Parallelization, Scalability, and Reproducibility in Next-Generation Sequencing Analysis

Dissertation
zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von
Johannes Köster

Dortmund
2014

Tag der mündlichen Prüfung: 9. März 2015

Dekan: Prof. Dr. Gernot A. Fink

1. Gutachter: Prof. Dr. Sven Rahmann
2. Gutachter: Prof. Dr. Axel Mosig

Abstract

The analysis of next-generation sequencing (NGS) data is a major topic in bioinformatics: short reads obtained from DNA, the molecule encoding the genome of living organisms, are processed to provide insight into biological or medical questions. This thesis provides novel solutions to major topics within the analysis of NGS data, focusing on parallelization, scalability and reproducibility.

The read mapping problem is to find the origin of the short reads within a given reference genome. We contribute the q-group index, a novel data structure for read mapping with particularly small memory footprint. The q-group index comes with massively parallel build and query algorithms targeted towards modern graphics processing units (GPUs). On top, the read mapping software PEANUT is presented, which outperforms state of the art read mappers in speed while maintaining their accuracy.

The variant calling problem is to infer (i.e., call) genetic variants of individuals compared to a reference genome using mapped reads. It is usually solved in a Bayesian way. Often, variant calling is followed by filtering variants of different biological samples against each other. With state of the art solutions, the filtering is decoupled from the calling, leading to difficulties in controlling the false discovery rate. In this work, we show how to integrate the filtering into the calling with an algebraic approach and provide an intuitive solution for controlling the false discovery rate along with solving other challenges of variant calling like scaling with a growing set of biological samples. For this, a hierarchical index data structure for storage of preprocessing results is presented and compression strategies are provided. The developed methods are implemented in the software ALPACA.

Depending on the research question, the analysis of NGS data entails many other steps, typically involving diverse tools, data transformations and aggregation of results. These steps can be orchestrated by workflow management. We present the general purpose workflow system Snakemake, which provides an easy to read domain-specific language for defining and documenting workflows, thereby ensuring reproducibility of analyses. The language is complemented by an execution environment that allows to scale a workflow to available resources, including parallelization across CPU cores or cluster nodes, restricting memory usage or the number of available coprocessors like GPUs. The benefits of using Snakemake are exemplified by combining the presented approaches for read mapping and variant calling to a complete, scalable and reproducible NGS analysis.

Acknowledgments

The final words I write for this thesis (although presented upfront) are dedicated to those who helped and supported me. Foremost, special thanks go to my advisor Prof. Sven Rahmann. As an employee, I could not imagine a better boss. As a PhD student, I am deeply thankful for enlightening discussions, advice and the many things I learned from him. Furthermore, I want to thank Prof. Axel Mosig for instantly agreeing to become the second examiner of this thesis, as well as Prof. Jens Teubner and Dr. Lars Hildebrand for completing the examination committee.

Thanks go to Dr. Eli Zamir for teaching me about cells and proteins and supporting the development of a feeling for biology. I thank Dr. Alexander Schramm and Prof. Johannes Schulte for almost four years of inspiring cooperation and providing biological motivation for algorithmic work.

Many thanks go to my current and former colleagues Marianna D'Addario, Dr. Daniela Beißer, Dr. Christina Czeschik, Corinna Ernst, Dominik Kopczynski, Prof. Tobias Marschall, Dr. Marcel Martin, Christopher Schröder, Henning Timm, Mareike Vogel and Dr. Inken Wohlers for sharing thoughts, providing feedback, having fun in the office and keeping me free of other duties in the decisive last months. Tobias and Marcel should be further thanked for teaching me essential basics during the beginning of my work, transforming me from a student into a PhD student.

I thank the many users of Snakemake for spreading the word about the software and providing feedback. Especially, I thank all early adopters and long-term supporters.

For proof reading and trying to understand alien topics, be it computer science, biology, or even both, special thanks go to Dr. Steven Engler, Stefan Gumprich, Dominik Kopczynski, Nicolas Potysch, Sven Strothoff and Martina Weiss. You provided extraordinarily helpful feedback.

I thank my parents and my parents-in-law for all their support and advice. Finally, I thank my wife Christine for her enormous support, love and patience in case of long work days, weeks of absence during conferences and delayed answers while sitting in front of my laptop or writing on the ugly whiteboard in our living room.

Johannes Köster
Essen, December 2014

Contents

1 Introduction

The genome of a living organism encodes its hereditary information. It serves as a blueprint for proteins, which form living cells, carry information and drive chemical reactions. Differences between populations, species, cancer cells and healthy tissue, as well as syndromes or diseases can be reflected and sometimes caused by changes in the genome. This makes the genome an major target of biological and medical research. Today, it is often analyzed with next-generation sequencing, producing gigabytes of data from a single biological sample. Analyzing this data entails creating complex workflows with tens of steps, applying various tools and converting between diverse representations of information. Two steps common to many such analyses are *read mapping* and *variant calling*. This thesis presents novel algorithms and data structures for read mapping and variant calling, and a workflow system that supports the analysis of next-generation sequencing data in a reproducible way.

For read mapping, we present the q-group index, a novel index data structure with a particularly small memory footprint, complemented by parallel algorithms for querying and building the index, targeting graphics processing units (GPUs). On top, we provide the novel read mapper PEANUT. By effectively exploiting parallelization on GPUs, it outperforms other read mappers while maintaining their accuracy.

With the variant caller ALPACA, we present an algebraic approach to variant calling, that is the first to allow intuitive control of the false discovery rate in complex filtering scenarios. ALPACA is designed to parallelize well on both central processing units (CPUs) and GPUs. We explore the use of hierarchical index data structures and their compression, ensuring scalability with the number of considered biological samples.

Finally, we present the workflow system Snakemake. While being designed with next-generation sequencing analysis in mind, it provides a general purpose, text-based, easy to read domain-specific language for workflow definition. Snakemake workflows entail implicit parallelization and scale from single-core workstations and multi-core servers to compute clusters without changing the workflow definition. The scheduling of Snakemake can be made aware of arbitrary resources, e.g., restricting the number of available GPU devices. Support for documentation and data provenance further enhance the reproducibility of Snakemake workflows.

The chapters 2 to 4 contain the three major topics of the thesis. Each chapter has its own introduction, providing the foundations needed for that chapter and a separate discussion summarizing the findings and outlining future work. Chapter 2 describes the read mapping problem and presents the q-group index data structure in combination with

the read mapper PEANUT as a massively parallel, efficient solution exploiting modern graphics cards. Chapter 3 describes the variant calling problem and shows how to solve it with the variant caller ALPACA in an algebraic approach, while keeping parallelization and scalability in mind. Chapter 4 introduces the workflow system Snakemake, defining the entailed domain-specific language and the scalable scheduling. The chapter ends with an example workflow which models a complete next-generation sequencing analysis combining the approaches from Chapter 2 and 3. The thesis is closed with the concluding Chapter 5. In the appendix, co-authored articles and the software published with this thesis are summarized. In the following, we introduce biological and algorithmic foundations that are particularly important in the context of this work.

1.1 The genome

We briefly introduce the necessary biological foundations based on the description of Alberts, Johnson, and Lewis (2008). The genome is encoded by deoxyribonucleic acid (DNA) molecules. DNA is a sequence of smaller units, the nucleotides. A nulceotide consists of a sugar (here deoxyribose), a phosphate group and an organic base. Within DNA, nucleotides with the bases adenin (A), cytosin (C), guanin (G) and thymine (T) occur. The nucleotides are connected via covalent bonds between the sugar of one and the phosphate of the next nucleotide. Hence, a DNA molecule has a direction, starting with a free sugar and ending with a free phosphate. The two ends are called 5' and 3'. When encoding the genome, DNA molecules occur double stranded, forming a helical structure, called the double helix. In this form, the two DNA strands are connected to each other via hydrogen bonds between their bases: A can bind to T, and C can bind to G. We say that A is complementary to T, and C is complementary to G. One strand is the reverse complement of the other, i.e., the 5' (first) base of the first strand is complementary to the 3' (last) base of the second strand, and so on. This maximizes the number of bonds between the two strands. From computer science perspective, a DNA strand can be seen as a string over the alphabet $\{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$. We interchangeably refer to A,C,G and T as *bases*, *nucleotides* or *basepairs* (referring specifically to their paired form in the double strand).

The storage of genomic DNA within a cell allows to divide species into prokaryotes (e.g., bacteria) and eukaryotes (e.g., mammals). The former consist of a single cell and their hereditary information is carried by a circular double stranded DNA molecule that may exist in multiple copies. In contrast, the DNA of eukaryotes is contained in a membrane surrounded structure, the nucleus, and occurs as *chromosomes*. A chromosome is a single, coiled, double stranded DNA molecule. Eukaryotic cells can have multiple copies of each chromosome. The number of copies is called *ploidy*. *Diploid* organisms like humans have in general two copies of each chromosome. An exception are the sex chromosomes X and Y: female individuals have two X chromosomes and no Y chromosome, males have an X and a Y chromosome. One copy of each chromosome is inherited from the mother, one from the father.

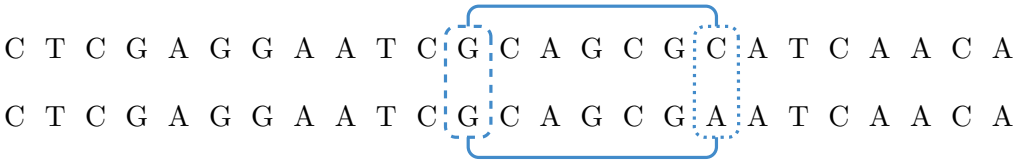

Figure 1.1: Example DNA sequences within two copies of the same chromosome of a diploid genome. We omit the sequence of the reverse complementary strands. The dashed box depicts a homozygous locus, the dotted box shows a heterozygous locus. The former has genotype GG, the latter exhibits the genotype CA. Combinations of alleles on the same chromosome depicted by solid lines are two haplotypes that can be observed here (GC and GA).

Each chromosome hosts a set of *genes*. Genes are segments in the DNA that encode instructions for creating a product. Usually, this product is a *protein*. Proteins are chains of amino acids, that execute various functions within the cell, build organelles and signalling channels, and form complexes. They are synthesized from genes in two steps. First, a gene is *transcribed* into an intermediate copy of itself in the form of ribonucleic acid (RNA) via the RNA polymerase enzyme. RNA is similar to DNA, having a ribose instead of deoxyribose sugar and a uracil (U) nucleotide instead of thymine. Second, the RNA molecule, called messenger RNA (mRNA) is *translated* into a protein by the ribosome, which itself is a complex of proteins. Each three consecutive nucleotides (called codon) in the mRNA encode for one amino acid. Genes are partitioned into *exons* and *introns*. Exons are the coding regions of a gene: during transcription, intronic regions are spliced out, i.e., they do not encode the amino acids that end up in the protein. The set of all exons of a genome is called *exome*.

The terms explained in the following are illustrated in Figure 1.1. Depending on the ploidy, each gene can be present in one or more copies of a chromosome. The different copies of a gene are called *alleles*. In this thesis, following DePristo et al. (2011), we also apply the term allele to individual positions on a chromosome, independently of the genes. In other words, alleles can also be the different bases occurring at the same position in all copies of a chromosome. With *locus*, we refer to a particular position on all copies of a chromosome. In a diploid organism, except on the sex chromosomes, each locus hosts two alleles. In an individual, the combination of all alleles of the same gene or locus is called *genotype*. In contrast, the combination of alleles of different loci on the same copy of a chromosome is called *haplotype*. The genomes of different individuals of the same species are, while being similar, not entirely the same. *Mutations* can cause nucleotide-level or even larger changes from one to the next generation (see Chapter 3). Even within a single individual, a locus may exhibit different alleles. We call such loci *heterozygous*. Loci which host the same allele on all chromosomes are called *homozygous*.

Eukaryotic genomes can be huge: e.g., the human genome has about 3.2 billion bases distributed to 24 different chromosomes and 25 thousand genes (Alberts, Johnson, and

Lewis 2008). Hence, the sequencing of genomes and the analysis of the obtained data is particularly challenging.

1.2 Next-generation sequencing

Sequencing of DNA is the determination of the nucleotide sequence of a DNA molecule. For long, the primary method of DNA sequencing was Sanger sequencing, proposed by Sanger, Nicklen, and Coulson (1977). In a huge and costly effort, the Sanger method allowed to initially sequence the human genome (Lander et al. 2001). From 2005 to 2008, a new class of sequencing methods emerged, which are commonly referred to as second- or *next-generation sequencing* (NGS). Shendure and Ji (2008) identify three steps common to all NGS approaches. First, the DNA subject to sequencing is fragmented randomly, and artificial adapter DNA molecules are attached. The resulting sequences are amplified (i.e., duplicated several times) with a polymerase chain reaction (PCR; see Mullis, Ferre, and Gibbs (1994)) and localized in clusters of duplicate single stranded fragments on some carrier material. Finally, the sequencing of the fragments is performed by cycles of biochemical treatment and imaging based data acquisition. The PCR amplification is necessary to make the signal detectable by the imaging. Together, n cycles yield the sequence of the first n nucleotides of all fragment clusters in parallel. We call these sequences *reads*. Read lengths range from 32 to a few hundred, with 100 to 200 being a common size in current studies.

Today, the most popular (Lam et al. 2012) implementation of next-generation sequencing is sold by Illumina[1]. With this technology, also called SBS (sequencing by synthesis), each cycle works as follows. First, a solution of free nucleotides is flooded over the fragment clusters. A DNA polymerase now synthesizes at each originally single stranded fragment another nucleotide of the reverse complementary strand (synthesis). The nucleotides are equipped with a fluorescent label, with different colors for A, C, G and T. The label prevents the synthesis of more than one nucleotide per cycle. At the end of each cycle, remaining free nucleotides are washed away (washing), the fluorescent color of each cluster is measured, providing information about the nucleotide attached in that cycle, and the nucleotide labels are removed (label removal), allowing the synthesis of another nucleotide in the next cycle. Figure 1.2 shows an example.

Compared to Sanger sequencing, NGS is less accurate: the observed error rates with Sanger sequencing range from 0.01% to 0.001% (Kircher and Kelso 2010). With Illumina sequencing, error rates of 1% are observed (Ross et al. 2013). In return, NGS yields more reads (hundreds of millions) at low costs.

An important variant of the sequencing protocol is *paired-end sequencing*. Here, fragments are sequenced from both ends. This results in two reads per fragment cluster, which are known to come from the same fragment. Usually, the size of the fragments

[1] `http://www.illumina.com`, visited 11/2014

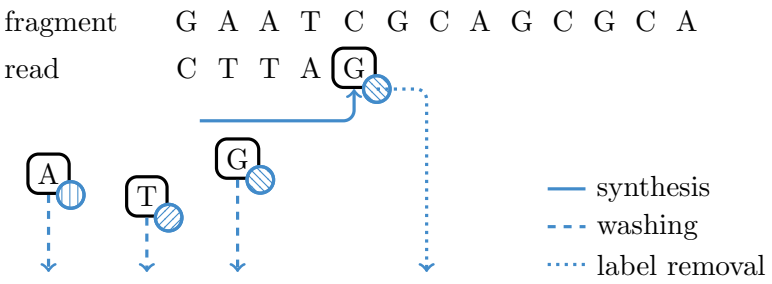

Figure 1.2: Next-generation sequencing cycle with Illumina SBS. Free labelled nucleotides are used to synthesize the next read base. Then, remaining free nucleotides are washed away and the synthesized read base is determined by its label. Finally, labels are removed.

(called *insert size*) is controlled, such that a certain distance between the read pairs can be expected. Paired-end sequencing has several advantages: e.g., it helps to find the true origin of ambiguously mappable reads (see Chapter 2). It can also be used to detect structural variants (Marschall et al. 2012) or help to infer transcript expressions (Trapnell et al. 2010).

With automated sequencing technologies being in place since about 30 years, the genomes of many species are well known by now. These *reference genomes* are representative mixtures, assembled from several individuals. Various applications of NGS have been developed that make use of an already known reference genome. Here, the read mapping problem occurs: for the reads obtained from the biological sample, the origin in the genome is unknown. The *read mapping* problem is to determine this origin by finding the most likely position of each read within the reference genome (see Chapter 2). When sequencing DNA, comparisons of the differences between reads and the reference genome can be used to detect genomic variants (see Chapter 3). Further, e.g., RNA can be sequenced to quantify the expression of genes and transcripts (RNA-seq; Wang, Gerstein, and Snyder 2009), and binding sites of transcription factors can be determined by combining NGS with antibody-based selection of fragments (ChIP-seq; Park 2009). The sequencing of DNA can also be targeted towards exonic regions (exome sequencing). With only about 1.5% of the 3.2 billion bases of the human genome being exonic (Alberts, Johnson, and Lewis 2008), the capacity of the sequencer can be used to generate a much higher read depth at the expense of loosing intronic and intergenic regions, without increasing or even lowering sequencing costs.

Reads are typically provided in the FASTQ format (Cock et al. 2010). The format provides read bases as a string over the IUPAC alphabet[2] which encodes in addition to $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$ classes of uncertain bases with additional letters, e.g., an N represents that the base is entirely unknown, a W represents uncertainty between A and T. Typically, only Ns occur in addition to Σ. Along with each base, a base quality is provided.

[2] `http://www.chem.qmul.ac.uk/iubmb/misc/naseq.html`, visited 11/2014

The quality is the probability that the base was miscalled by the sequencer, given in the so-called PHRED scale: Let p be a probability, then the PHRED scaled probability is obtained as

$$q := -10 \log_{10} p.$$

With FASTQ, the PHRED-scaled base qualities are stored in single bytes, allowing to encode miscall probabilities from 1.0 to $10^{-9.3}$.

In the context of this thesis, the term *sample* will, if not stated otherwise, usually describe a sample from some biological tissue that has been sequenced by some NGS technology to obtain reads.

1.3 Designing for efficient GPU usage

Chapter 2 and 3 present parallel algorithms targeted towards GPUs. This requires architecture specific considerations, which we elaborate here based on the work of Köster and Rahmann (2014). We first describe the GPU architecture and its implications. Then, we define the *parallel random access machine* (PRAM) that serves as an abstraction for assessing the time complexity of parallel algorithms. Finally, we introduce the prefix scan programming pattern that is used widely in this work.

We use the terminology of NVIDIA[3], while the general concepts are also applicable to the hardware of competitors like AMD[4]. A GPU is partitioned into *Streaming Multiprocessors* (SMs), each of which has its own on-chip memory, cache and processing cores. By adjusting the *thread block size* it can be controlled how threads are distributed among the SMs. One thread block is executed on one SM and stays resident until all threads in the block are completed. Once a thread block is finished, another will be scheduled to the SM if any blocks are left. An SM can execute 32 threads in parallel (restricting the thread block size to be a multiple of 32); such a group of threads is called a *warp* or *wavefront*. At any time, each of these threads has to execute the same instruction in the code, but may do so on different data; this concept is called *single instruction, multiple threads* (SIMT). Hence, conditionals with diverging branches should be avoided, since threads taking an if-branch have to wait for threads taking the corresponding else-branch to finish and vice versa. All SMs may access a slow common global memory (often less than 3 GB) in addition to their fast on-chip cache and memory. While the size of the fast cache is extremely limited, accessing global memory is slow and should be minimized. The memory latency can be reduced by *coalescing* the access, i.e., letting threads in a warp access contiguous memory addresses, such that the same memory transaction can serve many threads. In addition, an SM can execute a different warp while waiting on a transaction to finish, thereby hiding the latency. For the latter, threads should minimize their register usage such that the number of warps

[3] `http://www.nvidia.com`, visited 11/2014
[4] `http://www.amd.com`, visited 11/2014

that can reside on an SM is maximized. Finally, data transfers from the main system memory to the GPU's global memory are comparatively slow. Hence, it is advisable to minimize them as well.

To implement parallel algorithms for both GPU and CPU, we use OpenCL[5]. With OpenCL, parallel algorithms are implemented as kernels in OpenCL-C, a dialect of the C language. For given input data, typically in the form of one or more arrays, a kernel is executed with potentially many threads. Each thread executes the kernel code on one or more data points of the input data: e.g., a kernel executing with n threads can replace a loop over n items of an array. The kernel code would be equal to the body of the loop here. OpenCL provides an API for accessing various compute devices (e.g., the CPU or the GPU), managing memory and launching kernels. As all provided software is implemented in the Python programming language[6], we use the Python package PyOpenCL (Klöckner et al. 2012) to access OpenCL from within Python.

1.3.1 Parallel random access machines

A parallel random access machine (PRAM) is a theoretical model for analyzing parallel algorithms, defined as a collection of synchronous processors that have random access to a common shared memory (Dehne and Yogaratnam 2010; Reif 1993). The behavior of different processors accessing the same memory unit classifies PRAMs into

1. the EREW (exclusive-read, exclusive-write) PRAM, which allows only one processor at a time to read from and write to a memory unit,
2. the CREW (concurrent-read, exclusive-write) PRAM, which allows multiple processors at a time to read from a memory unit and only one to write,
3. the CRCW (concurrent-read, concurrent-write) PRAM, which allows multiple processors at a time to read from and write to a memory unit.

The PRAM abstracts from caching and communication between threads. In principle, a GPU could be seen as an CRCW PRAM, since it has multiple processors that can concurrently write to and read from a common memory. We choose to assess time complexity of the algorithms presented in this work with the PRAM model. Some of the presented algorithms require exclusive access to a memory unit (e.g., an entry within an array) for certain operations like counting, which we implement with OpenCL atomic operations[7]. Hence, we conservatively decide to assess complexity under the CREW PRAM model.

In practice, there are differences between the PRAM and a GPU (Dehne and Yogaratnam 2010). First, the processors of a GPU are grouped into SMs, the processors of which

[5] `https://www.khronos.org/opencl`, visited 11/2014
[6] `http://www.python.org`, visited 11/2014
[7] `https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/atomicFunctions.html`, visited 11/2014

can only work in parallel if they execute the same instruction. Second, memory accesses that do not coalesce between the processors can impact the performance. Hence, we complement the PRAM-based theoretical complexity results with discussions of GPU specific optimizations.

1.3.2 Prefix scans

A useful programming pattern that is used extensively in the algorithms presented in this work are *parallel prefix scans* (see Cormen et al. 2001; Blelloch 1990). A special case of prefix scans is the computation of a cumulative sum, which at first appears to be a serial process. Parallel prefix scans are used to solve these problems in a data parallel way with a minimum amount of branching, thus nicely fitting above considerations for GPU programming. In the following, we first define the prefix scan operation and then discuss its parallel implementation on a PRAM as presented by Blelloch (1990).

Definition 1.1 (Prefix scan)**.** *Let $A = (a_1, a_2, \ldots, a_n)$ be a sequence of n elements and $\oplus$ be an associative operator. Then, the prefix scan on A with operation $\oplus$ is given as*

$$(a_1, a_1 \oplus a_2, \ldots, a_1 \oplus a_2 \oplus \cdots \oplus a_n).$$

In the following, we refer to this operation as the *scan* operation. We further call a scan on $A' = (e, a_1, a_2, \ldots, a_{n-1})$ with e being the neutral element of $\oplus$, the *prescan* on A. While scan and prescan appear to be inherently sequential, they can be efficiently parallelized on a PRAM with ρ processors. In the following, we outline a parallel implementation of the prescan (Blelloch 1990). The scan can be obtained from this by removing the leading e and appending the $\oplus$-sum of a_n and the last element of the prescan.

The idea is to explore the levels of a binary tree with height $\lceil \log_2 n \rceil$ over the sequence. The nodes of the binary tree shall represent intermediate results of our computation. Initially, the leaves of the tree are determined by the values of the sequence. We calculate the values of each level iteratively, overwriting previous results in place, i.e., we map the value of the i-th node (from left to right) with height h at element a_k with $k = 2^h i$. On each level, the values of each node are calculated in parallel from those of the previous iteration. To obtain a prescan, we perform three steps (see Figure 1.3):

1. Explore the binary tree bottom-up, and set the value of each vertex to the $\oplus$-sum of its children.

2. Set the root to the neutral element.

3. Explore the binary tree top-down in preorder. For each node with value v set (a) the value of the right child to the $\oplus$-sum of the value of the left child and v and set (b) the value of the left child to v.

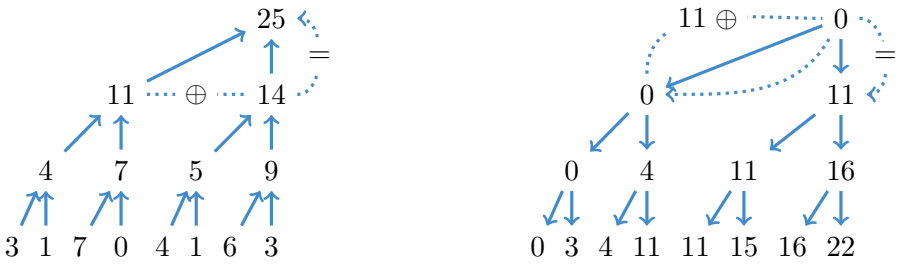

Figure 1.3: Example for the parallel implementation of the prescan operation with $\oplus$ being the addition and neutral element 0. The algorithm first calculates the left tree bottom up and then the right tree top down. The node values of each level are calculated in parallel.

After step one, element $k = 2^h i$ of the sequence contains the result of $a_1 \oplus \cdots \oplus a_k$. In step three each node shall be set to the sum of all leaves preceding it according to the preorder traversal. Since the root is preceded by no leaf, we set the root node to 0 in step two. After step two, the sum of the leaves preceding the right child is always equal to the sum of the left child after step one and the value of the parent. Hence, substep (a) of the third step sets the right child to that value. The left child has the same preceding leaves as the parent, therefore the value of the parent can be copied in substep (b). A detailed proof for this concept is presented by Blelloch (1990).

Usually, there will be more values in the sequence than processors on the PRAM, i.e., $n > \rho$. Then, the sequence can be divided into partitions of size $\lceil \frac{n}{\rho} \rceil$. For each partition, the $\oplus$-sum and the prescan can be calculated sequentially by one of the PRAM processors and used as leaves of the tree. The prescan of the $\oplus$-sums can then be used as an offset for a prescan over each partition. This leads to a time complexity of

$$\mathcal{O}\left(\frac{n}{\rho} + \log_2 \rho\right) \tag{1.1}$$

composed of the time for calculating the partition sums and calculating the $h = \log_2 \rho$ levels of the tree in parallel on an EREW PRAM (Blelloch 1990).

An example application of a prefix scan is to calculate a predicated copy of a sequence: from a sequence $A = (a_1, a_2, \ldots, a_n)$ and a predicate $f : A \to \{0, 1\}$ assigning a boolean value to each element of A we want to generate the sequence A' of exactly those elements $a_k \in A$ where $f(a_k) = 1$. To achieve this, we calculate the prescan over the sequence $(f(a_1), f(a_2), \ldots, f(a_n))$. This generates a sequence B where the k-th element denotes the target address of element a_k with $f(a_k) = 1$ in the predicated copy A': e.g., consider a sequence $A = (1, 7, 3, 4, 5)$ with predicate $f(a_k) = \mathbf{1}_{a_k > 3}$ being the indicator function selecting all elements greater than 3. A prescan is calculated over the sequence

$(0, 1, 0, 1, 1)$, resulting in the sequence $B = (0, 0, 1, 1, 2)$. By copying elements a_2, a_4 and a_5 to the corresponding addresses specified in B we obtain $A' = (7, 4, 5)$.

PyOpenCL (see Section 1.3) provides a flexible mechanism of defining prefix scans using templates. Further, it implements various primitives via prefix scans, e.g., cumulative sums and predicated copies.

2 A massively parallel read mapper

Next-generation sequencing produces millions of small reads that represent the DNA sequence of a biological sample (or specific parts thereof; see Section 1.2). The read lengths range, depending on the technology and biological question, from tens to several hundreds of bases. Today, read lengths of 100 – 200 are common. Typically, a genomic region is covered by more than one read but the information about the originating position of each read in the genome is lost during the sequencing process. For many species, a reference genome, representing a consensus among several individuals is already known (see Section 1.1). The *read mapping problem* is to find the origin of each read in the reference genome. Here, we present a novel data structure that helps to solve the read mapping problem efficiently on GPUs. On top of this, we introduce a new read mapping algorithm.

2.1 Introduction

In the following, we denote the set of strings over an alphabet Σ as Σ^*. Depending on the context, we call a string also word or text. We denote with $s[i]$ the i-th character (or letter) of a string $s \in \Sigma^*$. Further, $s[i, i+k]$ denotes the substring of length k beginning at position i. The term $|s|$ refers to the size of string s. The empty string is denoted as ε and $\circ$ is the string concatenation. We denote the set of k-combinations over a set M as M^k.

The reference genome usually consists of multiple sequences (i.e., chromosomes). We denote these as the *reference sequences*. We interpret the reference sequences and any read as strings over the alphabet $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$. Since a read can come from either strand of the chromosome (see Section 1.1) we also have to consider the reverse complement of each reference sequence.

An exact occurrence of the read in a reference sequence or its reverse complement is the most likely origin of a read. Therefore, at first sight, the read mapping problem can be approached by finding all occurrences of a pattern P in a text T, namely applying classical *pattern matching* (see Cormen et al. 2001), with P being a read, and T being a reference sequence or its reverse complement. Pattern matching searches for all positions i in a text T such that P is a substring of T beginning at position i.

In practice, many reads will not match exactly to their origin because of two reasons. First, the sequenced sample will have local differences to the reference due to the natural

genetic variation (see Section 1.1). Second, the sequencing itself can introduce technical errors and artifacts. Within reads, these differences manifest as *substitutions*, *insertions* and *deletions* of bases compared to the reference sequence. Therefore, the matching has to be approximate (or error tolerant), leading to alignments (see Gusfield 1997) of the reads against the reference sequences:

Definition 2.1 (Alignment)**.** *Let $s, t \in \Sigma^*$ be two strings over an alphabet Σ. A global alignment A of s and t is a string over the alphabet $\Sigma' = (\Sigma \cup \{-\})^2 \setminus \{(-,-)\}$ with $\pi_1(A) = s$ and $\pi_2(A) = t$. Here, π_1 and π_2 are homomorphisms with $\pi_1((a,b)) := a$, $\pi_1((-,b)) = \varepsilon$, $\pi_2((a,b)) := b$, $\pi_2((a,-)) = \varepsilon$ and $\pi_i(B \circ C) = \pi_i(B) \circ \pi_i(C)$ for $a, b \in \Sigma$, $B, C \in \Sigma'^*$ and $i \in \{1, 2\}$.*

We call the global alignment of substrings of s and t a local alignment, and the global alignment of s against a substring of t a semi-global alignment.

When printing the pairs of the alignment alphabet as columns, one (but not necessarily the best) global alignment between the strings `EAT` and `PEANUT` is:

```
-EA--T
PEANUT
```

The quality of an alignment can be assessed using a distance measure d that typically quantifies the amount of edit operations needed to turn one string into the other. In the following, we will sometimes call edit operations errors and use the term error rate as a synonym for a distance measure. For the read mapping problem, it suffices to consider substitutions, insertions and deletions as edit operations. The *edit* or *Levenshtein distance* (Levenshtein 1966) weights each operation equally, and can be written as

$$d(sa, tb) = \min \begin{cases} d(s,t) + \mathbf{1}_{a \neq b}, \\ d(s, tb) + 1, \\ d(sa, t) + 1 \end{cases} \tag{2.1}$$

with $d(s, \epsilon) = |s|$, $d(\epsilon, t) = |t|$, $s, t \in \Sigma^*$ and $a, b \in \Sigma$ (Navarro and Raffinot 2008). Here, $\mathbf{1}_{a \neq b}$ is the indicator function evaluating to 1 if $a \neq b$ and 0 otherwise. Further, sa and tb are the strings s and t extended by the letters a and b, respectively. In the recurrence, the first case handles substitution and match, followed by deletion from and insertion into s. Alternatively, the quality can be calculated via an *alignment score*. Alignment scores often weight matching characters with 1, penalize substitutions with -1 and use affine costs for insertions and deletions (also called *affine gap costs*), i.e., opening a gap has a higher cost than extending it. Edit distances or alignment scores and (optionally) the concrete alignment can be calculated by variants of the Smith-Waterman algorithm (Smith and Waterman 1981), which makes use of the score (or edit) matrix E. For strings s and t, the matrix contains one row for each letter $a \in s$ and one column for each letter $b \in t$. The value E_{ij} is the maximal score (or minimal distance) between a prefix of s of length i and a prefix of t of length j. Interpreted as a graph with a node for each matrix entry and directed edges from (i, j) to $(i + 1, j + 1), (i, j + 1)$ and

$(i+1, j)$, a path in E represents a local alignment between s and t. A path from the top to the bottom row represents a semi-global alignment whereas a path from the top left to the bottom right represents a global alignment. The variant that finds the latter path is also called Needleman-Wunsch algorithm (Needleman and Wunsch 1970). The Smith-Waterman Algorithm explores the score matrix with dynamic programming in time complexity $\mathcal{O}(|s| \cdot |t|)$ to find the best alignment score. Backtracking can be used to reconstruct a concrete alignment for the best score.

Now, the read mapping problem can be solved by finding the best semi-global alignment (according to some distance measure) of a read against the reference sequences and their reverse complements. Sometimes, the reads are expected to exhibit larger differences to the reference than local substitutions or small insertions or deletions. For example, they may contain technical adapters at one end (see Section 1.2), or a read may span a structural variant (e.g., a fusion of two chromosomes) that occurs in the sequenced sample. When this can be expected, it is advisable to look for local alignments instead of semi-global alignments, allowing to omit larger parts of a read without penalizing them in the distance measure.

The sizes of genomes (approximately 3.2 billion basepairs for human, 2.7 billion basepairs for mice; see Section 1.1) and the hundreds of millions of reads produced by a single modern next-generation sequencing experiment, render the application of the Smith-Waterman algorithm for each read against the complete reference prohibitive. Hence, various filtering methods and approximations have been developed. These can be roughly classified into methods based on backward search using the Burrows-Wheeler Transform (BWT; see Section 2.2) and methods based on q-gram indexes (see Section 2.3). Examples for the former are BWA (Li and Durbin 2009) and Bowtie 2 (Langmead and Salzberg 2012), examples for the latter are RazerS 3 (Weese, Holtgrewe, and Reinert 2012) and MrFast (Alkan et al. 2009).

Among the best alignments of a read, it is sometimes not obvious which one represents the true origin of the read. In the following, the candidate origins of a read reported by a read mapper are called *hits*. Weese, Holtgrewe, and Reinert (2012) categorize read mapping implementations into best-mappers that try to find the (or any) best hit of a read (e.g., BWA-MEM) and all-mappers that provide a comprehensive enumeration of all possible locations (e.g., RazerS 3 or MrFast) up to a given error threshold. While all-mappers can be much slower (depending on the number of hits), their strategy is beneficial whenever suboptimal hits are of relevance. For example this is the case when the originating genome of the reads is unknown (e.g., when sequencing a mixture of samples). Roberts and Pachter (2013) mention the mapping to alternative transcripts (see Section 1.1) and Alkan et al. (2009) motivate all-mapping with the detection of copy number variations (i.e. duplications of parts of chromosomes). An intermediate strategy is to report all hits of the best stratum, i.e., all hits with the same lowest error level (instead of only the first or a random such hit).

Recently, exploiting the parallelization capabilities of GPUs for read mapping has become popular and GPU-based BWT read mappers appeared, e.g., SOAP3 (Liu et al.

2012), SOAP3-dp (Luo et al. 2013) and CUSHAW2-GPU (Liu and Schmidt 2014). Using a q-gram index on a GPU is not a common choice because of its large size. Therefore, to the best of our knowledge, q-gram based mappers so far only use the GPU for calculating the alignments and keep the index on the CPU, e.g., NextGenMap (Sedlazeck, Rescheneder, and von Haeseler 2013) and Saruman (Blom et al. 2011).

Here, by introducing the *q-group index*, a new approach to solving the read mapping problem on the GPU is presented. The q-group index is a variant of the traditional q-gram index, with a smaller memory footprint. We show how the q-group index can be efficiently built and queried on GPUs using parallel algorithms. The q-group index is used in a *filtration and validation approach*: Exact matches of a given length q between each read and each reference sequence are detected quickly and alignments are computed only where such matches are found. The resulting read mapper is called PEANUT (ParallEl AligNment UTility) and available as open source software under the MIT license (see Section A.1).

