Computational challenges in genome wide association studies: data processing, variant annotation and epistasis

Pablo Cingolani

PhD.

School of Computer Science - Bioinformatics

McGill University

Montreal,Quebec

March 2015

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctor of Philosophy

ABSTRACT

Abstract...abstract...abstract...abstract...abstract...abstract...abstract...abstract...abstract

ABRÉGÉ

Abstract...abstract...abstract...abstract...abstract...abstract...abstract...abstract...abstract

TABLE OF CONTENTS

ABSTRACT ii

ABRÉGÉ iii

LIST OF TABLES vi

LIST OF FIGURES vii

1 Introduction 1

1.1 Motivation 1
1.1.1 Reference genome and genetic variants 3
1.1.2 DNA and disease 6
1.1.3 Type II diabetes 7
1.1.4 Missing heritability 8
1.1.5 Conclusions 10
1.2 Identification of genetic variants 10
1.2.1 Sequencing data 11
1.2.2 Sequence alignment 13
1.2.3 Read mapping 13
1.2.4 Mapping quality 15
1.2.5 Variant calling 15
1.3 Functional annotations of genomic variants 17
1.3.1 Functional annotations of coding variant 17
1.3.2 Non-coding annotations 18
1.3.3 Conclusions 18
1.4 Genome wide association studies 19
1.4.1 Single variant tests and models 19
1.4.2 Multiple variant tests 20
1.4.3 Continuous traits and correcting for co-factors . . . 21
1.4.4 Population structure 22
1.4.5 Population as confounding variable 22
1.4.6 Common and Rare variants 24
1.4.7 Rare variants test 24
1.4.8 Conclusions 25
1.5 Epistasis 25
1.5.1 Epistatic GWAS 28
1.6 Thesis roadmap and Contributions 30
1.6.1 Other contributions 32

2 BigDataScript: A scripting language for data pipelines 35

2.1 Preface 35
2.2 Introduction 36
2.3 Methods 40
2.3.1 Language overview 41
2.3.2 Abstraction from resources 42
2.3.3 Robustness 44
2.3.4 Other features 46
2.3.5 BDS implementation 49
2.4 Results 52
2.5 Discussion 56

Appendix A 58

A.1 Algorithm details 59

References 60

LIST OF TABLES

Table page

LIST OF FIGURES

Figure page

2–1 BDS report showing pipelines task execution timeline 51

2–2 Execution example. (A) Script `pipeline.bds`. (B) The script is executed from a terminal. The GO executable invokes main BDS, written in JAVA, performs lexing, parsing, compilation to AST and runs AST. (C) When the task statement is run, appropriate checks are performed. (D) A shell script `task1.sh` is created, and a bds-exec process is fired. (E) bds-exec reports PID, executed the script `task1.sh` while capturing stdout and stderr as well as monitoring timeouts and OS signals. When a process finishes execution, the exit status is logged 53

2–3 Whole-genome sequencing analysis pipelines flow chart, showing how computations are split across many nodes 54

2–4 Architecture independence example. Notes: Running the same BDS-based pipeline, a sequence variant calling and analysis pipeline, on the same dataset (chr20) but different architectures, operating systems and cluster management systems. 56

2–5 Scaling dataset sized by a factor of 1000. Notes: The same sample pipeline run on dataset of 2 GB (reads mapping to human chromosome 20) and 1.5 TB (whole-genome data set). Computational times vary according to systems resources, utilization factor and induced hardware failures. 56

CHAPTER 1
Introduction

1.1 Motivation

How does your DNA influence your risk of getting a disease? Contrary to popular belief, your future health is not "hard wired" in your DNA. Only in a few diseases, referred as "Mendelian diseases", there are well known, almost certain, links between genetic mutations and disease susceptibility. For the majority of what are known as "complex traits", such as cancer or diabetes, genomic predisposition is subtle and, so far, not fully understood.

With the rapid decrease in the cost of DNA sequencing, the complete genome sequence of large cohorts of individuals can now be routinely obtained. This wealth of sequencing information is expected allow the identification of genetic variations linked to complex traits. In this work, I investigate the analysis of genomic data in relation to complex diseases, which offers a number of important computational and statistical challenges. We tackle several steps necessary for the analysis of sequencing data and identifying the links to disease. Each step, which will correspond to a chapter in my thesis, is characterized by very different problems that need to be addressed.

i The first step is to analyze large amounts of information generated by sequencers to obtain a set of "genomic variants" that distinguish each individual. To address these big data processing problems, Chapter 2 shows how we designed a programming language (BigDataScript or BDS), that simplifies the creation robust, scalable data pipelines.

ii Once genomic variants are obtained, we need to prioritize and filter them to discern which variants should be considered "important" and which

ones are likely to be less relevant. In this process, known as "functional variant annotation" or simply "variant annotation", we calculate how the protein product would be affected and add information from relevant genomic databases (such as protein structure, deleteriousness scores or how often the variant is present in a population). We created SnpEff & SnpSift [**?**, **?**] packages that, using optimized algorithms, solve several annotation problems: a) standardize the annotation process, b) calculate putative genetic effects, c) estimate genetic impact, d) add several sources of genetic information, and e) facilitate variants filtering. We applied our methods in two large Genome Wide Association Studies (GWAS) for type II diabetes projects, in order to prioritize variants for statistical analysis. As a result of these studies, novel genes associated with diabetes and glycemic traits were found.

iii Finally, we address the problem of finding associations between "interacting genetic loci" and disease. One of the main problems in GWAS, known as "missing heritability", is that most of the phenotypic variance attributed to genetic causes remains unexplained. Since interacting genetic loci have been pointed out as one of the possible causes of missing heritability, finding links between such interactions and disease has great significance in the field. We propose a methodology to increase the statistical power of this type of approaches by combining population-level genetic information with evolutionary information.

In the rest of this introduction we give the background required to understand the material shown in Chapters 2 to 5 while providing motivations for our research. The transformation of raw sequencing data into biological insight in the aetiology of complex disease poses a series of computational,

analytical, algorithmic and methodological challenges that we address in the rest of this thesis.

1.1.1 Reference genome and genetic variants

DNA is composed of four basic building blocks, called "bases" or "nucleotides". These four nucleotides, usually abbreviated $\{A, C, G, T\}$, are Adenine, Cytosine, Guanine, and Thymine. Bases form pairs, either as $A - T$ or $C - G$, that pile-up forming two long polymers, with backbones that run in opposite directions giving rise to a double-helix structure. Arbitrarily, one of the polymers is called the positive strand and the other is called the negative strand.

The human genome has a total of 3 Giga-base-pairs (Gb), and those bases are divided into 23 chromosomes. We have two copies of each "autosomal" chromosomes, one inherited from our mother and one from our father. There are 22 autosomal chromosomes. The longest, being roughly 250 Mega-bases (Mb), is called "Chromosome 1" and the shortest, being 50 Mb is called "chromosome 22". We also have two sex chromosomes, called 'X' and 'Y'.

In order to be able to compare different objects length, we need some reference measure, such as the reference meter. Similarly, in order to compare DNA from different individuals (or samples), we need a "reference genome". The human reference genome (e.g. GRCh37) does not correspond to the DNA of any particular person, but to a "mosaic" of thirteen anonymous volunteers from Buffalo, New York [**?**].

When samples are sequenced, the DNA is compared to the "reference genome". Most of the DNA is the same, but there are differences. These differences, generically known as "genomic variants" (or "variants", for short), describe the particular genetic makeup of each individual. There are several

different ways a sample can differ from a reference genome. These are known as "variant types" and can be roughly categorized in the following way:

Single nucleotide variants (SNV) or Single nucleotide polymorphism (SNP) are the simplest and more common variants produced by single base difference (e.g. a base in the reference genome, at a given coordinate, is an A, whereas the sample is C). There are several biological mechanisms responsible for this type of variants: i) replication errors, ii) errors introduced by DNA repair mechanism, iii) deamination (a base is changed by hydrolysis which may not be corrected by DNA repair mechanisms), iv) tautomerism (and alteration on the hydrogen bond that results in an incorrect pairing).

Multiple nucleotide polymorphism (MNP) are differences of more than one base (e.g. reference is ACG whereas the sample is TGC).

Insertions (INS) refer to a sample having extra base(s) compared to the reference genome (e.g. reference is AT and sample is ACT). Some small insertion are usually attributed to DNA polymerase slipping and replicating the same base/s (this produces a type of insertion known as duplication). Large insertions are can be caused by unequal cross-over event (during meiosis) or transposable elements.

Deletions (DEL) are the opposite of insertions, the sample has some base(s) removed respect to the reference genome (e.g. reference is ACT and sample is AT). As in the case of insertions, deletions can also be caused by ribosomal slippage, cross-over events during meiosis and transposable elements.

Mixed variants can happen as a more complex combinations of combining SNV/MNP + Ins/Del.

Copy number variations (CNVs) arise when the sample has two or more copies of the same genomic region (e.g. a whole gene that has been duplicated or triplicated) or conversely, when the sample has less copies than the reference genome. Copy number variations can be attributed to problem during homologous recombination events.

Rearrangements are some complex variants that involve joining different regions (e.g. a translocation between chromosomes). Inversions, a type of rearrangement, result from a whole genomic region being inverted. These types of mutations are often attributed to cross-over events during meiosis.

As humans have two copies of each chromosome, variants could affect zero, one or two of the chromosomes and are called "homozygous reference", "heterozygous", and "homozygous alternative" respectively. Variants are also be classified on how common they are within the population: common, low frequency, or rare (see sections **??**). How these types of genetic variants influence traits or risk of disease is a topic of intense research that will be discussed throughout this thesis.

Proteins are composed by chains of amino acids and, as explained by the central dogma of biology, DNA is the template that instructs cellular machinery how to produce proteins. There are 4 bases in the DNA. There are 20 amino acids, which are the building blocks of all proteins. Each of the twenty amino acids is encoded by a group of three DNA bases called "codon". More than one codon can code for the same amino acid (i.e. $4^3 = 64$ codons > 20 amino acids) allowing for code redundancy. Additionally, there are codons that mark the end of the protein, these are called "STOP" codons and signal molecular machinery to end the transcription process. So variations in DNA may sometimes have direct effects on the protein product. We will talk

about this in section **??** and Chapter 3 where we cover the topic of "functional annotations".

1.1.2 DNA and disease

It would be fair to say that the Garrod family was fascinated by urine. As a physician at Kings College, Alfred Baring Garrod, discovered gout related abnormalities in uric acid [**?**]. His son, Sir Archibald Garrod, was interested in a condition known as alkaptonuria, in which children are mostly asymptomatic except for producing brown or black urine, but by the age of 30 individuals develop pain in joints of the spine, hips and knees. In 1902, Archibald observed that the family inheritance pattern of alkaptonuria resembled Mendels recessive pattern and postulated that a mutation in a metabolic gene was responsible for the disease. Publishing his finding he gave birth to a new field of study known as "Human biochemical genetics" [**?**].

Diseases having simple inheritance patterns, such as Cystic fibrosis, Phenylketonuria and Huntington's are also known as Mendelian diseases [**?**]. The genetic components of several Mendelian diseases have been discovered since the mechanism was first elucidated by Garrod in 1902 and the process has been accelerated in recent years, thanks to the application of DNA sequencing techniques [**?**].

