvement in the cis algorithm. In particular the identification of cis SNPs for a given gene could be significantly improved by utilising prior information about the SNP positions, e.g. by providing them in a sorted manner and employing a bisection search. Furthermore, the structural obstacle that requires the mapping of suitable SNPs onto each gene, instead of making the distributed genotype data the pivotal object of computation, could potentially be overcome and facilitate search and analysis for cis genes for an RDD of SNPs in parallel.

Tab. I: **Genes with numerous trans interactions** – Corresponding to the hotspot regions in Fig 4, genes with the highest number of significant trans eQTL and the associated count of cis and trans eQTL are listed. Variants within one megabase of the gene location are called cis, others trans.

Gene	Chromosome	cis eQTL	trans eQTL
IL36RN	2	0	86
HDGFL1	6	6	137
HCRTR2	6	1	128
LRFN5	14	1	68
IGFL3	19	1	98

The analysis of eQTL as described above is embarrassingly parallel on several levels. Implementing it in a GPU environment therefore seems natural. However, this task would require a skilled software engineer and the implementation would presumably lack generalisability to less parallel problems. In contrast, the implementation in Spark on an existing cluster is easily extendable and offers APIs in the most common languages for scientific programming. Nevertheless, it remains to be investigated whether Spark is suitable for less embarrassingly parallel tasks. In fact, the difficulties in designing a cis algorithm have already revealed some of the obstacles that Spark faces for more complex calculations. For instance operations that combine two large datasets or cases where it is unclear which part of the data is needed at which computation node, are problematic. Having said this, these difficulties are mostly inherent to distributed computing.

In conclusion, Spark allows to design complex analyses in a simple but versatile framework, with its design being well suited to exploit the strength of distributed computing. It is under vivid development, promising constant improvements in stability and new features. Together with its horizontal scalability, this makes Spark a propitious analysis tool for the ever increasing amounts of genetic sequencing data.

Acknowledgements

I would like to thank Satu Nahkuri at Roche for her dedicated and knowledgeable supervision and the whole Roche pREDi data science team in Basel for giving me a warm welcome and the opportunity to conduct this work. Furthermore, I would like to thank my academic supervisors Chris Yau and Peter Humburg for their dedicated and most helpful guidance and comments. I would also like to thank Florian Klimm for the rewarding discussions and companionship during this project.

References

[1] National Human Genome Research Institue. *The Human Genome Project Completion: Frequently Asked Questions*. https://www.genome.gov/11006943. Acessed: 2015-06-29. Oct. 2010.

[2] Antonio Regaldo. *llumina Says 228,000 Human Genomes Will Be Sequenced This Year*. http://www.technologyreview.com/news/531091/emtech-illumina-says-228000-human-genomes-will-be-sequenced-this-year/. Acessed: 2015-06-29. Sept. 2014.

[3] Genomics England. *100,000 Genomes Project*. http://www.genomicsengland.co.uk/the-100000-genomes-project/. Acessed: 2015-06-29.

[4] Lin Liu et al. "Comparison of next-generation sequencing systems." eng. In: *J Biomed Biotechnol* 2012 (2012), p. 251364. DOI: 10.1155/2012/251364. URL: http://dx.doi.org/10.1155/2012/251364.

[5] Lincoln D. Stein. "The case for cloud computing in genome informatics." eng. In: *Genome Biol* 11.5 (2010), p. 207. DOI: 10.1186/gb-2010-11-5-207. URL: http://dx.doi.org/10.1186/gb-2010-11-5-207.

[6] Ying-Chih Lin, Chin-Sheng Yu, and Yen-Jen Lin. "Enabling large-scale biomedical analysis in the cloud." eng. In: *Biomed Res Int* 2013 (2013), p. 185679. DOI: 10.1155/2013/185679. URL: http://dx.doi.org/10.1155/2013/185679.

[7] Broad Institute. *Broad Institute, Google Genomics combine bioinformatics and computing expertise to expand access to research tools*. https://www.broadinstitute.org/news/6994. Acessed: 2015-06-30. June 2015.