This chapter is based on previously published work (Köster and Rahmann 2014). First, related work is summarized (Section 2.2). Then, the traditional q-gram index is introduced (Section 2.3). Next, the novel q-group index (Section 2.4) and the read mapping algorithm of PEANUT (Section 2.5) are defined. The chapter ends with evaluations of the performance and accuracy, comparing PEANUT with other read mappers (Section 2.6) and a discussion (Section 2.7).

2.2 Related work

We describe BWA and RazerS 3 as representatives of BWT and q-gram index based approaches. Additionally, we describe the two GPU based read mappers CUSHAW2 and NextGenMap.

BWA A popular read mapper is BWA, the Burrows-Wheeler-Aligner (Li and Durbin 2009), which simulates an error tolerant search in a suffix array using the BWT. For a text T over an alphabet Σ ended by a sentinel \$ that is lexicographically smaller than all other letters in the text, the suffix array S is a permutation of the text positions such that $S[i]$ is the start position of the i-th suffix according to lexicographical order. The BWT B of the text T is a permutation of T such that the i-th position contains the character before the i-th suffix in the suffix array, i.e., $B[i] = \$$ when $S[i] = 0$ and $B[i] = T[S[i] - 1]$ otherwise. For each sequence read, BWA searches for intervals in the suffix array, which represent all occurrences of the read in the text. This happens using the following observation of Ferragina and Manzini (2000). For a letter $a \in \Sigma$, let $C(a)$ denote the number of letters lexicographically smaller than a in the text without the sentinel ($T[0, |T| - 1]$) and $O(a, i)$ denote the number of occurrences of letter a in the prefix of the BWT $B[0, i]$. If a string W is a substring of T, then the lower bound of the suffix array interval containing all occurrences of aW is $l(aW) = C(a)+O(a, l(W)-1)+1$

and the upper bound is $u(aW) = C(a)+O(a, u(W)-1)$. The BWT together with C and O represented as tables is also called the *FM-Index*. Since O can become huge for large genomes, it is usually sampled into entries at regular intervals and the intermediate values are calculated on the fly using the BWT. By performing a *backward search* from the end of the string W, updating the interval iteratively with the FM-Index, all exact occurrences of W in the text T can be found. To obtain error tolerance, BWA explores substitutions, insertions and deletions in W in a breadth-first way. A precomputed lower bound for the number of differences in the remaining portion of W is maintained and used for canceling the search early. The time complexity for finding an exact match of a single string (i.e., read) W is $\mathcal{O}(|W|)$, independently of the text size. The time complexity of the error tolerant version is exponential in the number of allowed errors. Therefore, backward search is typically divided into two phases, the *seed* and the *extend* phase. The error rate is limited to a small value during the seed phase (e.g., the first 32 letters) and relaxed in the extend phase. Newer versions of BWA provide an additional mode that finds *maximal exact matches* (MEMs, see below) as seeds over a variant of the FM-Index and extends these using the Smith-Waterman algorithm.

CUSHAW2 Similar to BWA, CUSHAW2 (Liu and Schmidt 2012) is based on BWT and FM-Index. Here, the FM-Index is used to find MEMs between the read and the reference sequence. The string W (i.e., the read) is evaluated from left to right. At position i in the string, an exact backward search (see above) is performed on the FM-index and stopped at the first mismatch. The result is the suffix array interval for the MEM between the prefix $W[0, i]$ and the text. All MEMs larger than a predetermined threshold are extended to local alignments with the Smith-Waterman algorithm. The GPU variant of CUSHAW2 (Liu and Schmidt 2014) performs both the MEM search and the extension to local alignments on the GPU.

RazerS 3 RazerS 3 (Weese, Holtgrewe, and Reinert 2012) uses the filtration and validation approach outlined above. It builds a q-gram index (see Section 2.3) over the sequence reads and searches for exact q-gram matches between the reference sequences and the index. These matches are projected onto a potential starting position of the read in the reference. Each potential starting position is validated using a bit-parallel algorithm (Myers 1999) to determine the edit distance for the semi-global alignment between the read and the reference at that position. If the distance is small enough, the alignment is calculated and reported. With a configurable sensitivity, RazerS 3 reports all alignments of a read up to a given distance. The PEANUT algorithm presented in this work borrows two ideas of RazerS 3, namely to build the index data structure over the reads instead of the reference (see Section 2.5.1) and to use a bit-parallel algorithm for the validation of potential hits (see Section 2.5.2).

NextGenMap In a preprocessing step, NextGenMap (Sedlazeck, Rescheneder, and von Haeseler 2013) builds a hash table over the q-grams of the reference sequences, storing

their occurrence positions. To map a read, the occurrences of its q-grams in the reference are obtained from the hash table. For regions with sufficiently many matching q-grams the best possible alignment score is calculated. For the region with the best alignment score, the alignment is computed and reported. The latter step uses the alignment library MASon (Rescheneder, von Haeseler, and Sedlazeck 2012) and can optionally be performed on the GPU.

2.3 Q-gram index

A q-gram is a string of length q over the DNA alphabet $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$. A classical DNA q-gram index of a text T stores for each q-gram at which positions in T it occurs and allows to retrieve each position in constant time. It is commonly implemented via two arrays that we call the *address table* A and the *position table* P.

Q-grams are encoded as machine words of appropriate size with two subsequent bits encoding one genomic letter (i.e., A = 00, C = 01, G = 10, T = 11). Unknown nucleotides (usually encoded as letter N) are converted randomly to A, C, G or T, and larger subsequences of Ns are omitted from the index. Hence, a q-gram needs $2q$ bits in hardware and is represented (encoded) as a number $g \in \{0, \ldots, 4^q - 1\}$. The address table provides for each (encoded) q-gram g a starting index $A[g]$ that points into the position table such that $P[A[g]], P[A[g]+1], \ldots, P[A[g+1]-1]$ are the occurrence positions of g.

Deciding about the q-gram length q entails a trade-off between specificity of the q-grams and the size of the data structure. Array A needs $2^{2q} = 4^q$ integers and thus grows exponentially with q, while array P needs $|T|$ integers, independently of q. Larger values of q lead to fewer hits per q-gram that need to be validated or rejected in later stages. Further, the choice of q determines the sensitivity or error-tolerance. Following directly from the pigeonhole principle, we observe (Jokinen and Ukkonen 1991):

Lemma 2.2 (Q-gram lemma)**.** *Let $s, t \in \Sigma^*$ with $|s| \geq |t|$ be two strings with edit distance e according to their optimal global alignment. Then, at least $|t| + 1 - (e+1)q$ of the $|t| - q + 1$ q-grams in t occur in s.*

In other words, each edit operation leads to q q-grams less to be shared between s and t.

2.4 Q-group index

The idea of the *q-group index* is to have the same functionality as the q-gram index (i.e., to retrieve all positions where a given q-gram occurs in constant time per position), but with a smaller memory footprint for large q. This is achieved by introducing additional

layers in the data structure. In the following, we always consider a q-gram as its numeric representation $g \in \{0, \ldots, 4^q - 1\}$.

We divide all 4^q q-grams into groups of size w, where w is the GPU word size (typically $w = 32$). Q-gram g is assigned to group number $\lfloor g/w \rfloor$. Thus, the i-th group is the set $G_i = \{g \mid \lfloor g/w \rfloor = i\}$ of w consecutive q-grams according to their numeric order. The set of all q-groups is $\mathcal{G}_q := \{G_0, G_1, \ldots, G_{\lceil \frac{4^q}{w} \rceil - 1}\}$. We write g_{ij} for the j-th q-gram in G_i.

For a given q and text T, the q-group index is a tuple of arrays

$$\mathcal{I}_{T,q} := (I, S, S', O). \tag{2.2}$$

Array I consists of $|\mathcal{G}_q|$ words with w bits each (overall $w\frac{4^q}{w} = 4^q$ bits). Bit j of $I[i]$ indicates whether g_{ij} occurs at all as a substring in the text, i.e.,

$$I[i]_j = \begin{cases} 1 & \text{if } g_{ij} \text{ is a substring of } T, \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

The array O corresponds to the position table P of a regular q-gram index: it is the concatenation of all occurrence positions of each q-gram in sorted numeric q-gram order. To find where the positions of a particular q-gram g begin in O, we first determine the group index i and j such that $g = g_{ij}$. With the bit pattern of $I[i]$, we determine whether q-gram g_{ij} occurs in the text T. If not, there is nothing else to do. If yes, i.e., $I[i]_j = 1$, we determine j' such that bit j is the j'-th one-bit in $I[i]$. We call j' the *group-rank* of the q-gram. The group-rank can be also seen as the number of smaller q-grams of the q-group that occur in the text.

The address array S contains, for each q-group i, an index into another address array S', such that $S'[S[i] + j']$ is the starting index in O where the positions of g_{ij} can be found. This implies that $S[i]$ is defined as the number of one bits in all previous entries of S, i.e.,

$$S[i] = \sum_{i'=0}^{i-1} \textsc{Popcount}(I[i'])$$

with the population count $\textsc{Popcount}(x)$ returning the number of one-bits in x. All occurrence positions are now listed as

$$O[S'[S[i] + j']], O[S'[S[i] + j'] + 1], \ldots, O[S'[S[i] + j' + 1] - 1].$$

See Figure 2.1 for an illustration. Similar to a plain q-gram index, access is in constant time per position.

Theorem 2.3. *For a text T, let $\mathcal{I}_{T,q} := (I, S, S', O)$ be a q-group index and g be an arbitrary q-gram with n occurrences in the text. With $0 \leq k < n$, accessing the k-th occurrence of g has complexity $\mathcal{O}(1)$.*

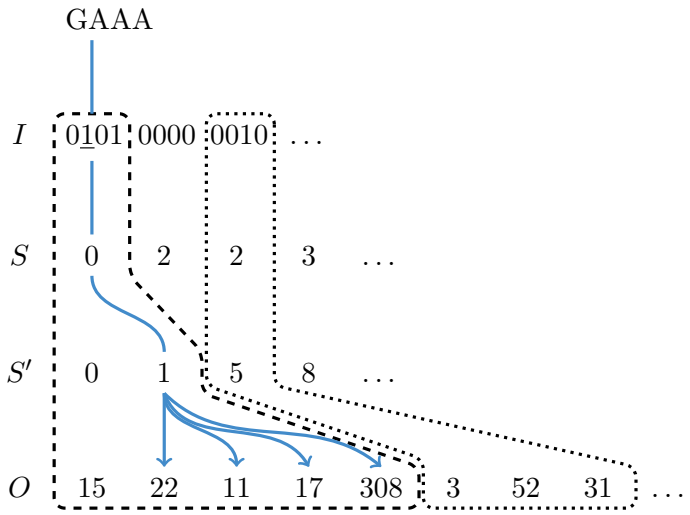

Figure 2.1: The q-group index consists of four arrays I, S, S', O. The dashed and dotted areas show the search spaces in the data structure for q-grams assigned to the first and the third q-group, respectively. The arrows illustrate how the four layers of the index are traversed to reach the occurrences of the queried q-gram GAAA.

Proof. To determine (i, j) such that $g = g_{ij}$, we simply compute $i = \lfloor g/w \rfloor$ and $j = g - wi = g \bmod w$ in constant time. To compute the group-rank j', i.e., find how many one-bits occur up to bit j in $I[i]$, we use the population count instruction with a bit mask

$$j' = \textsc{Popcount}(I[i] \,\&\, (2^j - 1)).$$

Given that w is set to the GPU word size (see above), population counts are available as a hardware instruction. Hence, this needs constant time as well. The k-th occurrence (starting from zero) of g can then be calculated as

$$O[S'[S[i] + j'] + k]$$

with complexity $\mathcal{O}(1)$. □

Comparison with rank data structures For a sequence of bits, the *rank* of the k-th bit is the number of 1-bits up to position k. A rank data structure provides the rank for the k-th bit in constant time. Jacobson (1988) calls it *succinct*, if it needs $n + o(n)$ bits for a bit sequence of length n. The classical succinct rank data structure uses the following strategy (González et al. 2005). The bit sequence is partitioned into equally sized superblocks which are themselves divided into blocks. For each superblock, the rank of the first bit is stored in a table. For each block, the rank of the first bit within the superblock is stored. Finally, the rank of each bit in a block can be computed using

the two tables and either the population count operation or an additional table. Since the tables have to store ranks for fewer bits, properly chosen block sizes render their space requirements sublinear in the number of bits in the sequence. When interpreting I as a sequence of bits, I and S implement a rank data structure, with $S[i] + j' + 1$ being the rank of the g-th bit in I for q-gram g with $i = \lfloor g/w \rfloor$ and $j = g \bmod w$. While the implementation is not succinct, a single query needs less table lookups. This is beneficial on the GPU since accessing memory is expensive (see Section 1.3).

2.4.1 Size

We note that both I and S consist of $\lceil 4^q/w \rceil$ words, S' contains an index for each occurring q-gram and hence of up to $\min\{4^q, |T|\}$ words, and O is a permutation of text positions consisting of $|T|$ words. The size of the q-group index follows directly as the sum of its components:

Theorem 2.4. *Let T be a text and $\mathcal{I}_{T,q} := (I, S, S', O)$ be the corresponding q-group index. Then, $\mathcal{I}_{T,q}$ needs up to $2\left\lceil \frac{4^q}{w} \right\rceil + \min\{4^q, |T|\} + |T|$ words.*

We evaluate how above worst case size of the q-group index behaves compared to the size of the q-gram index (i.e., $4^q + |T|$; see Section 2.3). Both depend on the size of the underlying text and the value of q, manifesting in the number of possible q-grams 4^q. Therefore, we consider the ratio K between the possible q-grams and the text size, i.e., $4^q = K|T|$. We further assume a word size of $w = 32$ bits.

If $4^q \lessapprox |T|$, the conventional q-gram index has a small advantage because each q-gram can be expected to occur (even multiple times). With $K \leq 1$, the size ratio between q-group index and q-gram index is

$$\begin{aligned}\frac{\frac{2}{32}4^q + 4^q + |T|}{4^q + |T|} &= \frac{\frac{K}{16}|T| + K|T| + |T|}{K|T| + |T|} \\ &= \frac{\frac{K}{16} + K + 1}{K + 1} \\ &= 1 + \frac{K}{16(1+K)}.\end{aligned}$$

For $K = 1$ (or $|T| = 4^q$), this means a small size disadvantage of 3% for the q-group index.

If q becomes larger for fixed text size (such that q-grams become sparse), the q-group index saves memory, up to a factor of 16. The size ratio is

$$\begin{aligned}\frac{\frac{2}{32}4^q + |T| + |T|}{4^q + |T|} &= \frac{\frac{K}{16}|T| + |T| + |T|}{K|T| + |T|} \\ &= \frac{\frac{K}{16} + 2}{K + 1}\end{aligned}$$

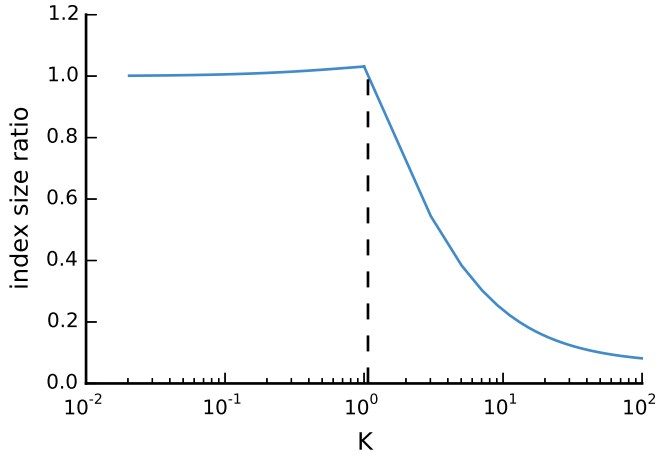

Figure 2.2: Ratio between the worst case size of the q-group index and the size of the q-gram index for different $K = 4^q/|T|$. The dashed line marks the break-even point, i.e., the K beyond which the q-group index guarantees to be smaller than the classical q-gram index.

for $K > 1$ and tends to $\frac{1}{16}$ for large K. The break-even point is reached for $K = \frac{16}{15}$. Figure 2.2 shows the behavior of the index size ratio depending on K.

In practice, we use $q = 16$ because of the following reasons. First, a bit-encoded 16-gram exactly fills a 32-bit word and hence avoids wasting bits. Second, todays usual reads contain 100 bases or more and the tendency is to produce rather larger than shorter reads. In this scenario $q = 16$ offers reasonable error tolerance and high specificity. With current GPU memory size, we use $|T| = 10^8$, i.e., we process 100 million nucleotides at a time. In the worst case, this results in a q-group index size of approximately 1.8 GB. The ratio between q-grams and text size is $K = 42.95$, and the q-group index needs in the worst case $\frac{42.95/16+2}{42.95+1} \approx 10\%$ of the memory of the conventional q-gram index.

2.4.2 Construction

Algorithm 1 shows how the index is built. The outline of the algorithm is as follows. First, I is created from the q-grams of the text (line 2). Then, S is calculated as the cumulative sum over the population counts of I (line 6). Next, the number of occurrences for each q-gram is calculated (line 10) and S' is created as the cumulative sum over these counts (line 14). Finally, the q-gram positions are written into the appropriate intervals of O (line 16).

Each step is implemented on the GPU with parallel OpenCL kernels (see Section 1.3). The cumulative sums are implemented with parallelized prefix scan operations (see Section 1.3.2). Importantly, the algorithm needs hardly any branching (hence maximizing concurrency) and makes use of coalescence along the reads to minimize memory latency.

Algorithm 1 Building the q-group index. For a q-gram g, the function Group-And-Bit(g) computes (i, j) with group $i := \lfloor g/w \rfloor$ and bit $j := g \bmod w$. The function Group-Rank(I, i, j) computes $j' := \text{Popcount}(I[i] \,\&\, (2^j - 1))$, as explained in the text.

Input: a text T, machine word size w, q-gram size q
Output: the q-group index (I, S, S', O)

```
1:  initialize I with ⌈4^q/w⌉ zeros
2:  for p ← 0, ..., |T| − q in parallel
3:      (i, j) ← Group-And-Bit(q-gram at position p in T)
4:      I[i]_j ← 1
5:  allocate S with space for |I| + 1 integers
6:  for i ← 0, ..., |I| − 1 in parallel
7:      S[i + 1] ← Popcount(I[i])
8:  S ← cumulative sum of S
9:  initialize S' of length S[|I|] + 1 with zeros
10: for p ← 0, ..., |T| − q in parallel
11:     (i, j) ← Group-And-Bit(q-gram at position p in T)
12:     j' ← Group-Rank(I, i, j)
13:     increment S'[S[i] + j' + 1] by 1
14: S' ← cumulative sum of S'
15: Allocate O of length |T|
16: for p ← 0, ..., |T| − q in parallel
17:     (i, j) ← Group-And-Bit(q-gram at position p in T)
18:     j' ← Group-Rank(I, i, j)
19:     k ← next free entry in O after S'[S[i] + j']
20:     O[k] ← p
```

All major data structures are kept in GPU memory. Therefore, between the steps, only constant amounts of data (e.g., a single integer defining the size of an array; see line 9) have to be transferred between the GPU and the host.

Theorem 2.5. *For a text T and a machine word size w, Algorithm 1 calculates the q-group index $\mathcal{I}_{T,q} := (I, S, S', O)$ with time complexity*

$$\mathcal{O}\left(\frac{1}{\rho}\left(\frac{4^q}{w} + \min\{4^q, |T|\} + |T|\right) + \log \rho\right)$$

on a CREW PRAM (Section 1.3.1) with ρ processors. If $|T| < 4^q$, complexity is

$$\mathcal{O}\left(\frac{1}{\rho}\left(\frac{4^q}{w} + |T|\right) + \log \rho\right).$$

Proof. We first prove the time complexity for the general case. Each initialization of an array A with zeros needs $|A|$ operations. Hence, line 1 and 9 need $|I| = 4^q/w$ and

$|S'| = \min\{4^q, |T|\}$ operations. The operations in all kernels (lines 2, 6, 10 and 16) need constant time (see Theorem 2.3). Therefore, the kernels need $\mathcal{O}(|T|)$ and $\mathcal{O}(|I|) = \mathcal{O}(4^q/w)$ operations. Considering a CREW PRAM with ρ processors and excluding the cumulative sums, we obtain a time complexity $\mathcal{O}\left((4^q/w + \min\{4^q, |T|\} + |T|)/\rho\right)$. A cumulative sum on an array A has time complexity $\mathcal{O}(|A|/\rho + \log\rho)$ on an EREW PRAM (see Section 1.3.2) and therefore also on a CREW PRAM. Since the cumulative sums (line 8 and 14) are calculated on S and S', the total time complexity is

$$\mathcal{O}\left(\frac{1}{\rho}\left(\frac{4^q}{w} + \min\{4^q, |T|\} + |T|\right) + \log\rho\right).$$

If $|T| < 4^q$, the minimum can be eliminated, resulting in

$$\mathcal{O}\left(\frac{1}{\rho}\left(\frac{4^q}{w} + |T|\right) + \log\rho\right).$$

For correctness, we first observe that array I is build correctly by the definition of Group-And-Bit. Any q-gram g not contained in the text T does not occur in the other data structures. It remains to show by induction that $O[S'[S[i] + j'] + k]$ contains the k-th occurrence[1] of an arbitrary q-gram g_{ij} contained in the text T with group-rank j' after executing Algorithm 1.

We first consider the smallest q-gram $g^{(0)} = g_{ij}$ contained in the text T as basis. Since $g^{(0)}$ is the smallest occurring q-gram, $S[i+1]$ is the first nonzero entry in S set in line 7. Therefore, the cumulative sum of S in line 8 yields $S[i] = 0$. Further, the rank of $g^{(0)}$ is $j' = 0$, such that line 13 increments $S'[1]$ by 1. The cumulative sum in line 14 yields $S'[0] = 0$. The last step of the algorithm fills consecutive entries of O beginning with $S'[0] = 0$ with the occurrence position p. Hence, the k-th occurrence of $g^{(0)}$ is located at $O[S'[S[i] + j'] + k] = O[k]$.

Now, we assume that the algorithm is correct for the n-th smallest q-gram $g^{(n)} = g_{i_n j_n}$ occurring m times in the text with group-rank j'_n. By induction, we know that $O[S'[S[i_n] + j'_n] + m - 1]$ contains the last occurrence of q-gram $g^{(n)}$. Let $g^{(n+1)} = g_{ij}$ be the $(n+1)$-th smallest q-gram occurring in the text with group-rank j'. It suffices to show that $S'[S[i] + j'] = S'[S[i_n] + j'_n] + m$, which follows directly from the observation that the cumulative sum over S' may only increase by m between $S'[S[i_n] + j'_n]$ and $S'[S[i] + j']$, because there is no occurring q-gram between $g^{(n+1)}$ and $g^{(n)}$. □

In practice, the memory requirements for the q-group index can be reduced further without changing access (Theorem 2.3) and construction (Theorem 2.5) time complexities. By storing every even position of S, the q-group index needs only up to $\lceil \frac{4^q}{w} \rceil + \lceil \frac{4^q}{2w} \rceil + \min\{4^q, |T|\} + |T|$ words. The values of the odd positions i of S can be obtained as $S[i-1] + \textsc{Popcount}(I[i-1])$ in constant time.

[1] Note that, in practice, the synchronization between threads when writing of O does not guarantee that the k-th occurrence in the index is also the k-th occurrence of the q-gram in the text. The sequence of occurences of a q-gram in O is rather a permutation thereof.

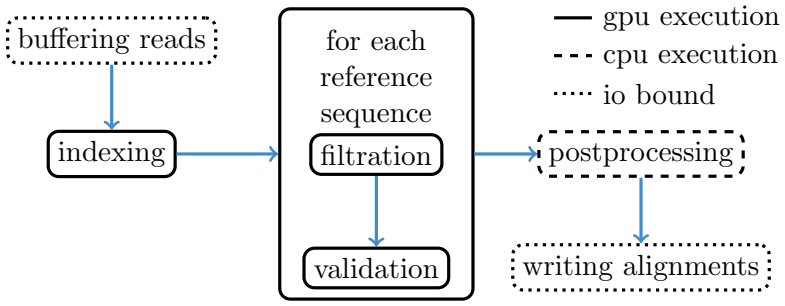

Figure 2.3: The PEANUT algorithm. Read sequences are buffered and a q-group index is created from them on the fly. Filtration (detection of q-gram hits) and validation are performed on the GPU until all reference sequences (e.g., chromosomes) are processed. The hits are postprocessed and streamed out in SAM format. All steps (here shown as boxes) operate independently in parallel and communicate via queues. Arrows between the steps represent a data transfer via a queue.

2.5 Algorithm

On top of the q-group index, we define the PEANUT algorithm for read mapping. The algorithm consists of three main steps:

1. filtration,
2. validation,
3. postprocessing.

The first two steps, filtration and validation, are handled on the GPU, while the postprocessing is computed on the CPU. The steps are conducted on a stream of reads. Reads are collected until buffers of configurable size are saturated. Then, any computation is done in parallel for all buffered reads (see Figure 2.3).

In the filtration step, potential hits between the reference sequence and the reads are detected using the q-group index. Next, the potential hits are validated using a variant of Myers' bit-parallel alignment algorithm (Myers 1999). The validated hits undergo a postprocessing that annotates them with a mapping quality and calculates the actual alignment. The postprocessed hits are streamed out in SAM format (Li et al. 2009). Because of memory constraints on the GPU, all steps are performed per reference sequence (e.g., chromosome) instead of using the reference as a whole (see Section 2.5.1). Paired-end reads (see Section 1.2) are handled independently during filtration and validation. During postprocessing, read-pair information is used to sort hits into strata of decreasing confidence and obtain mapping qualities (see Section 2.5.3). In the following, each step is described in detail.

2.5.1 Filtration

For a given subset of reads loaded into the buffer (see above), the filtration step aims to yield a set of potential hits, i.e., candidate origins of each read. For this, we seek for matching q-grams between each reference sequence and the reads using the q-group index (Section 2.4).

Following Weese, Holtgrewe, and Reinert (2012), we decide to build the q-group index over the concatenation of the buffered reads. At first sight, it would be more reasonable to build the index over the reference. However, building the q-group index over the whole reference (i.e., the concatenation of all reference sequences) would exceed the memory of most GPUs, since most q-grams would occur and the size advantage of the q-group index would vanish (see Section 2.4.1). Alternatively the index could be build over parts of the reference (as, e.g., naturally given by the reference sequences). Since not all indexes for all reference sequences could stay in memory, either building them online or loading them from a preprocessed storage would be necessary. Hence, either $|R|$ large data transfers or $|R|$ index builts would have to be repeated for each set of buffered reads and $|R|$ reference sequences. Since the q-group index is per definition larger than the underlying text, it is better to keep the reference sequences in GPU memory and build the index once over the set of buffered reads.

Given the q-group index (I, S, S', O), we assume that there is a function $\textsc{Indexpair}(g)$ that returns, for a q-gram g, an index pair $(k_{\text{start}}, k_{\text{end}})$ such that the occurrence positions in the indexed text are all $O[k]$ with $k_{\text{start}} \leq k < k_{\text{end}}$. The function $\textsc{Indexpair}(g)$ is implemented as follows. Let $(i, j) := \textsc{Group-And-Bit}(g)$ and the group-rank $j' := \textsc{Group-Rank}(I, i, j)$. Then $k_{\text{start}} = S'[S[i] + j']$ and $k_{\text{end}} = S'[S[i] + j' + 1]$. We further assume that $k_{\text{start}} = k_{\text{end}}$ if g does not occur in the index, i.e., $I[i]_j = 0$.

Algorithm 2 shows how putative hits are generated by querying the q-group index of buffered reads with q-grams of a reference sequence. Instead of considering all positions within the reference sequence, we allow to only use q-grams starting at a given subset P of reference positions. This allows to omit uninformative regions (see below). First, the number of hits per reference position is counted in parallel and stored in the array C (line 2). In the following, only positions with at least one hit are considered (line 5). The cumulative sum of the counts generates an interval for each position that determines where its hits are stored in the output array of the algorithm (line 6). Finally the occurrences for each reference q-gram are translated into hits that are stored in the corresponding interval of the output array (loop in line 8). We translate the position inside the text of concatenated reads into a read number (line 12) and a "hit diagonal" that denotes the putative start of the read in the reference (line 13, see Figure 2.4). For the read number, we assume that all reads are of the same length m. The practical implementation uses padding where this is not the case.

Each step of Algorithm 2 is implemented on the GPU with parallel OpenCL kernels. The filtering of P (line 5) and the cumulative sum (line 6) uses parallel prefix scans

Algorithm 2 Filtration of reference positions.

Input: reference sequence, ordered set P of considered reference positions with $p_l \in P$ being the l-th element of P, maximum read length m, q-group index (I, S, S', O)
Output: array H of hits as pairs (d, r) of diagonal d and read id r

1: Initialize array C of length $|P| + 1$ with zeros to count hits
2: **for** $p_i \in P$ **in parallel**
3: $(k_{\text{start}}, k_{\text{end}}) \leftarrow$ IndexPair(q-gram at reference position p_i)
4: $C[l + 1] \leftarrow k_{\text{end}} - k_{\text{start}}$
5: $P \leftarrow \{p_l \in P \mid C[l + 1] > 0\}$
6: $C \leftarrow$ cumulative sum of C
7: Allocate array H of length $2 \cdot C[|P|]$ to store hits
8: **for** $p_i \in P$ **in parallel**
9: $(k_{\text{start}}, k_{\text{end}}) \leftarrow$ IndexPair(q-gram at reference position p_i)
10: **for** $k \leftarrow 0 \ldots, k_{\text{end}} - k_{\text{start}} - 1$ **do**
11: $p' \leftarrow O[k_{\text{start}} + k]$
12: $r \leftarrow \lfloor p'/m \rfloor$
13: $d \leftarrow p - (p' \bmod m)$
14: $H[C[l] + k] \leftarrow (d, r)$

(see Section 1.3.2). All data structures reside in GPU memory; between the steps, at most constant amounts of data have to be transferred between host and GPU (e.g., a single integer). When fixing the number of PRAM processors, Algorithm 2 has the best possible time complexity, becoming linear in the number of investigated positions and obtained hits:

Theorem 2.6. *For a reference sequence with positions P, a q-group index (I, S, S', O) over reads of maximum length m and a machine word size w, Algorithm 2 calculates putative hits H between reads and reference sequence with time complexity*

$$\mathcal{O}\left(\frac{|H| + |P|}{\rho} + \log \rho\right)$$

on a CREW PRAM with ρ processors.

Proof. The array initializations in line 1 and 7 need $|P| + 1$ and $|H|$ operations. The first parallel kernel (line 2) has time complexity $\mathcal{O}(|P|/\rho)$ on the CREW PRAM. The second kernel (line 8) calculates each hit in constant time (see Theorem 2.3) and hence has time complexity $\mathcal{O}(|H|/\rho)$ on the PRAM. Finally the position filtering (line 5), implemented using a prefix scan (see Section 1.3.2), and the cumulative sum have time complexity $\mathcal{O}(|P|/\rho + \log \rho)$, resulting in above total complexity.

For correctness, we show that the resulting array H contains exactly all hits between the reference sequence at positions P and the reads. For the l-th position $p_l \in P$, we say that it occurs in the reads if the q-gram starting at position p_l occurs in the reads. Analogously, we say that p_l does not occur if the q-gram does not occur in the reads.

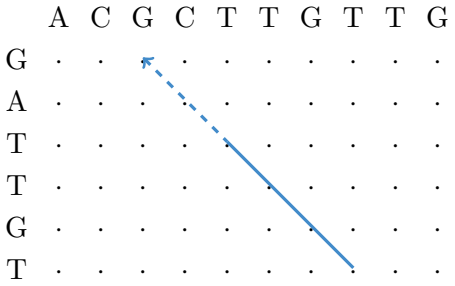

Figure 2.4: Determining the putative start position of a read from a matching q-gram via a hit diagonal (Rasmussen, Stoye, and Myers 2006). In the edit matrix between the reference sequence (top) and the read (left), a matching q-gram (solid line) induces (under the assumption that the read does not contain insertions or deletions before the q-gram) a diagonal that points to a putative start position of the read in the reference (dashed arrow).

We observe that before line 6, $C[l+1]$ contains the occurrence counts of the q-gram at the l-th position $p_l \in P$ by definition of the function INDEXPAIR. Then, it follows directly that the cumulative sum in line 6 results in $C[l]$ to contain the sum of the occurrence counts of positions $p_{l'} \in P$ with $l' < l$. Hence, the array H has the same number of entries as there are hits between the reference sequence at positions P and the reads (line 7). It remains to be shown that all hits of occurring positions are written into disjoint entries of H and that no hits are generated from positions that do not occur in the reads.

We first consider the second case: Let $p_l \in P$ be any non-occurring position. We note that $C[l+1] = 0$ because $k_{\text{start}} = k_{\text{end}}$ and p_l is removed from the set P in line 5. It therefore does not occupy any entries in H.

We prove the other case by induction. As basis, let $p_l \in P$ be the first reference position occurring in the reads. Let m be the number of occurrences of the q-gram starting at position p_l. Since p_l is the first occurring reference position, it holds $C[l] = 0$. Hence, the k-th hit (with $0 \leq k < m$) determined from the q-group index is stored in entry $H[k]$ and all m hits are stored in $H[0], \ldots, H[m-1]$.

Now, we assume that the n-th position $p_n \in P$ occurs m_n times in the reads. We consider the $(n+1)$-th position $p_{n+1} \in P$ with m_{n+1} occurrences. Without loss of generality, we assume $m_n > 0$ and $m_{n+1} > 0$ (which can always be achieved by reordering the set P). Since there are no occurring positions in between, the cumulative sum ensures that $C[n+1] = C[n] + m_n$. By induction, we know that $H[0] \ldots H[C[n] + m_n - 1]$ are filled with correct hits. Finally, the k-th hit of p_{n+1} is stored in $H[C[n+1] + k]$ such that $H[0], \ldots, H[C[n+1] + m_{n+1} - 1]$ is correct, too. □

The filtration step can lead to clusters of multiple potential hits pointing to the same

starting position by their diagonal (see Figure 2.4). This occurs in regions where the identity between read and reference is high. Instead of reducing these to a single hit, we found it to be faster to validate all hits and perform the removal of duplicates on those remaining after the validation step (see Section 2.5.3).

Reference sequence preprocessing The set P of reference positions to investigate and the reference sequences are retrieved from a precomputed data structure stored in HDF5 format[2]. First, this speeds up access to the reference. Second, we omit exceedingly frequent q-grams and stretches of unknown nucleotides (i.e., subsequences of Ns, see Section 2.3) from the set P. For q-gram index implementations this is common to avoid uninformative hits (Weese, Holtgrewe, and Reinert 2012). Further, we choose P such that the considered q-grams are disjoint, i.e., we investigate only every q-th position. Finally, P is sorted in numerical order of the q-grams. This increases the memory coalescence when accessing the q-group index, since subsequent threads will have a higher probability to access the same region in the index and hence the same memory bank in the global GPU memory.

2.5.2 Validation

The validation step takes the potential hits of the filtration step and calculates the edit distance (see Section 2.1) between a read and the reference sequence at its putative start position. If the edit distance is below a configurable threshold, the hit is considered good enough for being a possible origin of the read and will be postprocessed in the next step.

The edit distance is calculated with Myers' bit-parallel algorithm (Myers 1999) that simulates the edit matrix E between reference sequence and read. Myers' algorithm calculates the edit matrix column-wise, making use of bit-parallelization to obtain linear running time compared to the quadratic running time of the Smith-Waterman Algorithm. In contrast to the Smith-Waterman Algorithm it is though limited to report the minimal edit distance, not the actual alignment. The algorithm makes use of two observations: First, calculating the j-th column needs only column $j-1$. Second, each transition changes the edit distance by at most 1. Therefore it can maintain distance deltas between columns in bit vectors. A transition from one column to the next happens via a constant amount of bit-parallel operations on the bit vectors. In iteration j, the minimal distance between the read and any substring of the reference sequence that ends at position j can be retrieved. If the accepted error rate is limited, only a part (i.e., a band) of the edit matrix is needed to calculate the optimal edit distance. Our implementation of the algorithm follows a version that calculates only the relevant diagonal band of the edit matrix (Weese, Holtgrewe, and Reinert 2012; Hyyrö 2003). The implementation keeps the considered part of each column in a single machine word

[2] `http://www.hdfgroup.org/HDF5`, visited 11/2014

of size w (currently 32 bits). Thereby it provides a time complexity of $\mathcal{O}(|r|)$ with $|r|$ being the read length. While the reduction to the diagonal band restricts the maximum insertion or deletion size in a single alignment, mismatches are not affected. Hence, the procedure allows to discover partial matches of the read, as needed for local alignments. Large insertions or deletions can be rescued later in the postprocessing if a sufficiently large portion of the read aligns in this step.