In complex diseases (or complex traits), such as diabetes, cancer or Alzheimers, affected individuals cannot be segregated within pedigrees (i.e. no patterns of inheritance can be identified). As opposed to Mendelian diseases the aetiologically of complex traits is complicated due to factors such as: incomplete penetrance (symptoms are not always present in individuals who have the disease-causing mutation), oligogenic inheritance (characterized by more than

one gene) and genetic heterogeneity (caused by any of a large number of alleles). This makes it difficult to pinpoint the genetic variants that increase risk of complex disease.

1.1.3 Type II diabetes

Although this thesis focusses on the development of computational approaches that could be applied to the study of a number of complex diseases, our focus has been on type II diabetes mellitus (T2D), a complex disease first described by the Egyptians in 1500 BCE. Later the Greeks in 230 BCE used the term "diabetes" meaning "pass through" (or "siphon") denoting the constant thirst and frequent urination of the patients. In the 1700s the term "mellitus" (from honey) was added to denote that the urine was sweet and would "attracts ants".

Diabetes symptoms include frequent urination, thirst, and constant hunger, high blood sugar (hyperglycemia) and insulin resistance. Long term complication from T2D may include eyesight problems, heart disease, strokes and kidney failure. Type II diabetes, is highly correlated with obesity and disease rate has increased dramatically during the last 50 years. According to the World Health Organisation the prevalence of diabetes is 9% in adults and an estimated 1.5 millions deaths were caused by diabetes in 2012 [**?**], which is predicted to be the 7th leading cause of death by 2030. The costs associated to treating diabetes patients only in the U.S. are estimated around $245 billion dollars.

In recent years, over 80 genetic loci related to T2D have been identified [**?**]. Nevertheless, the overall effect sizes of these loci account for less than 10% of the overall disease predisposition [**?**]. This poses the question of why, given that so much efforts has been directed at finding the genetic components of this disease, the loci found so far have such modest effects. This lack of

large genetic effects do not only arise in T2D but also in almost all complex traits and could be explained by what is known as the "missing heritability" problem.

1.1.4 Missing heritability

We all know that "tall parents tend to have tall children", which is an informal way to say that height is a highly heritable trait. It is said that there are 30 cm from the tallest 5% to the shortest 5% of the population and genetics are accountable for 80% to 90% of this variation, which means that 27cm of variance are assumed to be "carried" by DNA variants from parents to offspring. Since 2010 the GIANT consortia has been investigating the genetic component of complex traits like height, body mass index (BMI) and waist to hip ratio (WHR). Even though they found many variants associated those traits, their findings only explain 10% of the phenotypic variance which corresponds to only a few centimeters in height [**?**].

In order to calculate heritability, we need to be able to measure it, so we need a formal definition. Heritability is defined as the proportion of phenotypic variance that is attributed to genetic variations. The total phenotypic variation is assumed to be caused by a combination of "environmental" and genetic variations $Var[P] = Var[G] + Var[E] + 2Cov[G, E]$ [**?**].

The environmental variance $Var[E]$ is the phenotypic variance attributable only to environment, that is the variance for individuals having the same genome $Var[E] = Var[P|G]$. Since cloning humans to calculate this term may be an overkill, we resort to approximate it based on phenotypic differences observed in monozygotic and dizygotic twins.

If the covariance factor $Cov[G, E]$ is assumed to be zero, we can define heritability as $H^2 = \frac{Var[G]}{Var[P]}$. This is called "broad sense heritability" because $Var[G]$ takes into account all possible forms of genetic variance:

$Var[G] = Var[G_A] + Var[G_D] + Var[G_I]$, where $Var[G_A]$ is the additive variance, $Var[G_D]$ is the variance form dominant alleles, and $Var[G_I]$ is the variance form interacting alleles (epistasis). Non-additive terms are difficult to estimate, so a simpler form of heritability called "narrow sense heritability" that only takes into account additive variance is defined as $h^2 = \frac{Var[G_A]}{Var[P]}$ [**?**].

Focusing on narrow sense heritability, the concept of "explained heritability" is defined as the part of heritability due to known variants with respect to all phenotypic variation ($\pi_{explained} = h^2_{known}/h^2_{all}$). Similarly, missing heritability is defined as $\pi_{missing} = 1 - \pi_{explained} = 1 - h^2_{known}/h^2_{all}$. When all variants associated with traits are known, then $\pi_{missing} = 0$.

Until recently, it was widely assumed by the research community that the problem of missing heritability lied in finding the appropriate genetic variants to account for the numerator of the equation (h^2_{known}) [**?**]. However, in a series of theorems published recently, it has been proposed that there is a problem in the way the denominator is estimated [**?**]. The authors created a limiting pathway model ($LP(k)$) that accounts for epistasis (gene-gene interactions) in k biological pathways. They showed that a severe inflation of h^2_{all} estimators occurs even for small values of k (e.g. $k \in [2, 10]$). As a result, genetic variants estimated to account only for 20% of heritability, could actually account for as much as 80% using an appropriate model [**?**].

Even though this result is encouraging, the problem is now shifted to detecting epistatic interactions, a problem that we analyze in section **??** and Chapter 4. In the same work [**?**], the authors show an example of power calculation assuming relatively large genetic effect that would require sequencing roughly $5,000$ individuals to detect links to genetic variants, which is a large but nowadays not uncommon, sample size. Nevertheless other estimates place the sample size requirements as high as $500,000$ individuals [**?**]. Even though

this sounds as an extremely large number of samples, it is quickly becoming possible thanks to large technological advances and cost reductions in sequencing and genotyping technologies.

1.1.5 Conclusions

Although some genetic causes of complex traits, such as type II diabetes, have been found, only a small portion of the phenotypic variance can be explained. This might indicate that many risk variants are yet to be discovered. Recent studies on the topic of missing heritability report that these "difficult to find genetic variants" might be in epistatic interaction (analyzed in section **??**) or rare variants (see section **??**), analysis of either them requires more complex statistical models and larger sample sizes. In Chapter 4 of this thesis, we focus on methods for finding epistatic interactions related to complex disease and develop computationally tractable algorithms that can process data from sequencing experiments involving large number of samples in a reasonable amount of time.

1.2 Identification of genetic variants

Two of the main milestones in genetics were the discovery of the DNA structure in 1953 [**?**], followed by the first draft of the human genome in 2004 [**?**]. The cost of sequencing the first human reference genome was around $3 billion (unadjusted US dollars) and it was an endeavor that took around 10 years. Since that time, sequencing technology has evolved substantially so that a human genome can now be sequenced in a three days for a price of less than $1,000, according to prices estimated by Illumina, one of the main genome sequencer manufacturers.

Having a standard reference sequence facilitates comparisons and analysis. For most well known organisms, "reference genome" sequences are available and current large scale sequencing projects are extending significantly the

number of genomes known, e.g. one project seeks to sequence 10,000 mammalian genomes [**?**], another is targeting all microbes that live within humans guts [**?**].

The amount of information delivered by sequencing devices is growing much faster than computer speed (Moore's law) and data storage capacity. Having to process huge amounts of sequencing information poses several challenges, a problem informally known as "data deluge". In the following sections, we explain how sequencing data is generated and how the huge amount of information delivered by a sequencer can be handled in order to make the problem tractable. Just as a crude example, a leading edge sequencing system is advertized to be capable of delivering 18,000 human genomes at $30x$ coverage per year, yielding over 3.2 PB of information. We want to transform this raw data into knowledge of genomic variants that contribute to disease risk with the ultimate goal to translate these risk variants into biological knowledge that can help to design drugs to treat or prevent disease. As expected, processing huge datasets consisting of thousands of sample is a complex problem. In Chapter 2 we show how mitigate or solve some of these issues, by designing a computer language specially tailored to tackle what are know as "Big data" problems.

1.2.1 Sequencing data

Different technologies for sequencing machines (or sequencers) exists. In a nutshell, a sequencer detects polymers (or chains) of DNA nucleotides and outputs a string of A, C, G, and Ts. Unfortunately, current technological limitations make it impossible to "read" a full chromosome as one long DNA sequence. Instead, modern sequencers produce a large number of "short reads", which range 100 bases to 20 Kilo-bases (Kb) in length, depending on the technology. Since sequencers are unable to read long DNA chains, preparing the

DNA for sequencing involves fragmenting it into small pieces. These DNA fragments are a random sub-samples of the original chromosomes. Reading each part of the genome several times allows to increase accuracy and ensure that the sequencer reads as much as possible of the original chromosomes. The coverage of a sequencing experiment is defined as the number of times each base of the genome is read on average. For instance, if the sequencing experiment is designed to produce one billion reads, and each read is 150 bases long, then the total number of bases read is 150Gb. Since the human genome is 3Gb, the coverage is said to be 50.

After sequencing a sample, we have millions of reads but we do not know where these reads originate from in the genome. This is resolved by aligning (also called mapping) reads to the reference genome, which is assumed to be very similar to the genome being sequenced. Once the reads are mapped, we can infer if the samples DNA has any differences with respect to the reference genome, a problem is known as "variant calling".

Using current technologies and computational methods for variant calling, detection accuracy varies significantly for different variant types. SNV are by far the most accurately detected. Insertions and deletions, collectively referred as InDels, can be detected less efficiently depending on their sizes. Small InDels consisting of ten bases or less are easier to detect than large InDels consisting of 200 bases or more. The reason being that the most commonly used sequencers reads DNA in stretches roughly 200 bases long. Due to this technological limitations, detection is less reliable for more complex variant types.

Although sequencing costs are dropping fast, it is still relatively expensive to sequence thousands of samples and in some cases it makes sense to focus on specific areas of the genome. A popular experimental setup is to focus on

coding regions (exons). A technique called "exome sequencing" consists of capturing exons using a DNA chip and then sequencing the captured DNA fragments only. Exons are roughly 3% of the genome, thus this technique reduces sequencing costs significantly, for which it has been widely used by many research groups.

1.2.2 Sequence alignment

Given two sequences s_1 and s_2 from an alphabet (e.g. $\Sigma = \{A, C, G, T\}$), the alignment problem is to add gap characters ('-') to both sequences, so that a distance, such as Levenshtein distance, $d(s_1, s_2)$ is minimized.

This problem has a well known solution, the Smith-Waterman algorithm [**?**], which is a variation of the global sequence alignment solution from Needleman-Wunsch [**?**]. The main problem is that the algorithm is $O(l_1.l_2)$ where l_1 and l_2 are the length of the sequences. So, Smith-Waterman algorithm is slow for very long sequences, such as the human genome.

In order to speed up sequence alignments, several heuristic approaches emerged. Most notably, BLAST [**?**], which is used for mapping sequences several thousand nucleotides long (i.e. longer than a typical sequencer read) to a reference genome. BLAST uses an index to map parts of the query sequence, called seeds, to the reference genome. Once these seeds have been positioned against the reference, BLAST joins the seeds performing an alignment. Since the alignment is performed only using a small part of the reference, the algorithm is much faster.