[8] P. Marjoram, A. Zubair, and S. V. Nuzhdin. "Post-GWAS: where next? More samples, more SNPs or more biology?" eng. In: *Heredity (Edinb)* 112.1 (Jan. 2014), pp. 79–88. DOI: 10.1038/hdy.2013.52.

[9] Miguel Arenas. "Advances in computer simulation of genome evolution: toward more realistic evolutionary genomics analysis by approximate bayesian computation." eng. In: *J Mol Evol* 80.3-4 (Apr. 2015), pp. 189–192. DOI: 10.1007/s00239-015-9673-0.

[10] Michael I. Jordan. "On statistics, computation and scalability". In: *Bernoulli* 2013, (Sept. 2013), Vol.19, No.4, 1378–1390.

[11] Ariel Kleiner et al. "A Scalable Bootstrap for Massive Data". In: (Dec. 2011).

[12] Stephen B. Montgomery and Emmanouil T. Dermitzakis. "From expression QTLs to personalized transcriptomics." eng. In: *Nat Rev Genet* 12.4 (Apr. 2011), pp. 277–282. DOI: 10.1038/nrg2969.

[13] Alexandra C. Nica and Emmanouil T. Dermitzakis. "Expression quantitative trait loci: present and future." eng. In: *Philos Trans R Soc Lond B Biol Sci* 368.1620 (2013), p. 20120362. DOI: 10.1098/rstb.2012.0362. URL: http://dx.doi.org/10.1098/rstb.2012.0362.

[14] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1863103.1863113.

[15] 1000 Genomes Project Consortium et al. "An integrated map of genetic variation from 1,092 human genomes." eng. In: *Nature* 491.7422 (Nov. 2012), pp. 56–65. DOI: 10.1038/nature11632. URL: http://dx.doi.org/10.1038/nature11632.

[16] Tuuli Lappalainen et al. "Transcriptome and genome sequencing uncovers functional variation in humans." eng. In: *Nature* 501.7468 (Sept. 2013), pp. 506–511. DOI: 10.1038/nature12531. URL: http://dx.doi.org/10.1038/nature12531.

[17] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed 2015-06-27]. 2001–. URL: http://www.scipy.org/.

[18] Andrey A. Shabalin. "Matrix eQTL: ultra fast eQTL analysis via large matrix operations." eng. In: *Bioinformatics* 28.10 (May 2012), pp. 1353–1358. DOI: 10.1093/bioinformatics/bts163. URL: http://dx.doi.org/10.1093/bioinformatics/bts163.

[19] Alkes L. Price et al. "Principal components analysis corrects for stratification in genome-wide association studies." eng. In: *Nat Genet* 38.8 (Aug. 2006), pp. 904–909. DOI: 10.1038/ng1847. URL: http://dx.doi.org/10.1038/ng1847.

[20] Nicoló Fusi, Oliver Stegle, and Neil D. Lawrence. "Joint modelling of confounding factors and prominent genetic regulators provides increased accuracy in genetical genomics studies." eng. In: *PLoS Comput Biol* 8.1 (Jan. 2012), e1002330. DOI: 10.1371/journal.pcbi.1002330. URL: http://dx.doi.org/10.1371/journal.pcbi.1002330.

[21] Black Duck. *Open Hub - Apache Spark Project Summary*. https://www.openhub.net/p/apache-spark. Acessed: 2015-06-25.

[22] Holden Karau et al. *Learning Spark*. Ed. by Ann Spencer and Marie Beaugureau. O'Reilly, 2015.

[23] Sandy Ryza et al. *Advanced Analytics with Spark*. Ed. by Ann Spencer. O'Reilly, 2014.

[24] *Spark Documentation*. https://spark.apache.org/documentation.html. Acessed: 2015-06-26.

[25] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[26] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28. ISBN: 978-931971-92-8. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[27] Michael Armbrust et al. "Spark SQL: Relational Data Processing in Spark". In: (2015).

[28] Patrick Wendell. *Tuning and Debugging in Apache Spark*. http://www.slideshare.net/pwendell/tuning-and-debugging-in-apache-spark. Acessed: 2015-06-27.

[29] Sandy Ryza. *How-to: Tune Your Apache Spark Jobs*. http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/. Acessed: 2015-06-27. Mar. 2015.