Similar to Weese, Holtgrewe, and Reinert (2012), we use the algorithm to calculate the edit distance of a semi-global alignment in backward direction, thereby obtaining the best starting position of the alignment. For each hit, the fraction of matches or *percent identity* is obtained which we define here in compliance with Weese, Holtgrewe, and Reinert (2012) as $100 \cdot (|r| - k)/|r|$ where k is the edit distance and $|r|$ is the read length. Hits with a percent identity less than a given threshold are discarded. The default for this threshold is 80 percent which provides a decent sensitivity in our benchmarks (see Section 2.6). Decreasing it hurts performance, since more hits have to be postprocessed and written to disk.

2.5.3 Postprocessing

The goal of the postprocessing is to prepare the hits remaining after the validation step for output. Subsequent tools expect a read mapper to provide one or several hits in terms of positions in the reference along with the actual alignment to the positions. The alignments are then for example used for variant calling (see Chapter 3).

First, the postprocessing removes duplicate hits generated by clusters of matching q-grams (see Section 2.5.1). Intuitively, a particular hit is more likely to be the true origin of a read the fewer hits with the same or with a better score occur. Therefore, the next step sorts the hits into strata of the same percent identity. Upon invocation, PEANUT can be configured to discard hits based on their stratum, e.g., providing only the best stratum, all strata (in the following called *best-stratum* and *all-mode*) or a given number of strata. Let s be the desired number of strata. The sorting can be implemented by iterating over the hits of a read and adding the percent identity to a binary search tree. If the tree has already s nodes and the percent identity of the next hit to add is smaller than all values in the tree, the hit is discarded. This procedure is faster than sorting n hits and discarding afterwards (with complexity $\mathcal{O}(n \log n)$), because hits with a too low percent identity can be skipped beforehand. Since the height of the tree is limited by the desired number of strata, we obtain a time complexity of $\mathcal{O}(n \log s)$, scaling with the number of strata and reaching linear time when reporting only the best stratum.

For the remaining strata, we strive to provide for each hit a score for the confidence that this hit is the true origin of the read. In the desired output format SAM (Li et al. 2009), this is called *mapping quality*. Let O be the random variable denoting the true origin of the read r within all reference positions P. For each hit, the mapping quality (given in PHRED-scale) is expected to approximate the probability $1 - \Pr(O = p \mid r)$

that the hit position p is not the true origin of the read r in the reference. Li, Ruan, and Durbin (2008) define $\Pr(O = p \mid r)$ in a Bayesian way as

$$\Pr(O = p \mid r) = \frac{\Pr(r \mid O = p)}{\sum_{p' \in P} \Pr(r \mid O = p')}. \tag{2.4}$$

with P being the set of all reference positions. This assumes that reads are uniformly sampled from the reference. The likelihood $\Pr(r \mid O = p)$ of read r having been sampled from position p (in the following called the *sampling likelihood*) is estimated as the product of the base qualities of mismatching bases (see Section 1.2), i.e.,

$$\Pr(r \mid O = p) \approx \prod_{i \in S_{r,p}} q_{r,i}$$

with $q_{r,i}$ being the base quality of the i-th base in the read and $S_{r,p}$ being the positions in the read with a substitution according to the optimal alignment of the read at position p. Above estimation assumes that a substitution compared to the reference is most likely a sequencing error. In practice, directly calculating the posterior probability $\Pr(O = p|r)$ is infeasible since the optimal alignments over all reference positions would be needed. Many best-mappers (Li, Ruan, and Durbin 2008; Li and Durbin 2009; Li and Durbin 2010; Liu and Schmidt 2012) apply rough approximations[3], considering only the best and the second best hit of a read.

Since PEANUT shall be able to provide mapping qualities for all hits in the extreme, these techniques are not applicable. However, in contrast to best-mappers, we have access to the percent identities of all hits down to a given threshold (see Section 2.5.2). We choose to approximate the sampling likelihood based on the percent identity and calculate the mapping quality as shown above (Equation (2.4)). We first approximate the sampling likelihood for a single hit with percent identity $s \in [0, 100]$. Each edit operation in the underlying (but unknown) alignment is either a substitution, insertion or deletion. If the alignment represents the true sampling position of the read, all three may occur either due to genetic variation in the sequenced sample compared to the reference sequence (see Section 1.1) or due to a sequencing error. Both cases are unlikely and dominated by the expected sequencing error rate (see Section 1.2). Hence, the sampling likelihood decays exponentially in the number of edit operations in the alignment. Therefore, we approximate it as

$$\Pr(r \mid O = p) \approx Ce^{-\lambda k}$$

with k being the error rate of the read alignment obtained as $100 - s$ from the percent identity $s \in [0, 100]$ (see Section 2.5.2). Per default, λ and C are set to 1. This is a rough but conservative and quite general approximation. Under the assumption that this estimate is almost 0 for hits discarded during validation (since they will have a small

[3]For example, CUSHAW2 (Liu and Schmidt 2012) estimates the mapping quality as $250a \cdot (s_1 - s_2)/s_1$ with s_1 and s_2 being the best and second best local alignment score and a being the ratio between the length of the best local alignment and the read length.

percent identity) we do not consider all reference positions P. Instead, we approximate the posterior probability $\Pr(O = p \mid r)$ as

$$\Pr(O = p \mid r) \approx \frac{\Pr(r \mid O = p)}{\sum_{p' \in P'} \Pr(r \mid O = p')}$$

with P' being the validated hit positions of the read r. The PHRED-scaled mapping quality is then obtained as $-10 \log_{10}(1 - \Pr(O = p \mid r))$. Per default, we cap the mapping quality at 60 and force it to 0 for ambiguous hits (i.e., two or more best hits with the same percent identity) to generate values comparable to other read mappers like BWA (Li and Durbin 2009). This is useful to satisfy the expectations of downstream analysis steps (e.g., a mapping quality of 0 is often used to filter ambiguously mapping reads). Section 2.6.5 evaluates the quality of the approximation.

When mapping paired-end reads (see Section 1.2), it is beneficial to consider the hits of the mate read for determining the most likely origin of a read: two hits that lie within the expected insert size on the same chromosome are more likely the true origin of both reads than hits that appear alone. We call such hits *properly paired*. For these, we consider the sum of their scores (i.e., percent identities) in all above computations. The sampling likelihood for paired-end hits is estimated as $\Pr(r|p) \approx e^{s-200}$ since the maximum obtainable score is 200 for a properly paired hit. This results in properly paired hits appearing in better strata than other hits. The expected insert size is a configurable parameter of PEANUT.

2.6 Results

We evaluate the efficiency of GPU resource usage, the accuracy and the run time performance of PEANUT. Further, we evaluate the ability of the mapping quality measure defined in Section 2.5.3 to separate true hits from others. To ensure reproducibility and documentation, all analyses were implemented as a Snakemake workflow[4] (see Chapter 4).

2.6.1 GPU resource usage

To maximize utilization of the GPU hardware, idle cores have to be avoided. The two most important reasons for idle cores are branching and memory latency (see Section 1.3). The latter can be hidden if the SM (see Section 1.3) can execute a different warp while waiting on a memory transaction. The capability to do so can be measured as *occupancy*, that is the fraction of active warps among the maximum number of warps on an SM. The more active warps exist on an SM, the higher is the chance that latency can be hidden by executing another warp. Figure 2.5 shows the occupancy patterns of

[4] `http://peanut.readthedocs.org/analysis.html`, visited 11/2014

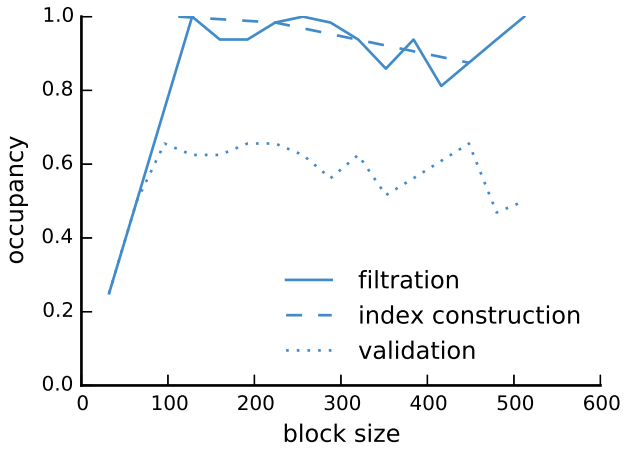

Figure 2.5: The occupancy of GPU cores depending on the thread block size. Shown are representative patterns for OpenCL kernels from the three main steps of the algorithm: index construction, filtration and validation.

the implemented OpenCL kernels, as measured with the NVIDIA™ CUDA command line profiler depending on the used thread block size (see Section 1.3). The thread block size influences the occupancy by limiting the number of potentially active warps and determining the amount of used registers and shared memory on the SM. Since the latter are limited, a bigger thread block size does not necessarily lead to a higher occupancy. As can be seen, the occupancy for all steps is high. For building of the q-group index (index construction) and the filtration step, it even reaches 1.0 which illustrates the benefit of the q-group index being tailored toward the GPU architecture.

2.6.2 Sensitivity

First, we strive to evaluate the sensitivity of PEANUT in terms of its ability (and hence that of the q-group index) to detect all alignments up to a given error rate. For this, we use the Rabema benchmark (Holtgrewe et al. 2011) that allows to compare mapping results based on a formalized framework. Rabema uses equivalence classes of ambiguous alignments, thereby letting the genomic origin of a read itself define how tight a mapping has to be in order to be considered correct. First, 10,000 Illumina reads (see Section 1.2) of length 100 were simulated using the read simulator Mason (Holtgrewe 2010) with the Saccharomyces cerevisiae genome (as provided by the Rabema data package[5]), default parameters and error rates. Second, the simulated reads were mapped to the genome using RazerS 3 (Weese, Holtgrewe, and Reinert 2012) with full sensitivity. In this configuration, RazerS 3 guarantees to report all alignments of a read up to a given error rate. For above notion of sensitivity, it would be insufficient to consider only the

[5] `http://www.seqan.de/projects/rabema`, visited 11/2014

true origins of a read as they are known from the simulation with Mason. The mapped reads were used to generate gold standards for Rabema to test against.

The sensitivity of the PEANUT algorithm is analyzed using q-grams of length 16, because it is computationally optimal on the current GPU hardware. PEANUT is configured to provide all semi-global alignments of a read and all other parameters are left at their default values. Sensitivity is assessed by the Rabema measure "Normalized found intervals" (Holtgrewe et al. 2011) and all alignments of a read are considered (all-mode). We investigate the relation between sensitivity and error rate of the gold standard alignments. Rabema defines the error rate as $100 - i$ with i being the percent identity as defined in Section 2.5.2. With a percent identity threshold of 60 (see Section 2.5.2) our algorithm provides 100% sensitivity for error rates below 5%, at least 99.86% sensitivity for error rates up to 10% and still 98.86% sensitivity with an error rate up to an unrealistically high 20%. With a stricter threshold of 80, PEANUT still reaches 98.81% sensitivity for the latter.

In general, the percent identity threshold should be set slightly more permissive than the expected error rate. This is because the replacement of N-characters in the reads and the reference (see Section 2.4) with random bases can introduce additional mismatches. Above rates are far better than the worst case sensitivity that can be expected by applying the pigeonhole principle (i.e., with reads of length 100 and q-grams of length 16, we can expect to find at least one perfectly matching q-gram for all alignments with at most 5 errors; see Section 2.3), such that using 16-grams appears to be a reasonable default choice in practice.

2.6.3 Comparison with other read mappers

We compare run time and accuracy of the PEANUT algorithm with other state of the art read mapping algorithms. The evaluation is conducted on 4 datasets:

1. 5 million simulated Illumina HiSeq 2000 reads,
2. 5 million real Illumina HiSeq 2000 reads from the human exome,
3. 10 million real paired-end Illumina HiSeq 2000 reads from the human exome,
4. 50 million real paired-end Illumina HiSeq 2000 reads from the whole human genome.

The simulated reads (dataset 1) were created from the ENSEMBL human reference genome[6] version 37 with Mason (see Section 2.6.2). The read length is set to 100 and all other parameters of Mason are left at their default values, such that reads with a typical error profile and mutation rate are generated. The second and third datasets are generated from real paired-end exome sequencing reads[7] (Martin et al. 2013) of

[6] `ftp://ftp.ensembl.org/pub/release-74/fasta/homo_sapiens/dna`, visited 11/2014
[7] `http://www.ebi.ac.uk/ena/data/view/ERR281333`, visited 07/2014

length 100 obtained from a patient suffering from uveal melanoma (a cancer of the eye) sequenced with an Illumina HiSeq 2000 sequencer. Dataset 2 consists of the first 5 million forward reads. Dataset 3 consists of both the first 5 million forward and backward reads, i.e., 10 million reads in total. Dataset 4 is generated from real paired-end whole genome reads of length 200 obtained from an african male[8]. The reads are part of the Illumina Platinum Genomes[9]. The first 25 million forward and backward reads were chosen, i.e., 50 million reads in total.

The benchmark was conducted on an Intel Core i7-3770™ system (4 cores with hyperthreading, 3.4 GHz, 16 GB RAM) with an NVIDIA Geforce™ 780 GPU (12 SMs, 3 GB RAM) and a 7200 rpm hard disk. We evaluated two modes of PEANUT. First, PEANUT was configured to find the best stratum of semi-global alignments (best-stratum mode) for each read. Second, PEANUT was configured to find all semi-global alignments (all mode) for each read. For comparison, we benchmarked the newest generation of BWA (BWA-MEM, version 0.7.5; Li (2013)), Bowtie 2 (version 2.0.2; Langmead and Salzberg (2012)), CUSHAW 3 (version 3.0.3; Liu, Popp, and Schmidt (2014)), CUSHAW2-GPU (version 2.1.8; Liu and Schmidt (2014)), NextGenMap (version 0.4.11; Sedlazeck, Rescheneder, and von Haeseler (2013)), RazerS 3 (version 3.2; Weese, Holtgrewe, and Reinert (2012)) and MrFast (version 2.6.0.1; Alkan et al. (2009)). All tools were configured to use 8 threads (the reasonable choice in case of 4 cores with hyperthreading). For MrFast, which does not support multithreading directly, this was achieved by partitioning the input files containing the reads into 100 equally sized chunks and running 8 parallel instances of MrFast with the Unix command *parallel* (the time for merging the resulting output was not included into the run time). NextGenMap was used in GPU mode, such that it makes maximum use of the available hardware. All read mappers were configured to output alignments in SAM format (Li et al. 2009) directly to the hard disk.

We outline the reasons for excluding several available read mappers from the benchmarks. At the time of writing (07/2014), no working installations of SOAP3 (Liu et al. 2012), SOAP3-dp (Luo et al. 2013) and BarraCUDA (Klus et al. 2012) could be obtained. A binary compiled against the setup of the test system (Ubuntu Linux 12.04 64-bit with NVIDIA CUDA 6) was not available for SOAP3 and SOAP3-dp. The compilation of SOAP3-dp-r177 and SOAP3-r146 failed on the used test system. BarraCUDA compiles but refuses to run on CUDA 6. Finally, read mappers specialized on RNA-Seq (see Section 1.2), e.g., STAR (Dobin et al. 2013) or TopHat (Trapnell, Pachter, and Salzberg 2009) were excluded, as this exceeds the scope of PEANUT. In principle, STAR (Dobin et al. 2013) could be applicable to DNA reads, and the authors claim speedups compared at least to other RNA-Seq focused read mappers. However, this comes at the cost of extensive memory usage by an uncompressed suffix array, which exceeds the capacity of the used test system.

[8] `http://www.ebi.ac.uk/ena/data/view/ERR091787`, visited 07/2014
[9] `http://www.illumina.com/platinumgenomes`, visited 07/2014

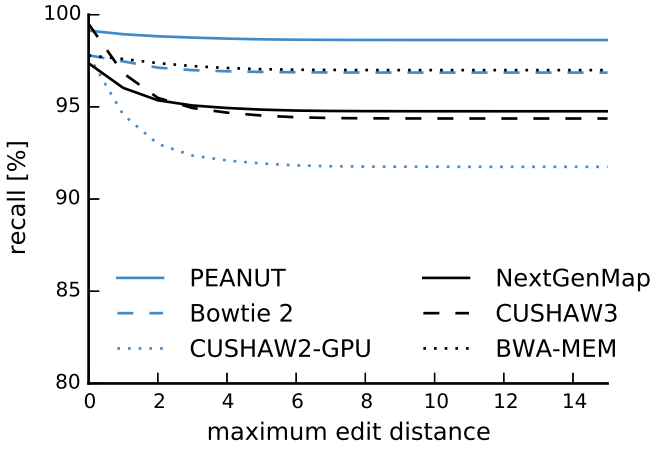

Figure 2.6: The obtained recall for different best-mappers given a maximum edit distance.

In the following, we distinguish between all-mappers and best-mappers (PEANUT occurs in both categories, using the all-mode and the best-stratum mode). Best-mappers only strive to find the single origin of a read on the reference sequence of a single organism. All-mappers provide all alignments of a read down to a given error rate. Hence, all-mapping is computationally more intensive.

Run time performance is measured three times as the total wall clock time for processing a dataset on the test system. Table 2.1 shows the run times for PEANUT and its competitors on all datasets. First, PEANUT in best-stratum mode outperforms all best-mappers (including the other GPU based mappers NextGenMap and CUSHAW2-GPU) on all datasets. On the biggest (and therefore most realistic) dataset, PEANUT is 2 times faster than the best competitor (BWA-MEM). Second, PEANUT in all-mode is 3 to 10 times faster than the all-mapper RazerS 3 and 4 to 6 times faster than the all-mapper MrFast. While Bowtie 2 provides an all-mode, too, it did not terminate in competetive time due to extensive memory requirements exceeding the capabilities of the test system.

The accuracy of the obtained alignments is assessed using Rabema (see Section 2.6.2). For best-mappers, using the simulated dataset 1, we measure precision and recall with Rabema as defined by Siragusa, Weese, and Reinert (2013): Recall is the fraction of reads correctly mapped to their original location. This location is known from the read simulation with Mason (see above). Precision is the fraction of correctly mapped reads among all reads that were mapped unambiguously (i.e., for which the mapper only provided exactly one hit). Figures 2.6 and 2.7 show the results with increasing maximum edit distance of the reads. Except when restricting to reads with zero errors (there, CUSHAW3 is slightly better), PEANUT slightly outperforms all other best-

Table 2.1: Performance of PEANUT and other read mappers on the human reference genome on four different datasets as defined in the text. Dataset sizes are given in gigabasepairs (Gbp; e.g., 1 Gbp is 10 million reads of length 100). Run times are listed for three consecutive repetitions. Dashes indicate that no run times could be obtained due to execution errors.

	mapper	type	time [min:sec]		
dataset 1 (0.5 Gbp)	PEANUT	best-stratum	**1:51**	**1:51**	**1:53**
	BWA-MEM	best	3:35	3:20	3:16
	Bowtie 2	best	5:13	5:12	5:12
	NextGenMap	best	3:06	3:08	3:06
	CUSHAW3	best	9:06	9:07	9:07
	CUSHAW2-GPU	best	2:38	2:38	2:39
	PEANUT	all	**22:26**	**22:37**	**22:42**
	RazerS 3	all	200:13	200:12	199:55
	MrFast	all	103:04	106:28	107:45
dataset 2 (0.5 Gbp)	PEANUT	best-stratum	**1:40**	**1:45**	**1:41**
	BWA-MEM	best	1:58	1:57	1:57
	Bowtie 2	best	3:30	3:14	3:12
	NextGenMap	best	2:28	2:38	2:29
	CUSHAW3	best	7:44	7:45	7:44
	CUSHAW2-GPU	best	2:20	2:22	2:21
	PEANUT	all	**13:43**	**13:57**	**13:57**
	RazerS 3	all	91:02	90:38	89:38
	MrFast	all	77:11	77:51	77:27
dataset 3 (1 Gbp)	PEANUT	best-stratum	**3:17**	**3:15**	**3:13**
	BWA-MEM	best	4:56	4:51	4:44
	Bowtie 2	best	8:20	8:18	8:20
	NextGenMap	best	4:46	4:42	4:45
	CUSHAW3	best	76:38	76:25	76:29
	CUSHAW2-GPU	best	6:30	6:09	6:07
	PEANUT	all	**28:05**	**28:11**	**28:11**
	RazerS 3	all	150:59	151:13	151:36
	MrFast	all	-	-	-
dataset 4 (10 Gbp)	PEANUT	best-stratum	**18:22**	**18:36**	**18:31**
	BWA-MEM	best	36:46	36:33	36:35
	Bowtie 2	best	54:38	54:22	55:51
	NextGenMap	best	-	-	-
	CUSHAW3	best	390:20	390:15	390:41
	CUSHAW2-GPU	best	30:23	30:30	30:34
	PEANUT	all	**254:43**	**254:49**	**254:19**
	RazerS 3	all	900:27	901:33	900:50
	MrFast	all	-	-	-

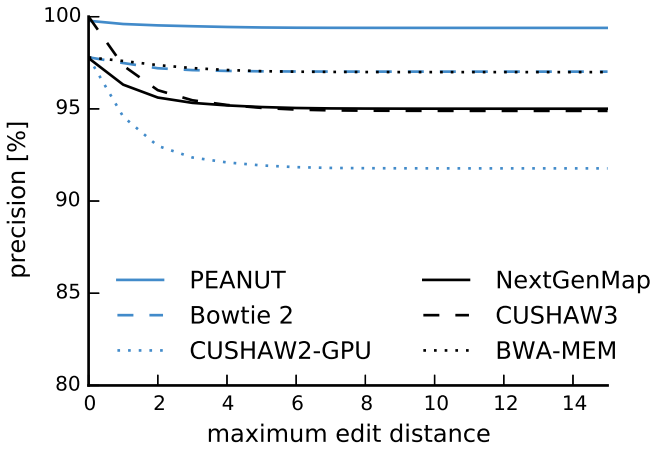

Figure 2.7: The obtained precision for different best-mappers given a maximum edit distance.

mappers in this benchmark. For best-mapping, the GCAT[10] provides an alternative, less formal approach to measure the accuracy of a mapper by testing whether a mapped read lies within 5 base pairs of its known origin in given simulated datasets. Köster and Rahmann (2014) additionally provide an evaluation of PEANUT with GCAT.

All-mappers are compared by their ability to find all alignments of a given edit distance or error rate. Here, we again use the sensitivity provided by Rabema (see Section 2.6.2). This involves creating a gold standard with RazerS 3 configured to full sensitivity, which is computationally expensive. We therefore perform this on only 1000 reads simulated with the same parameters as dataset 1. The gold standard is calculated for an edit distance of at most 15. Figure 2.8 shows that up to an edit distance of 4 PEANUT provides a sensitivity of almost 100%, similar to RazerS 3 and nearly as good as MrFast. Beyond an edit distance of 5, the sensitivity of PEANUT is superior to RazerS 3 and MrFast. This is due to RazerS 3 and MrFast being restricted to low error rates per default to achieve acceptable performance. In summary, at default settings, PEANUT provides similar or even better sensitivity than RazerS 3 and MrFast while being 3 to 10 times faster (see above).

While PEANUT requires at least 2.5 GB of GPU memory for filtration and validation, it is not restricted to running on high-end GPU models like the Geforce 780 used above. Table 2.2 shows that an advantage can be maintained when benchmarking on a different test system with an Intel Core i7-2600 (3.4 GHz, 16 GB RAM) and a four years old NVIDIA™ Geforce 580 GPU. We see that the older system is about 9% slower and still faster than the best competitor in Table 2.1 on the newer test system.

[10] http://www.bioplanet.com/gcat, visited 08/2014

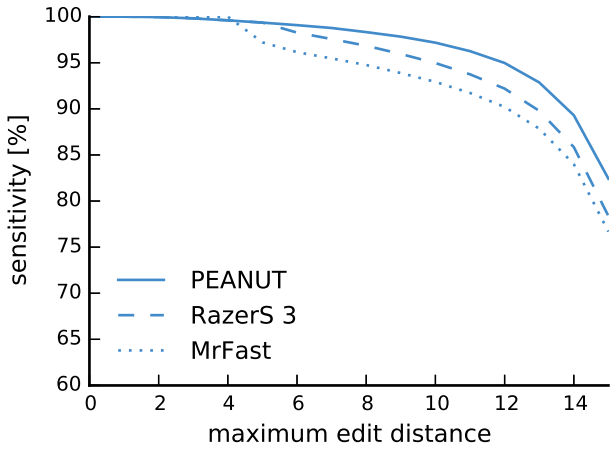

Figure 2.8: Sensitivity to find all alignments of a read given a maximum edit distance for different all-mappers with default parameters.

Table 2.2: Performance of PEANUT on a secondary test system with a four years old Geforce 580 GPU. See also Table 2.1.

dataset	type	time [min:sec]		
1	best-stratum	2:05	2:06	2:08
	all	24:04	24:33	24:50
2	best-stratum	1:56	1:43	1:47
	all	14:31	14:32	14:26
3	best-stratum	3:37	3:20	3:18
	all	29:55	30:06	30:13

2.6.4 Profiling algorithm steps

We profile the different steps of the PEANUT algorithm by recording the run time (as wall clock time) of indexing, filtration, validation, postprocessing and writing for each of above test datasets in best-stratum mode. Figure 2.9 shows the fraction of each run time. In practice, postprocessing runs in parallel to the other steps. The figure shows that this is reasonable, since it takes about 50% of the run time. Hence, during postprocessing of one set of buffered reads, we can perform the writing step for the last, as well as the indexing, filtration and validation steps for the next set of buffered reads.

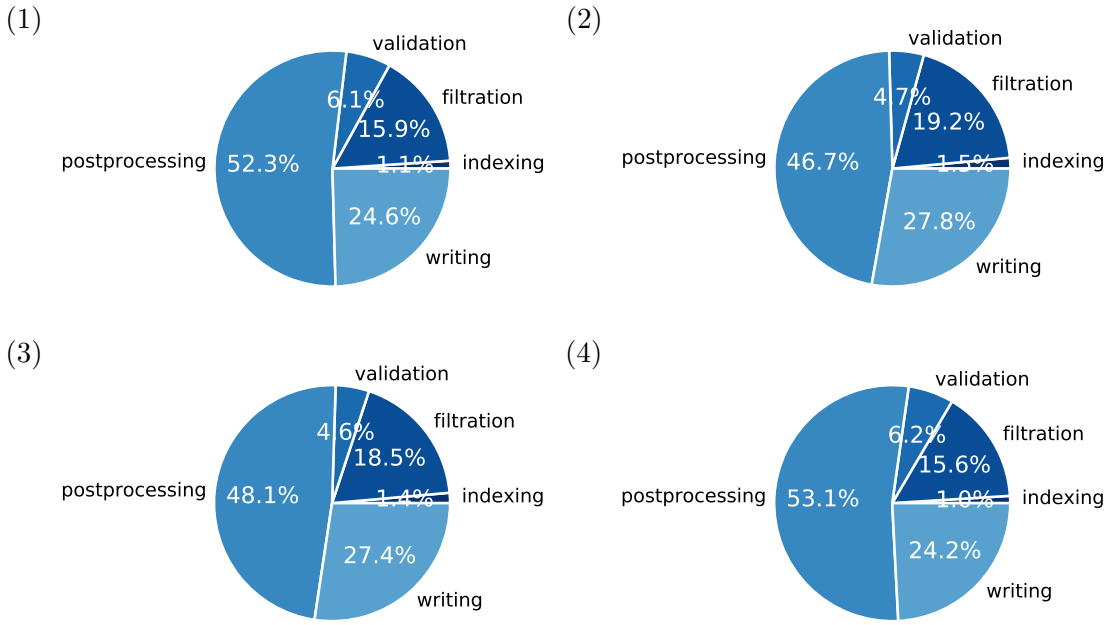

Figure 2.9: Run time fractions for steps of the PEANUT algorithm in best-stratum mode on dataset 1 to 4.

2.6.5 Evaluation of mapping qualities

Finally, the mapping qualities obtained by PEANUT are evaluated. These are intended to encode the probability of a read not being sampled from the reported position (see Section 2.5.3). The mapping qualities are given in PHRED scale (see Section 1.2), such that a value of 0 corresponds to a probability of 1. The higher the mapping quality, the smaller is the probability. In other words, if a hit has a small mapping quality, it shall be likely that the hit is a false positive, while a high mapping quality indicates a high confidence for the hit to be a true positive. Here, true positives are the correctly identified true sampling positions of the reads, whereas false positives are reported mapping locations that may have the same alignment score but are not the true origins of a read. For each PHRED-scaled mapping quality Q, the corresponding probability $10^{-Q/10}$ equals the expected false positive rate. Figure 2.10 shows the measured and expected false positive rate (i.e., the fraction of false positives among all hits) at increasing mapping qualities for the hits reported by PEANUT in all-mode on dataset 1. As can be seen, the measured false positive rate rapidly decays when increasing the mapping quality from zero. A mapping quality above 10 (i.e., a probability of 0.1) already guarantees almost no false positives. Comparison with the dashed line, depicting the expected false positive rate for each mapping quality, suggests that the mapping qualities provided by PEANUT are conservative in the sense of underestimating the true probabilities. Despite that, they provide a reasonable way to distinguish between true and false pos-

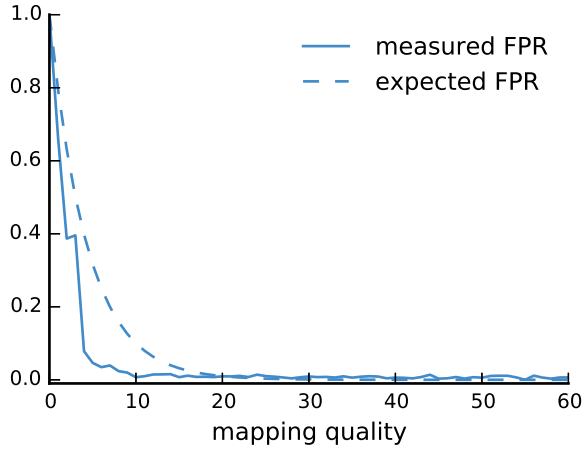

Figure 2.10: Mapping quality versus measured false positive rate for the hits reported by PEANUT in all-mode on the simulated dataset 1. The dashed line depicts the false positive rate as it would be expected from the probabilities encoded by the PHRED scaled mapping qualities (i.e., $10^{-\frac{Q}{10}}$ for mapping quality Q).

itives without the computational overhead needed for calculating concrete alignments of many suboptimal hits (see Section 2.5.3).

2.7 Discussion

In this chapter, we presented the q-group index, a variant of the q-gram index with a particularly small memory footprint, along with parallel algorithms for index building and querying. The algorithms fit nicely to the GPU architecture by using a combination of parallel element-wise and prefix scan operations over large arrays and requiring hardly any data transfer between the host and the GPU during their execution. We showed that the q-group index needs significantly less memory in practical scenarios than a conventional q-gram index while maintaining constant access time. The q-group index has been adopted by NVIDIA and is implemented in their NVBIO library[11].

On top of the q-group index we implemented the read mapper PEANUT. The q-group index enables the mapper to be the first that can perform both filtration and validation of hits on the GPU. So far, the GPU was leveraged only by BWT-based read mappers, e.g., by Liu and Schmidt (2014), or for calculating alignments, e.g., by Sedlazeck, Rescheneder, and von Haeseler (2013). PEANUT can be configured to either find all hits of a read or the best stratum. In both categories, it outperforms its competitors in terms of speed. It is 3 to 10 times faster than other all-mappers. Further, it is faster

[11] `http://nvlabs.github.io/nvbio`, visited 09/2014

than all other evaluated best mappers; in particular, it is 2 times faster than the fastest best-mapper on the most realistic dataset. The speed improvements do not come at the cost of mapping quality. In fact, our Rabema benchmarks show PEANUT to be even slightly more sensitive than the other all-mappers with default parameters and to have a slightly better recall and precision compared to the other best-mappers. In general the results suggest that PEANUT provides an accuracy comparable to other read mappers.

Apart from the GPU algorithms (i.e., filtration and validation) which have been implemented with PyOpenCL (Klöckner et al. 2012), other performance critical parts of the algorithm (reading, writing and postprocessing) have been implemented in Cython (Behnel et al. 2011). Cython compiles Python code to plain C or C++ which, among other optimizations, avoids the overhead generated by the Python interpreter. Still, the profiling in Section 2.6.4 identifies postprocessing as the current bottleneck of PEANUT. Future versions might therefore also parallelize parts of the postprocessing on the GPU with OpenCL.

In the current implementation, PEANUT requires the preprocessing of the reference into an HDF5 file (see Section 2.5.1), eliminating highly frequent q-grams, and sorting considered reference positions to maximize coalescence. Since the reference genome is often the same for many samples (e.g., the human genome), this is reasonable to save time during the mapping. Future work will try to eliminate this need by porting these steps to the GPU as well. For example, the q-group index itself can be used to efficiently count the occurrences of q-grams in the reference sequences to mask those being highly frequent.

The implementation of PEANUT in OpenCL allows it to be executed on other devices than the GPU. Here, coprocessors like field programmable gate arrays (FPGAs), as provided, e.g., by Altera[12] or the Intel Xeon Phi[13] architecture should be evaluated.

For the concept of the q-group index, other applications than plain read mapping are thinkable. Decoupled from validation, it can be used to, e.g., estimate contamination in a sample, by counting hits between a set of reads and a collection of bacterial or viral genomes, compared to the hits between the reads and the target organism. For this purpose, it is intended to separate the q-group index into a library that can be used independently from PEANUT.

[12] `http://www.altera.com/products/software/opencl`, visited 11/2014

[13] `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`, visited 11/2014

3 An algebraic variant caller

Mutations are alterations in the genomic sequence of an individual that occur, e.g., during cell division. We distinguish between *germline mutations*, appearing in the germ cells, which are responsible for sexual reproduction, and *somatic mutations*, occurring in other cells. Germline mutations are inherited by the next generation, but do not affect the originating individual. Somatic mutations behave vice versa: while they usually cannot be passed to the next generation, they can affect the individual. Mutations range from the substitution of single nucleotides (also called point mutations) over small insertions and deletions to large-scale structural alterations (e.g., inversions or duplications).

When sequencing a sample with NGS, mutations themselves are not directly observable. Instead, we can only infer *variants* compared to a given reference genome (see Section 1.1). In this context, *single nucleotide variants* (SNVs; a single nucleotide being different to the reference sequence), *indels* (small insertions and deletions), and structural variants (large insertions or deletions, inversions or duplications) can be considered. The process of finding such variants is commonly referred to as *variant calling*. Here, we focus on the calling of small variants (i.e., SNVs and indels) and present a novel approach to variant calling that is motivated by several problems with current solutions.

3.1 Introduction

Current variant callers for small variants often rely on a Bayesian approach to estimate at any genomic locus (see Section 1.1) the posterior probability for the null hypothesis of having no variant given the data. Then, a variant is *called* (i.e., reported), if the probability is small enough. The data consists of the pileup of reads mapped to this locus.

Definition 3.1 (Pileup)**.** *Let $\Sigma = \{A, C, G, T\}$ the alphabet of DNA bases, i be a genomic locus and s be a sample. The pileup of sample s at locus i is the pair*

$$\mathcal{P}_{s,i} := (\mathcal{R}_{s,i}, \mathcal{M}_{s,i})$$

where $\mathcal{R}_{s,i}$ is the sequence of read alignments overlapping locus i, and $\mathcal{M}_{s,i}$ is the sequence of corresponding mapping qualities (see Section 2.5.3). For any pileup, we further

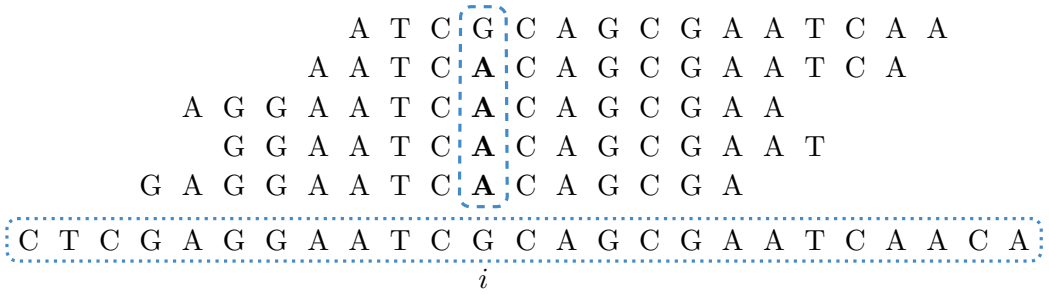

Figure 3.1: Pileup of reads mapped to a reference sequence (the dotted box at the bottom) over locus i for an exemplary sample s. Mapping qualities are omitted. The dashed column is the sequence $\mathcal{B}_{s,i}$ of read bases composed of matches and substitutions over the locus. Here, four of five reads report the base A instead of G, indicating a variant.

denote the sequence of read bases within the alignments being either a match or a substitution at locus i as $\mathcal{B}_{s,i} = (b_1, b_2, \ldots, b_n)$ with $b_i \in \Sigma$. By $\mathcal{Q}_{s,i} = (q_1, q_2, \ldots, q_n)$ with $q_i \in [0, 1]$ we denote the corresponding base qualities (see Section 1.2). For a set of samples S, let $\mathfrak{P}_{S,i}$ denote the set of pileups of all samples $s \in S$ at locus i.