1.2.3 Read mapping

Sequence alignment has an exact algorithm solution and several faster heuristic solutions. But even the fastest solutions are too slow to be used with the millions of reads generated in a typical sequencing experiment. Faster algorithms can be used if we relax our requirements in two ways: i) we allow

for sub-optimal results, and ii) instead of requiring information of where each base of the read maps to the reference genome, we just want to know where the first base maps. This relaxed version of the alignment algorithm is called "read mapping" and the reduced complexity is enough to speed up the computations significantly. An implicit assumption in this formulation, is that the read will be very similar to the reference and that there will be no big gaps. Once the mapping is performed, the read is locally aligned, a strategy similar to BLAST algorithm [**?**].

Reformulating the problem this way, allows us to use other methods, such as suffix array [**?**]. Suffix arrays algorithms are fast, but memory requirements are $O[n\ log(n)]$ and this becomes the limiting factor. In order to reduce memory footprint of suffix arrays, Ferragina and Manzini [**?**] created a data structure based on the Burrows-Wheeler transform. This structure, known as an FM-Index, is memory efficient yet fast enough to allow mapping high number of reads. An FM-index for the human genome can be built in only 1Gb of memory, compared to 12Gb required for an equivalent suffix array [12]. Given a genome G and a read R, an FM-index search can find the N_{occ} occurrences of R in G in $O(|R| + N_{occ})$ time, where $|R|$ is the length of R [12].

Efficient indexing and heuristic algorithms can decrease mapping time considerably. Nevertheless, these algorithms are not guaranteed to find an optimal mapping. Several parameters, such as read length, sequencing error profile, and genome complexity profile can affect performance. The most commonly used implementation of the FM-index mapping algorithms are BWA [12, 13] and Bowtie [10, **?**]. Each of them provide optimized versions for the two most common sequencing types: i) short reads with high accuracy [12, 10] or ii) longer reads with lower accuracy [13, **?**].

It is worth noting that the mapping problem appears as a consequence of the technological limitations of sequencers. Having long, highly accurate reads, the problem becomes much easier to solve. As an extreme example, having only one read which is as long as a chromosome and has no errors requires no mapping processing.

1.2.4 Mapping quality

Sequencers not only provide sequence information, but also provide an error estimate for each base [**?**]. This is often referred as a quality (Q) value, which is the probability of an error, measured in negative decibels $Q = -10\ log_{10}(p)$.

Mapping quality is an estimation of the probability that a read is incorrectly mapped to the reference genome. Mapping algorithms provide estimates of mapping errors. In the MAQ model [**?**], which is one of the earliest models for calculating mapping quality, three main sources of error are explored: i) the probability that a read does not originate from the reference genome (e.g. sample contamination); ii) the probability that the true position is missed by the algorithm (e.g. mapping error); and iii) the probability that the mapping position is not the true one (e.g. if we have several possible mapping positions). It is assumed that the total error probability can be approximated as $\epsilon \approx max(\epsilon_1, \epsilon_2, \epsilon_3)$.

1.2.5 Variant calling

Once the sequencing reads have been mapped to the reference genome, we can try to find the differences between a sequenced sample and the reference genome. This is referred as "variant calling". Several factors complicate this task, the two main ones being sequencing errors and mapping errors, described in **??**. Using sequencing and mapping error estimates, a maximum likelihood model can infer when there is a mismatch between a sample and the reference

genome [?]. This method works best for differences of a single base (SNV), but it can also work with different degrees of success for short insertions or deletions (InDels) usually consisting of less than 10 bases.

Due to the nature of short reads, this family of methods does not work for structural genomic variants, such as large insertions, deletions, copy number variations, inversions, or translocations. A different family of algorithms are used to identify structural variants, but their accuracy so far has been low compared to SNV calling algorithm [?].

Aligning sequences that contain InDels (gaps) is more difficult than ungapped alignments since finding optimal gap boundary depends on the scoring method being used. This biases variant calling algorithms towards detecting false SNVs near InDels [?]. An approach to reduce this problem is to look for candidate InDels and perform a local realignment in those regions. This local re-alignment process reduces significantly the number of false positive SNVs [?]. Another approach to reduce the number of false positive SNVs calls near InDels involves the "Base Alignment Quality" (BAQ) [?], which is the probability of misalignment for each base. It can be shown that replacing the original base quality with the minimum between base quality and BAQ produces an improvement in SNV calling accuracy. The BAQ can be calculated using a special type of "Hidden Markov Model" (HMM) designed for sequence alignment [?, ?]. A more sophisticated option for reducing errors consist of performing a local genome re-assembly on each polymorphic region (e.g. HaplotypeCaller algorithm [?]).

Finally, the error probabilities inferred by the sequencers are far from perfect. Once the variants have been called, empirical error probabilities can be easily calculated [?] by comparing sequenced variants to a set of "gold standard variants" (i.e. variants that have been extensively validated). This

allows to re-calibrate or re-estimate the error profile of the reads. This is know as a re-calibration step, and usually improves the number of false positives calls [?].

1.3 Functional annotations of genomic variants

Once DNA is sequenced, reads are mapped and variants are called as described in previous sections, variants identified are annotated in order to gain biological insight. For instance, we would like to know if it is located in a gene and if so whether the variant could be deleterious to the functionality of the protein encoded by the gene. This is the focus of Chapter 3 of my thesis.

The simplest case of a genetic annotation would be to know whether a genetic variant lies onto a gene or not. This would be trivial to calculate, since it only requires comparing the genomic location of the variant with the genomic location of known genes. However, in a sequencing experiment there are usually millions of variants and hundreds of thousands of genomic "features" such as genes, transcripts, exons, introns, splice regions, promoters, etc. The sheer volume of data requires time and memory efficient algorithms and data structures.

1.3.1 Functional annotations of coding variant

Genetic variants that are located within the coding region of a protein-coding gene are called coding variants. Although they form a small subset of all variants, they are the ones whose function can best be predicted. As explained by the central dogma of biology, genetic information flows from DNA molecules to mRNA molecules, which are used as a template to produce proteins.

A coding variant that produces a codon change is called "synonymous" or "non-synonymous" depending on whether the resulting amino acid remains the same of changes. If the variant is synonymous, we can be reasonably

confident that there will be almost no effect in protein function, conversely if a non-synonymous variant creates a new STOP codon, it might be a strong indicator that protein function will be disrupted thus the variant is deleterious.

Estimating the putative effect of large coding variants (duplications, inversions or fusions) is much more challenging than in the cases of simple variants (SNVs, MNVs and small InDels) since there is still not enough studies to determine what effects large coding variants have in protein expression or function.

1.3.2 Non-coding annotations

For variants in non-coding regions of the genome, annotations are more difficult than in coding regions, mostly due to the fact that not only the location of most non-coding features (such as transcription factor binding sites, chromatin modifications, methylation) are not exactly known, but also non-coding features tend to be tissue specific. How DNA variants affect non-coding features is mostly speculative, and even for the few non-coding feature that have predictive models, these are riddled with false positives.

Assuming that non-coding features, or parts of them, have selective pressure to keep their functionality, conservation scores can be used as a proxy on how "important" these regions are. Nevertheless, this might not apply for certain classes of non-coding features, such as some transcription factors, where there is evidence of negative selection (meaning that transcription factors binding sites might not only not be conserved, but also change more rapidly than other genomic regions).

1.3.3 Conclusions

In Chapter 3 we show two software packages we designed for efficiently performing functional annotations of sequencing variants. These packages, SnpEff & SnpSift, allow to annotate, prioritize, filter and manipulate variant

annotations as well as combine several public or custom-created databases. It should be noted SnpEff was one of the first annotation packages and has become one of the most widely used annotation software in both research and clinical environments.

1.4 Genome wide association studies

A genome wide association study aims at identifying genetic variants associated to a particular phenotype. First, the genomes (or exome, depending on the study design) of affected individuals (cases) and healthy individuals (controls) need to be sequenced, variants called, annotated and filtered. Then, the goal is to find variants that exhibit some statistical association with the trait or phenotype of interest, which could be a disease status (e.g. diabetes vs healthy), a biomedical measurement (e.g. cholesterol level), or any measurable characteristic (e.g. height). Since the genome is so large, patterns of mutations that suggest correlation may be encountered by chance, so we need to establish statistical significance in order to distinguish true association from spurious ones. Like most studies, we will focus on SNVs, but most methods can be extended to other genomic variants.

1.4.1 Single variant tests and models

Let's imagine that there is only one variant in the whole genome for the cohort we are analyzing. Since each individual has two sets of chromosomes, the variant can be present in one, both, or neither chromosomes. When a variant is in both chromosomes is said to be "homozygous", whereas if present in only one of the chromosomes, it is said to be "heterozygous". So the number of times a non-reference allele is present in an individual, is $N_{nr} = \{0, 1, 2\}$.

When the trait of interest is binary (e.g healthy vs disease), a cohort can be divided into cases and controls and we can build a 3 by 2 contingency table:

	$HomozygousReference$ ($N_{variant} = 0$)	$Heterozygous$ ($N_{nr} = 1$)	$Homozygousnon - reference$ ($N_{nr} = 2$)
$Cases$	$N_{ca,ref}$	$N_{ca,het}$	$N_{ca,hom}$
$Controls$	$N_{co,ref}$	$N_{co,het}$	$N_{co,hom}$

Further assumptions about how many variants are required to increase disease risk can reduce this 3×2 table to a 2×2 table. In the "dominant model", the effect of a mutated gene dominates over the healthy one, so one variant is enough to increase risk. The opposite, called "recessive model", is when both chromosomes have to be mutated in order to increase risk [**?**, **?**]. In these models, we can count how many cases and controls have at least one variant (dominant model) or two variants (recessive model). This simplifies the previous table, yielding a 2×2 contingency table, than can be tested using either a χ^2 test or a Fisher exact test [**?**].

Two other commonly used models, are the "multiplicative" and the "additive" models [**?**, **?**]. In these models, a disease risk is assumed to be multiplied (or increased) by a factor γ with every variant present. We cannot simplify the contingency table, so we assess significance using a Cochran-Armitrage test [**?**].

1.4.2 Multiple variant tests

In a real case scenario there are thousands or millions of variants. We can extend the concept shown in the previous section by performing individual tests for each variant present in the cohort. Multiple testing can be addressed either by performing a correction, such as False Discovery Rate [**?**, **?**], or using a stricter genome wide significance level. There are 3×10^9 bases in the

genome, but taking into account the correlation between nearby variants (linkage disequilibrium), the genome wide significance level is generally accepted to be $p_{value} \leq 10^{-8}$.

In order to check if the null hypothesis of a significance tests is adequate, a QQ-plot is used (i.e. plotting the $y = -log(p_{value})$ vs $x = -log[rank(p_{value})/(N+1)]$, where N is the total number of variants). Adherence of the p-values to a 45 degree line on most of the range implies few systematic sources of association [**?**, **?**]. If the p-values have a higher slope than the $y = x$ line, there might be "inflation", possibly due to co-factors, such as population structure (see section **??**). If the inflation is not too high (e.g. less than 5%), this bias can be corrected by shifting the p-values towards the 45 degree slope. More sophisticated methods are explained in section **??**.

1.4.3 Continuous traits and correcting for co-factors

Methods analyzed so far are suitable for binary "traits" or "phenotypes" (e.g. disease vs. healthy individuals). Statistical methods that link genetic information to traits can also be used on continuous or "quantitative" traits (e.g. weight, height, cholesterol level, etc.). A linear regression can be used assuming the traits are approximately normally distributed [**?**, **?**]. A significance test (p_{value}) for linear models can be calculated using an F statistic, but more sophisticated methods are also available [**?**, **?**].

Using linear models, it is easy to include known co-factors to correct for biases or inflation. For instance, if it is known that a risk increases with age or that males are more susceptible than females, age and sex can be added to the linear equation in order to correct for these effects [**?**, **?**]. In a similar manner, we can add co-factors to binary traits using logistic regression.

1.4.4 Population structure

It is widely accepted that humans started in Africa and migrated to Europe, then to Asia and later to America [**?**]. Out of an initial population, a few individuals migrate and colonize a new territory. This implies that the genetic variety of the new colony is significantly reduced, compared to the previous population, since the genetic pool is only a small "founder population". The "Out of Africa" hypothesis implies that each new migration produced a reduction in genetic variety, also known as a "population bottleneck" [**?**].

As we previously mentioned, each individual inherits two chromosome sets, a maternal and a paternal one. In a process known as recombination, a chromosome that is formed by part of the maternal chromosome and part of the paternal one, is inherited to the offspring. As a result of recombination, a child has two sets of chromosomes that are one from each parent and, on average, half of a chromosome from each grandparent. This breaking and shuffling of chromosomes every generation, increases genetic diversity. Nevertheless if variants are located nearby in the chromosome, the chances that they are broken apart by recombination event are smaller than if they are further away from each other. This produces a correlation of close variants or "linkage disequilibrium" (LD). Nearby highly correlated variants are said to be in the same "LD-block" [**?**]. If a population has low genetic variety, the LD-blocks are large. So African population has more variety (smallest LD-blocks) and conversely, European, Asian and Amerindian populations have less variety (larger LD-blocks) [**?**].

1.4.5 Population as confounding variable

Imagine that we have a cohort of individuals drawn from two populations (P_A and P_B) and that individuals in P_A have much higher risk of diabetes than individuals from P_B. Now imagine that individuals from P_A have a variant v_A

more often, but v_A is actually neutral and has no health effects whatsoever. If we do not take into account population factors, our study would conclude that $variant_A$ is the cause of diabetes, just because we see $variant_A$ more often in affected individuals. In this case is clear that population structure is being a confounding variable. We could avoid this problem by analyzing each population separately [**?**], but this would cause a loss of statistical power since we have fewer samples.

A population that is a mixture of two or more populations, is known as an "admixed population". For instance the "African-American" population is a mixture of, roughly, 80% African and 20% European genomes [**?**, **?**]. This means that analyzing a cohort of African-American individuals, we would get population structure as a confounding variable because of population admixture [**?**]. Obviously, in this case we cannot analyze each population separately, because each individual in the sample is a mixture of two populations.

The admixed population problem can be studied by performing a correction using the eigen-structure of the sample covariance matrix [**?**]. Samples can be arranged as a matrix C where each row is a sample and each column represents a position in the genome where there is a variant. The numbers $C_{i,j}$ in the matrix indicate whether a sample (row i) has a non-reference allele at a genomic position (column j). Since the allele can be present in zero, one, or two chromosomes in each individual, the possible values for $C_{i,j}$ are $\{0, 1, 2\}$. The covariance matrix is calculated as $M = \hat{C}^T.\hat{C}$, where $\hat{C}$ is the matrix C corrected to have zero mean columns. Usually, the first two to ten principal components of M are used as factors in linear models (see section **??**) to correct for population structure [**?**].

Whether a cohort has any population structure and needs correction or not, can be tested using two methods: a) plotting the projections of the first

two principal components and empirically observing the number of clusters in the chart, or b) using a statistic of the eigenvalues of M based on Tracy-Widom's distribution [**?**].

1.4.6 Common and Rare variants

The "allele frequency" (AF) is defined as the frequency a variant appears in a population. Variants are usually categorized according to AF into three groups: i) Common variants ($AF \geq 5\%$), "low frequency" ($1\% < AF < 5\%$), and iii) "rare variants" ($AF < 1\%$). Common variants originated earlier in the population while rare variants are either relatively recent or selected against.

There are three main models for disease susceptibility [**?**, **?**]:i) the Common-Disease-Common-Variant hypothesis (CDCV) assumes that if disease is common, it must be caused by a common variant; ii) the "infinitesimal hypothesis" proposes that there are many common variants each having small risk effects; and iii) the Common-Disease-Rare-Variant hypothesis proposes that there exists many rare variants, each one having large risk effects.

1.4.7 Rare variants test

The "rare variant model" assumes that multiple rare variants have large effects on a trait. The problem is that, since these variants are rare, huge sample sizes are required for tests to identify statistically significant associations. To overcome this problem, methods known as "burden tests", collapse several rare variants and perform statistical significance tests on grouped variants [**?**]. An example of collapsing technique is to count the number or rare variant in a given window and apply a Fisher exact test, as shown in section **??**. A limitation of some burden tests is that they implicitly assume that all rare variants have the same direction of effect, although rare variants might have no effect, be deleterious, or protective [**?**, **?**].

Several techniques allow weighting rare variants by collapsing them using a kernel matrix. This allows to incorporate other information, such as allele frequency and functional annotations. It can be shown that the statistic induced by kernel weighting functions follows a mixture of χ^2 distributions and there is an efficient way to approximate it [**?**, **?**], avoiding computationally expensive permutations tests.

1.4.8 Conclusions

In this section we introduced the basic concepts and methodologies used in GWAS. Although fairly mature, there is still heavy research and continuous improvement on GWAS statistical methods. Not only it is well known that traditional (i.e. single marker) GWAS methods fail under non-additive models [**?**], but also variants so far discovered using these methods do not account for all the expected phenotypic variance attributed to genetic causes (i.e. missing heritability). As other authors pointed out, this might be because we need to look for epistatic variants which are not taken into account using these methods. In the next section, and in Chapter 4, we cover the topic of epistatic GWAS analysis.

1.5 Epistasis

Proteins are the most important part of the cell composing up to 50% of a cells dry weight compared to 3% of the DNA [**?**]. Proteins perform their functions mainly by interacting with other proteins, forming complex pathways that lead to a vast array of cellular functions including catalysis of chemical reactions, cell signaling, and structural conformation of the cell. The 3-dimensional structure of the protein, also called "tertiary structure", is tailored to bind to other proteins in a specific manner to accomplish a functionality.

Genome wide association studies focus on single variants or nearby groups of variants. An often cited reason for the lack of discovery of high impact risk factors in complex disease is that these models ignore loci interactions [**?**] and recently they have been pointed out as a potential solution for the "missing heritability" problem [**?**]. With interactions being so ubiquitous in cell function, one may wonder why they have been so neglected by GWAS. We should point out that there are several reasons: i) models using interactions are much more complex [**?**] and by definition non-linear, ii) information on which proteins interacts with which other proteins is incomplete [**?**], iii) in the cases where there protein-protein interaction information is available, precise interacting sites are unknown [**?**]. Taking into account the last two items, we need to explore all possible loci combinations, thus the number of Nth order interactions grows as $O(M^N)$ where M is the number of variants [**?**]. This requires exponentially more computational power than single loci models. This also severely reduces statistical power, which translates into requiring larger cohort, thus increasing sample collection and sequencing costs [**?**].

In Chapter 4 we develop a computationally tractable model for analyzing putative interaction of pairs of variants from sequencing experiments involving large case / control cohorts of complex disease. Our model is based on combining multiple sequence alignments using a coevolutionary model in order to perform GWAS analysis of pairs of non-synonymous variants that may interact.

5.1 Detection of interacting sites in proteins using co-evolution

Proteins interactions and interaction loci are expensive to identify reliably experimentally and difficult to predict computationally. Some computational prediction methods are based on the assumption that protein interactions sites are under evolutionary pressure to avoid mutations [**?**], because such mutations

reduce the efficiency or even disrupt pathways. Assuming that evolutionary pressure maintains favorable interaction between loci, compensatory mutations can happen more often than non-compensatory ones. The underlying idea is that fitness is higher for compensatory mutations and higher fitness co-occurring mutations would be fixed in the population. Since several organisms have been sequenced, we can try compare orthologous protein sites occurring in all these organisms and seek evidence of coevolution. It should be noted that this approach can be used to detect interacting sites within two different proteins or two interacting sites within the same protein. Detecting interacting sites within the protein can be valuable for determining protein structure. The most widely used method for inferring co-evolution starts from protein multiple sequence alignments ($\mathcal{M}_{sa}$), and identifies a pair of sites (e.g. one site from each interacting protein) that maximizes mutual information (MI) [**?**]. It is known that MI has some limitations [**?**] and is biased due to the fact that the multiple sequence alignment are related by an evolutionary process [**?**]. This means that some sequences will be very similar since they are evolutionarily close to each other (e.g. human and chimp), whereas other sequences will be very different (e.g. mouse and coelacanth).

A proper albeit more complex statistical analysis takes into account MSAs phylogenetic tree. Sophisticated coevolutionary models are usually designed with the intent of aiding protein structure predictions, which require to pinpoint the exact loci in each protein. These complex models can take anywhere from minutes to days to run for each pair of proteins, thus making them unfit for GWAS-scale analysis.

We propose to make use of co-evolutionary information to increase the interaction priors in a GWAS model. Since the goal is to increase GWAS

priors instead of pinpointing the exact interaction loci, we can relax coevolutionary methods requirements to design computationally tractable models. In Chapter 4, we introduce an epistatic GWAS approach that while combining coevolutionary and sequencing information it is efficient enough to be applied to GWAS-scale, large cohort, datasets.

1.5.1 Epistatic GWAS

Arguably, the most common model linking binary phenotypes (disease vs. healthy) to genotypes is the logistic regression model which relates log odds probability of disease using multiple regression, $ln(\frac{p}{1-p}) = \hat{\beta}^T \hat{g}$, where p is the probability of disease, $\hat{g}$ are the models input variables (usually including genotype, sex, age, population structure, etc.), and $\hat{\beta}$ are the logistic regression coefficients.

Given a set of genotypes (typically genotype analysis includes 2,000,000 variants and tens of thousands of samples), the simplest way to look for interactions is an exhaustive search of all combinations. This raises two issues: i) multiple testing, which is often resolved by stringent significance threshold, and ii) computational feasibility, which is solved by efficient algorithms, parallelization, and heuristic approaches to quickly discard uninformative loci combinations.

The definition of epistasis, form a statistical perspective, is a "departure from a linear model" [**?**]. This means that in a logistic regression model the input includes terms with each of the genotypes (g_i and g_j), as well as an "interaction term" $gi.g_j$ [**?**]. Although we mainly talk about interaction between two loci, higher order interactions (three or more loci combinations) can be analyzed, but these models require more parameters and extremely large samples are required to accurately fit them.