Figure 3.1 provides an example. The number of read alignments in a pileup, i.e., $|\mathcal{R}_{s,i}|$ is also called the *read depth* or *coverage* of sample s at locus i. In the following, we will make use of $\mathcal{B}_{s,i}$ and $\mathcal{Q}_{s,i}$. As defined, these do not contain insertions or deletions, i.e., it holds $|\mathcal{B}_{s,i}| = |\mathcal{Q}_{s,i}| \leq |\mathcal{R}_{s,i}|$.

Auxiliary scores Bayesian approaches to detect variants rely mainly on the base qualities within the pileup. Sometimes, sequencing errors are poorly reflected by the base qualities, though. Therefore, auxiliary scores have been proposed to exclude such artifacts. On average, sequencers can be expected to sequence both DNA strands (see Section 1.1) equally well. Hence, for a locus exhibiting a variant, the null hypothesis is that the strand should be independent from the allele supported by a read. A deviation from the null hypothesis is called *strand bias* and a significant strand bias indicates a technical problem rather than a true variant. DePristo et al. (2011) test the null hypothesis using Fisher's exact test (Fisher 1922) by setting up a contingency table with a row for the reference and the alternative allele and a column for the forward and reverse strand reads supporting the alleles. Apart from the strand bias, other measures to avoid artifacts have been proposed (DePristo et al. 2011), mostly based on testing for independence of distributions. This includes the *read position bias* and the *mapping quality bias*. The read position bias describes that observing the alternative allele is not independent of the position within the read: e.g., if all alternative alleles in a pileup occur at the end of their supporting reads, they might be caused by a technical problem that occured in the last cycles of the sequencing. The mapping quality bias describes

whether reads supporting a variant have a systematically lower mapping quality. Such reads might in reality come from another locus with a similar sequence.

In the end, a variant caller provides a list of putative variants. Often, these variants are further processed in a *call and filter* approach. For example, to obtain the somatic mutations contained in a certain tumor sample, one can remove all variants that also occur in the corresponding healthy blood sample. Alternatively, one might strive for the de novo mutations of a patient with a certain syndrome and healthy parents. Such mutations are either somatic mutations of the patient himself or inherited from the germline mutations of the parents (and hence not affecting the parents). They can be obtained by subtracting those variants that are also present in the genomes of the parents. We identify three major problems with call and filter approaches that shall be outlined in the following.

FDR problem When calling variants over, e.g., the human genome, we have to consider up to three billion loci (see Section 1.1). With Bayesian approaches, similar to frequentist hypothesis testing, a multiple testing or multiple comparison problem occurs (Wasserman 2004; Müller, Parmigiani, and Rice 2006). For each locus, we reject the null hypothesis (i.e., report a variant) if the posterior probability falls below a given threshold. Doing this for many loci, we can expect a considerable amount of false discoveries depending on the used threshold. Without the filtering, controlling the false discovery rate (FDR) using the posterior probabilities is almost straightforward (Müller, Parmigiani, and Rice 2006). With filtering though, the obtained posterior probabilities do not properly reflect the significance of the variants since the used null hypothesis does not consider the filtering scenario. Hence, controlling the false discovery rate becomes difficult. We call this the *FDR problem*.

Insufficient evidence problem Both the calling and the filtering are sensitive to the used thresholds, e.g., the threshold used to decide about the null hypothesis. Consider a call and filter scenario where the variants of a sample s' shall be subtracted from the variants of a sample s, such that variants exclusive to sample s remain. Here, insufficient read depth in sample s' can lead to a true variant not being called, maybe with a confidence only slightly beyond some threshold. In turn, that variant might remain after subtraction if it is also called in sample s. Hence, insufficient evidence in the filter sample s' leads to a wrongly predicted variant. We call this the *insufficient evidence problem*.

N+1 problem The third problem arises when trying to fix the second: errors caused by insufficient evidence can be avoided by performing a joint calling of multiple samples, obtaining the probability of the event that no sample contains a variant at a given genomic locus. Often, studies seeking for mutations causing a certain disease use tens or hundreds of samples. It can happen that certain samples are added during a later

stage. When performing a joint calling, this leads to redundant computations, since the whole calling has to be repeated when adding a new sample. This is called the *N+1 problem*.

This chapter presents an algebraic approach for calling single nucleotide variants that overcomes above limitations. The N+1 problem is solved by creating per-sample indexes of precomputed likelihoods. Before the actual variant calling, the sample indexes are merged into an optimized index, which omits loci without any evidence for variation. This is a lightweight and massively parallel step that can be repeated with little overhead upon the addition of new samples. On top of the optimized index, we implement a novel algebraic SNV calling procedure that allows to estimate the posterior probability for having a variant that behaves like specified in a given algebraic query expression. The query expression can be used to flexibly model filtering scenarios. Since the posterior probability reflects the filtering, we can easily control the false discovery rate. The usage of the optimized index makes the variant calling fast, such that different filter scenarios and thresholds can be explored within seconds. The approach is implemented in the new variant caller ALPACA (ALgebraic PArallel CAller), available as open source software under the MIT license (see Section A.1).

The chapter first summarizes related work (Section 3.2). Then, a Bayesian approach to joint variant calling over multiple samples is described (Section 3.3). Section 3.4 presents the novel algebraic variant calling method, followed by a description of its applications (Section 3.5). In Section 3.6, a description of efficient data structures and parallel algorithms for the approach is presented. The chapter is closed by an evaluation of the method (Section 3.7), a description of the software (Section 3.8) and a discussion (Section 3.9).

3.2 Related work

Many variant callers have been published so far. We focus on those being most popular, namely GATK, SAMtools and FreeBayes. While all three can solve the insufficient evidence problem by allowing to jointly call variants in multiple samples together, only GATK provides a mechanism for solving the N+1 problem, and none addresses the FDR problem.

Apart from traditional variant calling, we review two specialized algorithms for detecting somatic mutations in tumors called MuTect and Strelka. In a sense, these solve the three identified problems by restricting themselves to a limited setup: instead of allowing to jointly call variants on arbitrary sets of samples, both tools expect a pair consisting of a tumor and a normal sample (e.g., unaffected blood of the same patient). Tumors can be heterogeneous, can develop metastases or reoccur after treatments. In such cases, the ancestry of a tumor sample is often unclear and it has to, e.g., be compared to various other instances of the tumor. In consequence, general approaches are even useful when studying somatic mutations in tumors.

Variant calling is usually embedded into a larger workflow, responsible for the filtering but also for annotating the variants with biological knowledge and compiling results into a human readable form. We describe Exomate as an example of such a system.

GATK The Genome Analysis Toolkit, GATK (DePristo et al. 2011), is a general framework for the analysis of next-generation sequencing data. Part of GATK is the Bayesian variant caller UnifiedGenotyper, which allows to jointly call all variants of a given set of samples. The Bayesian variant calling presented in Section 3.3 is based on the procedure of UnifiedGenotyper, while introducing some changes. UnifiedGenotyper uses a modified version of that procedure to also call indels. Recently, GATK was extended by the HaplotypeCaller. Instead of investigating each genomic locus independently, it creates local assemblies of reads and derives haplotypes (see Section 1.1) over a putatively variant region. This allows to provide phasing information for the reported variants (i.e., whether two variants occur on the same or different chromosomes; see Chapter 1). HaplotypeCaller entails a solution to the N+1 problem: preprocessing results can be written to a per-sample text file in the genomic variant call format[1] (GVCF), which also contains averaged information about non-variant genomic regions. The actual calling combines these intermediate results for calculating posterior probabilities. If a sample is added, the preprocessing results of the other samples can be reused for another calling. Further, GATK provides two commonly used preprocessing steps. First, it provides a *base quality recalibration* by learning systematic deviations between the reported base qualities and the empirical base quality for several covariates, including the machine cycle (i.e., the position of the base within the read) and the dinucleotide context (i.e., the preceding base). Under the assumption that most loci do not host a variant, the empirical base quality for each covariate is estimated by considering substitutions between the read and the reference as sequencing errors. Second, GATK allows to perform a local multiple sequence realignment over loci which host indels (*indel realignment*). Since read mappers align each read separately, the optimal alignment of the same indel within different reads might differ. The multiple sequence realignment considers all reads covering a locus together, thereby increasing the consistence between the reported indels. Finally, GATK provides a method to estimate the false discovery rates of the reported variants using gaussian mixture model trained with a set of high confidence known variants (*variant quality score recalibration*). However, this approach does not consider filtering between samples, such that the reported false discovery rates are only accurate when jointly calling all variants in a single set of samples without performing additional filtering.

SAMtools Li et al. (2009) provide SAMtools as a collection of tools for handling mapped reads. Among other features, it implements a Bayesian variant calling procedure similar to that of GATK. Around loci which contain an indel, reads are often misaligned. Especially if the indel is small, assuming one or more substitutions might attain a better

[1] `https://sites.google.com/site/gvcftools/home/about-gvcf`, visited 06/2014

alignment score (see Chapter 2) than assuming an insertion or deletion. Such misaligned read bases can occur systematically around an indel and cause spurious variant calls. With a Hidden Markov Model (Li 2011), SAMtools calculates a base alignment quality (BAQ) that is used to lower the base qualities of likely misaligned read bases such that they contribute less to variant calls. ALPACA uses SAMtools to obtain read pileups together with the recalibrated base qualities (see Section 3.6.5) from a BAM file of mapped reads.

FreeBayes Similar to GATK's HaplotypeCaller, FreeBayes (Garrison and Marth 2012) performs a local assembly of haplotypes: the genotypes used in FreeBayes contain multiple loci and are inferred from the assembled haplotypes. FreeBayes considers the posterior probability for concrete genotype combinations of the investigated samples instead of allele frequencies. At the expense of increased computational complexity, this allows to find the set of genotypes that maximizes the posterior probability in a gradient search. The authors argue that this is superior to only considering allele frequencies, where the maximum likelihood genotype of each sample has to be reported (see Equation (3.6) in Section 3.4).

MuTect Instead of calling variants, MuTect (Cibulskis et al. 2013) tries to directly detect somatic point mutations. It does so for pairs of tumor and normal samples (i.e., a sample from unaffected tissue of the same patient). MuTect considers the family of models M_f^m with m being the assumed sequenced allele and $f \in [0, 1]$ being its frequency. At each genomic locus i, for the tumor sample t with pileup $\mathcal{P}_{\mathrm{t},i}$, it determines the log odds score

$$\log_{10} \frac{\Pr(\mathcal{P}_{\mathrm{t},i} \mid M_f^m)}{\Pr(\mathcal{P}_{\mathrm{t},i} \mid M_0^m)}$$

of the model likelihoods for the case of observing alternative allele m at frequency f against observing only the reference allele (i.e., observing allele m at frequency 0). The alternative allele m and the alternative allele frequency f is determined heuristically from the distribution of read bases at that locus. For the normal sample n with pileup $\mathcal{P}_{\mathrm{n},i}$, the log odds score for observing no variant against a heterozygous variant

$$\log_{10} \frac{\Pr(\mathcal{P}_{\mathrm{n},i} \mid M_0^m)}{\Pr(\mathcal{P}_{\mathrm{n},i} \mid M_{0.5}^m)}$$

is calculated analogously. A somatic mutation in the tumor is reported if both scores exceed thresholds motivated by the expected mutation rates. Cibulskis et al. (2013) claim to provide superior sensitivity on tumor samples that are contaminated or a heterogeneous mixture of more than one tissue, because the expected allele frequency is estimated from the data. In contrast to the prior assumption of a ploidy (see Section 3.3) this does not weight down (and thereby allows to detect) mutations that only occur in a small subset of the reads. However, deviations from the expected ploidy can also indicate sequencing errors, which are henceforth not properly reflected in the log odds

scores. MuTect introduces an extensive set of auxiliary scores to avoid that this causes too many false positives.

Strelka Similar to MuTect, Strelka (Saunders et al. 2012) detects somatic mutations by investigating allele frequencies in a tumor and a normal sample. At any genomic locus i, Strelka calculates likelihoods $\Pr(\mathcal{P}_{\mathrm{t},i}|f_\mathrm{t})$ and $\Pr(\mathcal{P}_{\mathrm{n},i}|f_\mathrm{n})$ to observe the pileup $\mathcal{P}_{\mathrm{t},i}$ of the tumor and $\mathcal{P}_{\mathrm{n},i}$ of the normal sample given the alternative allele frequencies $f_\mathrm{t}, f_\mathrm{n} \in [0, 1]$. Then, the posterior probability for a somatic mutation is approximated by integration over all combinations of unequal alternative allele frequencies, i.e.,

$$\int_0^1 \int_0^1 \mathbf{1}_{f_\mathrm{t} \neq f_\mathrm{n}} \Pr(\mathcal{P}_{\mathrm{t},i} \mid f_\mathrm{t}) \Pr(\mathcal{P}_{\mathrm{n},i} \mid f_\mathrm{n}) \Pr(f_\mathrm{t}, f_\mathrm{n}) \, \mathrm{d}f_\mathrm{t} \mathrm{d}f_\mathrm{n}$$

with $\mathbf{1}_{f_\mathrm{t} \neq f_\mathrm{n}}$ being the indicator function for $f_\mathrm{t} \neq f_\mathrm{n}$ and $\Pr(f_\mathrm{t}, f_\mathrm{n})$ being the prior probability of observing the two allele frequencies in a tumor and a normal sample. The case of $f_\mathrm{t} \approx f_\mathrm{n}$ is not handled explicity. It is rather avoided by sampling the space of considered allele frequencies in practice. Further, the obtained posterior probability is multiplied by the probability for having the reference genotype in the normal sample (calculated in a Bayesian way; see Section 3.3). This is necessary to avoid false positives caused by copy number variations, i.e., mutational changes duplicating parts of a chromosome.

Exomate Practical implementations of variant calling workflows need to perform various additional steps. We exemplify this by describing the Exomate framework (Martin 2014). Exomate consists of three parts: a variant calling workflow, a PostgreSQL[2] database and a web frontend. The variant calling workflow is implemented with Snakemake (Chapter 4). It uses BWA to map the sequence reads to the reference genome. Afterwards, it performs various preprocessing steps on the mapped reads proposed by DePristo et al. (2011), e.g., PCR duplicates are detected, indels are realigned and base qualities are recalibrated. The removal of PCR duplicates (see Section 1.2) is useful since they can bias the distribution of alleles at a locus. Variants are called with GATK's UnifiedGenotyper. Unlike the new HaplotypeCaller (see above), UnifiedGenotyper does not handle the N+1 problem. The Exomate workflow therefore does not jointly call all samples together, but calls samples of the same patient and its family jointly in a group. This partially solves the insufficient evidence problem, and largely avoids the N+1 problem since the addition of a new sample only causes one group to be called again. The called variants are imported into the PostgreSQL database. The information about a variant alone is often not sufficient to judge over its biological impact: e.g., a variant that lies within a gene (see Section 1.1) and leads to truncation of the encoded protein is more likely the cause of a disease than a variant that occurs in the intron of a gene. Hence, Exomate annotates variants with biological information using the tool

[2] `http://www.postgresql.org`, visited 11/2014

VEP (McLaren et al. 2010). Finally, a web frontend allows to access the called variants easily, filter variants of samples against each other and evaluate the consequence of a variant by its annotation.

3.3 Bayesian variant calling

Given a set of samples $S = \{s_1, s_2, s_3, \ldots, s_n\}$ and an arbitrary genomic locus i, we first describe how to estimate the probability of having a single nucleotide variant in any of the samples. Here we assume that all samples have the same ploidy $\Phi \in \mathbb{N}$ (i.e., the number of copies of each chromosome; see Section 1.1). Human samples are diploid, i.e., the ploidy is $\Phi = 2$. It is common practice to ignore the different ploidy of the sex chromosomes in male samples (see Section 1.1) and correct the obtained calls in later steps (DePristo et al. 2011). Since we are only interested in single nucleotide variants we can observe the alleles $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$ and the genotypes Γ_Φ as the set of all Φ-combinations of alleles with replacement. For a ploidy $\Phi = 2$, there are 10 possible genotypes, namely

$$\mathrm{AA},\ \mathrm{CC},\ \mathrm{GG},\ \mathrm{TT},\ \mathrm{AC},\ \mathrm{AG},\ \mathrm{AT},\ \mathrm{CG},\ \mathrm{CT},\ \mathrm{GT}.$$

Definition 3.2, Lemma 3.3 and Lemma 3.5 are taken from DePristo et al. (2011) and Lemma 3.6 is inspired by Li (2010). For clarification, we add two proofs that have been omitted in the literature. We first define the probability that a read base has been sampled from a given allele, i.e., the likelihood of the read base given the allele.

Definition 3.2 (Allele likelihood)**.** *Let $s \in S$ be a sample with pileup $\mathcal{P}_{s,i}$ at locus i. Let A be the random variable indicating the true allele and B be the random variable indicating the observed base. For any k, let $b \in \mathcal{B}_{s,i}$ be the k-th base in the pileup of sample s at locus i. Let $q \in \mathcal{Q}_{s,i}$ be the corresponding k-th base quality. Then, the likelihood to observe base b given that $A = a \in \Sigma$ is the true allele at the locus is*

$$\Pr(B = b \mid A = a) = \begin{cases} 1 - q & \text{if } b = a, \\ q \cdot \xi_{b|a} & \text{otherwise,} \end{cases}$$

where $\xi_{b \mid a}$ is the probability of base b to be observed given that allele a was miscalled.

The probability $\xi_{b|a}$ can be obtained from a technology specific confusion matrix (see Table 3.1). The rows of the confusion matrix sum up to 1. This is reasonable since, given that the sequencer reported a wrong base, there are only three wrong choices for each allele. In consequence, for a fixed base quality q and allele a, the sum of the allele likelihoods for all four possible bases equals 1:

$$\sum_{b \in \Sigma} \Pr(B = b \mid A = a) = 1 - q + q \sum_{b \in \Sigma, b \neq a} \xi_{b|a} = 1.$$

Recalling that a genotype is composed of Φ alleles, we can use the allele likelihood to infer the likelihood to observe a certain base under a given genotype.

Table 3.1: Confusion matrix for base miscalls of Illumina HiSeq sequencers as reported by DePristo et al. (2011). Column b in row a depicts the probability $\xi_{b|a}$ to observe base b given that the true allele a in the DNA sequence is miscalled.

	A	C	G	T
A	-	0.58	0.17	0.25
C	0.35	-	0.11	0.54
G	0.32	0.05	-	0.63
T	0.46	0.22	0.32	-

Lemma 3.3 (Genotype likelihood)**.** *Let Φ be the expected ploidy of sample $s \in S$. Then the likelihood to observe base $b \in \mathcal{B}_{s,i}$ under genotype $G = g \in \Gamma_\Phi$ is*

$$\Pr(B = b \mid G = g) = \frac{1}{\Phi} \sum_{a \in g} \Pr(B = b \mid A = a).$$

Assuming independence between the sequence reads, the likelihood to observe the pileup $\mathcal{P}_{s,i}$ under genotype $G = g$ is

$$\Pr(\mathcal{P}_{s,i} \mid G = g) = \prod_{b \in \mathcal{B}_{s,i}} \Pr(B = b \mid G = g).$$

Proof. Initially, we see that

$$\Pr(B = b \mid G = g) = \sum_{a \in g} \Pr(B = b \mid A = a) \Pr(A = a \mid G = g).$$

A priori, the read base could have been sampled from any of the alleles of the genotype with equal probability, i.e., $\Pr(A = a \mid G = g) = \frac{1}{\Phi}$, and therefore

$$\Pr(B = b \mid G = g) = \frac{1}{\Phi} \sum_{a \in g} \Pr(B = b \mid A = a).$$

Then, $\Pr(\mathcal{P}_{s,i} | G = g)$ follows directly from assuming independence between the reads. □

We now want to calculate the likelihood of a pileup to be observed under a given alternative allele frequency. Here, the alternative allele frequency is the number of non-reference alleles in the genotype, denoted as $|g|$ for $g \in \Gamma_\Phi$. For any ploidy Φ, there is always one genotype $g \in \Gamma_\Phi$ with $|g| = 0$. This genotype represents that no allele differs from the reference. If assuming a uniform distribution, for a given alternative allele frequency $M_s = m \leq \Phi$ of sample $s \in S$ at locus i, we can denote the prior probability of a certain genotype $G = g$ with $|g| = m$ as

$$\Pr(G = g \mid M_s = m) = \binom{3 + m - 1}{m}^{-1}, \tag{3.1}$$

i.e., the inverse of the number of possibilities to draw m alternative alleles from the set of possible alternative alleles with replacement. With four alleles $\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}$, the set of possible alternative alleles has always a cardinality of three, since one allele is the reference allele. In the diploid case (i.e., $\Phi = 2$), this means that for an alternative allele frequency of 1, the prior probability for any genotype $g \in \Gamma_2$ with $|g| = 1$ is $\Pr(G = g \mid M_s = 1) = \frac{1}{3}$. For an alternative allele frequency of 2, the prior for any genotype g with $|g| = 2$ is $\Pr(G = g \mid M_s = 2) = \frac{1}{6}$, since all homozygous genotypes except the one with the reference allele and three heterozygous genotypes have an alternative allele frequency of 2 (recall that, in contrast to heterozygous, homozygous means that all chromosomes carry the same allele; see Section 1.1). The likelihood of the pileup given the allele frequency follows directly.

Lemma 3.4 (Allele frequency likelihood)**.** *For any sample $s \in S$, at any genomic locus i, the likelihood to observe the pileup $\mathcal{P}_{s,i}$ given an allele frequency $M_s = m$ is*

$$\Pr(\mathcal{P}_{s,i} \mid M_s = m) = \sum_{g \in \Gamma_\Phi, |g| = m} \Pr(\mathcal{P}_{s,i} \mid G = g) \Pr(G = g \mid M_s = m).$$

Recall that $\mathfrak{P}_{S,i} = \{\mathcal{P}_{s,i} \mid s \in S\}$ denotes the set of pileups at locus i for a set of samples S. For multiple samples, the likelihood of observing no alternative allele in all samples, i.e., a total alternative allele frequency $M = 0$ is given as

$$Pr(\mathfrak{P}_{S,i} \mid M = 0) = \prod_{s \in S} \Pr(\mathcal{P}_{s,i} \mid M_s = 0). \tag{3.2}$$

Here, we assume that the samples are independent. Then, Bayes' Theorem allows to calculate the probability of having a total alternative allele frequency of zero given the pileups of the samples.

Lemma 3.5 (Reference genotype probability)**.** *Let $\mathfrak{P}_{S,i}$ be the set of pileups of samples S at any genomic locus i. The probability to have in total zero alternative alleles is*

$$\Pr(M = 0 \mid \mathfrak{P}_{S,i}) = \frac{\Pr(M = 0) \Pr(\mathfrak{P}_{S,i} \mid M_s = 0)}{\Pr(\mathfrak{P}_{S,i})}.$$

The event $M = 0$ can be seen as our null hypothesis. A genomic locus with a small probability for the null hypothesis can be considered to exhibit a variant in any of the samples S. For Lemma 3.5, we need the prior probability $\Pr(M = m)$ for a total alternative allele frequency of m. Here, the following simplifying assumptions are made. It is assumed that the samples S come from a certain population. The size of the population is finite and constant over time. From one generation to the next, all individuals die and are replaced by offspring. Reproduction is a random process: some individuals may have no offspring, while some can have multiple. This is called the Wright-Fisher model (Wakeley 2008). Further, during reproduction, mutations can appear, but only at a locus previously not mutated. This is called the infinite-sites model (Wakeley 2008).

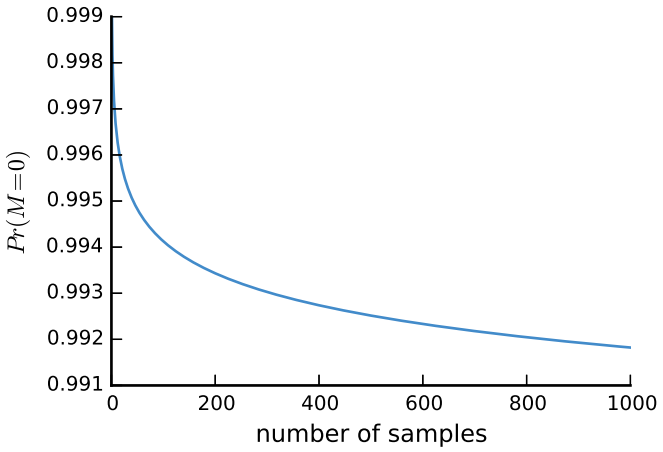

Figure 3.2: The prior probability for zero alternative alleles depending on the number of samples.

Using these assumptions, the prior probability of a total alternative allele frequency of m is approximated as (Li 2010; DePristo et al. 2011; Wakeley 2008)

$$\Pr(M = m) = \begin{cases} \frac{\Theta}{m} & \text{if } m > 0, \\ 1 - \Theta \sum_{i=1}^{|S| \cdot \Phi} \frac{1}{i} & \text{otherwise.} \end{cases} \tag{3.3}$$

The parameter Θ denotes the heterozygosity, the expected fraction of heterozygous loci. For human samples, DePristo et al. (2011) propose to set $\Theta = 0.001$. Figure 3.2 shows the development of $\Pr(M = 0)$ for increasing number of samples with heterozygosity $\Theta = 0.001$ and ploidy $\Phi = 2$. For a single sample, the probability is $\Pr(M = 0) = 1 - (\Theta + \Theta/2) = 0.9985$. With more samples, it decreases, since it becomes more likely that a sample with a mutation at that locus is included. However, even when considering large numbers of samples, we see that with the proposed heterozygosity Θ, most loci are expected to exhibit the reference genotype.

We finally have to calculate the marginal probability $\Pr(\mathfrak{P}_{S,i})$ of all pileups. This could be done using the law of total probability and summing over all combinations of genotypes (DePristo et al. 2011), which becomes infeasible with larger numbers of samples. Here, we adapt an approach of Li (2010) developed for the Bayesian model of SAMtools (see Section 3.2) and present a proof by structural induction. It can be calculated by dynamic programming (see Section 3.6.5) instead of having to sum over an exponential set of genotype combinations.

Lemma 3.6 (Marginal pileup probability)**.** *Let $S = \{s_1, s_2, s_3, \dots\}$ be an ordered set of samples with a set of pileups $\mathfrak{P}_{S,i}$ at an arbitrary genomic locus i. We define*

$$z_{j,k} := \Pr(\mathfrak{P}_{\{s_1,\dots,s_j\},i} \mid M = k)$$

as the likelihood of the pileups given that k alternative alleles are contained in the first j samples. Then, we claim

$$z_{j,k} = \sum_{k'=0}^{\Phi} z_{j-1,k-k'} \Pr(\mathcal{P}_{s_j,i} \mid M_{s_j} = k')$$

with $z_{0,0} = 1$ *and* $z_{j,k} = 0$ *for* $0 > k > |S| \cdot \Phi$ *or* $j \leq 0$ *and* $k > 0$*. Then, the likelihood to observe* $\mathfrak{P}_{S,i}$ *given an alternative allele frequency* $M = m$ *can be calculated as* $\Pr(\mathfrak{P}_{S,i} \mid M = m) = z_{|S|,m}$ *such that the law of total probability yields*

$$\Pr(\mathfrak{P}_{S,i}) = \sum_{m=0}^{|S|\cdot\Phi} \Pr(\mathfrak{P}_{S,i} \mid M = m) \Pr(M = m)$$

as the probability for observing the pileups $\mathfrak{P}_{S,i}$*.*

Proof. It suffices to show $\Pr(\mathfrak{P}_S \mid M = m) = z_{|S|,m}$ by structural induction. Let $S = \{s\}$ be an arbitrary set of a single sample. It holds that

$$\begin{aligned} z_{1,m} &= \sum_{k'=0}^{\Phi} z_{0,m-k'} \Pr(\mathcal{P}_{s,i} \mid M_s = k') \\ &= z_{0,0} \Pr(\mathcal{P}_{s,i} \mid M_s = m) \\ &= \Pr(\mathcal{P}_{s,i} | M_s = m) \\ &= \sum_{g \in \Gamma_\Phi, |g| = m} \Pr(\mathcal{P}_{s,i} | G = g) \Pr(G = g | M_s = m). \end{aligned}$$

If $m > \Phi$, there is no such genotype and the latter sum is $0 = \Pr(\mathfrak{P}_{S,i} \mid M = m)$. Else, the sum describes the likelihood of the pileup of sample s given an alternative allele frequency of m, which is exactly the definition of $\Pr(\mathfrak{P}_{S,i} \mid M = m)$.

Let $S = \{s_1, s_2, \ldots, s_{n-1}, s_n\}$ be a set of multiple samples. We assume that

$$\Pr(\mathfrak{P}_{S',i} \mid M = k) = z_{|S'|,k}$$

holds for any k and subset $S' \subset S$ with $|S'| = |S| - 1$. Without loss of generality we assume that $S' = \{s_1, s_2, \ldots, s_{n-1}\}$ (which can always be achieved by relabeling samples). It holds

$$\begin{aligned} z_{n,m} &= \sum_{k'=0}^{\Phi} z_{n-1,m-k'} \Pr(\mathcal{P}_{s_n,i} \mid M_{s_n} = k') \\ &= \sum_{k'=0}^{\Phi} \Pr(\mathfrak{P}_{S'} \mid M = m - k') \Pr(\mathcal{P}_{s_n,i} \mid M_{s_n} = k') \\ &= \Pr(\mathfrak{P}_{S,i} \mid M = m) \end{aligned}$$

by induction. □

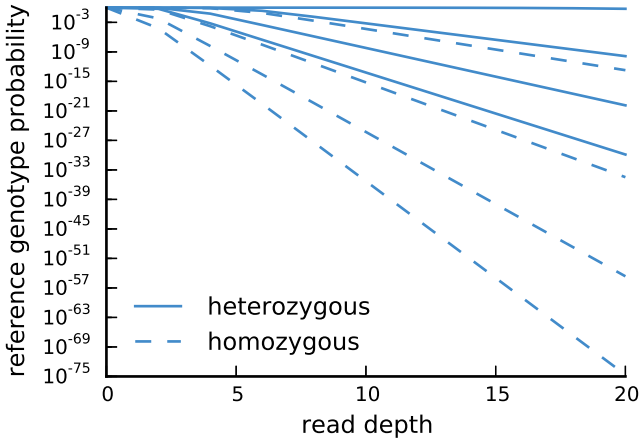

Figure 3.3: Exemplary development of reference genotype probability depending on read depth. Solid lines depict the heterozygous case, dashed lines the homozygous case (see text) with ploidy $\Phi = 2$, heterozygosity $\Theta = 0.001$, reference allele A and alternative allele C. For each case, base qualities are uniformly set to 0.2, 0.02, 0.002 and 0.0002 (from top to bottom line). With zero reads, the reference genotype probability equals the prior probability for an allele frequency of zero, here $\Pr(M = 0) = 1 - (0.001 + 0.0005) = 0.9985$.

In the following, we consider some examples. First, we investigate the development of the reference genotype probability depending on the read depth, the base quality and the fraction of non-reference bases in the pileup. For this, we assume a pileup with $2n$ bases and investigate two cases. In the heterozygous case, n of these represent the same alternative allele. In the homozygous case, all $2n$ bases represent the alternative allele. Further, for both cases we consider a range of base qualities, namely 0.2, 0.02, 0.002 and 0.0002 which are set uniformly for all bases. Figure 3.3 shows the resulting reference genotype probabilities. As expected, increasing the read depth increases the degree of belief in observing a variant, reflected in decreasing the reference genotype probability exponentially. Naturally, the homozygous case yields lower probabilities as there is more evidence for the alternative allele. Further, the better the base qualities, the lower the reference genotype probability.

Now, we investigate the influence of number and read depths of samples. We consider an arbitrary locus with reference allele A and two samples s_1 and s_2, showing different counts of allele A and alternative allele C in their pileup. Table 3.2 shows two aspects. First, as long as the fraction of alternative bases over both samples remains constant, the reference genotype probability is almost the same, independently of the depths of the individual samples. Second, sample s_2 showing a pileup of reference bases or having no coverage at all causes almost no difference in the resulting probability. The first observation indicates that shallowly sequenced samples can be rescued with others of the same or better read depth. The second observation illustrates that the model works

Table 3.2: Reference genotype probability for an example scenario of two samples with varying support for a reference and an alternative allele. The pileup of each sample is characterized as $n + m$ with n being the number of A bases and m being the number of C bases. A is assumed to be the reference allele. A uniform base quality of 0.002 was assumed.

s_1	s_2	probability
18+18	2+2	3.6751e-43
10+10	10+10	3.6756e-43
15+15	5+5	3.6756e-43
20+20	40+0	6.1261e-44
20+20	0+0	3.3415e-44

as desired: we calculate the probability of having zero reference alleles at the considered loci; with one sample showing evidence for a variant, subsequent samples that show no evidence for a variant should not have a major effect on the probability. Only adding more samples with evidence for a variant will further improve the probability.

3.4 Algebraic variant calling

On top of the posterior probabilities for having zero alternative alleles at a given genomic locus, we can now define an algebraic variant calling procedure that allows to incorporate a desired filtering scenario into the obtained posterior probabilities. This will allow us to report, e.g., somatic mutations along with their probability while controlling the rate of false discoveries.

A variant locus is a genomic locus that hosts a variant. We strive to model filtering in terms of subtraction and union of variant loci. For this, we first define an algebra of variant loci over the true variant genomic loci of a given set of samples. Further, we define a set of expressions over this algebra that is restricted to the relevant operations union and difference.

Definition 3.7 (Algebra of variant loci)**.** *For a finite set of samples $S = \{s_1, s_2, s_3, \dots\}$, let $V_S = V_{s_1} \cup V_{s_2} \cup V_{s_3} \cup \dots$ be the true set of variant genomic loci in the samples. The algebra of variants is the set algebra $\mathcal{A}_S := (2^{V_S}, \cup, \cap, \setminus)$ with powerset 2^{V_S} as carrier set and binary operations union ($\cup$), intersection ($\cap$) and difference ($\setminus$). Further, let $\mathcal{Q}_S$ denote the smallest set of expressions over the algebra of variants with*

$$\begin{aligned} V_s &\in \mathcal{Q}_S \\ \phi_1 \cup \phi_2 &\in \mathcal{Q}_S \\ \phi_1 \setminus \phi_2 &\in \mathcal{Q}_S. \end{aligned}$$

for $\phi_1, \phi_2 \in \mathcal{Q}_S$ and $s \in S$.

Now, the desired filtering scenarios can be written as expressions of $\mathcal{Q}_S$: e.g., all variant loci that are exclusively in sample s_1 compared to samples s_2 and s_3 are expressed as $V_{s_1} \setminus (V_{s_2} \cup V_{s_3})$. In the following, depending on the context, we refer to these expressions either as *sets* of variant loci (i.e., referring to the result of their evaluation) or *queries* (i.e., referring to the syntactic construct). The set of allowed expressions $\mathcal{Q}_S$ does not contain the empty expression. This would describe the empty set of variant loci, but is of no use in combination with the other possible expressions. Further, intersections are omitted. The intersection of the variant loci of multiple samples would describe those loci that are variant in all samples. In practice, this is quite restrictive and of limited use: a single sample not exhibiting a particular variant locus would remove the locus from the result of the intersection, which might contain many other samples all exhibiting the variant locus. Instead, we want to be robust against such outlier samples and opt for providing a relaxed form of the intersection in a future work (see Section 3.9).