Although a comprehensive review is out of the scope of this thesis, it is worth mentioning that several other approaches for epistatic GWAS exist. Here we mention a few (shown in alphabetic order):

- Allele frequency: In [**?**], an analysis of imbalanced allele pair frequencies is performed under the assumptions that an implicit test for fitness can be achieved looking for over/under-represented allele pairs in a given population.
- Bayesian model: In [**?**], a "Bayesian partitioning model" is used by providing Dirichlet prior distributions for each partition and computing posterior probabilities using Markov chain Monte Carlo (MCMC) algorithms. The methodology first test individual makers and picks only the top 10% to further investigate for epistasis, because it is prohibitive to test all loci.
- Linkage disequilibrium: Studying LD patterns in a population under two-loci model it was shown [**?**] that interactions creates LD in disease population. The authors show how LD-based p-values can uncover interaction and sometimes (in their simulations) outperform logistic regression tests.
- Machine learning: From a machine learning point of view, finding interacting variants is simply an optimisation and attribute selection procedure [**?**]. Several approaches have emerged to tackle the "interaction problem" and used a variety of different techniques [**?**, **?**] , such as neural networks, cellular automata, random forests, multifactor dimensionality reduction, support vector machines, etc.

Although all these models have advantages under some assumptions, none of them seems to be a "clear winner" over the rest [**?**], thus currently there are no de-facto standards in epistatic analysis. In light of this, there is need of

different approaches to be explored. In Chapter 5 we combine coevolutionary models and GWAS epistasis of pairs of putatively interacting loci, by using Bayes Factors to combine information.

1.6 Thesis roadmap and Contributions

The original research presented in this thesis covers topics related to the computational and statistical methodologies related to the analysis of sequencing variants to unveil genetic links to complex disease. Broadly speaking, we address three types of problems: (i) Data processing of large datasets from high throughput biological experiments such as resequencing in the context of a GWAS (Chapter 2); (ii) functional annotations, i.e. calculating variants impact at molecular, cellular or even clinical level (Chapter 3); (iii) identification of genetic risk factors for complex disease using models that combine population-level and evolutionary-level data to detect putative epistatic interactions (Chapter 4). It should be pointed out that the chapters are ordered similar to the analysis steps we used when analyzing our data for type II diabetes, starting from raw sequencing data and ending with GWAS analysis. When applicable, background material specific to each chapter is presented in a preface, together with an explanation of how that chapter ties in with the rest of the thesis.

This thesis comprises text and figures of scientific articles which have either been published, submitted for publication, or ready to be submitted (waiting upon data embargo restrictions):

Chapter 2 For this paper, PC conceptualized the idea and performed the language design and implementation. RS & MB helped in designing robustness testing procedures. PC, RS & MB wrote the manuscript.

- **Cingolani, Pablo**, Rob Sladek, and Mathieu Blanchette. "BigDataScript: a scripting language for data pipelines." Bioinformatics 31.1 (2015): 10-16.

Chapter 3 For this paper, PC designed, implemented and tested SnpEff & SnpSift. RS & MB suggested several extensions for common research use cases. PC, RS & MB wrote the manuscript. The manuscript was submitted to Nature Protocols, and the editor suggested for it to be published after the main T2D paper is accepted for publication (see next paragraph).

- **Cingolani, Pablo**, Rob Sladek, and Mathieu Blanchette. "Genomic variant annotation and prioritization" Ready for submission (waiting on consortia paper submission).

The following studies are T2D (type II diabetes) consortia projects which used SnpEff and SnpSift extensibly, several modules were designed with these projects in mind. This are part of a large consortia involving several institutions:

- McCarthy M., et al (T2D Genes Consortia). "Variation in protein-coding sequence and predisposition to type 2 diabetes", Ready for submission.
- Mahajan, Anubha, et al. "Identification and Functional Characterization of G6PC2 Coding Variants Influencing Glycemic Traits Define an Effector Transcript at the G6PC2-ABCB11 Locus." PLoS genetics 11.1 (2015): e1004876-e1004876.

The original SnpEff and SnpSift publications are provided in the appendices:

- **Cingolani, Pablo**, et al. "A program for annotating and predicting the effects of single nucleotide polymorphisms, SnpEff: SNPs in

the genome of Drosophila melanogaster strain w1118; iso-2; iso-3." Fly 6.2 (2012): 80-92.

- **Cingolani, Pablo**, et al. "Using Drosophila melanogaster as a model for genotoxic chemical mutational studies with a new program, SnpSift." Toxicogenomics in non-mammalian species (2012): 92.

Chapter 4 For this paper, PC designed the methodology under the supervision of MB and RS. PC implemented the algorithms. PC, RS & MB wrote the manuscript. This work uses data from the T2D consortia, thus it cannot be published until the main T2D paper is accepted for publication (according to T2D data embargo).

- **Cingolani, Pablo**, Rob Sladek, and Mathieu Blanchette. "A co-evolutionary approach for detecting epistatic interactions in genome-wide association studies" Ready for submission (data embargo restrictions).

1.6.1 Other contributions

Other scientific articles (grouped by topic) published, submitted for publication, or ready to be submitted, not mentioned in this thesis:

Epigenetics

- **Cingolani, Pablo**, et al. "Intronic Non-CG DNA hydroxymethylation and alternative mRNA splicing in honey bees." BMC genomics 14.1 (2013): 666.
- Senut, Marie-Claude, et al. "Lead exposure disrupts global DNA methylation in human embryonic stem cells and alters their neuronal differentiation." Toxicological Sciences (2014).

- Ruden D., "Epigenetics as an answer to Darwins special difficulty Part 2: Natural selection of metastable epialleles in honeybee castes", Submitted.
- Arko S, et al. "Lead exposure induces changes in 5-hydroxymethylcytosine clusters in CpG islands in human embryonic stem cells and umbilical cord blood", Submitted.
- Senut, Marie-Claude, et al. "Epigenetics of early-life lead exposure and effects on brain development." Epigenomics 4.6 (2012): 665-674.

GWAS & Disease

- Oualkacha, Karim, et al. "Adjusted sequence kernel association test for rare variants controlling for cryptic and family relatedness." Genetic epidemiology 37.4 (2013): 366-376.
- Bongfen, Silayuv E., et al. "An N-ethyl-N-nitrosourea (ENU)-induced dominant negative mutation in the JAK3 kinase protects against cerebral malaria." PloS one 7.2 (2012): e31012.
- Hawn, Thomas R., et al. "Host-directed therapeutics for tuberculosis: can we harness the host?." Microbiology and Molecular Biology Reviews 77.4 (2013): 608-627.
- Meunier, Charles, et al. "Positional mapping and candidate gene analysis of the mouse Ccs3 locus that regulates differential susceptibility to carcinogen-induced colorectal cancer." PloS one 8.3 (2013): e58733.
- Caignard, Grgory, et al. "Genome-wide mouse mutagenesis reveals CD45-mediated T cell function as critical in protective immunity to HSV-1." PLoS pathogens 9.9 (2013): e1003637.

- Bouttier M., et al. "Genomics analysis reveals elevated LXR signaling reduces M. tuberculosis viability", Submitted.
- Bouttier M., et al. "Genomic analysis of enhancers engaged in M. tuberculosis-infected macrophages reveals that LXR signaling reduces mycobacterial burden", Submitted.

Other

- **Cingolani, Pablo**, and Jesus Alcala-Fdez. "jFuzzyLogic: a robust and flexible Fuzzy-Logic inference system language implementation." FUZZ-IEEE. 2012.
- **Cingolani, Pablo**, and Jess Alcal-Fdez. "jFuzzyLogic: a java library to design fuzzy logic controllers according to the standard for fuzzy control programming."International Journal of Computational Intelligence Systems

CHAPTER 2
BigDataScript: A scripting language for data pipelines

2.1 Preface

The overall goal in this thesis is to find genetic loci related to complex disease. In order to have enough statistical power to find these risk loci, we need to sequence thousands of cases and controls (i.e. patients and healthy individuals). Obviously the first step is to find all these patients, obtain patients consent, take samples and keep track of clinically relevant variables (such as age, sex, BMI, and glycemic traits). Just by the sheer number of patients involved, its easy to see that the logistics are challenging, to say the least.

Once the sequencing of each patients DNA is performed, we need to process the raw sequencing information by performing what is known as primary sequencing analysis, which involves mapping reads to the reference genome, calling variants, as well as performing several types of quality controls. The term primary analysis makes it sound as if this step is simple,but it is not. Managing such volume of information is a huge task that requires large computational resources, and coordinating the process involved at every stage of the analysis is not trivial, even if the jobs are relatively easy to parallelize.

As an example of the complexity and data volumes involved in these analysis pipelines, mapping the raw reads to the reference genome (i.e. the first stage of the primary analysis) for our T2D sequencing data is estimated to take over 12,000 CPU hours, that is over 32 CPU/years, under the most optimistic assumptions. At this magnitude hardware and failures become a significant issue since the probability of one or more nodes malfunction, while the data is being processed, is quite high.

We designed and implemented a simple script-like programming language called BigDataScript (BDS), with a clean and minimalist syntax to develop and manage pipeline execution and provide robustness to various types of software and hardware failures as well as portability. This programming language specifically tailored for data processing pipelines, improves abstraction from hardware resources and assists with robustness. Hardware abstraction allows BDS pipelines to run without modification on a wide range of computer architectures, from a small laptop to multi-core servers, server farms, clusters, clouds or even whole datacenters. BDS achieves robustness by incorporating the concepts of absolute serialization and lazy processing, thus allowing pipelines to recover from errors. By abstracting pipeline concepts at programming language level, BDS simplifies implementation, execution and management of complex bioinformatics pipelines, resulting in reduced development and debugging cycles as well as cleaner code. BDS was used to create data analysis pipelines required for our research, including the ones described throughout this thesis, and is currently used by other research groups and sequencing facilities in both academic and private environments.

The rest of the chapter is published in: Cingolani, Pablo, Rob Sladek, and Mathieu Blanchette. "BigDataScript: a scripting language for data pipelines." Bioinformatics 31.1 (2015): 10-16.

2.2 Introduction

Processing large amounts of data is becoming increasingly important and common in research environments as a consequence of technology improvements and reduced costs of high-throughput experiments. This is particularly the case for genomics research programs, where massive parallelization of microarray and sequencing-based assays can support complex genome-wide experiments involving tens or hundreds of thousands of patient samples [**?**]. With

the democratization of high-throughput approaches and simplified access to processing resources (e.g. cloud computing), researchers must now routinely analyze large datasets. This paradigm shift with respect to the access and manipulation of information creates new challenges by requiring highly specialized skill, such as implementing data-processing pipelines, to be accessible to a much wider audience.

A data-processing pipeline, referred as "pipeline" for short, is a set of partially ordered computing tasks coordinated to process large amounts of data. Each of these tasks is designed to solve specific parts of a larger problem, and their coordinated outcomes are required to solve the problem as a whole. Many of the software tools used in pipelines that solve big data genomics problems are CPU, memory or I/O intensive and commonly run for several hours or even days. Creating and executing such pipelines require running and coordinating several of these tools to ensure proper data flow and error control from one analysis step to the next. For instance, a processing pipeline for a sequencing-based genome-wide association study may involve the following steps [**?**]: (i) mapping DNA sequence reads obtained from thousands of patients to a reference genome; (ii) identifying genetic changes present in each patient genome (known as "calling" variants); (iii) annotating these variants with respect to known gene transcripts or other genome landmarks; (iv) applying statistical analyses to identify genetic variants that are associated with differences in the patient phenotypes; and (v) quality control on each of the previous steps. Even though efficient tools exist to perform each of these steps, coordinating these processes in a scalable, robust and flexible pipeline is challenging because creating pipelines using general-purpose computer languages (e.g. Java, Python or Shell scripting) involves handling many low-level process synchronization and scheduling details. As a result, process coordination usually depends on

specific features of the underlying systems architecture, making pipelines difficult to migrate. For example, a processing pipeline designed for a "multi-core server" cannot directly be used on a cluster because running tasks on a cluster requires queuing them using cluster-specific commands (e.g. qsub). Therefore, if using such a language, programmers and researchers must spend significant efforts to deal with architecture-specific details that are not germane to the problem of interest, and pipelines have to be reprogrammed or adapted to run on other computer architectures. This is aggravated by the fact that the requirements change often and the software tools are constantly evolving.

In the context of bioinformatics, there are several frameworks to help implement data-processing pipelines; although a full comparison is beyond the scope of this article, we mention a few that relate to our work: (i) Snakemake (Koster and Rahmann, 2012) written as a Python domain-specific language (DSL), which has a strong influence from `make` command. Just as in `make`, the workflow is specified by rules, and dependencies are implied between one rules input files and another rules output files. (ii) Ruffus (Goodstadt, 2010), a Python library, uses a syntactic mechanism based on decorations. This approach tends to spread the pipeline structure throughout the code, making maintenance cumbersome [**?**]. (iii) Leaf [**?**], which is also written as a Python library, expresses pipelines as graphs drawn using ASCII characters. Although visually rich, the authors acknowledge that this representation is harder to maintain than the traditional code. (iv) Bpipe [**?**] is implemented as a DSL on top of Groovy, a Java Virtual Machine (JVM)-based language. Bpipe facilitates reordering, removing or adding pipeline stages, and thus, it is easy for running many variations of a pipeline. (v) NextFlow (www.nextflow.io), another Groovy-based DSL, is based on data flow programming paradigm. This paradigm simplifies parallelism and lets the programmer focus on the

coordination and synchronization of the processes by simply specifying their inputs and outputs.

Each of these systems creates either a framework or a DSL on a pre-existing general-purpose programming language. This has the obvious benefit of leveraging the languages power, expressiveness and speed, but it also means that the programmer may have to learn the new general-purpose programming language, which can be taxing and take time to master. Some of these pipeline tools use new syntactic structures or concepts (e.g. NextFlows data-flow programming model or Leafs pipeline drawings) that can be powerful, but require programming outside the traditional imperative model, and thus might create a steep learning curve.

In this article, we introduce a new pipeline programming language called BigDataScript (BDS), which is a scripting language designed for working with big data pipelines in system architectures of different sizes and capabilities. In contrast to existing frameworks, which extend general-purpose languages through libraries or DSLs, our approach helps to solve the typical challenges in pipeline programming by creating a simple yet powerful and flexible programming language. BDS tackles common problems in pipeline programming by transparently managing infrastructure and resources without requiring explicit code from the programmer, although allowing the programmer to remain in tight control of resources. It can be used to create robust pipelines by introducing mechanisms of lazy processing and absolute serialization, a concept similar to continuations (Reynolds, 1993) that helps to recover from several types of failures, thus improving robustness. BDS runs on any Unix-like environment (we currently provide Linux and OS.X pre-compiled binaries) and can be ported to other operating systems where a Java runtime and a GO compiler are available.

Unlike other efforts, BDS consists of a dedicated grammar with its own parser and interpreter, rather than being implemented on top of an existing language. Our language is similar to commonly used syntax and avoids inventing new syntactic structures or concepts. This results in a quick-to-learn, clean and minimalistic language. Furthermore, creating our own interpreter gives better control of pipeline execution and allows us to create features unavailable in general-purpose language (most notably, absolute serialization). This comes at the expense of expressiveness and speed. BDS is not as powerful as Java or Python, and our simple interpreter cannot be compared with sophisticated just-in-time execution or JVM-optimized byte-code execution provided by other languages. Nonetheless, in our experience, most bioinformatics pipelines rely on simple programmatic constructs. Furthermore, in typical pipelines, the vast majority of the running time is spent executing external programs, making the executing time of the pipeline code itself a negligible factor. For these reasons, we argue that BDS offers a good trade-off between simplicity and expressiveness or speed.

2.3 Methods

In our experience, using general-purpose programming languages to develop pipelines is notably slow owing to many architecture-specific details the programmer has to deal with. Using an architecture agnostic language means that the pipeline can be developed and debugged on a regular desktop or laptop using a small sample dataset and deployed to a cluster to process large datasets without any code changes. This significantly reduces the time and effort required for development cycles. As BDS is intended to solve or simplify the main challenges in implementing, testing and programming data processing pipelines without introducing a steep learning curve, our main design goals

are (i) simple programming language; (ii) abstraction from systems architecture; and (iii) robustness to hardware and software failure during computationally intensive data analysis tasks. In the next sections, we explore how these concepts are implemented in BDS.

2.3.1 Language overview

BDS is a scripting language whose syntax is similar to well-known imperative languages. BDS supports basic programming constructs (if/ else, for, while, etc.) and modularity constructs such as functions and `include` statements, which are complemented with architecture-independent mechanisms for basic pipeline runtime control (such as task, sys, wait and checkpoint). At runtime, the BDS backend engine translates these high-level commands into the appropriate architecture-dependent instructions. At the moment, BDS does not support object-oriented programming, which is indeed supported by other pipeline tools based on libraries/DSL extending general-purpose programming languages. The complete language specification and documentation is available online at http://pcingola.github.io/BigDataScript.

Unlike most scripting languages, BDS is strongly typed, allowing detection of common type conversion errors at the initial parsing stage (pseudo-compilation) rather than at runtime (which can happen after several hours of execution). As the syntax of strict typing languages tends to be more verbose owing to longer variable declaration statements, we provide a type inference mechanism (operator `:=`) that improves code readability. For example (Listing 1), the variables `in` and `out` are automatically assigned the types the first time they are used (in this case, the type is assigned to be string).

2.3.2 Abstraction from resources

One of the key features of BDS is that it provides abstraction from most architecture-specific details. In the same way that high-level programming

languages such as C or Java allow abstraction of the CPU type and other hardware features, BDS supports system-level abstraction, including the number and the type of computing-nodes or CPU-cores that are available to the pipeline and its component tasks, whether firing another process may saturate the servers memory or whether a process is executed immediately or queued.

Pipeline programming requires effective task management, particularly the ability to launch processes and wait for processes to finish execution before starting others. Task management can be performed using a single BDS statement, independently of whether this is running on a local computer or a cluster. Processes are executed using the task statement, which accepts an optional list of resources required by the task (for example, see Listing 1). The task consists of running a fictitious system command myProcess and diverting the output to `output.file`. BDS currently supports the following architectures: (i) local, single or multi-core computer; (ii) cluster, using GridEngine, Torque and Moab; (iii) server farm, using ssh access; and (iv) cloud, using EC2 and StarCluster. Depending on the type of architecture on which the script is run, the task will be executed by calling the appropriate queuing command (for a cluster) or by launching it directly (for a multi-core server).

Listing 2.1: `pipeline.bds` program. A simple pipeline example featuring and a maximum of 6 h of execution time (Line 5)

```
1 #!/usr/bin/env bds
2 in :=  input . file
3 out :=  output . file
4 task ( out <- in, cpus=2, timeout=6*HOUR ) {
5   sys myProcess $in > $out # Invoke command
6 }
```

BDS performs process monitoring or cluster queue monitoring to make sure all tasks end with a successful exit status and within required time limits. This is implemented using the `wait` command, which acts as a barrier to ensure that no statement is executed until all tasks finished successfully. Listing 2 shows a two-step pipeline with task dependencies using a `wait` statement (Line 13). If one or more of the `task` executions fail, BDS will wait until all remaining tasks finish and stop script execution at the `wait` statement. An implicit `wait` statement is added at the end of the main execution thread, which means that a BDS script does not finish execution until all tasks have finished running. It is common for pipelines to need multiple levels of parallel execution; this can be achieved using the `parallel` statement (or `par` for short). Wait statements accept a list of task IDs/parallel IDs in the current execution thread.

In addition to supporting explicitly defined task dependencies, BDS also automatically models implicit dependencies using a directed acyclic graph (DAG) that is inferred from information provided in the dependency operators (`<`) contained in `task` statements (see Listing 2, line 8). Finally, the `dep` expression defines a task whose conditions are not evaluated immediately (as it happens in `task` expressions) but only executed if required to satisfy a `goal`. Using `dep` and `goal` makes it easier to define pipelines in a `declarative` manner that is similar to other pipeline tools, as tasks are executed only if the output needs to be updated with respect to the inputs, independent of the intermediate results file, which might have been deleted.

2.3.3 Robustness

BDS provides two different mechanisms that help create robust pipelines: lazy processing and absolute serialization. When a processing pipeline fails, BDS automatically cleans up all stale output files to ensure that rerunning the pipeline will produce a correct output. If a BDS program is interrupted,

typically by pressing Ctrl-C on the console, all scheduled tasks and running jobs are terminated or deallocated from the cluster. In addition to immediately releasing computing resources, a clean stop means that users do not have to manually dequeue tasks, which allows them to focus on the problem at hand without having to worry about restoring a clean state.

Lazy processing.. Complex processing pipelines are bound to fail owing to unexpected reasons that range from data format problems to hardware failures. Rerunning a pipeline from scratch means wasting days on recalculating results that have already been processed. One common approach, when using general-purpose scripting languages, is to edit the script and comment out some steps to save processing time, which is inelegant and error prone. A better approach is to develop pipelines that incorporate the concept of lazy processing [**?**], a concept popularized by the `make` command (Feldman, 1979) used to compile programs, and which simply means the work is not done a task invoking a fictitious command `myProcess` defined to require 2 CPUs twice. This concept is at the core of many of the pipeline programming tools, such as SnakeMake, Ruffus, Leaf and Bpipe. By design, when lazy processing pipelines are rerun using the same dataset, they avoid unnecessary work. In the extreme case, if a lazy processing pipeline is run on an already successfully processed dataset, it should not perform any processing at all.

BDS facilitates the creation of lazy processing pipelines by means of the dependency operator (`<-`) and conditional task execution (see Listing 1, line 5 for an example). The task is defined as `task (out < in)`, meaning that it is executed only if `out` file needs to be updated with respect to `in` file: for example, if `output.file` file does not exist, has zero length, is an empty directory or has been modified before `input.file`.

Absolute serialization.. This refers to the ability to save and recover a snapshot of the current execution state, compiled program, variables, scopes and program counter, a concept similar to continuations (Reynolds, 1993). BDS can perform an absolute serialization of the current running state and environment, producing checkpoint files from which the program can be re-executed, either on the same computer or on any other computer, exactly from the point where execution terminated. Checkpoint files (or `checkpoints` for short) also allow all variables and the execution stack to be inspected for debugging purposes (`bds -i checkpoint.chp`). The most common use of checkpoints is when a task execution fails. On reaching a `wait` statement, if one or more tasks have failed, BDS creates a checkpoint, reports the reasons for task execution failure and terminates. Using the checkpoint, pipeline execution can be resumed from the point where it terminated (in this case, at the most recently executed `wait` statement) and can properly re-execute pending tasks (i.e. the tasks that previously failed execution).