As the true variant loci in each sample are unknown, so is the carrier set of our algebra. Hence, an expression $\phi \in \mathcal{Q}_S$ cannot be calculated directly. Instead, our goal is to approximate ϕ by calculating posterior probability for any locus i not being in the set ϕ given the read pileups $\mathfrak{P}_{S,i}$, i.e., $\Pr(i \notin \phi | \mathfrak{P}_{S,i})$. We rely on the Bayesian variant calling method described in Section 3.3, which yields the probabilities for the base case, where we want to know if any sample from a set of samples exhibits a variant. We assume independence between the samples and demand that no sample occurs twice in ϕ. Then, the following posterior probability can be obtained.

Theorem 3.8 (Posterior query probability)**.** *Let $S = \{s_1, s_2, s_3, \dots\}$ be a set of independent samples with pileups $\mathfrak{P}_{S,i}$ at genomic locus i and $\phi \in \mathcal{Q}_S$ be a query. The posterior probability for $i \notin \phi$ is*

$$\Pr(i \notin \phi \mid \mathfrak{P}_{S,i}) := \begin{cases} \Pr(M = 0 \mid \mathfrak{P}_{S',i}) & \text{if } \phi = \bigcup_{s \in S' \subseteq S} V_s \\ 1 - \Pr(i \in \phi_1 \mid \mathfrak{P}_{S,i}) \cdot \Pr(i \notin \phi_2 \mid \mathfrak{P}_{S,i}) & \text{if } \phi = \phi_1 \setminus \phi_2 \\ \Pr(i \notin \phi_1 \mid \mathfrak{P}_{S,i}) \cdot \Pr(i \notin \phi_2 \mid \mathfrak{P}_{S,i}) & \text{if } \phi = \phi_1 \cup \phi_2 \end{cases}$$

with $\Pr(i \in \phi_1 | \mathfrak{P}_{S,i}) = 1 - \Pr(i \notin \phi_1 | \mathfrak{P}_{S,i})$.

Proof. We show the correctness by structural induction over ϕ. As base case, let $\phi = \bigcup_{s \in S' \subseteq S} V_s$. Lemma 3.5 tells us that $\Pr(M = 0 \mid \mathfrak{P}_{S',i})$ is the probability for the samples S' exhibiting zero alternative alleles at locus i. Hence, it holds $\Pr(M = 0 \mid \mathfrak{P}_{S',i}) = \Pr(i \notin \phi \mid \mathfrak{P}_{S,i})$. By setting $S' = \{s\}$, we see that this also holds for $\phi = V_s$.

We now assume that $\Pr(i \notin \phi_1 \mid \mathfrak{P}_{S,i})$ and $\Pr(i \notin \phi_2 \mid \mathfrak{P}_{S,i})$ are correct for $\phi_1, \phi_2 \in \mathcal{Q}_S$. If $\phi = \phi_1 \setminus \phi_2$, we see

$$\begin{aligned} 1 - \Pr(i \in \phi_1 \mid \mathfrak{P}_{S,i}) \cdot \Pr(i \notin \phi_2 \mid \mathfrak{P}_{S,i}) &= 1 - \Pr(i \in \phi_1 \text{ and } i \notin \phi_2 \mid \mathfrak{P}_{S,i}) \\ &= 1 - \Pr(i \in \phi_1 \setminus \phi_2 \mid \mathfrak{P}_{S,i}) \\ &= \Pr(i \notin \phi_1 \setminus \phi_2 \mid \mathfrak{P}_{S,i}) \\ &= \Pr(i \notin \phi \mid \mathfrak{P}_{S,i}). \end{aligned}$$

Finally, if $\phi = \phi_1 \cup \phi_2$ and not $\phi = \bigcup_{s \in S' \subseteq S} V_s$ for some S', it holds

$$\begin{aligned}\Pr(i \notin \phi_1 \mid \mathfrak{P}_{S,i}) \cdot \Pr(i \notin \phi_2 \mid \mathfrak{P}_{S,i}) &= \Pr(i \notin \phi_1 \text{ and } i \notin \phi_2 \mid \mathfrak{P}_{S,i}) \\ &= \Pr(i \notin \phi_1 \cup \phi_2 \mid \mathfrak{P}_{S,i}) \\ &= \Pr(i \notin \phi \mid \mathfrak{P}_{S,i}).\end{aligned}$$

□

If the query probability $\Pr(i \notin \phi \mid \mathfrak{P}_{S,i})$ at a genomic locus i is sufficiently small, we can expect i to be contained in the set ϕ. We can use this to define the set of putatively variant loci.

Definition 3.9 (Putatively variant loci)**.** *Let L be the set of all genomic loci. For a query $\phi \in \mathcal{Q}_S$ and a threshold $\alpha \in [0, 1]$, we denote the set of putatively variant loci as*

$$\phi^*_\alpha := \{i \in L \mid \Pr(i \notin \phi \mid \mathfrak{P}_{S,i}) \leq \alpha\}.$$

The set ϕ^*_α is an approximation of ϕ. Calculating ϕ^*_α involves a collection of hypothesis tests for each locus $i \in L$, with $i \notin \phi$ being the null hypothesis. For each test, we can make two kinds of errors. The type I error is to erroneously reject the null hypothesis, i.e., to state $i \in \phi^*_\alpha$ for a locus i not contained in the unknown set ϕ in reality. Such a locus i would be called a false positive. The type II error is to falsely accept a null hypothesis, i.e., to state $i \notin \phi^*_\alpha$ for a locus i which in reality is contained in the set ϕ. Such loci i are called false negatives. The trade-off between these two errors can be controlled with the threshold α. The set of genomic loci to be considered is very large, given that, e.g., the human genome is about 3.2 billion basepairs long. With so many hypothesis tests, the choice of α becomes crucial to avoid excessive amounts of false positives. We choose to control the false discovery rate (FDR), i.e., the fraction of false positives among all predicted loci. Here, in contrast to frequentist approaches based on p-values, we use posterior probabilities in a Bayesian approach (see Wasserman 2004). While controlling the FDR was originally introduced by Benjamini and Hochberg (1995) for the frequentist approach, it is also possible, even more straightforwardly, with Bayesian probabilities (Müller, Parmigiani, and Rice 2006).

First, as shown by Müller, Parmigiani, and Rice (2006), for a given α we calculate the expected false discovery rate, assuming independence of the hypotheses, from the posterior probabilities as the expected number of false positives divided by the total number of positives, i.e.,

$$\overline{FDR}_\alpha = \frac{1}{|\phi^*_\alpha|} \sum_{i \in \phi^*_\alpha} \Pr(i \notin \phi | \mathfrak{P}_{S,i}). \tag{3.4}$$

We can then control the expected false discovery rate by choosing the maximum α such that the expected FDR does not exceed a given threshold α^*, i.e.,

$$\alpha = \max_{\alpha' : \overline{FDR}_{\alpha'} \leq \alpha^*} \alpha'. \tag{3.5}$$

Now, we can link the putatively variant loci to the set of true variant loci and describe the approximation quality with the FDR, which follows directly from Equations (3.4) and (3.5).

Corollary 3.10. *For a query ϕ and a desired false discovery rate α^*, ϕ^*_α with α from Equation (3.5) is the largest set of putative variants with an expected FDR of at most α^*.*

In general, the information about a locus being variant alone is of limited use. Since the posterior probability only considers allele frequencies, it cannot be used to find the most likely genotype. Therefore, at each predicted variant locus i, we estimate the concrete variant in each sample s by calculating the maximum likelihood genotype as

$$\arg\max_{g \in \Gamma_\Phi} \Pr(B_{s,i} | GT = g). \tag{3.6}$$

3.5 Applications of algebraic variant calling

The algebraic approach described above allows to flexibly define various filtering scenarios. The main applications shall be outlined in the following. The most simple application is to call all variants in a set of samples, e.g., a population. For this, we obtain all putative variants by formulating

$$\phi = V_{s_1} \cup V_{s_2} \cup V_{s_3} \cup \dots$$

the estimation of which is reduced to Bayesian variant calling by Theorem 3.8. To estimate the variant loci that are exclusive to a group of samples S compared to another group of samples S', we can write

$$\phi = \bigcup_{s \in S} V_s \setminus \bigcup_{s' \in S'} V_{s'}.$$

The sets of samples S and S' could, e.g., represent two different outcomes of a disease. The resulting set of variant loci might contain the genetic reason for the different outcomes, for example a certain variant that damages an important protein.

Such queries can also be used to infer mutations from variants: e.g., S can be a set of children, whereas S' can be their parents. Then, variant loci exclusive to the children are putative de novo mutations occuring in the child generation. Note that, mutations of a certain kind are missed here, namely when a locus hosts already a variant compared to the reference genome in a parent and a mutation induces a further change to this locus in the child. Such cases are rare though, because they would imply a mutation event to happen twice at the same locus within some generations. In fact, in an in-house database (Exomate; see Section 3.2) of 54,844,721 loci with single nucleotide variants called in 326 human samples, only 424,752 loci, less than 1%, exhibit more than one alternative allele.

Algebraic variant calling can also be used to find somatic mutations that might explain the development of a tumor. Such somatic mutations should have occurred at some point during or before tumor development, and should not be present in the healthy tissue of the same individual. Assume that t is a sample of the tumor, and h is a healthy sample (e.g., from blood) of the same patient. Somatic mutations occuring in the tumor can be estimated by writing

$$V_{\mathrm{t}} \setminus V_{\mathrm{h}}.$$

Tumors are often heterogeneous and may change over time. Algebraic variant calling allows to flexibly select mutations of different tumor branches, e.g.,

$$((V_{\mathrm{t}_1} \cup V_{\mathrm{t}_2}) \setminus (V_{\mathrm{t}_3} \cup V_{\mathrm{t}_4})) \setminus V_{\mathrm{h}}$$

if $\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_4$ are samples of these different branches and we are interested in the first two branches.

Often, investigating multiple instances of the same disease is advisable, which can be accomplished easily with the query language. Let t', h' be another tumor-healthy sample pair, then the somatic mutations in any of the two tumors can be estimated as

$$(V_{\mathrm{t}} \cup V_{\mathrm{t}'}) \setminus (V_{\mathrm{h}} \cup V_{\mathrm{h}'}).$$

This grouped filtering has the advantage that regions of weak evidence might be compensated by the additional samples. Instead of pooling healthy and tumor samples, it is sometimes advisable to investigate a paired scenario, i.e., only consider the corresponding healthy sample and not all healthy samples for each candidate somatic mutation. Such pairing can be formulated as

$$(V_{\mathrm{t}} \setminus V_{\mathrm{h}}) \cup (V_{\mathrm{t}'} \setminus V_{\mathrm{h}'}).$$

This combined query has still an advantage over creating two separate queries $V_{\mathrm{t}} \setminus V_{\mathrm{h}}$ and $V_{\mathrm{t}'} \setminus V_{\mathrm{h}'}$ because the combined probability might be significant at loci where the single probabilities are too weak to pass the FDR control.

Along with the putatively variant genomic loci, the procedure yields posterior probabilities for each reported locus being in the set ϕ of true variants and controls the false discovery rate, hence solving the FDR problem (see Section 3.1). Further, no separate thresholds need to be applied for the calling of the filter samples, thereby solving the insufficient evidence problem (see Section 3.1).

We investigate the behavior of the involved probabilities for a filtering scenario in an example. Consider two samples s_1 and s_2 and the query $\phi := V_{s_1} \setminus V_{s_2}$. For an arbitrary locus, we assume that s_1 has a pileup with 20 bases equal to the reference allele, say A, and 20 bases showing an alternative allele, say C. In other words, sample s_1 shows strong evidence for a heterozygous variant. For our filter sample s_2, we now evaluate the depths $d = 0, 2, \ldots, 8, 10$ with an equal amount of A and C bases for each depth d. Figure 3.4 shows that, as expected, the reference genotype probability for the filter sample s_2

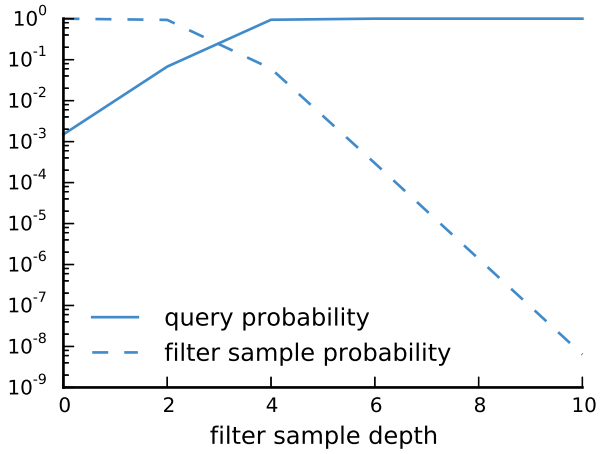

Figure 3.4: Example for the development of probabilities in a filtering scenario with increasing depth. The solid line shows the query probability $\Pr(i \notin V_{s_1} \setminus V_{s_2} \mid \mathfrak{P}_{s_2,i})$ (see Definition 3.8) with s_1 and s_2 as defined in the text. The dashed line depicts the reference genotype probability for the filter sample $\Pr(M = 0 \mid \mathfrak{P}_{s_2,i})$. Base quality was uniformly set to 0.002.

decreases with increasing depth. In turn, this increases the overall query probability for ϕ. With a depth of 4 in the filter sample, the query probability is already close to one here. In other words, the better the evidence for the variant locus in the filter sample is, the lower is the degree of belief for that locus to be in the set ϕ here.

3.6 Algorithm and data structure

Algebraic variant calling allows to retrieve a set of putatively variant genomic loci given a query which describes a filtering scenario. We now investigate its algorithmic implementation. An important observation is that for a given set of samples, different queries use the same allele frequency likelihoods (Lemma 3.4), while all following probabilities like reference genotype probability (Lemma 3.5), pileup probability (Lemma 3.6) and query probability (Theorem 3.8) differ. Hence, it is reasonable to avoid re-calculating allele frequency likelihoods for different queries. We therefore split our algorithm into three distinct steps:

1. sample indexing,
2. index merging,
3. calling.

The first step preprocesses a sample and stores the obtained information (e.g., the allele frequency likelihoods) in an index. The second step merges the indexes of the samples

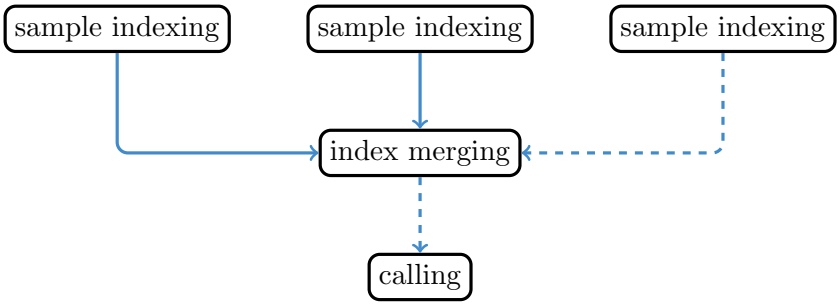

Figure 3.5: Outline of the ALPACA calling workflow for three samples. The dashed arrows indicate the part of the work that has to be repeated upon adding the third sample.

into a combined *optimized index*. Finally, the third step uses the optimized index to satisfy a given query. Upon the addition of a sample, only steps two and three have to be repeated. This solves the N+1 problem (see Section 3.1). The different steps are described in the following. Figure 3.5 visualizes the idea.

3.6.1 Sample indexing

We process each sample separately, and store the information in an index in the hierarchical data format[3] (HDF5). HDF5 provides facilities optimized for storing numerical data in a hierarchical structure. The data is stored as datasets, that are organized in a hierarchy of groups. Both groups and datasets can be accessed by names, and allow to store additional attributes. HDF5 supports transparent compression and fast access to slices of datasets.

For a sample s and each chromosome, we store five datasets. For each chromosome, we consider all genomic loci i covered by at least one read base. Certainly, a single read base could never yield a reliable variant call. Hence, one could think about storing only loci with three or five read bases. However, we will later call jointly on multiple samples. Here, that single read base is at least important to provide comprehensive information if the locus is called due to other samples. First, a dataset P^s of integers is stored, with P^s_i containing the position of genomic locus i in the chromosome. Second, a dataset L^s of floats is stored: Entry $L^s_{i,m} := \ln \Pr(\mathcal{P}_{s,i} \mid M_s = m)$ contains the allele frequency likelihood (Lemma 3.4) at genomic locus i represented in logarithmic space with $\mathcal{P}_{s,i}$ being the pileup at the genomic locus. Third, we store a dataset G^s of bytes, setting $G^s_i := j$ with $g_j \in \Gamma_\Phi$ being the maximum likelihood genotype (see Equation (3.6)). Fourth, we record for each genomic locus i and each allele $a \in \Sigma$ the total number of sequence reads d supporting it, i.e., $D^s_{i,a} := d$ in a dataset D^s of bytes. Finally, we record analogously the numbers of supporting sequence reads coming from the reverse strand

[3] `http://www.hdfgroup.org/HDF5`, visited 04/2014

Table 3.3: Datasets stored per chromosome in the sample index. Each dataset has as many rows as loci are covered by at least one read in the chromosome.

dataset	data type	bits per entry	columns	content
P^s	integer	32	1	positions
L^s	floating point	32	$\Phi + 1$	likelihoods
G^s	integer	8	1	maximum likelihood genotypes
D^s	integer	8	6	allele depths
R^s	integer	8	6	reverse strand allele depths

(see Chapter 1) in a dataset R^s. Sometimes, the sequencer cannot properly determine a read base. These bases are encoded as N (in contrast to A, C, G, T) in the read. Since many N bases can be an indicator for a technical problem, we store above two read counts also for a virtual N allele. Analogously we store the read counts for a virtual *indel* allele, counting all reads that support any insertion or deletion at that locus. We use bytes for D^s and R^s since one can expect the number of sequence reads to be less than 256 at most loci. Further, the confidence with 255 reads should be high enough. Similar to DePristo et al. (2011) we take a random subsample of size 255 in cases where the read depth is higher. Table 3.3 provides a summary of the datasets.

3.6.2 Index merging

Motivated by the following observation, this step merges the sample indexes together, creating an optimized index in HDF5 format. The prior probability for an alternative allele frequency greater than zero in a single sample is

$$\Pr(M > 0) = \sum_{m=1}^{\Phi} \Pr(M = m) = \sum_{m=1}^{\Phi} \frac{\Theta}{m}$$

with $\Pr(M = m)$ from Equation (3.3). DePristo et al. (2011) propose an expected heterozygosity of $\Theta = 0.001$ for humans. Together with a ploidy $\Phi = 2$, we obtain a prior probability $\Pr(M > 0) = 0.0015$ for a non-reference locus. In other words, loci that host a variant are rare. Since loci that do not host a variant in any of the samples are not relevant for calling, we can expect to save a considerable amount of space by including only those with at least one sample exhibiting a variant according to the maximum likelihood genotype. In practice, this makes the optimized index much smaller than the sample indexes, although it contains information for multiple samples. Further, it is much faster to process upon calling, since irrelevant loci are already filtered out.

For each chromosome, we store the datasets L^s, G^s, D^s and R^s of all samples s for these loci and a dataset P of integers with the genomic positions of the loci. If a sample has zero coverage (i.e., no reads) over any of the loci, the corresponding allele frequency

likelihoods of that sample are set to 1. This results in the posterior reference genotype probability (see Lemma 3.5) of that sample at such loci becoming equal to the prior probability $\Pr(M = 0)$ for zero alternative alleles.

3.6.3 Calling

Finally, the optimized index can be used to serve putative variants for a specific query $\phi \in \mathcal{Q}_S$. This entails calculating the query probability (Theorem 3.8) for each locus from the merged index, using the allele frequency likelihood as stored in L^s and calculating putatively variant loci as specified in Definition 3.9 and Corollary 3.10. The loci are provided in the variant call format[4] (VCF) along with the strand bias (see Section 3.1) that is calculated using the sequence read counts from D^s and R^s. Further, the maximum likelihood genotype is read from G^s and shown along with the read counts for each sample s occurring in the query ϕ.

3.6.4 Compression

The three steps of the algorithm involve accessing and creating two types of HDF5-based indexes. The merged optimized index (see Section 3.6.2) contains only putatively variant loci, and is therefore comparably small: e.g., the merged index for six exome sequencing samples described in Section 3.7 with a total BAM file size of 43 GB takes only about 300 MB (i.e., 0.7%) of disk space. Since it will be accessed for each query (i.e., each invocation of the calling step), it should be optimized for fast reading. Hence we choose to store the merged index uncompressed. In contrast, the sample index is accessed less often (only during the merge step), but can become huge because data for each covered genomic locus has to be stored, making it a valuable target for compression strategies. Table 3.3 shows that the sample index needs $4 + 4(\Phi + 1) + 1 + 6 + 6 = 21 + 4\Phi$ bytes per locus. When expecting a ploidy $\Phi = 2$, this results in 29 bytes for each sample and locus. For the human genome with about 3.2 billion basepairs, in the worst case with at least one read base covering each genomic locus the sample index would therefore need about 86 GB. Of these, likelihood storage (dataset L^s) occupies about 41%, allele depth and depth of reverse strand reads (datasets D^s and R^s) require 21% each, storing the chromosomal positions (dataset P^s) occupies 14% and the maximum likelihood genotypes (dataset G^s) require 3% of the storage. In the following, we discuss compression strategies.

HDF5 allows to compress datasets with several compression algorithms. While we want to compress numerical data, we talk about texts, words and characters in the following, since it appears more natural in combination with compression. Here, a text can also be a sequence of integers, the bytes of which are the characters and words being arbitrary subsequences. H5Py[5], the HDF5 bindings for Python used in our implementation, provide

[4] `http://samtools.github.io/hts-specs/VCFv4.2.pdf`, visited 11/2014
[5] `http://www.h5py.org`, visited 11/2014

P^s	60394	60395	60396	60397	60398	60399	61300	61301	61302	61303	61304
$\overline{P^s}$	60394	1	1	1	1	1	901	1	1	1	1

Figure 3.6: Transforming the position dataset P^s to the incremental form $\overline{P^s}$. Dashed boxes depict contiguously covered stretches, which are transformed into an offset together with a sequence of ones.

GZIP[6] and LZF[7] compression by default. Both are variants of the LZ77 algorithm by Ziv and Lempel (1977), while GZIP additionally performs a Huffman coding (Huffman 1952) to encode frequent characters with fewer bits. The LZ77 algorithm is a dictionary based compression method (Salomon 1998). Such compression methods store a text as a combination of uncompressed words and tokens that refer to a dictionary of frequent words. The dictionary can be either static (i.e., pre-defined, like a dictionary of common English words) or adaptively updated from the text to compress. LZ77 maintains an adaptive dictionary by using a sliding window approach. The window is divided into two parts, the search buffer and the look-ahead buffer. LZ77 identifies prefixes of the look-ahead buffer that have a matching substring starting in the search buffer. Such matches are encoded by a token denoting to the position in the search buffer and the length of the match. In each step, the window advances by the match length plus one. In consequence, within the scope of the sliding window each additional occurrence of a word occupies only a single token. Naturally, LZ77 achieves the best compression if patterns in the input data occur close together. In the extreme, a repeat limited by the length of the look-ahead buffer can be encoded with a single token. Next, we discuss the application of LZ77-based compression to the different datasets stored in the sample index.

The dataset P^s contains the chromosomal position of each locus covered by at least one read. Hence, it will contain contiguously covered stretches of at least the length of a single read (e.g., 100), where the positions are incremented by one at each entry. Typically these stretches will be much longer, up to covering almost the whole chromosome in the extreme. This knowledge can be used to improve compression. Instead of storing absolute positions, we choose to store the absolute position in the first entry of the dataset and the increment in the subsequent entries, i.e.,

$$\overline{P^s_i} := \begin{cases} P^s_i & \text{if } i = 0, \\ P^s_i - P^s_{i-1} & \text{otherwise.} \end{cases}$$

This results in covered stretches being represented as repetitions of the value 1, preceded by the distance to the last covered stretch (see Figure 3.6). When reading the index, P^s can be restored from $\overline{P^s}$ by calculating the cumulative sum (Section 1.3.2).

[6] `http://www.gzip.org`, visited 11/2014
[7] `http://oldhome.schmorp.de/marc/liblzf.html`, visited 11/2014

$m = 0$	$m = 1$	$m = 2$
-0.439842	-127.3984	-230.3234
-0.439842	-127.5699	-232.9806
-0.004961	-130.8980	-300.9984
-125.58412	-32.58962	-90.41858
-0.019537	-128.8984	-240.8698

Figure 3.7: Structure of the allele frequency likelihoods dataset L^s (likelihoods in logarithmic space). Each column contains the likelihoods for one allele frequency ($m = 0$, $m = 1$, $m = 2$). Within the columns, subsequent values are more similar than between columns, which motivates the column-wise compression of the dataset.

Alternatively, directly storing only start positions and lengths of the covered stretches would be possible. This is similar to a pure run-length encoding (see Salomon 1998) on the dataset P^s and would further increase compression since the detection of repeats is not limited by the size of the look-ahead buffer of the LZ77 algorithm. However it would increase the code complexity for reading the sample index by breaking with the locus-wise storage necessary for the other datasets. Since P^s only requires 14% of the storage space in the uncompressed form, we decide against this.

The dataset L^s of allele frequency likelihoods has a column for each possible alternative allele frequency (e.g., 0, 1, 2 for $\Phi = 2$). While we cannot expect the read depth (i.e., the number of reads covering a particular locus) to be constant over a covered stretch, it will still be similar for loci close to each other. This, together with the expectation that most loci are not hosting a variant (see Section 3.6.2), implies that in each column subsequent values will be at least similar. The likelihoods in the column for an alternative allele frequency of zero will be in a range close to 0 in logarithmic space for most loci because most of the reads will support the reference allele. The likelihoods of the other columns will have similar, larger negative values in logarithmic space, depending on the read depth. HDF5 datasets are compressed in chunks of configurable dimension. Above observations suggest that subsequent likelihoods in a column are more similar than subsequent likelihoods in a row. Therefore, we configure HDF5 to write the columns of L^s into separate chunks. Figure 3.7 illustrates the idea. Optionally, we consider two additional strategies. First, we allow to store the likelihoods in half-precision (i.e., 16 bit) floating point format (IEEE 2008). Here, the minimum representable value is -65504. As the likelihoods are stored in logarithmic scale, this is equivalent to a likelihood of $\exp(-65504)$, which is essentially zero. Half-precision introduces a loss of precision, which is less severe at values close to zero. Hence, in logarithmic space, we will loose some precision only with very small probabilities. Section 3.7.1 shows that the resulting posterior probabilities are almost not affected. Second, we allow to make use of the shuffling filter provided by HDF5. This filter reorders the bytes of a chunk (see above) such that the k-th bytes of each value are stored together. For single precision

G^s	0	1	0	3	2	9	1	2	3	3	0	3	1	0
ref	0	1	0	3	2	1	1	2	3	3	1	3	1	0
$\overline{G^s}$	0	0	0	0	0	8	0	0	0	0	-1	0	0	0

Figure 3.8: Transformation of dataset G^s. The middle row shows the reference genotype r_i. Dashed rectangles depict loci with non-reference genotype. All loci with reference genotype are transformed to zeros.

floating point values (consisting of 4 bytes), this means that we first store the first byte of each value, then the second byte of each value, up to the fourth byte of each value. This can increase the number of similar bytes within the search buffer of the LZ77 algorithm because values that are close together might only differ in their first bytes. Our results show that shuffling does not help though, at least with the used test data (see Section 3.7.1).

The maximum likelihood genotypes in dataset G^s are represented as 8 bit integers j pointing to genotype $g_j \in \Gamma_\Phi$. Since most of the loci can be expected to not host a variant, these integers will mostly point to the reference genotype. The reference genotype is always homozygous (e.g., AA, CC, GG or TT with ploidy $\Phi = 2$). Recalling that genotypes are enumerated such that the homozygous ones come first (see Section 3.3), most integers j will have a value between 0 and 3. This is already advantageous for compression since we can expect recurrent patterns of bases and even larger repetitive regions in the reference genome. We can do even better by storing the genotypes transformed in the following way. Let $g_{r_i} \in \Gamma_\Phi$ be the the reference genotype at the i-th covered locus on the chromosome. For each genotype, we store the difference to the reference genotype as

$$\overline{G_i^s} := G_i^s - r_i.$$

Then, most of the entries will be zero, since most loci will not host a variant and therefore the reference genotype will have the maximum likelihood (see Figure 3.8). The original dataset can be restored by adding r_i again, which can be easily obtained from the reference sequence.

Datasets D^s and R^s are expected to be sparse without transformation. Each row represents a locus, and each column an allele. The two virtual alleles (encoding indels and undetermined read bases; see Section 3.6.1) are rare. Hence, their columns will contain almost entirely zeros. For the other columns we assume all alleles to be equally frequent for illustration. Then, we can expect every fourth value to be greater than zero in each column. We therefore expect the zero-words in all six columns to compress nicely when configuring HDF5 to write each column into separate chunks.

3.6.5 Implementation

Before evaluating the compression performance for the different strategies presented above (see Section 3.7.1), we discuss the implementation of algebraic variant calling in the software ALPACA. All three steps follow a common workflow. For each chromosome, we iterate over intervals of loci. First, the data corresponding to the current interval of loci is loaded (coming from BAM files of mapped reads in case of the sample indexing and from the defined HDF5 index data structures in case of index merging and calling). Then, calculations are performed in parallel over the loci, using OpenCL kernels. Afterwards, we proceed to the next interval. In case of sample indexing and index merging, resulting datasets are written to the HDF5 index data structure once a chromosome has been processed. We delegate parsing of the BAM files and calculation of pileups to the SAMtools package (Li et al. 2009). Thereby, we also obtain recalibrated base qualities which help to avoid artifacts (see Section 3.2). To improve numerical stability, all probabilities are stored and handled in logarithmic space.

Fortunately, the calculations do not impose hardware specific considerations that would exclude certain computational devices, such that all OpenCL kernels work well on both CPU and GPU. For most calculations (e.g., Definition 3.2, Lemma 3.3 to 3.5 and Theorem 3.8) parallelization is even straightforward. In principle, each kernel thread could calculate the probabilities for a single genomic locus. While this might be efficient on a GPU where switching between threads happens with almost no overhead, thread scheduling, instantiation and switching can hurt performance on a CPU. Therefore it is advisable to calculate some neighboring loci together in a loop if the computation per locus is very fast. We delegate this decision to the PyOpenCL (Klöckner et al. 2012) implementation. In the following, we outline the parallelization for the less obvious cases of marginal pileup probability (Lemma 3.6) and controlling the false discovery rate (Equation (3.5)).

Marginal pileup probability By storing the values for $z_{j,k}$ (see Lemma 3.6) in a matrix, the marginal pileup probability $\Pr(\mathfrak{P}_{S,i})$ for a genomic locus i can be calculated by dynamic programming. Figure 3.9 illustrates the idea. Importantly, only the last $\Phi + 1$ columns need to be remembered. Algorithm 3 provides an efficient implementation. Instead of accessing column k for $z_{j,k}$, we access column $k \bmod (\Phi + 1)$ (see line 5). With $z_{j-1,k-k'}$, we have to handle $k - k'$ becoming negative: we access column $|\mathbf{1}_{k \geq k'} \cdot k - k'| \bmod (\Phi + 1)$ (see line 8) and initialize the matrix with zeros. The resulting algorithm is executed in parallel for multiple genomic loci. Since it avoids branching and thus divergent threads, it can be executed efficiently on a GPU.

Controlling FDR To choose the optimal probability threshold α such that the expected FDR does not exceed a given α^* (see Equations (3.4), (3.5) and Corollary 3.10), we first sort the obtained posterior probabilities $\Pr(i \notin \phi | \mathfrak{P}_{S,i})$ for all genomic loci i in the merged index and query expression ϕ in ascending order. Then, we calculate the

z	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$
$j=0$	1	0	0	0	0
$j=1$	1.852e-085	3.025e-013	1.266e-090	0	0
$j=2$	1.845e-085	3.013e-013	2.745e-025	1.148e-102	6.883e-265

Figure 3.9: Exemplary dynamic programming matrix z for calculating the marginal pileup probability of two diploid ($\Phi = 2$) samples $S = \{s_1, s_2\}$. Column $0 \leq k \leq |S|\Phi$ represents the probabilities for k alternative alleles, row $0 \leq j \leq |S|$ represents the considered number of samples. Arrows depict the values that have to be considered to calculate the dashed entry of the matrix: i.e., only three columns of the matrix have to be kept in memory. The pileup of sample s_1 suggests a heterozygous genotype with 10 reference bases and 10 bases supporting the alternative allele. The pileup of sample s_2 suggests the reference genotype. The matrix reflects this scenario: column $k = 1$ contains the largest probability which also dominates the resulting marginal pileup probability $\Pr(\mathfrak{P}_{S,i}) = 3.013\text{e-}16$.

cumulative sum over the probabilities via a parallel prefix scan (Section 1.3.2). The j-th element of the cumulative sum then contains the expected number of false positives when reporting the j most significant putative variants. Since the cumulative sum is monotonically increasing, we can determine the rightmost element j smaller or equal to α^* by binary search. The corresponding posterior probability $\Pr(j \in \phi|\mathfrak{P}_{S,j})$ is the desired threshold α.

Logarithmic probabilities In the implementation, all probabilities are stored and handled in logarithmic space. First, this provides improved numeric stability with small probabilities. Second, products and divisions are faster since they turn into addition and subtraction. In contrast, calculating the sum of probabilities becomes challenging. Consider calculating the sum $p = p_1 + p_2 + \cdots + p_n$ of probabilities which are represented as their logarithmic space. The naive idea of taking the exponential before summing can cause imprecision for small probabilities. Assuming that $p_1 \geq p_2 > \cdots \geq p_n$, Durbin (1998) shows that p can be obtained as

$$\begin{aligned}\log(p_1 + p_2 + \cdots + p_n) &= \log\left(p_1\left(1 + \frac{p_2}{p_1} + \cdots + \frac{p_n}{p_1}\right)\right) \\ &= \log p_1 + \operatorname{log1p}(\exp(\log p_2 - \log p_1) + \cdots + \exp(\log p_n - \log p_1))\end{aligned}$$

with $\operatorname{log1p}(x)$ being the common implementation of $\log(1+x)$ as it is provided by many standard libraries, avoiding a loss of precision for small x. If the difference $\log p_i - \log p_1$

Algorithm 3 Dynamic programming algorithm for calculating the marginal pileup probability $\Pr(\mathfrak{P}_{S,i})$ for a single genomic locus.

Input: a set of samples $S = \{s_1, s_2, \ldots, s_n\}$ with ploidy Φ and pileups $\mathfrak{P}_{S,i} = \{\mathcal{P}_{s_1}, \mathcal{P}_{s_2}, \ldots, \mathcal{P}_{s_n}\}$ at genomic locus i
Output: the marginal pileup probability $\Pr(\mathfrak{P}_{S,i})$

initialize $z_{j,k}$ with $0 \leq j \leq n$, $0 \leq k \leq \Phi + 1$ as a table of zeros
$p \leftarrow 0$
for $m \leftarrow 0, \ldots, |S| \cdot \Phi$ **do**
    $z_{0,0} \leftarrow m \leq \Phi$
    $k \leftarrow m \bmod (\Phi + 1)$
    for $j \leftarrow 1, \ldots, n$ **do**
        for $k' \leftarrow 0, \ldots, \Phi$ **do**
            $k^* \leftarrow |\mathbf{1}_{m \geq k'} \cdot m - k'| \bmod (\Phi + 1)$
            $z_{j,k} \leftarrow z_{j,k} + z_{j-1,k^*} \cdot \Pr(\mathcal{P}_{s_j,i} \mid M_{s_j} = k')$
    $p \leftarrow p + z_{n,k} \Pr(M = m)$
$\Pr(\mathfrak{P}_{S,i}) \leftarrow p$

is small, taking the exponential is not a problem. If the difference becomes larger, this is because p_i is so small compared to p_1 that the sum is dominated by p_1 anyway and a loss of precision when taking the exponential does not matter.

3.7 Results

Here, we evaluate our implementation of the presented algebraic variant calling in the software ALPACA. For this purpose, we use two exome sequencing datasets of the human individuals NA12878[8] and NA12892[9] (a daughter and her mother) from the 1000 genomes project (The 1000 Genomes Project Consortium 2012). The dataset of NA12878 contains 237,938,160, the one of NA12892 contains 262,954,976 Illumina reads of length 76 (see Section 1.2). The datasets come already mapped to a version of the human reference genome (hs37d5) and are free of PCR duplicates. Further, base qualities have been recalibrated and indels have been realigned with GATK (DePristo et al. 2011).