Limitations.. BDS is designed to afford robustness to the most common types of pipeline execution failures. However, events such as full cluster failures, emergency shutdowns, head node hardware failures or network problems isolating a subset of nodes may result in BDS being unable to exit cleanly, leading to an inconsistent pipeline state. These problems can be mitigated by a special purpose `checkpoint` statement that, as the name suggests, allows the programmer to explicitly create checkpoints. Given that the overhead of creating checkpoints is minimal (a few milliseconds compared with hours of processing time for a typical pipeline), carefully crafted checkpoint statements within the pipeline code can be useful to prevent losing processed data, mitigate damage and minimize the overhead when rerun, which can be critical for long running pipelines.

2.3.4 Other features

Here we mention some selected features that are useful in pipeline programming. Extensive documentation is available at http://pcingola.github.io/BigDataScript.

Automatic logging.. Logging all actions performed in pipelines is important for three reasons: (i) it helps debugging; (ii) it improves repeatability; and (iii) it performs audits in cases where detailed documentation and logging are required by regulatory authorities (such as clinical trials).

Listing 2.2: `pipeline_2.bds` program. A two-step pipeline with task dependencies. The first step (line 9) requires to run `myProcess` command on a hundred input files, which can be executed in parallel. The second step (line 19) processes the output of those hundred files and creates a single output file (using fictitious `myProcessAll` command). It should be noted that we never explicitly state which hardware we are using: (i) if the pipeline is run on a dual-core computer, as each process requires 2 CPUs, one `myProcess` instance will be executed at the time until the 100 tasks are completed; (ii) if it is run on a 64-core server, then 32 `myProcess` instances will be executed in parallel; (iii) if it is run on a cluster, then 100 `myProcess` instances will be scheduled and the cluster resource management system will decide how to execute them; and (iv) if it is run on a single-core computer, execution will fail owing to lack of resources. Thus, the pipeline runs independent of the underlying architecture. The task defined in line 18 depends on all the outputs from tasks in line 8 (`mainOut < outs`).

```
1 #!/usr/bin/env bds
2 // Step 1: Parallel processing of input files
3 string[] outs // Define a list of strings
4 for( int i=0 ; i < 100 ; i++ ) {
5     in  :=  input_$i . file
```

```
    out := output_$i . file
    task ( out <- in, cpus=2, timeout=6*HOUR ) {
        sys myProcess $in > $out
    }
    outs.add( out )         // Add all output files here
}
wait // Optional: Wait for all tasks to finish

// Step 2: Process all outputs from previous step
mainOut := main . txt
mainIn := outs.join(    ) // Create a string with all names
    (space-separated)
task ( mainOut <- outs, mem=10*G ) {
    sys myProcessAll $mainIn > $mainOut
}
```

Creating log files is simple, but it adds boilerplate code and increases the complexity of the pipeline. BDS performs automatic logging in three different ways. First, it directs all process StdOut/StdErr output to the console. Second, as having a single output can be confusing when dealing with thousands of processes running in parallel, BDS automatically logs each processs outputs (StdOut and StdErr) and exit codes in separate clearly identified files. Third, BDS creates a report showing both an overview and details of pipeline execution (Fig. 2–3).

Automatic command line parsing.. Programming flexible data pipelines often involves parsing command-line inputsa relatively simple but tedious task.

BDS simplifies this task by automatically assigning values to variables specified through the command line. As an example, if the program in Listing 1 is called `pipeline.bds`, then invoking the program as `pipeline.bds -in another.file` will automatically replace the value of variable `in` with `another.file`.

Task re-execution.. Tasks can be re-executed automatically on failure. The number of retries can be configured globally (as a command-line argument) or by a task (using the `retry` variable). Only after failing `retry+1` times will a task will be considered to have failed.

2.3.5 BDS implementation

BDS is programmed using Java and GO programming languages. Java is used for high-level actions, such as performing lexical analysis, parsing, creating abstract syntax trees (AST), controlling AST execution, serializing processes, queuing tasks, etc. Low-level details, such as process execution control, are programmed in GO. As BDS is intended to be used by programmers, it does not rely on graphical interfaces and does not require installation of complex dependencies or Web servers.

Figure 2–2 shows the cascade of events triggered when a BDS program is invoked. First the script pipeline.bds (Fig. 2–2A) is compiled to an AST structure (Fig. 2–2B) using ANTLR (Parr, 2007). After creating the AST, a runnable-AST (RAST) is created. RAST nodes are objects representing statements, expressions and blocks from our BDS implementation. These nodes can execute BDS code, serializing their state, and recover from a serialized file, thus achieving absolute serialization. The script is run by first creating a scope and then properly traversing the RAST (Fig. 2–2C). We note that if needed, this approach could be tuned to perform efficiently, as demonstrated by modern languages, such as Dart.

When recovering from a checkpoint, the scopes and RAST are deserialized (i.e. reconstructed from the file) and then traversed in recovery mode, meaning that the nodes do not execute BDS code. When the node that was executed at the time of serialization event is reached, BDS switches to run mode and the execution continues. This achieves execution recovery from the exact state at serialization time. Checkpoints are the full state of a programs instance and are intended as a recovery mechanism from a failed execution. This includes failures owing to corrupted or missing files, as BDS will re-execute all failed tasks when recovering, thus correcting outputs from those tasks. However, checkpoints are not intended to recover from programming errors, where the user modifies the program to fix a bug, as a previously generated checkpoint is no longer valid respect to the new source code.

When a task statement is invoked, process requirements, such as memory, CPUs and timeouts, can optionally be specified. Depending on the architecture, BDS either checks that the underlying system has appropriate resources (CPUs and memory) to run the process (e.g. local computer or ssh-farm) or relies on the cluster management system to appropriately allocate the task. If all task requirements are met, a script file is created (Fig. 2–2D), and the task is executed by running an instance of bds-exec, a program that controls execution (Fig. 2–2E). This indirection is necessary for five reasons, which are described in detail below: (i) process identification, (ii) timeout enforcement, (iii) logging, (iv) exit status report and (v) signal handling.

Process identification. means that bds-exec reports its process ID (PID), so that BDS can kill all child processes if the BDS script execution is terminated for some reason (e.g. the Ctrl-C key is pressed at the console).

Timeout enforcement. has to be performed by bds-exec as many underlying systems do not have this capability (e.g. a process running on a

Figure 2–1: BDS report showing pipelines task execution timeline

server). When a timeout occurs, bds-exec sends a kill signal to all child processes and reports a timeout error exit status that propagates to the user terminal and log files.

Logging. a process means that bds-exec redirects stdout and stderr to separate log files. These files are also monitored by the main BDS process, which shows the output on the console. As there might be thousands of processes running at the same time and operating systems have hard limits on the number of simultaneous file descriptors available for each user, opening all log files is not an option. To overcome this limit, BDS polls log file sizes, only opening and reading the ones that change.

Exit status. has to be collected to make sure a process finished successfully. Unfortunately, there is no unified way to do this, and some cluster systems do not provide this information directly. By saving the exit status to a file, bds-exec achieves two goals: (i) unified exit status collection and (ii) exit status logging.

Signal handling. is also enforced by bds-exec making sure that a kill signal correctly propagated to all subprocesses, but not to parent processes. This is necessary because there is no limit on the number of indirect processes that a task can run, and Unix/Posix systems do not provide a unified way to obtain all nested child processes. To be able to keep track of all subprocesses, bds-exec creates a process group and spawns the subprocess in it. When receiving a signal from the operating system, bds-exec traps the signal and propagates a kill signal to the process group.

2.4 Results

To illustrate the use of BDS in a real-life scenario, we present an implementation of a sequencing data analysis pipeline. This example illustrates

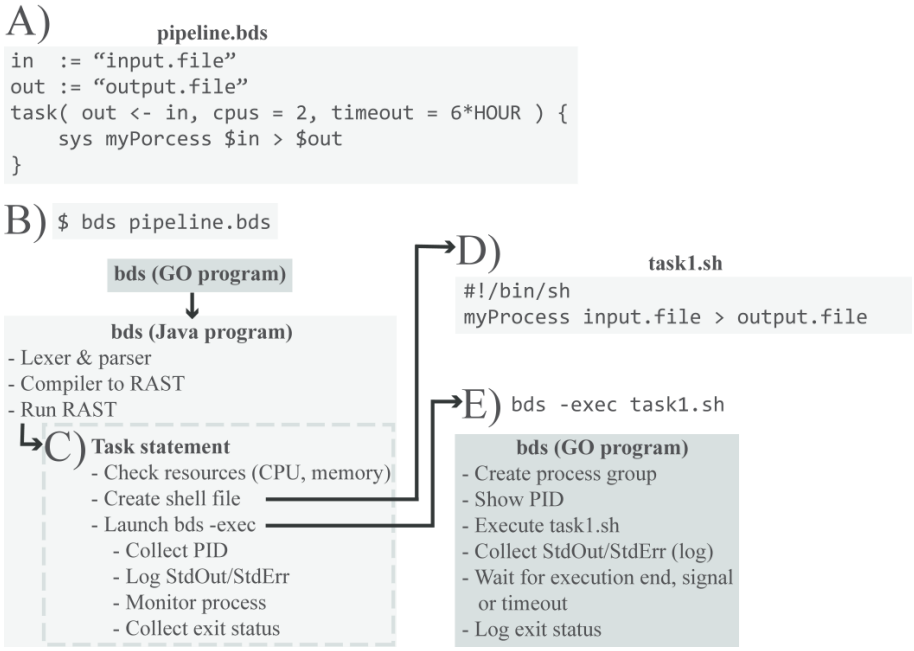

Figure 2–2: Execution example. (A) Script `pipeline.bds`. (B) The script is executed from a terminal. The GO executable invokes main BDS, written in JAVA, performs lexing, parsing, compilation to AST and runs AST. (C) When the task statement is run, appropriate checks are performed. (D) A shell script `task1.sh` is created, and a bds-exec process is fired. (E) bds-exec reports PID, executed the script `task1.sh` while capturing stdout and stderr as well as monitoring timeouts and OS signals. When a process finishes execution, the exit status is logged

three key BDS properties: architecture independence, robustness and scalability. The data we analyzed in this example consist of high-quality short-read sequences (200 coverage) of a human genome corresponding to a person of European ancestry from Utah (NA12877), downloaded from Illumina platinum genomes (http://www.illumina.com/platinumgenomes).

The example pipeline we created follows current best practices in sequencing data analysis [**?**], which involves the following steps: (i) map reads to a reference genome using BWA (Li and Durbin, 2009), (ii) call variants using GATKs HaplotypeCaller and (iii) annotate variants using SnpEff [**?**] and SnpSift [**?**]. The pipeline makes efficient use of computational resources by making sure tasks are parallelized whenever possible. Figure 2–3 shows a flowchart of our implementation, while the pipelines source code is available at `include/bio/seq` directory of our projects source code (https://github.com/pcingola/BigDataScript).

Architecture independence. . We ran the exact same BDS pipeline on (i) a laptop computer; (ii) a multi-core server (24 cores, 256 GB shared RAM); (iii) a server farm (5 servers, 2 cores each); (iv) a 1200-core cluster; and (v) the Amazon AWS Cloud computing infrastructure (Table 2–4). For the purpose of this example and to accommodate the fact that running the pipeline on a laptop using the entire dataset would be prohibitive, we limited our experiment to reads that map to chromosome 20. The architectures involved were based on different operating systems and spanned about three orders of magnitude in terms of the number of CPUs (from 4 to 1200) and RAM (from 8GB to 12TB). BDS can also create a cluster from a server farm by coordinating raw SSH connections to a set of computers. This minimalistic setup only requires that the computers have access to a shared disk, typically using NFS, which is a common practice in companies and university networks.

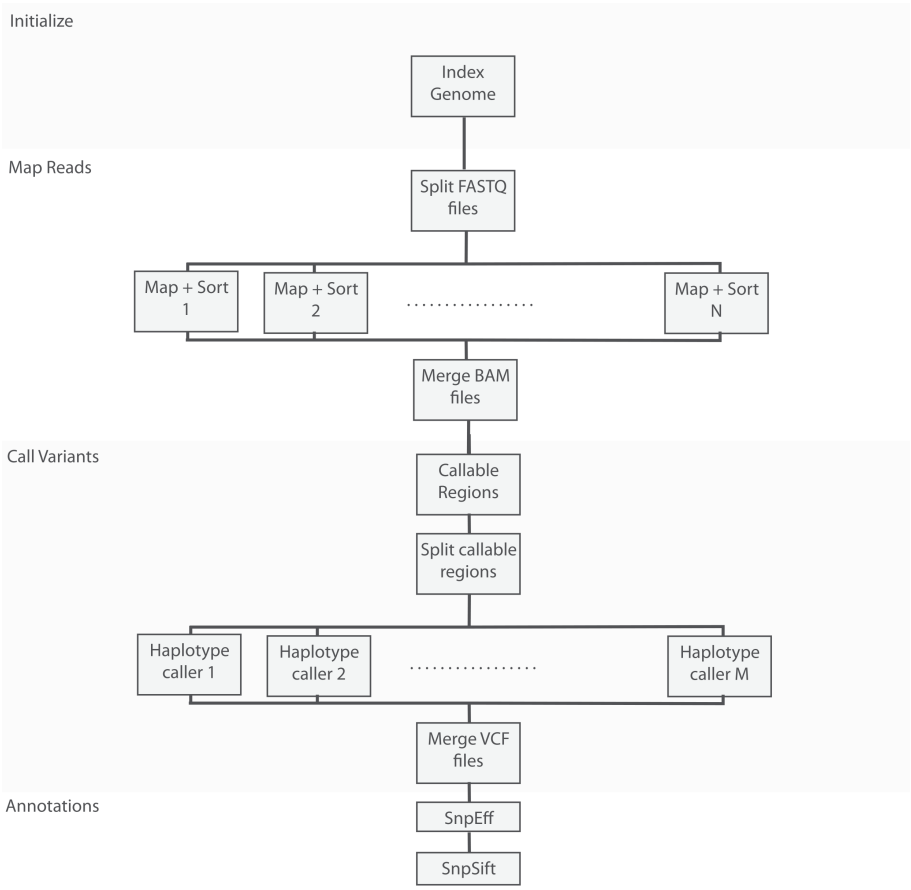

Figure 2–3: Whole-genome sequencing analysis pipelines flow chart, showing how computations are split across many nodes

System	CPUs	RAM	Notes
Laptop (OS.X)	4	8 GB	
Server (Linux)	24	256 GB	
Server farm (ssh)	16	8 Gb	Server farm using 8 nodes, 2 cores each.
Cluster (PBS Torque)	1200	12 TB	High load cluster (over 95%).
Cluster (MOAB) (Random failures)	1200	12 TB	High load cluster (over 95%). Hardware induced failures.
Cloud (AWS + SGE)	Inf.	Inf.	StarCluster, 8 m1.large instances.

Figure 2–4: Architecture independence example. Notes: Running the same BDS-based pipeline, a sequence variant calling and analysis pipeline, on the same dataset (chr20) but different architectures, operating systems and cluster management systems.

In all cases, the overhead required to run the BDS script itself accounted for 52 ms per task, which is negligible compared with typical pipeline runtimes of several hours.

Robustness.. To assess BDSs robustness, we ran the pipeline on a cluster where 10% of the nodes have induced hardware failures. As opposed to software failures, which are usually detected by cluster management systems, hardware node failures are typically more difficult to detect and recover from. In addition, we elevated the cluster load to 495% to make sure the pipeline was running on less than ideal conditions. As shown in Table 2–4, the pipeline finished successfully without any human intervention and required only 30% more time than in the ideal case scenario because BDS had to rerun several failed tasks. This shows how BDS pipelines can be robust and recover from multiple failures by using lazy processing and absolute serialization mechanisms.

Scalability.. To assess BDSs scalability, we ran exactly the same pipeline on two datasets that vary in size by several orders of magnitude (Table 2–5): (i) a relatively small dataset (chromosome 20 subset, 2GB) that would typically be used for development, testing and debugging and (ii) a high-depth whole-genome sequencing dataset (over 200 coverage, roughly 1.5 TB).

Dataset	Dataset size	System	CPUs	RAM
chr20	2 GB	Laptop (OS.X)	4	8 GB
Whole genome	1.5 TB	Cluster (MOAB)	22 000	80 TB

Figure 2–5: Scaling dataset sized by a factor of 1000. Notes: The same sample pipeline run on dataset of 2 GB (reads mapping to human chromosome 20) and 1.5 TB (whole-genome data set). Computational times vary according to systems resources, utilization factor and induced hardware failures.

2.5 Discussion

We introduced BDS, a programming language that simplifies implementing, testing and debugging complex data analysis pipelines. BDS is intended to be used by programmers in a similar way to shell scripts, by providing glue for several tools to ensure that they execute in a coordinated way. Shell scripting was popularized when most personal computers had a single CPU and clusters or clouds did not exist. One can thus see BDS as extending the hardware abstraction concept to data-center level while retaining the simplicity of shell scripting.

BDS tackles common problems in pipeline programming by abstracting task management details at the programming language level. Task management is handled by two statements (`task` and `wait`) that hide system architecture details, leading to cleaner and more compact code than general-purpose languages. BDS also provides two complementary robustness mechanisms: lazy processing and absolute serialization.

A key feature is that being architecture agnostic, BDS allows users to code, test and debug big data analysis pipelines on different systems than the ones intended for full-scale data processing. One can thus develop a pipeline on a laptop and then run exactly the same code on a large cluster. BDS also provides mechanisms that eliminate many boilerplate programming tasks, which in our experience significantly reduce pipeline development times. BDS

can also reduce CPU usage, by allowing the generation of code with fewer errors and by allowing more efficient recovery from both software and hardware failures. These benefits generally far outweigh the minimal overhead incurred in typical pipelines.

Appendix A

A.1 Algorithm details

Here we show the details on the algorithmic implementations...

References

[1] C.T. Abdallah. Mathematical controllability of genomic networks. *Proceedings of the National Academy of Sciences*, 108(42):17243–17244, 2011.

[2] N.J. Cowan, E.J. Chastain, D.A. Vilhena, J.S. Freudenberg, and C.T. Bergstrom. Nodal dynamics, not degree distributions, determine the structural controllability of complex networks. *Arxiv preprint arXiv:1106.2573*, 2011.

[3] A.P. Fejes, G. Robertson, M. Bilenky, R. Varhol, M. Bainbridge, and S.J.M. Jones. FindPeaks 3.1: a tool for identifying areas of enrichment from massively parallel short-read sequencing technology. *Bioinformatics*, 24(15):1729, 2008.

[4] M.J. Fullwood and Y. Ruan. ChIP-based methods for the identification of long-range chromatin interactions. *Journal of cellular biochemistry*, 107(1):30–39, 2009.

[5] D. Ghosh and Z.S. Qin. Statistical Issues in the Analysis of ChIP-Seq and RNA-Seq Data. *Genes*, 1(2):317–334, 2010.

[6] M. Hu, J. Yu, J.M.G. Taylor, A.M. Chinnaiyan, and Z.S. Qin. On the detection and refinement of transcription factor binding sites using ChIP-Seq data. *Nucleic acids research*, 38(7):2154, 2010.

[7] H. Ji, H. Jiang, W. Ma, D.S. Johnson, R.M. Myers, and W.H. Wong. An integrated software system for analyzing ChIP-chip and ChIP-seq data. *Nature Biotechnology*, 26(11):1293–1300, 2008.

[8] S. Jiao, C.P. Bailey, S. Zhang, and I. Ladunga. Probabilistic Peak Calling and Controlling False Discovery Rate Estimations in Transcription Factor Binding Site Mapping from ChIP-seq. *Methods in molecular biology (Clifton, NJ)*, 674:161–177, 2010.

[9] R. Jothi, S. Cuddapah, A. Barski, K. Cui, and K. Zhao. Genome-wide identification of in vivo protein-DNA binding sites from ChIP-Seq data. *Nucleic acids research*, 36(16):5221, 2008.

[10] B. Langmead, C. Trapnell, M. Pop, and S.L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol*, 10(3):R25, 2009.

[11] H. Li. Mathematical Notes on SAMtools Algorithms. *http://lh3lh3.users.sourceforge.net/download/samtools.pdf*, 2010.

[12] H. Li and R. Durbin. Fast and accurate short-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(5), 2009.

[13] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589, 2010.

[14] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078, 2009.

[15] F. Lin. Robust control design an optimal control approach. 2007.

[16] Y.Y. Liu, J.J. Slotine, and A.L. Barabási. Controllability of complex networks. *Nature*, 473(7346):167–173, 2011.

[17] P.J. Park. ChIP–seq: advantages and challenges of a maturing technology. *Nature Reviews Genetics*, 10(10):669–680, 2009.

[18] S. Pepke, B. Wold, and A. Mortazavi. Computation for ChIP-seq and RNA-seq studies. *Nature methods*, 6:S22–S32, 2009.

[19] W. Song, W. Jianmin, Z. Wei, P. Stanley, and C. Cheng. ChIP-PaM: an algorithm to identify protein-DNA interaction using ChIP-Seq data. *Theoretical Biology and Medical Modelling*, 7.

[20] A.M. Szalkowski and C.D. Schmid. Rapid innovation in ChIP-seq peak-calling algorithms is outdistancing benchmarking efforts. *Briefings in Bioinformatics*, 2010.

[21] C.M. Taniguchi, B. Emanuelli, and C.R. Kahn. Critical nodes in signalling pathways: insights into insulin action. *Nature Reviews Molecular Cell Biology*, 7(2):85–96, 2006.

[22] L. Teemu, R. Sunil, T. Soile, L. Riitta, and A. Tero. A practical comparison of methods for detecting transcription factor binding sites in ChIP-seq experiments. *BMC Genomics*, 10.

[23] Stanford University. Wnt home — stanford university, nussel lab.

[24] Wikipedia. Wnt signaling pathway — Wikipedia, the free encyclopedia. 2004.

[25] E.G. Wilbanks and M.T. Facciotti. Evaluation of algorithm performance in ChIP-seq peak detection. *PloS one*, 5(7):e11471, 2010.

[26] Y. Zhang, T. Liu, C.A. Meyer, J. Eeckhoute, D.S. Johnson, B.E. Bernstein, C. Nussbaum, R.M. Myers, M. Brown, W. Li, et al. Model-based analysis of ChIP-Seq (MACS). *Genome biology*, 9(9):R137, 2008.