For the analysis, we generate virtual samples of NA12878 and NA12892. We subsample four times 25% of the reads of NA12878 with different random seeds and label the

[8] `ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/technical/working/20120117_ceu_trio_b37_decoy/CEUTrio.HiSeq.WEx.b37_decoy.NA12878.clean.dedup.recal.20120117.bam`, visited 11/2014

[9] `ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/technical/working/20120117_ceu_trio_b37_decoy/CEUTrio.HiSeq.WEx.b37_decoy.NA12892.clean.dedup.recal.20120117.bam`, visited 11/2014

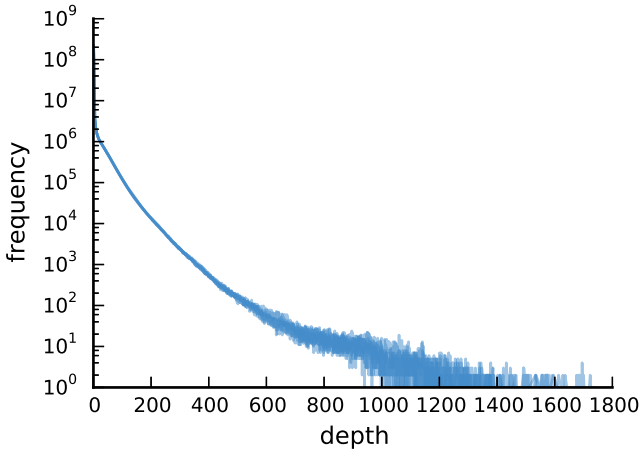

Figure 3.10: Frequencies of read depths of the virtual samples. Each virtual sample is one line. The mean depth ranges from 8 to 10.

resulting datasets A0 to A3. We do the same for NA12892, but only two times, labeling the resulting datasets B0 and B1. The six resulting virtual samples have a similar distribution of read depths over their covered loci, with a mean depth of 8 to 10 and a maximum read depth of about 1800 (see Figure 3.10). Recall that with exome sequencing, only about 1.5% of the genome is covered with reads (see Section 1.2). In turn, the same amount of reads can provide a much higher depth.

All evaluations were performed on a compute server with four 12-core AMD[10] Opteron 6176 CPUs with 512 GB RAM, equipped with a RAID 6 storage. To ensure reproducibility and a comprehensive documentation of the steps, the analyses were composed into a Snakemake workflow[11] (see Chapter 4). Snakemake was further used to systematically benchmark the run times.

3.7.1 Compression

As shown in Section 3.6.4, the sample index generated by ALPACA can become huge if no compression is used. Here, we evaluate the different compression strategies. Even in the uncompressed mode of ALPACA, we always apply the transformations on the datasets P^s and G^s, since they can be expected to not hurt performance compared to the other computations that have to be performed for each locus. The optional strategies are combined to 10 profiles listed in Table 3.4.

We perform sample indexing on the sample A0 with the 10 profiles and measure the size of the resulting index and the run time. Then, we merge (see Section 3.6.2) each version of the A0 index with a sample index of A1 compressed with the same profile

[10] `http://www.amd.com`, visited 12/2014
[11] `http://alpaca.readthedocs.org/analysis.html`, visited 12/2014

Table 3.4: Evaluated compression profiles.

profile	description	profile	description
n	no compression	ls	LZF with shuffling
h	half-precision likelihoods	lsh	LZF with shuffling and half precision
l	LZF compression	gh	GZIP with half precision
g	GZIP compression	gs	GZIP with shuffling
lh	LZF with half precision	gsh	GZIP with shuffling and half precision

and measure the run time. To account for fluctuations, we repeat measurements three times for the latter. Figure 3.11 provides the results.

We measure the efficiency of compression as compression ratio, the ratio between the compressed and uncompressed index size (Salomon 1998): the better the compression, the smaller the ratio (a ratio of 1 indicates no compression). The uncompressed sample index can be expected to have about the same size as the BAM file it was created from if its average read depth does not significantly increase over the amount of bytes needed to store a single locus in the sample index. In fact, it might be even slightly bigger since BAM files are compressed. Indeed, we find that the size of the sample index in its uncompressed form (profile n) of sample A0 is 6.86 GB, whereas the BAM file of the same sample occupies 6.81 GB. Storing half-precision likelihoods (profile h) already reduces the index size to 5.5 GB, obtaining a compression ratio of 0.8. Compression with LZF (profile l) reduces the index size to 2.7 GB (compression ratio 0.39). Compression with GZIP (profile g) provides an index size of 1.9 GB and a compression ratio of 0.28. The best compression is achieved by combining GZIP compression with half precision likelihoods (profile gh), providing an index size of 1.2 GB. This is equivalent to a compression ratio of 0.17, i.e. a size reduction by 83%. LZF compression combined with half precision likelihoods (profile lh) still attains 1.7 GB index size, namely a compression ratio of 0.25 or a size reduction by 75%. Interestingly, shuffling does not provide an improvement to the index size with the used samples. A likely reason is that the likelihoods within the search buffer of the LZ77 algorithm are already similar enough.

Regarding the run time for sample indexing (rightmost column of Figure 3.11), GZIP based compression profiles resulted in slightly higher run times and LZF compression had almost no impact compared to the uncompressed version. While storing half precision likelihoods (profile h) does not affect index merging run time, LZF and GZIP compression cause run time to increase by a factor of 1.4 to 1.5, with GZIP being slightly slower than LZF.

Finally, the loss of precision introduced by storing half precision likelihoods was evaluated. On the two merged indexes generated from (a) the A0 and A1 sample indexes with profile n and (b) the sample indexes with profile h, we performed an algebraic calling of $\phi = V_{\mathrm{A0}} \setminus V_{\mathrm{A1}}$ controlling the FDR at 0.05 (see Section 3.6.3 and 3.5). The resulting set of reported variants was identical for case (a) and (b). Of 1431 called

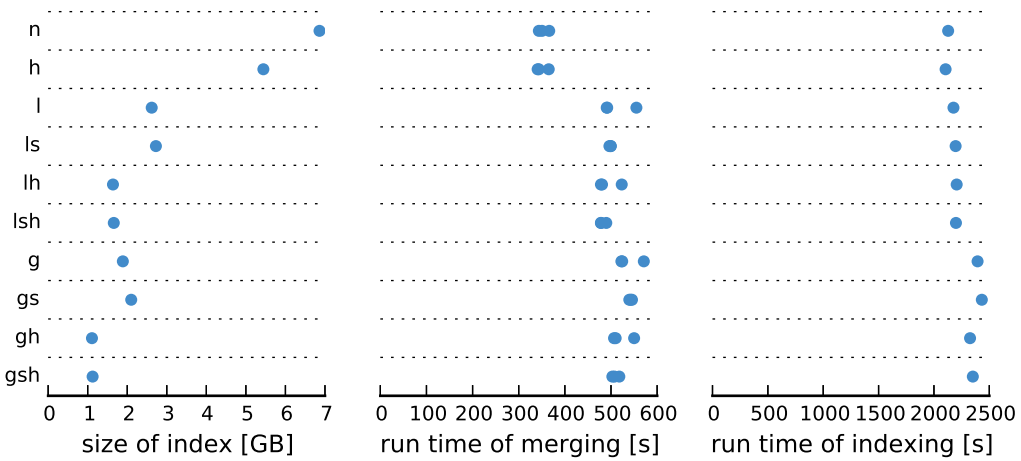

Figure 3.11: Comparison of different compression profiles (see Table 3.4). Each profile occupies a row. For each profile, the size of the resulting sample index (for sample A0), the run time for merging the sample indexes of A0 and A1, and the run time for indexing sample A0 is shown. Run time is measured as seconds of wall clock time; ALPACA was configured to use 12 CPU threads.

variants, 1384 (96%) had the same PHRED-scaled posterior probability. The other 47 variants exhibited a PHRED-scaled probability shifted by only 1. Hence, half precision storage can be used without considerably affecting precision.

In summary, both profile lh and gh appear as good default choices for ALPACA. Where the size of the index is critical, we advocate using profile gh, which imposes only slight speed penalties compared to profile lh while providing the best compression ratio.

3.7.2 Comparison with other variant callers

Here, we compare ALPACA with GATK's HaplotypeCaller 3.2, FreeBayes 0.9.18 and SAMtools 1.1 (see Section 3.2) in terms of true and false positive rate as well as run time performance in different sample filtering scenarios. We omit a comparison with MuTect and Strelka, as these are specialized for calling somatic mutations from pairs of tumor and normal samples. Based on the virtual samples A0 to A3 as well as B0 and B1, we formulate 8 filtering scenarios (in the following referred to as queries Q0 to Q7) in Table 3.5. We evaluate all callers at their default parameters. For ALPACA, this means to control the FDR at 0.05, to use the gh compression profile, and to perform all computations on the CPU.

Table 3.5: Evaluated filtering scenarios formulated as algebraic queries.

name	query	name	query
Q0	$V_{\mathrm{A0}} \setminus V_{\mathrm{A1}}$	Q4	$V_{\mathrm{A0}} \setminus V_{\mathrm{B0}}$
Q1	$V_{\mathrm{A0}} \setminus (V_{\mathrm{A1}} \cup V_{\mathrm{A2}})$	Q5	$V_{\mathrm{A0}} \setminus (V_{\mathrm{B0}} \cup V_{\mathrm{B1}})$
Q2	$(V_{\mathrm{A0}}) \setminus (V_{\mathrm{A1}} \cup V_{\mathrm{A2}} \cup V_{\mathrm{A3}})$	Q6	$(V_{\mathrm{A0}} \cup V_{\mathrm{A1}}) \setminus V_{\mathrm{B0}}$
Q3	$(V_{\mathrm{A0}} \cup V_{\mathrm{A1}}) \setminus (V_{\mathrm{A2}} \cup V_{\mathrm{A3}})$	Q7	$(V_{\mathrm{A0}} \cup V_{\mathrm{A1}}) \setminus (V_{\mathrm{B0}} \cup V_{\mathrm{B1}})$

Since all callers use a Bayesian variant calling model, there should be a similar trade-off between true positive rate and false positive rate, i.e., one caller might show a better false positive rate at the cost of loosing true positive rate and vice versa. We expect ALPACA to provide a better false positive rate at the expense of true positive rate since it controls the FDR. From a global perspective, it should be possible to obtain similar behaviors with all four callers by adjusting the default thresholds. We do not apply additional filtering criteria like strand bias, since they are independent of the used variant calling approach and can always be expected to reduce the false positive rate. In contrast to ALPACA, the other callers GATK, FreeBayes and SAMtools also call indels by default. Here, we only consider single nucleotide variants. Hence, we deactivate indel calling for FreeBayes and SAMtools, and filter out indels called by GATK, as deactivation is not possible there.

ALPACA is the only variant caller to include generic sample based filtering into the calling process. For SAMtools, FreeBayes and GATK, we therefore apply a typical call and filter approach: for any query Q0 to Q7, we first jointly call all samples together with the respective caller; then, we remove all calls from the output that do not comply with the query: e.g., for query Q3, we remove all variant loci that (a) are called in neither A0 nor A1 or (b) are called in either A2 or A3. We do not distinguish between heterozygous or homozygous calls. As all callers comply to the standardized output format VCF (Variant Call Format), we can implement the filtering with the GATK command "SelectVariants".

First, we evaluate the false positive rate (FPR) that can be expected using the different variant callers. FPR is defined as $FP/(FP + TN)$ with FP being the number of false positives and TN being the number of true negatives. Here, false positives are wrongly predicted variants and true negatives are variants that are correctly not predicted. Calculating FP and TN is challenging when dealing with variant calling, because of the following rationale. By applying deep sequencing and using orthogonal validation techniques, a highly confident set of variants for a sample can be determined. An example for this is the set of highly confident variant calls for the NA12878 individual published by Zook et al. (2014). This dataset is also called the *NIST-GIAB gold standard*. However, such variant calls most likely represent only a subset of the true variants of an individual. The values of TN and FP cannot be determined from these, since it is unknown whether a variant is not present in such a set because of missing evidence or

because a locus only hosts the reference allele in reality. One solution to overcome this is to simulate a dataset from an artificial genome where all variants are known. The interpretation of this is limited by the ability of the simulation to mimic a real next generation sequencer with all its artifacts and errors, though. An alternative that can use real data was proposed by Cibulskis et al. (2013), who subsampled two times the same dataset of reads from a sequenced individual. Since both resulting samples come from the same genome, any variant that is called to be in the first but not in the second sample is essentially a false positive, although the real set of variants of the individual can be entirely unknown. Likewise, the true negatives are all loci that are not called in this setup. Noting that our virtual samples A0 to A3 come from the same individual, we see that queries Q0 to Q3 generalize this approach towards the filtering of more than two samples.

Figure 3.12 shows the FPR per variant caller estimated from the four queries Q0 to Q3. For each caller, default parameters were used. We see that ALPACA and SAMtools perform equally good for high depth loci, while ALPACA has advantages at low-depth loci. In query Q1 and Q2, SAMtools is slightly better for depths between 10 and 40. In part, this can be attributed to discrete effects: ALPACA calls in total only 35 and 29 variants for Q1 and Q2, respectively. GATK and FreeBayes provide a worse FPR for all queries at default parameters.

Second, we evaluate the true positive rate (TPR) for the queries Q4 to Q7. TPR (also called sensitivity) is defined as $TP/(TP + FN)$ with TP being the number of true positives and FN being the number of false negatives. All four queries should result in a subset of the true variants of NA12878. Loci which are variant in NA12892 (i.e., virtual sample B0 or B1) should not occur in the resulting calls. Hence we define the set of true single nucleotide variants by the NIST-GIAB gold standard and remove all loci which are presumably variant in NA12892. For the latter, we use a set of calls from deep sequencing published within the Illumina platinum genomes[12]. Here, TP is the number of called variants that occur in the set of true variants, and FN is the number of true variants that are not called. Figure 3.13 shows, that the TPR of ALPACA is in general lower than the TPR of the competitors at low read depth. Beyond a read depth of 15, sensitivities are essentially the same. In other words, as expected, the improved FPR comes at the cost of losing TPR. To adjust TPR to the level of the other callers, the FDR control of ALPACA has to be relaxed. For this, we set the threshold for the minimum variant quality (i.e., the PHRED-scaled posterior query probability; see Theorem 3.8) to $10, 20, 30, \ldots, 70$ instead of controlling the FDR. Figure 3.14 shows the obtainable curves for the different thresholds. We see that lower thresholds allow for a similar TPR as the competitors, while also shifting the FPR to similar curves. Increasing the threshold beyond the one selected by controlling the FDR at 0.05 further reduces the FPR, while also hurting the TPR.

Next, we compare the run time performance of the different callers. The run time performance is measured as wall clock time (in min:sec) on the AMD Opteron system

[12] `ftp://ussd-ftp.illumina.com/NA12892_S1.genome.vcf.gz`, visited 11/2014

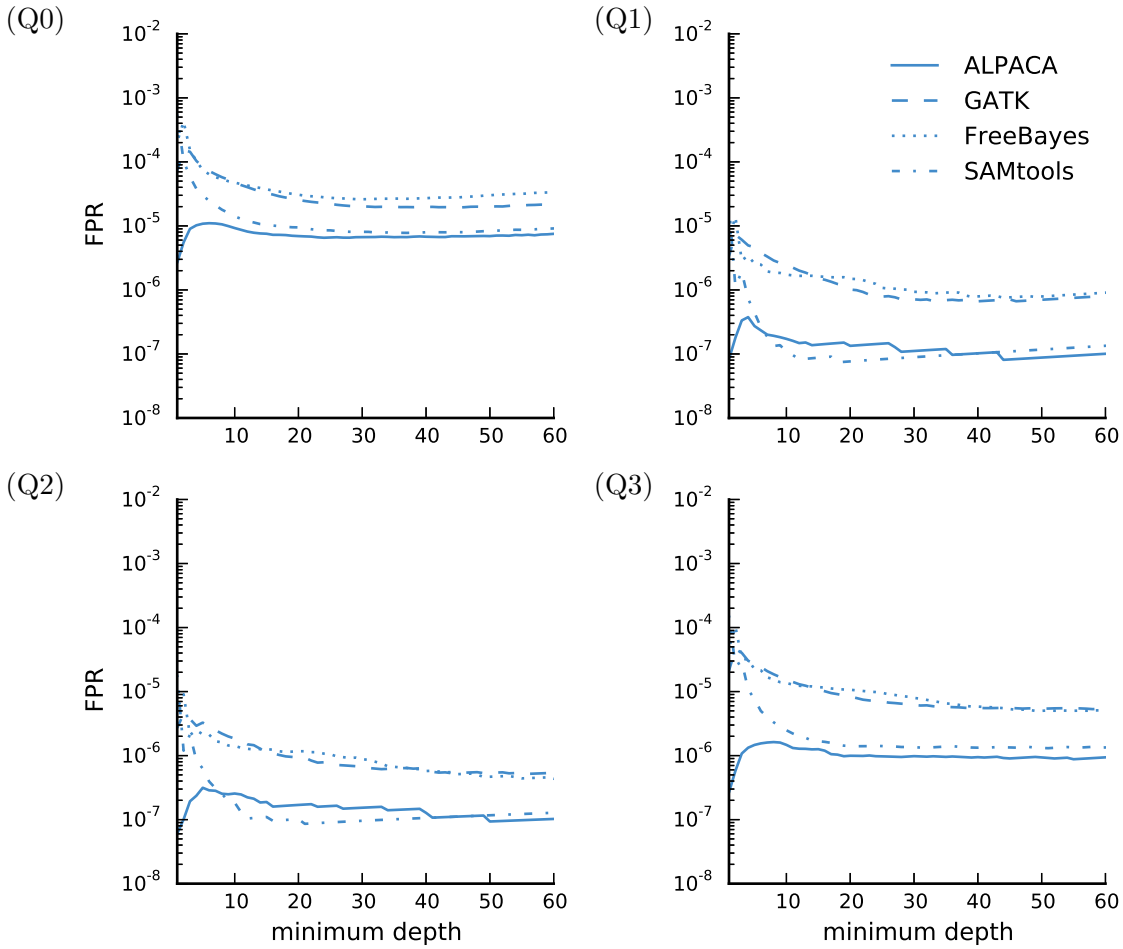

Figure 3.12: False positive rate (FPR) of ALPACA and competitors at default settings for query Q0 to Q3. FPR at minimum depth d is the FPR for all loci with a read depth of at least d. Read depth is calculated as the sum of read depths of all samples involved in the query.

described above, configuring each caller to use 12 CPU cores. For SAMtools, this was achieved using GNU Parallel[13] to process chromosomes in parallel, as the caller itself does not support multi-threading. We separately measure the run times of the different steps involved in the calling of query Q0 to Q7 with all evaluated callers (see Table 3.6). Recall that GATK, FreeBayes and SAMtools do not support calculating a query directly, such that we call the union of the samples in the query and use a filtering step afterwards: since run times are dominated by the calling and filtering might be further optimized, we do not measure the additional time needed for filtering. Both ALPACA and GATK preprocess each sample separately. With ALPACA, sam-

[13] `http://www.gnu.org/software/parallel`, visited 11/2014

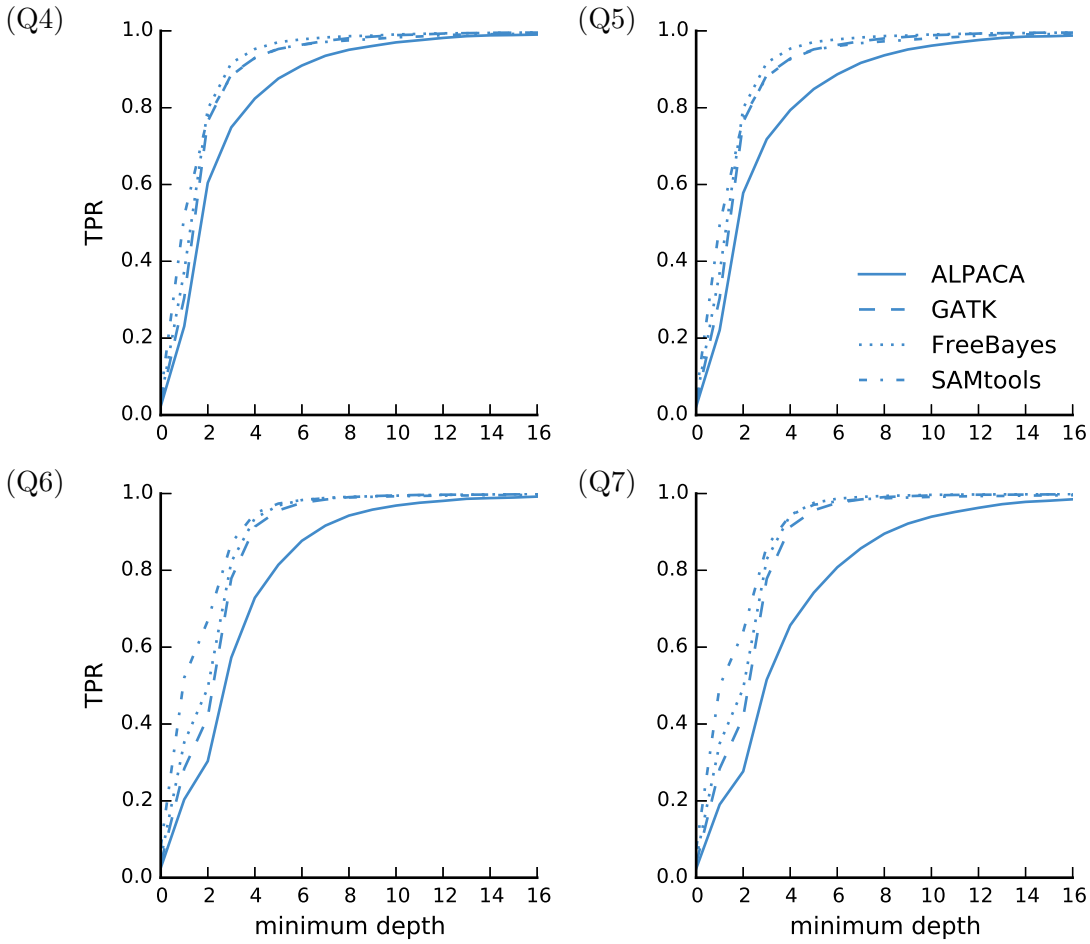

Figure 3.13: Sensitivity of ALPACA and competitors at default settings for query Q4 to Q7. See Figure 3.12 for a description of minimum depth.

ple indexing is performed, generating an HDF5 index (see Section 3.6.1); with GATK, haplotype calling is performed, generating a GVCF file with per-sample information (see Section 3.2). GATK haplotype calling can only use 2 threads, reflected in higher run times. However, the preprocessing of the 6 virtual samples can be done in parallel, saturating the 12 given cores. This results in a total preprocessing run time of 206:07 for ALPACA and 363:51 for GATK. FreeBayes and SAMtools do not allow separation of the steps. Therefore, the run time for calculating the calls of each query entails the whole calling process here. To better assess the benefits of the different approaches, we elaborate on different scenarios in the following.

First, we assume a de novo calling of a single query (e.g., Q0) for a fixed set of samples (e.g., A0 to B1). Here, for each caller, the sum of the run times of all steps matters, such that ALPACA needs 224:48 for sample indexing, generating the merged index

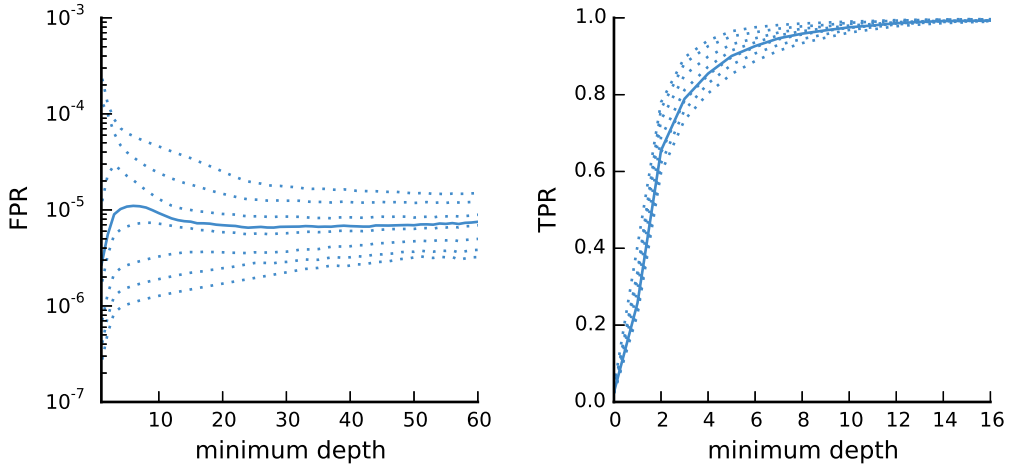

Figure 3.14: FPR and TPR for different variant quality thresholds. The solid line depicts the default behavior, controlling FDR at 0.05. See Figure 3.12 for a description of minimum depth.

for all samples and calling, whereas GATK needs 376:11 for preprocessing and calling. In contrast, FreeBayes and SAMtools need only 12:04 and 10:11, respectively. Even if ALPACA and GATK would preprocess only the samples needed for the particular query, this is much faster. The reason for this is that FreeBayes and SAMtools consider all given samples together from the start: they can skip many irrelevant loci that, e.g., do not host a read showing an alternative allele in any of the samples. GATK and ALPACA cannot skip such loci during preprocessing because they might become relevant later when the samples are considered jointly. Hence, callers like FreeBayes and SAMtools provide superior speed for such static scenarios.

Second, we assume the calling of many queries on a fixed set of samples, say, A0 to B1. With 8 queries (Q0 to Q7), ALPACA needs in total 226:07 and GATK needs 501:33. FreeBayes (143:49) and SAMtools (147:22) are still faster. The advantage of ALPACA becomes apparent when doubling the number of queries: then, since each query takes only a few seconds, we can expect an almost unchanged run time for ALPACA, whereas FreeBayes and SAMtools run times are doubled, exceeding that of ALPACA. The run time of GATK increases by about a third. Heuristically, this advantage of ALPACA compared to FreeBayes and SAMtools can still be minimized by not performing the whole calling procedure for each query, but performing a joined calling for all samples initially and only redo the filtering for each query. Such a strategy is, e.g., implemented with Exomate (see Section 3.2).

Third, we assume that a single query is called on a fixed set of samples, but an additional sample is added (e.g., because the study was extended to another patient) after the initial calls were calculated. First, we note that naturally the run time of FreeBayes and SAMtools increases with the number of samples involved in the query (see Ta-

Table 3.6: Run times of ALPACA and competitors. Dashes indicate that a step is not necessary in that caller. Run times are given as [min:sec] wall clock time using 12 threads, except of the preprocessing with GATK: here, only 2 threads per sample can be used.

task		ALPACA	GATK	FreeBayes	SAMtools
preprocessing	A0	32:14	297:04	-	-
	A1	32:37	302:03	-	-
	A2	32:30	302:13	-	-
	A3	32:07	297:06	-	-
	B0	37:52	360:18	-	-
	B1	38:47	363:51	-	-
merging		18:41	-	-	-
calling	Q0	0:07	12:20	12:04	10:11
	Q1	0:06	16:29	16:25	14:04
	Q2	0:08	21:34	21:13	18:41
	Q3	0:09	21:32	21:13	18:49
	Q4	0:12	12:06	12:52	11:52
	Q5	0:14	15:58	18:36	33:38
	Q6	0:15	16:24	17:51	16:44
	Q7	0:15	21:19	23:35	23:23

ble 3.6). Here, the preprocessing of ALPACA and GATK becomes advantageous, once the set of initial samples is large enough such that FreeBayes and SAMtools need more time for calculating the query than ALPACA or GATK to preprocess (and merge) the additional sample. This scenario is an example for the N+1 problem: the addition of data leads to redundant computations with SAMtools and FreeBayes, whereas the explicit consideration of such cases via the preprocessing allows ALPACA and GATK to be faster.

3.8 Command line interface

ALPACA implements a command line interface to invoke the three steps of the algorithm. For all commands, the number of threads and the OpenCL device to use (e.g., CPU or GPU) can be defined. To limit memory requirements, ALPACA processes the genome (and also the HDF5 indexes) in slices, which can be adjusted by modifying a buffer size. Given mapped reads for sample A0 in BAM format and a reference genome in FASTA format, a sample index can be created with:

```
$ alpaca index reference.fasta A0.bam A0.hdf5
```

Here, various parameters can be adjusted. The expected ploidy of the sample can be set (see Section 3.3), the sample name can be specified for later use in query expressions, and the compression strategy can be configured by specifying one of the profiles shown in Table 3.4. Since ALPACA delegates pileup calculation to SAMtools, this can be configured, too: e.g., base alignment quality calculation (see Section 3.2) can be deactivated and the minimum mapping quality for a read to be considered can be set. Merging indexes for samples A0 and A1 into an optimized index is achieved with:

```
$ alpaca merge A0.hdf5 A1.hdf5 all.hdf5
```

Finally, calling can be performed on the optimized index. ALPACA allows to specify query expressions at the command line by representing the union operator ($\cup$) with a plus sign and the difference operator ($\setminus$) with a minus sign. The variant calls are streamed out in VCF format[14], which is here piped into a file:

```
$ alpaca call all.hdf5 "A0-A1" > calls.vcf
```

To assess the biological importance of a variant, it is useful to annotate it with additional information like the gene it may be contained in, its effect on a protein that is encoded by the gene or whether it is already known and maybe associated to some disease (see Section 3.2). ALPACA can annotate a VCF file with such information, using the VEP web service (McLaren et al. 2010). Since the VCF format is rather technical, ALPACA can compose a human readable HTML file summarizing the calls. We can combine the two commands using Unix pipes:

```
$ alpaca annotate < calls.vcf | alpaca show > calls.html
```

The HTML file lists the calls with their annotations in a table that can be sorted and filtered. For each variant, links to external resources and details about allele counts and effects are provided. An example is available online[15].

3.9 Discussion

This chapter provides a novel solution to variant calling that entails filtering the variants of one or more samples against that of another group of samples. We first identified three major challenges of state of the art call and filter approaches: the FDR problem, the insufficient evidence problem, and the N+1 problem. None of the established approaches is able to solve them all. As a solution, we present an algebraic method for variant calling and its implementation in form of the variant caller ALPACA. It is the first variant caller to solve the FDR problem for arbitrary filtering scenarios. The key idea is to integrate the filtering process into the calling process: this allows to calculate the posterior probability for a locus to occur in an unknown set of true variant loci that is described by an expression over an algebra of variant loci. Any evidence for a variant

[14] http://samtools.github.io/hts-specs/VCFv4.2.pdf, visited 11/2014
[15] http://alpaca.readthedocs.org/_downloads/show_example.html, visited 12/2014

will affect the resulting probability, thereby solving the insufficient evidence problem. Since the posterior probabilities reflect the filtering, they can be used to control the FDR. The N+1 problem is solved by preprocessing each sample into a sample index, avoiding the need of redundant computations upon the addition of new samples. In this sense, ALPACA provides scalability especially for large, growing studies.

For the sample indexes, different compression strategies were developed, resulting in a size reduction of up to 83% without major performance regressions. ALPACA allows to obtain true and false positive rates (TPR and FPR) similar to other variant callers. At default parameters, it provides an improved FPR at the expense of losing TPR at low read depths. In contrast to the other variant callers, controlling the FDR with ALPACA allows to calibrate the trade-off between TPR and FPR intuitively. ALPACA is faster than GATK, while the run time performance of SAMtools and FreeBayes is superior for static scenarios where no samples are added subsequently and a single query is calculated. In more complex and larger studies with subsequently added samples (the N+1 problem) or many different queries, ALPACA becomes faster than its competitors, since the index data structure allows filtering queries to be calculated in seconds and the addition of a new sample avoids redundant computations.

In its current form, query expressions allow to express union ($\cup$) and difference ($\setminus$) of variant loci. In Section 3.4 we argue that the intersection ($\cap$) would be of limited use because it does not properly handle outlier samples. Instead, we will support a relaxed version of the intersection in the future. We extend the algebra of variant loci $\mathcal{A}_S$ with a family of operators $\bigotimes^k$ and write $\phi = \bigotimes^k_{s \in S'} V_s$ to express the k-relaxed intersection of the set of samples $S' \subseteq S$. The parameter k denotes the minimum number of samples $s \in S'$ in which a locus has to be variant in order to occur in the intersection: e.g., for a set of samples $S' = \{s_1, s_2, \ldots, s_{10}\}$, the 8-relaxed intersection $\bigotimes^8_{s \in S'} V_s$ yields all loci being variant in at least 80% of the samples. The expression $\bigotimes^{|S'|}_{s \in S'} V_s$ is equivalent to the ordinary intersection $\bigcap_{s \in S'} V_s$. Further, the 1-relaxed intersection $\bigotimes^1_{s \in S'} V_s$ is equal to the union $\cup_{s \in S'} V_s$. The k-relaxed intersection can be computed efficiently via dynamic programming. For this, at any locus i we can define $y_{j,k}$ as the posterior probability to have k non-reference genotypes in the first j samples in S'. Then, it holds

$$y_{j,k} = y_{j-1,k-1} \Pr(M > 0 \mid \mathfrak{P}_{\{s_j\},i}) + y_{j-1,k} \Pr(M = 0 \mid \mathfrak{P}_{\{s_j\},i})$$

with $y_{0,0} = 1$ and $y_{j,k} = 0$ for $0 > k > |S'|$ or $j \leq 0$ and $k \neq 0$. Finally, the posterior probability for $\phi = \bigotimes^k_{s \in S'} V_s$ can be calculated as

$$\Pr(i \notin \phi \mid \mathfrak{P}_{S,i}) = \sum_{k'=0}^{k-1} y_{|S'|,k'}.$$

Controlling the FDR with ALPACA provides an effective way of minimizing false positives. The current way (Corollary 3.10) maximizes the number of predicted variants while ensuring that the FDR does not exceed the given threshold. Müller, Parmigiani,

and Rice (2006) see this as one case in a class of loss functions that can be used to optimize the prediction for various targets. Future work on ALPACA will entail the evaluation of various alternative loss functions. It is possible to incorporate covariates like the read depth or auxiliary scores (Section 3.1) into the loss function. This can help to optimize the selection of individual variants. Imagine that two variants have the same weak posterior query probability, and the exclusion of one of them would suffice to reach the desired FDR. A loss function integrating, e.g., the strand bias as auxiliary score might help to decide which variant to exclude here. Rather than globally removing all variants based on a certain covariate (e.g., an auxiliary score measuring only a single aspect), this approach could make use of the covariate only in cases where the decision based on the posterior probabilities is not clear.

The calculation of the posterior query probability in Theorem 3.8 assumes independence between the samples. This is a common simplification in variant calling (DePristo et al. 2011; Li et al. 2009; Garrison and Marth 2012), especially since the true dependencies are often unknown. In the future, dependencies could be considered in the probability model, e.g., by adjusting the prior probabilities.

The bayesian variant calling model of ALPACA assumes that the number of copies sequenced at a genomic locus is determined by the given ploidy (i.e., the number of chromosomes). However, copy number variations (CNVs) can violate this assumption. CNVs result from structural alterations of the genome, that duplicate parts of a chromosome or even a whole chromosome. Further, especially tumor samples can sometimes be of poor quality, being contaminated with, e.g., healthy tissue. In such a case, modeling a genotype as a pair of alleles (e.g., AA) might be misleading. Future work could incorporate such prior information about CNVs or contamination into the model: e.g., if the number of expected copies differs from ploidy at a locus the genotype model can consider these as additional alleles.

4 A scalable text-based workflow system

Bioinformatics analyses typically involve the application of various command line tools that convert raw data (e.g., from a sequencing experiment) into results (e.g., putative mutations) via intermediate steps. The tools range from long-running optimized algorithms performing computationally expensive tasks like read mapping or variant calling, to filter utilities or format conversions and custom scripts for plotting or statistical tests. Usually, every step is configurable, and a single analysis might need multiple adjustments until the final results can be obtained. Often, the initial input data is subject to change, e.g., because new samples become available.

Such analyses can be seen as workflows. Aalst, Hee, and Mylopoulos (2002) define a workflow as a collection of tasks (also called *jobs*) needed to create a product. The tasks are executed as a whole by a resource, e.g., a machine. In the context of that theory, a process defines the order (induced by dependencies between the tasks) and necessity of task execution. In addition, a process can be enriched by conditions which determine subsequent tasks based on intermediate results. In this notion, a workflow management system determines the process needed to create a product and manages the distribution of tasks to the available resources until the product is created.

Workflow management is crucial for bioinformatics analyses: Apart from the automation of the task execution, it helps to ensure reproducibility of the obtained results, and documents the used data, methods and parameters. Here, we present an approach to workflow management, introducing a text-based workflow definition language and a flexible execution environment.

4.1 Introduction

Solutions to the workflow management problem that are of practical relevance in bioinformatics can be sorted into three categories. The first and most general solution is to implement a workflow in a scripting language. Second, a text-based domain specific language can be used to specify and execute a workflow. Among others, Ruffus (Goodstadt 2010), BPipe (Sadedin, Pope, and Oshlack 2012), Pwrake (Tanaka and Tatebe 2010) and GXP Make (Taura et al. 2013) are examples of such systems. A common ancestor of many text-based approaches is GNU Make[1], which was originally developed for the compilation of source code. The third way is to define the workflow by connecting the

[1] `https://www.gnu.org/software/make`, visited 04/2014

tasks graphically, i.e., drawing a graph which visualizes the process. Examples for this kind of workflow management are KNIME (Berthold et al. 2007), Taverna (Oinn et al. 2004), PegaSys (Shah et al. 2004), Biopipe (Hoon et al. 2003), GeneProf (Halbritter, Vaidya, and Tomlinson 2011) and Galaxy (Goecks, Nekrutenko, and Taylor 2010).

Using plain scripting languages often involves writing boilerplate code for common tasks that are already solved properly by workflow systems, e.g., parallelization and support for resuming a previously paused execution. The choice between graphical and text-based workflow systems is, apart from being a matter of taste, a question of the desired development model. On the one hand, a workflow defined in a human readable text file can be easily developed collaboratively via a version control system like Git[2] or Subversion[3]. Further, it is possible to quickly modify the workflow via a secure shell terminal on a server or cluster. On the other hand, a graphical workflow definition can be more intuitive in the sense that the definition already entails the graphical representation of the dependencies between the tasks.

Workflow definition languages can be divided into explicit and implicit approaches. The former (e.g., Ruffus, BPipe, and the graphical workflow systems) require the explicit definition of dependencies between the steps. The latter (e.g., Pwrake and GXP Make), whose common ancestor is GNU Make, allow the definition of generic rules with wildcards that describe how to obtain a single output file from one or several input files. The dependencies between the rules are inferred automatically by the system. At the expense of some control, this can be advantageous as less code is needed for the workflow definition, especially if certain steps (i.e., rules) occur repeatedly in an analysis.

Here, we present the text-based workflow system Snakemake which generalizes the rule-based implicit paradigm to multiple named wildcards and output files. Snakemake sets itself apart from existing text-based workflow systems (see Section 4.2) in the following way. Hooking into the Python interpreter, Snakemake offers a definition language that is an extension of the Python programming language[4] with syntax to define rules and workflow specific properties. This allows to combine the flexibility of a plain scripting language with a pythonic workflow definition. The Python language is known to be concise yet readable and can appear almost like pseudo-code. The syntactic extensions provided by Snakemake try to maintain this property for the definition of the workflow. Further, Snakemake supports implicit parallelization that can be constrained by priorities, provided cores and customizable resources and it provides a generic support for distributed computing (e.g., cluster or batch systems). Thereby, a Snakemake workflow scales without modification from single core workstations and multi-core servers to cluster or batch systems. Snakemake is available as open source software under the MIT license (see Section A.1).

This chapter is based on previously published work (Köster and Rahmann 2012a; Köster

[2] `http://git-scm.com`, visited 04/2014
[3] `http://subversion.apache.org`, visited 04/2014
[4] `http://www.python.org`, visited 04/2014

and Rahmann 2012b). In Section 4.2, other workflow systems are summarized. Next, the workflow definition language of Snakemake and the implementation of the parser is described (Section 4.3). Then, resolution of dependencies (Section 4.4), scheduling of compute jobs (Section 4.5) and support for distributed computing (Section 4.6) is presented. This is followed by a summary of data provenance support provided by Snakemake (Section 4.7). Finally, we exemplify how the other software presented in this work (PEANUT and ALPACA) can be combined into a Snakemake workflow that provides a complete and flexible parallel solution to a major task in the analysis of next-generation sequencing data (Section 4.8). The chapter is closed by a discussion (Section 4.9).

4.2 Related Work

We briefly summarize representative implementations of the different approaches for describing a workflow.

Pwrake Pwrake (Tanaka and Tatebe 2010) builds on top of Rake[5], which is an implementation of GNU Make in the Ruby language[6]. Rake (and thereby also Pwrake) allows to define rules (here called tasks) using special Ruby methods. Pwrake adds implicit parallelization to Rake, and allows to execute tasks locally or in a distributed environment. For the latter, in contrast to Snakemake, Pwrake does not support resource management systems (see Section 4.6) but relies on secure shell (SSH) access to the computing nodes of a distributed environment.

GXP Make GXP Make (Taura et al. 2013) uses GNU Make for workflow definition and extends it with support for distributed computing. For this, it allows to run an execution environment (the GXP daemon) on a collection of distributed machines. When invoking a GXP Make workflow, the jobs determined by GNU Make are distributed to these daemons. Thereby, the daemons can be instantiated using SSH or a resource management system. GXP Make starts one daemon per machine, which keeps running until the workflow is finished. On a resource management system, a submitted command usually has to wait in a queue before it is distributed and executed on a compute node. Since GXP Make only needs to submit the daemons once, jobs can be executed almost immediately. The downside is that an idle daemon still occupies a compute node. Since the workflow is defined in the language of GNU Make, tasks are limited to shell invocations and scripts.

[5] `https://github.com/jimweirich/rake`, visited 06/2014
[6] `https://www.ruby-lang.org`, visited 06/2014

Ruffus Rather than introducing additional syntax like Snakemake, Ruffus (Goodstadt 2010) allows to specify a workflow in pure Python. It follows an explicit approach of workflow definition, i.e., the dependencies between steps are part of the implemented description. Tasks are defined by Python functions which can be decorated to define parameters and dependencies. They can be parallelized automatically using the internal multiprocessing of Python. Further, Ruffus allows to execute jobs in a distributed environment via the DRMAA API (see Section 4.6).

BPipe BPipe (Sadedin, Pope, and Oshlack 2012) provides a domain specific language that allows to define a workflow via tasks that are shell commands. The tasks are composed into a workflow using mathematical operators. Thereby, parallelism has to be specified explicitly. BPipe allows to execute tasks in various distributed environments (see Section 4.6).

KNIME KNIME (Berthold et al. 2007) is a commercial, graphical workflow management system based on the Eclipse platform[7]. The main software is available for free. A workflow can be defined by graphically arranging nodes. It comes with predefined nodes that support, e.g., input and output, database functions and statistical tests. Custom nodes can be implemented in JAVA[8] and published in an online repository: e.g., there already exists a collection of nodes for the analysis of next-generation sequencing data[9]. Via a non-free extension, KNIME supports the execution of workflows in a distributed environment.

Galaxy A popular workflow management platform targeted towards the analysis of genomic data is Galaxy (Goecks, Nekrutenko, and Taylor 2010). Galaxy is a server application that allows to define workflows via a web interface. One goal of Galaxy is to provide an environment to perform analyses without programming skills. It offers a wide range of predefined tools that can be connected to a workflow for analyzing genomic data. Custom tools can be added by defining their invocation in an XML document. Galaxy can execute steps in distributed environments. With "Galaxy pages", it supports the documentation of workflows by composing HTML pages in a web-based editor. Together with a created workflow, these can be published for other researchers within the framework.

4.3 Workflow definition language

A file containing a Snakemake workflow definition is called *snakefile*. The Snakemake language extends the Python language, adding syntactic structures for rule definition

[7] `http://eclipse.org`, visited 11/2014
[8] `http://java.com`, visited 11/2014
[9] `http://tech.knime.org/community/next-generationsequencing`, visited 11/2014

and additional controls. All added syntactic structures begin with a keyword followed by a code block that is either in the same line or indented and consisting of multiple lines. The resulting syntax resembles that of original Python constructs.

We first describe the mandatory language elements and elaborate on more special keywords later in this chapter. A formal description of the language follows in Section 4.3.6. The most important construct of the Snakemake language is a rule. A rule describes how to obtain a set of output files from a set of input files. Typical rules have a name, any number of input and output files given as comma separated lists of strings, and either a shell command or python code that creates the output files from the input files. The rule

```
rule alpaca_index:
    input:  "reference.fasta", "reads.bam"
    output: "index.hdf5"
    shell:  "alpaca index {input} {output}"
```

describes how to obtain the output file `"index.hdf5"` from two given input files via the command following the **`shell`** keyword. Here, the rule creates an ALPACA sample index from the reads in the BAM file and the reference genome sequence (see Chapter 3). Inside the shell command, braces are used to include variables into the string, using the Python format mini-language[10]. Apart from local and global variables, this allows especially to include parameters of the rule into the shell command. Here, `"{input}"` and `"{output}"` are replaced by a space-separated list of the respective files. The resulting shell command is the following:

```
$ alpaca index reference.fasta reads.bam index.hdf5
```

A rule can be generalized by including named wildcards into the files, e.g.,

```
rule alpaca_index:
    input:  "reference.fasta", "{sample}.bam"
    output: "{sample}.hdf5"
    shell:  "alpaca index {input} {output}"
```

Here, the wildcard `"{sample}"` in the output file can be replaced by any string to let the rule generate a certain output. We call this replacement the *value* of a wildcard. The wildcard value is propagated to the input files, such that here, to generate the index for sample A0, say `"A0.hdf5"`, the resulting wildcard value `"A0"` would lead to the input file `"A0.bam"`, which contains the mapped reads (see Section 1.2) of the sample.

When a workflow is executed, Snakemake tries to generate given *target* files. Target files can be specified via the command line, e.g.,

```
$ snakemake A0.hdf5 A1.hdf5
```

[10] `https://docs.python.org/3/library/string.html#formatspec`, visited 04/2014

will try to generate the ALPACA index for samples A0 and A1. To generate the target files, Snakemake applies the rules given in the snakefile. The application of a rule to generate a set of output files is called *job*. For each input file of a job, Snakemake determines rules that can be applied to generate it, e.g., by replacing wildcards. This yields a directed acyclic graph of jobs where the edges represent dependencies (see Section 4.4). When no target files are provided via the command line interface, Snakemake tries to apply the first rule in the snakefile. Hence, a common pattern for defining default target files is to provide a rule *all* at the top of the snakefile which collects these, e.g.:

```
rule all:
    input: "A0.hdf5", "A1.hdf5", "A2.hdf5"
```

Such a rule is called a *target rule*. In general, hardcoding data dependent values (like sample names) into a snakefile should be avoided. Hence, it is possible to store these in a config file in JSON[11] format, e.g.,

```
{
    "samples": ["A0", "A1", "A2"],
    "reference": "path/to/reference.fasta"
}
```

and to load this file from the workflow by writing:

```
configfile: "path/to/config.json"
```

The contents of the config file are available as Python dictionary under the global variable **config**. In summary, we can compose above features into a first complete snakefile:

```
configfile: "path/to/config.json"

rule all:
    input: expand("{sample}.hdf5", sample=config["samples"])

rule alpaca_index:
    input:  config["reference"], "{sample}.bam"
    output: "{sample}.hdf5"
    shell:  "alpaca index {input} {output}"
```

The function **expand** is a helper to compose a list of strings from a pattern and given values for the wildcards in the pattern. Here, it yields a list

```
["A0.hdf5", "A1.hdf5", "A2.hdf5"]
```

which ends up being the set of desired targets that is assigned to the input of our target rule `all`. By issueing

```
$ snakemake --dag | dot -T pdf > dag.pdf
```

[11] `http://www.json.org`, visited 11/2014

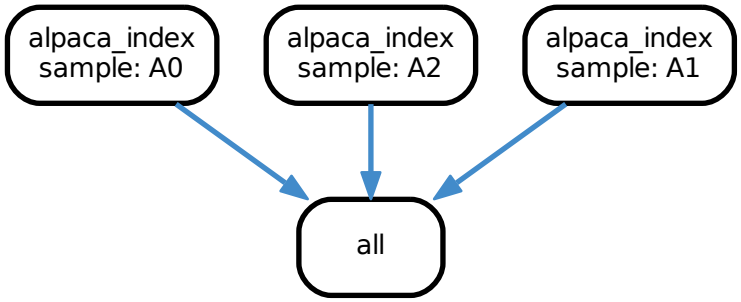

Figure 4.1: DAG of jobs for example workflow. Each node represents a job that will be scheduled for execution, annotated with the corresponding wildcard values.

Snakemake can be used in combination with the Graphviz `dot` command (Gansner and North 2000) to visualize the DAG of jobs that would be executed by this workflow (see Figure 4.1). The DAG shows that Snakemake would spawn three jobs to create each of the index files which serve as input for the job created from the target rule. The latter just finishes without execution since the rule does not specify a command. In the following sections, we describe several additional features of the Snakemake language.

4.3.1 Defining resource usage

The scheduler of Snakemake considers the resource usage of jobs and can be constrained to not exceed given resource limits (see Section 4.5). The most obvious resource is the number of threads a job spawned from a rule will use. This can be specified by the **`threads`** keyword:

```
rule alpaca_index:
    input:   config["reference"], "{sample}.bam"
    output:  "{sample}.hdf5"
    threads: 8
    shell:
        "alpaca --threads {threads} index {input} {output}"
```

The value assigned to **`threads`** can be accessed from within the shell command, similar to the input and output files. Arbitrary resources can be defined with the **`resources`** keyword:

```
rule alpaca_index:
    input:     config["reference"], "{sample}.bam"
    output:    "{sample}.hdf5"
    threads:   8
    resources: gpu=1
    shell:
        "alpaca --dev gpu -t {threads} index {input} {output}"
```

Here, we define that the rule needs one unit of the resource GPU. Together, both features allow to flexibly scale the workflow upon invocation, without changing the snakefile. For example, by invoking

```
$ snakemake --cores 16 --resources gpu=1
```

we tell the Snakemake scheduler to use up to 16 CPU cores and provide one unit of the resource GPU. Since each job of the rule `alpaca_index` needs one GPU, only one of them can run at a time, although the scheduler was given enough cores to run two of the jobs (since each job also needs 8 threads). The threads given to a rule are interpreted as a maximum. Hence, specifying

```
$ snakemake --cores 4 --resources gpu=1
```

will execute each job of the rule `alpaca_index` with only 4 threads.

4.3.2 Temporary and protected files

With Snakemake, almost every step of the workflow generates output files. Sometimes, such intermediate results can be discarded once not needed any more. Other files might be expensive to compute, such that their protection from accidental deletion is desirable. Therefore, output files can be marked as **`temp`** or **`protected`**, e.g.:

```
rule alpaca_index:
    input:  config["reference"], "{sample}.bam"
    output: protected("{sample}.hdf5")
    shell:  "alpaca index {input} {output}"
```

Files that are marked as **`protected`** are write protected in the file system, such that neither Snakemake nor other processes can modify or overwrite them accidentally. A file marked as **`temp`** is deleted by Snakemake once no job needing the file remains to be executed.

4.3.3 Additional functionality

The **`input`** and **`output`** keywords accept single strings or comma separated lists of strings. Apart from that, they also accept the output of arbitrary Python code if it returns a string or a list of strings. Finally, they can be provided with a pointer to a function, e.g.:

```
rule alpaca_index:
    input:  "reference.fasta", get_sample_path
    output: "{sample}.hdf5"
    shell:  "alpaca index {input} {output}"
```

Such a function has to accept a single parameter that gives access to the wildcard values, e.g.:

```
def get_sample_path(wildcards):
    return config["bam_files"][wildcards.sample]
```

Sometimes, it is necessary to access input or output files explicitly. For this purpose, it is allowed to define names for them, e.g.:

```
rule alpaca_index:
    input:  ref="reference.fasta", reads="{sample}.bam"
    output: "{sample}.hdf5"
    shell:  "alpaca index {input.ref} {input.reads} {output}"
```

Snakemake offers the **params** keyword that allows to separate parameters from the shell command, using the same syntax as **input** and **output**:

```
rule alpaca_index:
    input:  config["reference"], "{sample}.bam"
    output: "{sample}.hdf5"
    params: name=get_sample_name
    shell:
        "alpaca index "
        "--sample-name {params.name} "
        "{input} {output}"
```

Here, a function `get_sample_name` retrieves the name of the sample from some configuration. The resulting value is accessed from within the shell command. A rule can be further annotated with various additional keywords, e.g.:

```
rule alpaca_index:
    input:     config["reference"], "{sample}.bam"
    output:    "{sample}.hdf5"
    version:   shell("alpaca --version")
    log:       "logs/{sample}.log"
    benchmark: "benchmarks/{sample}.json"
    shell:     "alpaca index {input} {output} 2> {log}"
```

The **version** keyword allows to provide Snakemake with the version of the software used in the rule. The version can be retrieved directly from the software using the **shell** function provided by Snakemake. In a data provenance summary, Snakemake allows to view the version an output file was created with along with other information (see Section 4.7).

The **log** keyword allows to define a log file (which can be also generalized using wildcards). This file can be referenced in the shell command. Here, we pipe the output of ALPACA into the log file.

Snakemake can be used for benchmarking by specifying the **benchmark** keyword. It allows to define a JSON file that shall contain information about the run time of the job. Per default, the run time of a single execution of the job is stored in the JSON

file. If specified, Snakemake can execute a job multiple times to generate more reliable measurements.

4.3.4 Python rules

Sometimes, the shell is not convenient to describe a workflow step, e.g., when plotting some results. With the **run** keyword, Snakemake allows to define arbitrary Python code for execution by a rule:

```
rule plot_index_size:
    input:  expand("{sample}.hdf5", sample=config["samples"])
    output: plot="plots/index_size.pdf"
    run:
        plt.figure()
        sizes = [os.path.getsize(f) for f in input]
        plt.hist(sizes)
        plt.savefig(output.plot)
```

Within the run block, all keywords previously defined in the rule are accessible, similar to shell rules. Here we access input and output files and plot a histogram of index sizes via matplotlib (Hunter 2007).

4.3.5 Modularization

For modularization, it is useful to separate a large workflow into building blocks. Snakemake allows to include another snakefile into the current one, e.g.,

```
include: "path/to/other/Snakefile"
```

such that all code contained in the other snakefile is interpreted in the same namespace as the current snakefile. Instead of a path, it is also possible to use a URL pointing to a network resource (e.g., a snakefile on a server or in a source code repository).

A more separated way of modularization can be performed using sub-workflows. The sub-workflow definition

```
subworkflow other:
    workdir:   "path/to/other"
    snakefile: "path/to/other/Snakefile"
```

points to another Snakemake workflow residing in the directory `"path/to/other"` with the snakefile `"path/to/other/Snakefile"`. Inside the current workflow, one can refer to the output files of a sub-workflow by its name, e.g.,

```
rule alpaca_index:
    input:  config["reference"], other("{sample}.bam")
    output: "{sample}.hdf5"
    shell:  "alpaca index {input} {output}"
```

assuming that the other workflow generates the BAM file for the sample (e.g., by performing read mapping; see Chapter 2). Here, `other("{sample}.bam")` is replaced by the appropriate path pointing to the sub-workflow. When a wildcard value is determined for the rule, the corresponding input file is recorded as a target file for the sub-workflow, e.g., `"path/to/other/A0.bam"`. Before the current workflow is executed, Snakemake executes the sub-workflow, thereby ensuring that up-to-date versions of all recorded target files are generated. With sub-workflows, a hierarchical modularization structure is enforced, i.e., when the current workflow depends on the sub-workflow, the sub-workflow may neither directly nor indirectly depend on the current workflow.

4.3.6 Parsing

The syntax of the Snakemake language is defined with the extended Backus-Naur-Form (EBNF) in Listing 4.1. The undefined non-terminals refer to the corresponding constructs of the Python language[12]: `statement` is any Python statement, `NEWLINE` and `INDENT` are the corresponding characters for a line break and indentation. Indentation is always relative to the line before. Further, `stringliteral` is a string, `integer` is a Python integer, and `identifier` is a python compatible identifier, i.e., a word containing any alphanumeric character or the underscore, starting with an alphabetic character. The EBNF gives rise to three groups of keywords that extend Python in the Snakemake language: the *top-level keywords* `include`, `workdir`, `configfile`, `ruleorder`, `localrules`, `subworkflow` and `rule`; the *rule keywords* `input`, `output`, `params`, `message`, `log`, `threads`, `resources`, `version`, `run` and `shell`; and finally the *subworflow keywords* `workdir` and `snakefile`.

Typically, a parser for a domain specific language is generated from a complete grammar. The Snakemake language is an extension of Python, though, adding only few statements. A full parser would have to be maintained for all Python versions Snakemake is compatible with. To avoid this, we use the following strategy to parse a snakefile. First, the snakefile is tokenized with the Python tokenizer. Second, the stream of tokens obtained from the tokenizer (containing the non-Python keywords of the Snakemake language) is translated into plain Python tokens using a collection of Mealy automata (Mealy 1955). A Mealy automaton $\mathcal{A} = (S, s_0, F, \Sigma, \Omega, \delta, \lambda)$ with a set of states S, a start state $s_0 \in S$, a set of accepting states F, an input alphabet Σ, an output alphabet Ω, a transition function $\delta : S \times \Sigma \to S$ and an output function $\lambda : S \times \Sigma \to \Omega$ can be seen as a deterministic finite automaton that additionally emits a literal of the output alphabet

[12] `https://docs.python.org/3/reference`, visited 04/2014

Listing 4.1: Extended Backus-Naur-Form of the Snakemake language.

```
snakemake          = (statement | rule | include | workdir | ruleorder
                     | localrules | subworkflow)*.
include            = "include:" param_string.
workdir            = "workdir:" param_string.
configfile         = "configfile:" param_string.
ruleorder          = "ruleorder:"  identifier > identifier (> identifier)*.
localrules         = "localrules:" identifier (identifier)*.
subworkflow        = "subworkflow" identifier ":"
                     NEWLINE INDENT subworkflow_params.
subworkflow_params = ["workdir:" param_string]
                     ["snakefile:" param_string].
rule               = "rule" (identifier | "") ":"
                     NEWLINE INDENT rule_params.
rule_params        = [input] [output] [params] [message] [threads]
                     [log] [resources] [version] [(run | shell)].
input              = "input:" param_list.
output             = "output:" param_list.
params             = "params:" param_list.
message            = "message:" param_string.
log                = "log:" param_string.
threads            = "threads:" param_integer.
resources          = "resources:" param_kwlist.
version            = "version:" param_statement.
run                = "run:" NEWLINE INDENT statement NEWLINE DEDENT.
shell              = "shell:" param_string.
param_string       = (NEWLINE INDENT stringliteral NEWLINE DEDENT) |
                     (stringliteral NEWLINE).
param_integer      = (NEWLINE INDENT integer NEWLINE DEDENT) |
                     (integer NEWLINE).
param_statement    = (NEWLINE INDENT statement NEWLINE DEDENT) |
                     (statement NEWLINE).
param_list         = (NEWLINE INDENT statement_list NEWLINE DEDENT) |
                     (statement_list NEWLINE).
param_kwlist       = (NEWLINE INDENT statement_kwlist NEWLINE DEDENT) |
                     (statement_kwlist NEWLINE).
statement_list     = statement ("," statement)* [, statement_kwlist].
statement_kwlist   = identifier "=" statement
                     ("," identifier "=" statement)*.
```

upon each transition. Finally, the resulting tokens are interpreted with the Python interpreter. This strategy allows the Python language to evolve almost independently of the Snakemake language.

In the following, let Σ be the alphabet of tokens and Σ^* be the set of all words (i.e., sequences) of these with ε denoting the empty word. In Python as well as in the Snakemake language, indentation has a semantic meaning as it determines code blocks (e.g., the body of a for loop). We make the indentation level a part of the input by considering Mealy automata with pairs of tokens and indentation levels $\Sigma \times \mathbb{N}$ as input alphabet and Σ^* as output alphabet. We further assume that the output function has access to the type of a token. We first define the *identity automaton* which leaves the sequence of tokens unmodified as the Mealy automaton

$$\mathcal{I} = (S, s_0, F, \Sigma \times \mathbb{N}, \Sigma^*, \delta, \lambda) \tag{4.1}$$

with a single state $S = \{s_0\}$, $F = \emptyset$, $\delta(s_0, (t, i)) = s_0$ and $\lambda(s_0, (t, i)) = t$ for any $(t, i) \in \Sigma \times \mathbb{N}$. Next, we define the keyword automaton which shall accept the code block of a keyword (see Section 4.3). The output of the automaton shall be the code block translated to plain Python code. We call the Mealy automaton

$$\mathcal{K}_i = (S, s_0, F, \Sigma \times \mathbb{N}, \Sigma^*, \delta, \lambda) \tag{4.2}$$

an i-indented keyword automaton, if and only if it has at least one state $s \neq s_0$ with $\delta(s, (t, i)) \in F$ and t being the `NEWLINE` token. The intuition is that the i-indented keyword automaton shall stop in an accepting state if the indentation returns to level i. In between, it shall translate the code block of a Snakemake keyword into plain Python tokens, allowing larger indentation levels. For each Snakemake keyword, an i-indented keyword automaton accepting its code block follows directly from its EBNF definition (see Section 4.3). Unexpected tokens can be reported as syntax errors. An i-indented keyword automaton for the `include` keyword (see Section 4.3.5) is shown in Figure 4.2. The automaton first requires a colon, and then either a `stringliteral` or a `NEWLINE` token. In the former case, it accepts if it is followed by a `NEWLINE` token. In the latter case, an indented `stringliteral` token is expected.

To translate a snakefile into plain Python code, we apply, depending on the occurring keywords, above Mealy automata to the sequence of tokens. Whenever a new Snakemake keyword is reached, we recurse into the corresponding keyword automaton. Algorithm 4 outlines the approach. We start the algorithm with the identity automaton (Equation (4.1)), position $k = 1$ in the sequence of tokens and K (the set of valid Snakemake keywords) being the set of top-level keywords. If the snakefile contains only Python code, the algorithm does not recurse and just yields the identity. Else, whenever a Snakemake keyword is reached (see line 8), the corresponding keyword automaton is spawned and the algorithm recurses until the automaton accepts or the end of the file is reached. Thereby, the set of expected Snakemake keywords is updated accordingly. For example, when recursing into a rule code block, the keywords `input`, `output`, ... are expected.

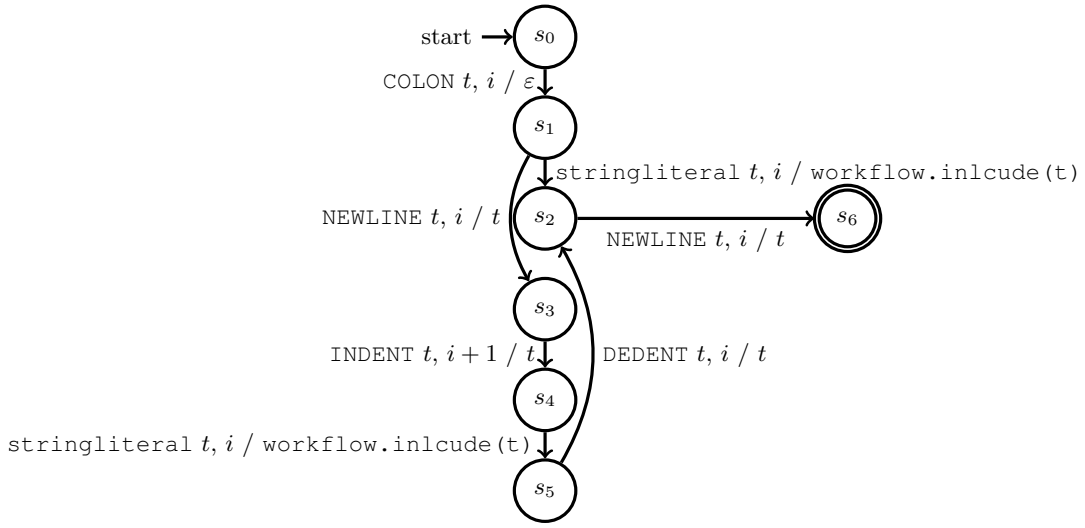

Figure 4.2: Keyword automaton for the `include` keyword. Input and output are denoted separated by a slash beside the transition. Input consists of a token with its type (e.g., `COLON`, `NEWLINE`, `stringliteral`) and the current indentation level. For simplicity, the output may refer to the input token given as t. Unexpected input shall lead to an error.

To map the Snakemake language to Python code, the implemented keyword automata make use of Python function decorators[13]. The example rule from above would be translated to the following Python code:

```
@workflow.rule(name="alpaca_index")
@workflow.input(config["reference"], "{sample}.bam")
@workflow.output("{sample}.hdf5")
@workflow.run
def __alpaca_index(
    input, output, wildcards, threads, resources, log, version
):
    shell("alpaca index {input} {output}")
```

4.4 Dependency resolution

For a given set of target files, Snakemake determines rule applications (i.e., jobs) to generate these. Multiple jobs with different wildcard values can be spawned from a single rule. If the input files of a job are not present, they need to be created by other jobs. Hence, jobs can depend on others. This gives rise to a directed graph of jobs,

[13] `https://docs.python.org/3/reference/compound_stmts.html#function-definitions`, visited 06/2014

Algorithm 4 Translating the sequence of tokens from a snakefile into plain Python tokens using Mealy automata.

Input: The sequence of tokens $t_1, t_2, t_3, \ldots, t_n$ annotated with indentation levels $i_1, i_2, i_3, \ldots, i_n$, a position $1 \leq k \leq n$ in the sequence, a Mealy automaton $\mathcal{A} = (S, s_0, F, \Sigma \times \mathbb{N}, \Sigma^*, \delta, \lambda)$ and a set of expected Snakemake keywords K.

Output: Yields the translated tokens, returns the updated position k in the token sequence.

$s \leftarrow s_0$
$k' \leftarrow k$
while $k' \leq n$ **do**
    $s \leftarrow \delta(s, (t_{k'}, i_{k'}))$
    yield $\lambda(s, (t_{k'}, i_{k'}))$
    if $s \in F$ **then**
        return k'
    if $t_{k'} \in K$ **then**
        $k' \leftarrow k' + 1$
        $K' \leftarrow \emptyset$
        if $t_{k'}$ = rule or $t_{k'}$ = subworkflow **then**
            set K' to be set of Snakemake keywords that may occur in the code block
        $\mathcal{K}_{i_{k'}} \leftarrow$ keyword automaton for keyword $t_{k'}$
        $k' \leftarrow$ **recursion** with $\mathcal{A} = \mathcal{K}_{i_{k'}}$, $K = K'$, $k = k'$
    $k' \leftarrow k' + 1$
return k'

with jobs as nodes and dependencies as edges. The graph needs four properties to be a feasible solution that generates the desired target files.

1. For each target file, there has to be a job that creates it.
2. For each job in the graph, all input files need to be either present in the filesystem or created by another job. Jobs that meet this requirement are called *feasible*.
3. No pair of jobs may exist in the graph that create the same output file. We call such jobs *ambiguous*.
4. The graph has to be acyclic.

The latter requirement is implied by the design of the Snakemake language. A cycle in the graph would mean a loop that executes the same jobs multiple times. As the rule based definition (similar to GNU Make) does not provide a syntax for specifying the number of iterations of such a loop, they are disallowed here. In practice, such a loop can be moved inside a single rule. We call a directed acyclic graph (DAG) that fulfills above four critera a *Job-DAG*. In the following we discuss how a Job-DAG can be found.

The dependencies of a job are determined by finding rules which generate output files that match the input files of the job. Since output files may contain wildcards, we use regular expressions for this matching. The basic idea is to replace wildcards with regular expressions accepting any non-empty word. An output file then matches an input file if the latter is contained in the corresponding regular language of the output file. In the following, we use the extended regular expression syntax of Python[14].

We first give a formal description of the wildcard notation in output files. A wildcard w is a substring of an output file f that is in the language generated by the regular expression $\{n(,c)?\}$ with $n = [a-zA-Z0-9\_]+$ being the name of the wildcard and c being a constraint. Here, { and } are literals. Per default, a wildcard is replaced with the regular expression .+, matching any non-empty word. The constraint is optional and can be used to override the default regular expression that is used to replace the wildcard. This is useful if a wildcard shall only allow certain values, e.g., it would be possible to constrain the wildcard of the example `alpaca_index` rule (Section 4.3) to lower case characters with `{genome,[a-z]+}.hdf5`. We denote with $\mathcal{L}(f)$ the regular language obtained by replacing wildcards in the output file f as described above. Then, a file f' *matches* f if and only if $f' \in \mathcal{L}(f)$. Now, the set of jobs J_f that can create a file f can be determined by finding all rules with an output file f' such that f matches f'. In turn, we define the potential dependencies of a job j with input files I_j as

$$d(j) := \bigcup_{f \in I_j} J_f \tag{4.3}$$

For a given set of target files, the potential dependencies induce a directed graph of jobs with dependencies as edges.

Definition 4.1 (Graph of jobs)**.** *For a job j, we define the dependency graph $G_j = (V_j, E_j)$ with nodes*

$$V_j := d(j) \cup \bigcup_{j' \in d(j)} V_{j'}$$

and edges

$$E_j := \{(j', j) \mid j' \in d(j)\} \cup \bigcup_{j' \in d(j)} E_{j'}.$$

Then, for a given set of target files T, the graph of jobs $G = (V, E)$ is the union of the dependency graphs for each $j \in J_f$ with $f \in T$.

Above graph is the search space for the desired Job-DAG. The Job-DAG can be obtained by traversing the graph in depth-first order as follows. We denote with G_j^* the subgraph of G_j satisfying the Job-DAG properties and obtained by the traversal starting from j. First, an artificial sink node connected to all jobs that create target files is added to the graph. The traversal starts at the sink node and moves along the edges in reverse direction. Let j be the currently visited node. The subgraph G_j^* is obtained by linking j to the union of all $G_{j'}^*$ with $j' \in J'$. The set J' is the set of all j' with $(j', j) \in E$ and

[14] `https://docs.python.org/3/library/re.html`, visited 06/2014

1. adding $G^*_{j'}$ does not close a cycle in G^*_j,
2. j' is feasible,
3. there is no feasible j'' with j' and j'' being ambiguous for an input file $f \in I_j$.

Importantly, these constraints can also render the subgraph G^*_j a singleton only containing the node j. Further, we observe that unresolved dependencies can help to solve ambiguities: if job j'' is not feasible, it will not conflict with the ambiguous job j'. This allows to have multiple rules generating the same output file but being sorted out based on the present input files. If the resulting Job-DAG after complete traversal contains non-feasible jobs, an error is reported. In the following, we call the jobs that (directly or indirectly) depend on a job j, i.e., the set

$$\{j' \in V^* \mid \text{there is a path from } j \text{ to } j' \text{ in } G^*\},$$

downstream jobs of j. Figure 4.4 in Section 4.8 shows the Job-DAG for an example workflow.

Sometimes above procedure does not suffice to sort out ambiguous jobs. One solution in such a case is to constrain the wildcards (see above). Another solution provided by Snakemake is to give rules an order. By defining

```
ruleorder: a > b
```

we can tell Snakemake to prefer jobs of rule `a` over jobs of rule `b`. Then, two jobs of rule `a` and rule `b` that can create the same output file are no longer ambiguous. Instead, the job of rule `b` is discarded if the one of rule `a` is feasible.

4.5 Job scheduling

Once the Job-DAG is created, Snakemake decides which jobs need to be executed. A job shall be considered for execution only if

1. its output files are not present and either a target file or needed by another executed job,
2. its input files are newer than its output files or will be updated by another job,
3. its execution is enforced by the user.

This avoids unnecessary job executions. For example, if a workflow has been completed and all output files are present, the first case will not occur and hence no job will be executed upon re-invocation of the workflow. If the initial execution has been aborted at some step, the jobs whose output files are still missing will be scheduled for execution in contrast to those who have been already successfully completed. If a certain input file is updated in the filesystem, e.g., a biological experiment yielding some raw data

Algorithm 5 A breadth-first search algorithm to determine jobs that shall be executed.

Input: The Job-DAG $G^* = (V^*, E^*)$ as determined above (Section 4.4). The sets O_j and I_j of output and input files for each job j.

Output: Jobs to execute.

1: $J \leftarrow \emptyset$
2: $V \leftarrow J$
3: init queue Q with jobs creating target files not present in file system
4: append jobs with at least one input file being newer than one output file to queue
5: append forced jobs to queue
6: **while** $Q \neq \emptyset$ **do**
7: $\quad j \leftarrow$ first job in Q
8: $\quad J \leftarrow J \cup \{j\}$
9: $\quad$ **for** $j' : (j', j) \in E^*$ **do**
10: $\quad\quad$ **if** $j' \notin V$ and there is an $f \in O_{j'} \cap I_j$ missing in file system **then**
11: $\quad\quad\quad$ append j' to Q
12: $\quad$ **for** $j' : (j, j') \in E^*$ **do**
13: $\quad\quad$ **if** $j' \notin V$ **then**
14: $\quad\quad\quad$ append j' to Q
15: **return** J

has been repeated, the second case applies and leads to re-execution of the necessary jobs.

Based on a breadth-first search (BFS) on the Job-DAG in two directions, Algorithm 5 efficiently implements the decision about the execution of jobs. First, the queue and the set of visited nodes is initialized with forced jobs, those that create missing target files, and those that have updated input files (lines 1-5). Next, the BFS is performed. For a job taken from the queue, first the predecessors in the DAG (i.e., dependencies) are investigated. A predecessor creating a missing input file is appended to the queue if not already visited. Second, all successors of the job are appended to the queue, since the job will be run and thus update its output files.

Above procedure yields a connected subgraph of the Job-DAG which contains only jobs that need to be executed. In the following, we consider this subgraph as the Job-DAG.

Classical scheduling problems (Brucker 2004) consider jobs and machines, with jobs being partitioned into operations of known processing requirements (i.e., the number of machine instructions needed). Typically, the machines are the cores of a CPU, with a known instruction throughput. The scheduling problem is then to assign the jobs to the machines, while jobs can optionally be preemptive (i.e., may be paused and continued later) and may depend on each other (e.g., synchronization between threads). We delegate these decisions to the system scheduler, and only decide which jobs to execute at a specific time point. The Snakemake job scheduling can hence be thought of as a meta-scheduling.

A simple solution is to execute all jobs serially according to the topological sorting of the Job-DAG. Here, the goal is to execute as many jobs as possible in parallel. A path in the Job-DAG specifies a sequence of jobs that have to be executed serially, while two not connected paths can be executed in parallel. At any time point during execution of the workflow, we consider the set J of jobs ready for execution. A job is ready for execution when the execution of its dependencies has been completed. The job scheduling problem of Snakemake is to select a subset of these for immediate execution. Snakemake allows to constrain this by specifying resources that shall not be exceeded by the selection. The only default resource is the number of available CPU cores. The resource usages of a job can be specified in the rule definition (see Section 4.3.1). Any resource requirement defined in the Snakefile that exceeds a given limit is reduced accordingly, i.e., $r' = \min\{r, R\}$ with r being the original resource requirement and R being the given limit.

Apart from the available resources, this selection shall be guided by three properties of a job:

1. the priority,
2. the number of downstream jobs,
3. the total size of the input files,

while higher values are preferred. Per default, each job has a priority of 0, which can be overwritten by its rule and via the command line; e.g.,

```
$ snakemake --prioritize homo_sapiens.hdf5
```

would ensure that `homo_sapiens.hdf5` is created as fast as possible by setting all needed jobs (i.e., the subgraph of the Job-DAG with `homo_sapiens.hdf5` as output of the sink node) to maximum priority. The second and third property are heuristics to avoid idle cores. The intuition is that a job with many downstream jobs (see Section 4.4) should be executed as early as possible, in order to have more choices in J to fill up the available resources at a later time point. Finally, a job with a large total input file size will likely be long running. It is therefore resonable to execute these jobs early to avoid waiting on a single long running job at the end of the workflow execution. The complete job scheduling problem can be defined as the following optimization problem.

Definition 4.2 (Job scheduling)**.** *Let J be the set of jobs ready for execution. Suppose $n \geq 1$ is the number of given resources and $R_i \in \mathbb{N}$ the free amount of resource $i = 1, \ldots, n$. Among all subsets $E \subseteq J$ we search the set of jobs E^* that maximizes*

$$\sum_{j \in E} (p_j, d_j, i_j)^T$$

under lexicographical order, subject to

$$\sum_{j \in E} r_{i,j} \leq R_i \quad \text{for } i = 1, 2, \ldots, n$$

with priority p_j, number of descendants d_j, input size i_j and usage $r_{i,j}$ of resource i.

Here, $(p_j, d_j, i_j)^T$, in the following called the *job reward*, shall be a vector with the usual element-wise sum. The lexicographical order over the job reward is defined as

$$(a,b,c)^T < (a',b',c')^T \text{ if and only if } a < a' \vee (a = a' \wedge b < b') \vee (a = a' \wedge b = b' \wedge c < c')$$

with $\wedge, \vee$ being the boolean logic operators "and" and "or". We chose the lexicographical order instead of weighting the components of the job reward, thereby generating a combined score, because the components have different units and may appear in diverse ranges. Hence, finding universally applicable weights appears difficult. In contrast, the lexicographical order ensures that priority is the most and input file size is the least important property for job selection.

Similar to other specialized scheduling problems (Vanderster, Dimopoulos, and Sobie 2006; Mounie, Rapine, and Trystram 1999), this problem can be seen as a knapsack problem. We consider the multi-dimensional knapsack problem (MDKP), which Lin (1998) defines as follows. Assume an n-dimensional knapsack with b_i being the capacity of the i-th dimension. For the k-th of m different items, let u_k be the number of copies, $a_{i,k}$ be its requirement of dimension i, and c_k be the reward when including a copy of item k in the knapsack. Then, the multi-dimensional knapsack problem is to

$$\begin{aligned} \text{maximize } & \sum_{k=1}^{m} c_k x_k & & \\ \text{subject to } & \sum_{k=1}^{m} a_{i,k} x_k \leq b_i & & \text{for } i = 1, 2, \ldots, n \\ & 0 \leq x_k \leq u_k & & \text{for } x_k \in \mathbb{N}, k = 1, 2, \ldots, m \end{aligned}$$

with x_k being the decision variable selecting x_k copies of item k for inclusion. For job scheduling, the knapsack has n dimensions, one for each resource with $b_i = R_i$. Each job ready for execution becomes an item k with only a single copy $u_k = 1$. This is also called a 0-1 MDKP, analog to the 0-1 knapsack problem (Cormen et al. 2001). For each item k, the resource usage of resource i is the requirement of dimension, i.e., $a_{i,k} = r_{i,k}$. The item reward equals the job reward, i.e., $c_k = (p_k, d_k, i_k)^T$. This guarantees that the selected items, i.e., jobs, do not exceed the given resources and maximize the reward.

The 0-1 MDKP is a generalization of the one-dimensional knapsack problem. Hence, it is NP-hard (Lin 1998). While an exact solution via a dynamic programming algorithm follows directly from the one-dimensional knapsack problem, it is much more challenging since a table of size $\mathcal{O}(mb^n)$ with b being the maximum capacity has to be filled (Kellerer, Pferschy, and Pisinger 2004). Since scheduling has to be fast even for thousands of jobs and arbitrary resources, it is advisable to use a heuristic approximation. Here, we use a primal greedy heuristic (see Kellerer, Pferschy, and Pisinger 2004) published by Akçay, Li, and Xu (2007) for approximating the 0-1 MDKP, namely we greedily select the jobs with the maximum rewards that can be accepted with the remaining capacity of the knapsack.

4.6 Support for distributed computing

By default, Snakemake executes jobs on the local machine it is invoked on. Alternatively, it can execute jobs in distributed environments, e.g., compute clusters or grids. These are networked sets of possibly heterogeneous machines (nodes), controlled by a resource management system (RMS) that manages shared resources and schedules submitted compute jobs (see Krauter, Buyya, and Maheswaran 2002). If the nodes share a common file system, Snakemake supports two alternative approaches of using a resource management system.

On such systems, compute jobs are usually submitted as shell scripts via commands like `qsub` (e.g., Oracle/Univa Grid Engine[15] and Torque[16]) or `sbatch` (e.g., SLURM[17]). In a generic approach, Snakemake can compile each job into a shell script executing it. A cluster submission command specified via the command line inteface, e.g.,

```
$ snakemake --cluster qsub
```

is then used to submit each job shell script to the resource management system. The specified submission command can also be decorated with additional parameters taken from the submitted job. For example, the number of used threads can be accessed in braces similarly to the formatting of shell commands (see Section 4.3):

```
$ snakemake --cluster "qsub -pe threaded {threads}"
```

This way, Snakemake supports many resource management systems without the need of specific implementations. Alternatively, Snakemake can use the Distributed Resource Management Application API (DRMAA)[18]. This API provides a common interface to control various resource management systems. The DRMAA support can be activated by invoking Snakemake as follows:

```
$ snakemake --drmaa
```

A DRMAA supporting RMS registers itself as a backend to the DRMAA library. Snakemake uses the Python bindings of DRMAA[19] to submit jobs as shell scripts (see above), which are automatically delegated to the registered RMS by the DRMAA library.

4.7 Data provenance

Documentation and reproducibility of computational scientific analyses has been identified as a major challenge (Mesirov 2010). This includes the automation as well as documentation of the analysis steps and the ability to track the provenance of each

[15] `http://www.univa.com/products/grid-engine.php`, visited 06/2014
[16] `http://www.adaptivecomputing.com/products/open-source/torque`, visited 06/2014
[17] `https://computing.llnl.gov/linux/slurm`, visited 06/2014
[18] `http://www.drmaa.org`, visited 06/2014
[19] `http://drmaa-python.readthedocs.org`, visited 11/2014

generated result. The latter is also called data provenance. Snakemake provides data provenance by various means. The source code describing the workflow is readable because of the simple pythonic syntax that splits the workflow into small, self-contained parts. By issuing

```
$ snakemake --summary
```

a table associating each output file in a workflow with the rule used to generate it, the creation date and optionally the version of the tool used for creation (see Section 4.3.3) is provided. Further, the table informs about updated input files and changes to the source code of the rule after creation of the output file.

An important part of data provenance is the communication of results together with the performed analysis steps to researchers of other disciplines. Here, (similar to Goecks, Nekrutenko, and Taylor 2010) Snakemake provides a method to compose rich HTML reports that allow to semantically connect the results with each other and to describe the analysis steps. The reports are written in RestructuredText[20], a simple markup language that can be converted to HTML. Typically a report is generated by an extra rule:

```
rule report:
    input:  "benchmarks/A0.json"
    output: report="report.html"
    run:
        snakemake.utils.report("""
        ==================
        Benchmarking ALPACA
        ==================

        A benchmark of ALPACA sample
        indexing (see File F1_) shows...

        """, output.report, F1=input[0],
        metadata="Author: Johannes Koester")
```

Files (e.g., figures or tables) provided as keyword arguments (`F1=input[0]`) can be referred from within the document. All provided files are embedded into a single HTML file as base64 encoded data URLs[21]. The resulting report works as a self-contained document storage. Instead for providing large archives or folders with many files, a single, portable HTML file can be provided to collaborators that informs about the performed analysis and provides the results as files that can be saved to the local workstation, e.g., for use in a publication. Figure 4.3 shows the report generated from above example.

[20] http://docutils.sourceforge.net/rst.html, visited 11/2014
[21] http://tools.ietf.org/html/rfc2397, visited 11/2014

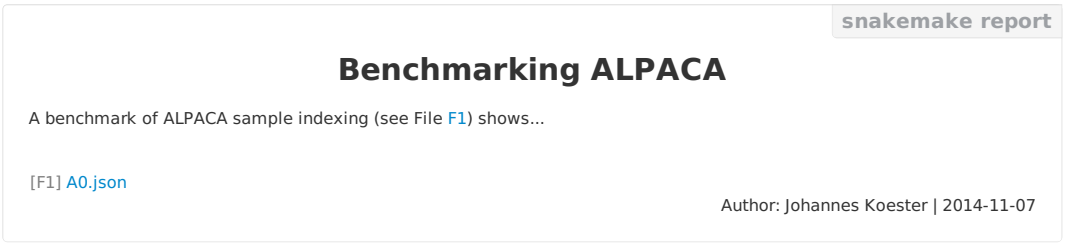

Figure 4.3: Example HTML report generated with Snakemake. Embedded files are provided as links at the bottom of the page. They can be referred to from within the text, thereby allowing to connect results semantically with the described analysis.

4.8 An example workflow

This work provides new algorithms for two major tasks in the analysis of NGS data, i.e., PEANUT for read mapping (Chapter 2) and ALPACA for variant calling (Chapter 3). Now, we show how to combine these into a Snakemake workflow for finding variants from genome sequencing data. We decide to divide the workflow into three modules:

1. Read mapping,
2. BAM file postprocessing,
3. Variant calling.

Apart from increasing the readability, this allows to replace parts of the workflow, e.g., with other tools or alternative strategies. A workflow always has to be adapted to the underlying data, here the NGS reads obtained from the biological samples. We configure the workflow with a JSON file, using the **`configfile`** keyword (see Section 4.3). In the following we refer to this as the *workflow configuration*.

We distinguish between samples and units. A sample is a biological section of a tissue or blood. The sequencing of such a sample yields a unit, a collection of sequence reads. A unit is typically given as a FASTQ file (Cock et al. 2010). Sometimes, multiple units exist for a single sample, e.g., if a sample was sequenced repeatedly to increase data quality. We model the relation between samples and units in the workflow configuration.

The outline of the workflow is as follows. First, sequence reads of each unit are mapped separately to a given reference genome with PEANUT. Second, the resulting BAM files are postprocessed, preparing them for variant calling by sorting and indexing. Third, the ALPACA sample index is built for each sample and merged before the variants are called according to queries defined in the workflow configuration. We describe each module separately.

Listing 4.2: Read mapping module

```
UNIT_TO_SAMPLE = {
    unit: sample for sample, units in config["samples"].items()
    for unit in units}


rule peanut_map:
    input:
        lambda wildcards: config["units"][wildcards.unit],
        lambda wildcards: config["references"][wildcards.reference]
    output:
        "mapping/{reference}/units/{unit}.bam"
    params:
        sample=lambda wildcards: UNIT_TO_SAMPLE[wildcards.unit],
        custom=config.get("params_peanut", "")
    log:
        "mapping/log/{reference}/{unit}.log"
    threads: 8
    resources: gpu=1
    shell:
        "peanut map {params.custom} "
        "--read-group ID:{wildcards.unit} "
        "SM:{params.sample} PL:{config[platform]} "
        "--threads {threads} {input} 2> {log} "
        "| samtools view -Sbh - > {output}"
```

Read mapping The read mapping is handled with a single rule `peanut_map` (Listing 4.2). The rule maps the reads of a unit to a given reference using PEANUT (Chapter 2). The input, consisting of FASTQ files with the reads of the unit and a FASTA file with the reference are determined by evaluating lambda expressions, i.e., inline Python functions[22]. They take the wildcard values determined from the output files as argument and fetch the paths to the FASTQ and FASTA files from the workflow configuration. The rule uses up to 8 threads and one GPU. The requirement of the GPU is modeled by the custom `gpu` resource (see Section 4.3.1). PEANUT is made aware of the sequencing platform and the sample the unit belongs to by specifying a read group. This information is stored inside the BAM file for later use.

BAM file postprocessing For processing with ALPACA, the BAM files created by the mapping have to be sorted by position and indexed. This is achieved by two rules `bam_index` and `bam_sort` (Listing 4.3). The rules are generic and can be applied to any BAM file by using a wildcard `{prefix}` instead of requesting a more specific path. Hence, they can be re-used in a different context. Here, an advantage of using a rule-based system becomes apparent. Instead of having to model indexing and sorting as a dependency at possibly multiple locations in the workflow, it is incorporated automatically by requiring the `.bai` file as an input and adding `sorted` to the filename.

[22] `https://docs.python.org/3/reference/expressions.html#lambda`, visited 06/2014

Listing 4.3: BAM handling module

```
rule bam_index:
    input:
        "{prefix}.bam"
    output:
        "{prefix}.bam.bai"
    shell:
        "samtools index {input}"


rule bam_sort:
    input:
        "{prefix}.bam"
    output:
        "{prefix}.sorted.bam"
    shell:
        "samtools sort {input} {wildcards.prefix}.sorted"
```

Variant calling Finally, the postprocessed BAM files containing the mapped reads of each unit can be used for variant calling with ALPACA (Listing 4.4). We use ALPACA in GPU computing mode (and therefore let the rules require the custom resource `gpu`). As outlined in Section 3.6, ALPACA consists of three steps. First, each sample is indexed (rule `alpaca_index`). Here, the expected ploidy (i.e., the expected number of chromosome copies; see Chapter 3) is read from the workflow configuration. Second, after all samples have been indexed, the indexes are merged and irrelevant genomic sites are removed (see Section 3.6). In the third step, the merged index is used for calculating the variant calls for a given query (see Section 3.4). The name of the query is encoded into the name of the output file. The actual query expression is fetched from the workflow configuration and stored in the rule parameter `query`.

The workflow can be stitched together by including above modules into a master snakefile (Listing 4.5). Initially, the workflow configuration is parsed from the JSON config file. Then, the modules are included. Instead of paths to local files, HTTP urls are allowed here (see Section 4.3.5). This eases the distribution of a workflow, since rules can be fetched directly from a trusted online repository. Since the includes can be placed inside conditional statements, parts of the workflow could be replaced, e.g., depending on some parameter. For example, the PEANUT module could be replaced by a module using BWA (Li and Durbin 2009) for read mapping. Finally, a target rule `all` is defined, that collects the VCF files with the variant calls for the queries and thresholds defined in the workflow configuration. Figure 4.4 shows the Job-DAG for two samples A and B, with one and two units and the queries A+B, A-B and B-A.

Listing 4.4: Variant calling module. The helper functions `_sample_units` and `_get_ref` return the corresponding paths from the workflow configuration.

```
rule alpaca_index:
    input:
        _sample_units("mapping/{reference}/units/{unit}.sorted.bam.bai"),
        bams=_sample_units("mapping/{reference}/units/{unit}.sorted.bam"),
        ref=_get_ref
    output:
        "snv_calling/{reference}/{sample}.index.hdf5"
    log:
        "snv_calling/log/{reference}/{sample}.index.log"
    threads: 8
    resources: gpu=1
    shell:
        "alpaca --dev gpu --threads {threads} index --ploidy {config[ploidy]} "
        "--sample-name {wildcards.sample} {input.ref} "
        "{input.bams} {output} 2> {log}"


rule alpaca_merge:
    input:
        _get_ref,
        expand(
            "snv_calling/{{reference}}/{sample}.index.hdf5",
            sample=config["samples"])
    output:
        "snv_calling/{reference}/index.hdf5"
    log:
        "snv_calling/log/{reference}/merge.log"
    resources: gpu=1
    shell:
        "alpaca --dev gpu merge {input} {output} 2> {log}"


rule alpaca_call:
    input:
        "snv_calling/{reference}/index.hdf5"
    output:
        "snv_calling/{reference}/{query}.vcf"
    log:
        "snv_calling/log/{reference}/{query}.call.log"
    params:
        query=lambda wildcards: config["alpaca_queries"][wildcards.query]
    resources: gpu=1
    shell:
        "alpaca --dev gpu call --fdr {config[fdr]} "
        "--max-strand-bias {config[max_strandbias]} "
        "--heterozygosity {config[heterozygosity]} "
        "{input} '{params.query}' > {output} 2> {log}"
```

Listing 4.5: Master snakefile

```
configfile: "config.json"

include:
    "peanut.rules"
include:
    "samfiles.rules"
include:
    "alpaca_calling.rules"

rule all:
    input:
        expand(
            "snv_calling/genome/{alpaca_queries}.vcf",
            alpaca_queries=config["alpaca_queries"]
        )
```

4.9 Discussion

Among the text-based workflow systems (see Section 4.2), Snakemake provides novel distinguishing features like the easy to read yet powerful pythonic language and the advanced scheduling capabilities. Together with the cluster support, the scheduling which considers priorities, threads and custom resources allows to scale a Snakemake workflow from single core machines and multi-core servers to clusters or batch systems without changing the workflow definition. Further, the custom resources allow flexible incorporation of heterogeneous computing solutions like PEANUT and ALPACA. Apart from the command line interface, Snakemake provides a Python API, that allows the programmatic invocation of a workflow. The API supports custom log handlers, such that the status of the workflow can be displayed, e.g., in a web interface. In the future, this might lead to even tighter integration of Snakemake into analysis frameworks like Exomate (Martin et al. 2013).

Snakemake is a reasonable tool to perform reproducible science. In the extreme, it is possible to model the whole process from raw data to the figures and tables of a publication, while providing documentation and provenance information, as well as self-contained portable reports connecting the results with the description of the method.

Since its publication, Snakemake has been widely adopted and was used to build analysis workflows for a variety of publications, e.g., by Patterson et al. (2014), Chang et al. (2014), Marschall and Schönhuth (2013), Martin et al. (2013), Czeschik et al. (2013), Rahmann et al. (2013), Althoff et al. (2013), and Marschall et al. (2012). With around 2200 homepage visits from January to April 2014 of around 850 different visitors[23], and more than 12,000 downloads since the first release[24] it appears to have a stable

[23] http://www.google.com/analytics, visited 05/2014
[24] http://pypi-ranking.info, visited 06/2014

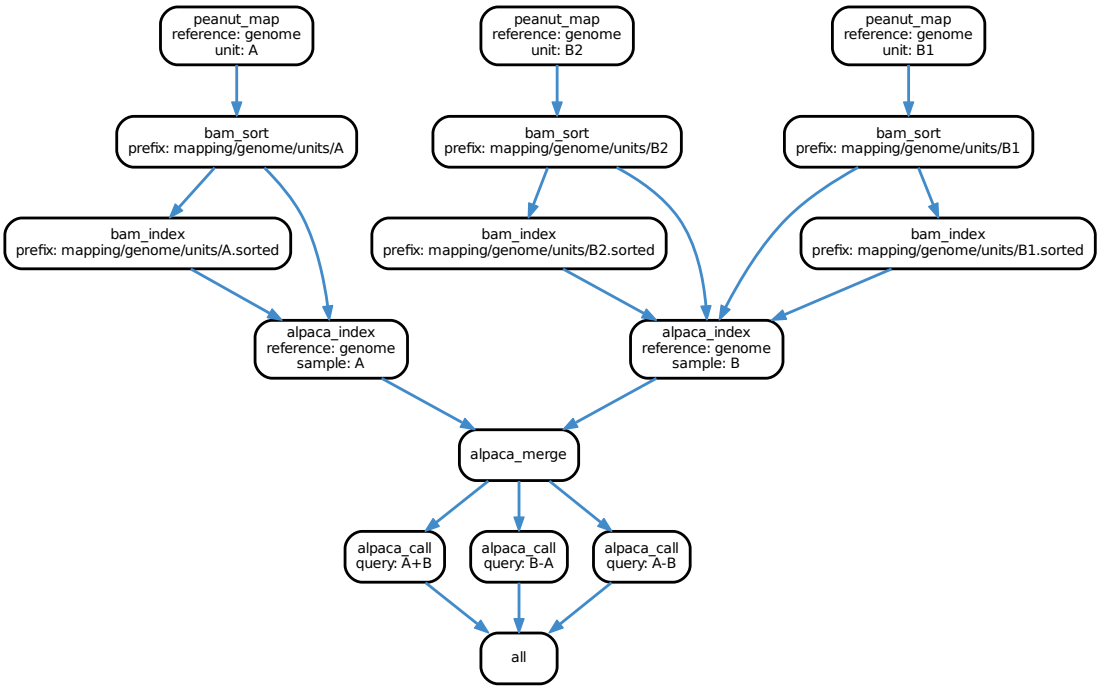

Figure 4.4: Example Job-DAG automatically generated with Snakemake and Graphviz.

community of regular users.

Snakemake is complemented by the Snakemake Workflow Repository[25], providing a collection of high quality rules and workflows for common analysis tasks. So far, the repository is limited to the analysis of next-generation sequencing data, but contributions might extend it to other fields and disciplines as well. The contained rules are grouped into modules following conventions to allow for interchangeability. Among others, the modules of above example workflow (Section 4.8) are contained in the repository.

[25] `https://bitbucket.org/johanneskoester/snakemake-workflows`, visited 11/2014

5 Conclusion

This thesis contributes to three major topics in NGS analysis. Taken together, the three presented approaches to read mapping, variant calling and workflow management can execute a complete NGS analysis, focusing on parallelization, scalability and reproducibility, providing various advances compared to competing solutions.

In Chapter 2, we present the q-group index, a variant of the q-gram index with particularly small memory footprint, making it feasible even for consumer level GPUs. We show that the parallel algorithms for querying and building of the q-group index can saturate GPU computing capabilities. The benefit of the q-group index is illustrated by the implementation in the PEANUT read mapper, which outperforms state of the art competitors while at least maintaining their accuracy. When requiring all alignments of a read down to a given error rate, PEANUT is three to ten times faster than the best competitor.

Chapter 3 introduces an algebraic variant calling approach and its implementation in the variant caller ALPACA. We incorporate the filtering between samples into the calling process. Based on expressions over the set algebra of unknown variant loci which are approximated via posterior probabilities, algebraic variant calling is the first method that allows to intuitively control the false discovery rate for arbitrary filtering scenarios. Using preprocessed indexes, ALPACA avoids redundant computations upon the addition of new samples to an existing study and allows to calculate queries within seconds, providing improved scalability compared to competitors.

Although being designed with NGS analysis in mind, the workflow system Snakemake presented in Chapter 4 has become a generic approach to describe and execute workflows. It provides an easy to read workflow definition language and a flexible execution environment that allows to implicitly parallelize a workflow. Providing a scheduling approach that can adapt to arbitrary given resources like the number of available CPU cores or even GPUs, Snakemake workflows scale to the used hardware without the need to modify the workflow definition. Data provenance support and HTML reports, that allow to semantically connect results with the used methods, help to create reproducible analyses.

While being an established technology, NGS still evolves and new approaches appear. Therefore, it is reasonable to ask about the impact of new developments on the algorithms presented in Chapter 2 and 3. A general tendency is to increase the length of the

reads. An example is the emerging single molecule real time (SMRT) sequencing platform of Pacific Biosciences[1], which provides reads of length between 1000 and 40.000 bases. Today, the typical read depths do not allow variant calling with SMRT, but it can be used to, e.g., infer haplotypes from called variants (Patterson et al. 2014), such that it can be seen as a complement to Illumina NGS. When read depths become higher, algebraic variant calling (Chapter 3) can be directly applied, profiting from the more random error distribution of SMRT sequencing. With longer reads, the challenges during read mapping are shifted. The handling of paired end sequencing is unnecessary and ambiguous hits become less an issue. In contrast, it is more likely that the proper read alignment is split over various regions, e.g., when sequencing RNA. Here, a read will align to the exons of a gene while skipping introns (see Section 1.1). Still, the q-group index (Chapter 2) is applicable: in an ongoing effort, we use it to find anchor points for local alignments between the read and the reference, which will be combined to a semi-global alignment in a later step, e.g., considering prior knowledge about exonic and intronic regions and genes. In summary, the presented algorithms are not so tightly coupled with the current technology that the foreseeable new developments will render them obsolete.

[1] `http://www.pacificbiosciences.com`, visited 11/2014

A Appendix

A.1 Software

The software presented in this thesis is available open source under the MIT license[1]. In the following, we provide information about each software and refer to the implementations of the presented algorithms, together with the Git[2] commits that contain the state described in this thesis.

PEANUT The read mapper PEANUT, presented in Chapter 2, is available at

`https://peanut.readthedocs.org`

as a Python package. The implementation is located in the folder `peanut` in a corresponding Git repository[3]. This thesis describes version 1.3.1 of PEANUT (commit 61bae9f). The implementation of the q-group index with build and query algorithms (Section 2.4, Algorithm 1, and Algorithm 2) can be found in `filtration.py` and `filtration.cl`. The implementation of the validation (Section 2.5.2) can be found in the files `validation.py` and `validation.cl`. Postprocessing (Section 2.5.3) is implemented in the files `postprocessing.pyx` and `alignment.pyx`. The analysis workflow implemented with Snakemake as it was used for this thesis (Section 2.6) can be found in the file `analysis-pipeline/Snakefile` at commit 86f820d.

ALPACA The variant caller ALPACA, presented in Chapter 3, is available at

`https://alpaca.readthedocs.org`

as a Python package. The implementation is located in the folder `alpaca` in a corresponding Git repository[4]. This thesis describes version 0.2.2 of ALPACA (commit 695c66b). The implementation of the underlying Bayesian variant calling (Section 3.3) and the index data structures (Section 3.6) can be found in the folder `index`. Algebraic variant calling and FDR control (Section 3.4) is implemented in the folder `caller`.

[1] `http://opensource.org/licenses/MIT`, visited 12/2014
[2] `http://git-scm.com`, visited 12/2014
[3] `https://bitbucket.org/johanneskoester/peanut.git`, visited 12/2014
[4] `https://bitbucket.org/johanneskoester/alpaca.git`, visited 12/2014

The file `analysis-pipeline/Snakefile` at commit 0e41d10 contains the analysis workflow implemented with Snakemake as it was used to generate the results in Section 3.7.

Snakemake The workflow system Snakemake, presented in Chapter 4, is available at

`https://bitbucket.org/johanneskoester/snakemake`

as a Python package. The implementation is located in the folder `snakemake` in a corresponding Git repository[5]. This thesis describes version 3.1.1 of Snakemake (commit 498db51). The file `parser.py` contains the implementation of the language parser (Section 4.3.6). Dependency resolution (Section 4.4) and Algorithm 5 is implemented in the file `dag.py`. The file `scheduler.py` contains the implementation of the scheduling (Section 4.5).

A.2 Contributions to co-authored articles

Several articles were published together with co-authors during the work on this thesis. Sven Rahmann assisted in all stages as advisor. The following three articles describe research that is extended in this thesis. Chapter 2 is based on the article

> J. Köster and S. Rahmann (2014). "Massively parallel read mapping on GPUs with the q -group index and PEANUT". In: *PeerJ* 2:e606

for which I developed the q-group index and the PEANUT algorithm and performed a comparison with other algorithms. Sven Rahmann assisted in writing, analyzed the index size (Section 2.4.1) and helped with the estimation of the mapping quality (Section 2.5.3). Chapter 4 is based on the article

> J. Köster and S. Rahmann (2012a). "Building and Documenting Workflows with Python-Based Snakemake". In: *German Conference on Bioinformatics 2012*. Ed. by S. Böcker et al. Vol. 26. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, pp. 49–56

in which I present my first version of the scheduling (Section 4.5) and the parsing mechanism of Snakemake (Section 4.3.6). The article

> J. Köster and S. Rahmann (2012b). "Snakemake - a scalable bioinformatics workflow engine". In: *Bioinformatics* 28.19, pp. 2520–2522

[5] `https://bitbucket.org/johanneskoester/snakemake.git`, visited 12/2014

is a shorter version, in which I present the workflow definition language (Section 4.3) and major properties of Snakemake at a higher level. To both articles, Sven Rahmann contributed by writing and discussions about functionality. An initial test framework for Snakemake and various useful feature requests were contributed by Tobias Marschall and Marcel Martin.

The following articles did not directly contribute, but shaped a general understanding of biology, next-generation sequencing analysis and interdisciplinary research that has helped with and inspired the research done in this thesis:

> J. Köster, E. Zamir, and S. Rahmann (2012). "Efficiently mining protein interaction dependencies from large text corpora". In: *Integrative Biology* 4.7, pp. 805–812
>
> A. Schramm, B. Schowe, K. Fielitz, M. Heilmann, M. Martin, T. Marschall, J. Köster, J. Vandesompele, J. Vermeulen, K. d. Preter, J. Koster, R. Versteeg, R. Noguera, F. Speleman, S. Rahmann, A. Eggert, K. Morik, and J. H. Schulte (2012). "Exon-level expression analyses identify MYCN and NTRK1 as major determinants of alternative exon usage and robustly predict primary neuroblastoma outcome". In: *British Journal of Cancer* 107.8, pp. 1409–1417
>
> K. Althoff, A. Beckers, A. Odersky, P. Mestdagh, J. Köster, I. M. Bray, K. Bryan, J. Vandesompele, F. Speleman, R. L. Stallings, A. Schramm, A. Eggert, A. Sprüssel, and J. H. Schulte (2013). "MiR-137 functions as a tumor suppressor in neuroblastoma by downregulating KDM1A". In: *International Journal of Cancer* 133.5, pp. 1064–1073
>
> S. Rahmann, M. Martin, J. H. Schulte, J. Köster, T. Marschall, and A. Schramm (2013). "Identifying transcriptional miRNA biomarkers by integrating high-throughput sequencing and real-time PCR data". In: *Methods* 59.1, pp. 154–163
>
> A. Schramm, J. Köster, T. Marschall, M. Martin, M. Schwermer, K. Fielitz, G. Büchel, M. Barann, D. Esser, P. Rosenstiel, S. Rahmann, A. Eggert, and J. H. Schulte (2013). "Next-generation RNA sequencing reveals differential expression of MYCN target genes and suggests the mTOR pathway as a promising therapy target in MYCN-amplified neuroblastoma". In: *International Journal of Cancer* 132.3, E106–E115

Bibliography

Aalst, W. v. d., K. V. Hee, and J. Mylopoulos (2002). *Workflow Management: Models, Methods, and Systems*. MIT Press. 384 pp.

Akçay, Y., H. Li, and S. H. Xu (2007). "Greedy algorithm for the general multidimensional knapsack problem". In: *Annals of Operations Research* 150.1, pp. 17–29.

Alberts, B., A. Johnson, and J. Lewis (2008). *Molecular Biology of the Cell*. Garland Pub. 1601 pp.

Alkan, C., J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, S. C. Sahinalp, R. A. Gibbs, and E. E. Eichler (2009). "Personalized copy number and segmental duplication maps using next-generation sequencing". In: *Nature Genetics* 41.10, pp. 1061–1067.

Althoff, K., A. Beckers, A. Odersky, P. Mestdagh, J. Köster, I. M. Bray, K. Bryan, J. Vandesompele, F. Speleman, R. L. Stallings, A. Schramm, A. Eggert, A. Sprüssel, and J. H. Schulte (2013). "MiR-137 functions as a tumor suppressor in neuroblastoma by downregulating KDM1A". In: *International Journal of Cancer* 133.5, pp. 1064–1073.

Behnel, S., R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith (2011). "Cython: The Best of Both Worlds". In: *Computing in Science & Engineering* 13.2, pp. 31–39.

Benjamini, Y. and Y. Hochberg (1995). "Controlling the false discovery rate: a practical and powerful approach to multiple testing". In: *Journal of the Royal Statistical Society. Series B. Methodological* 57.1, pp. 289–300.

Berthold, M. R., N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, C. Sieb, K. Thiel, and B. Wiswedel (2007). "KNIME: The Konstanz Information Miner". In: *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*. Springer.

Blelloch, G. E. (1990). *Prefix Sums and Their Applications*. Tech. rep. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University.

Blom, J., T. Jakobi, D. Doppmeier, S. Jaenicke, J. Kalinowski, J. Stoye, and A. Goesmann (2011). "Exact and complete short-read alignment to microbial genomes using Graphics Processing Unit programming". In: *Bioinformatics* 27.10, pp. 1351–1358.

Brucker, P. (2004). *Scheduling Algorithms*. Springer. 367 pp.

Chang, H., J. Lim, M. Ha, and V. N. Kim (2014). "TAIL-seq: Genome-wide Determination of Poly(A) Tail Length and 3' End Modifications". In: *Molecular Cell* 53.6, pp. 1044–1052.

Cibulskis, K., M. S. Lawrence, S. L. Carter, A. Sivachenko, D. Jaffe, C. Sougnez, S. Gabriel, M. Meyerson, E. S. Lander, and G. Getz (2013). "Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples". In: *Nature Biotechnology* 31.3, pp. 213–219.

Cock, P. J. A., C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice (2010). "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants". In: *Nucleic Acids Research* 38.6, pp. 1767–1771.

Cormen, T. H., C. Stein, C. E. Leiserson, and R. L. Rivest (2001). *Introduction to Algorithms*. The Mit Press. 1202 pp.

Czeschik, J. C., C. Voigt, Y. Alanay, B. Albrecht, S. Avci, D. FitzPatrick, D. R. Goudie, U. Hehr, A. J. Hoogeboom, H. Kayserili, P. O. Simsek-Kiper, L. Klein-Hitpass, A. Kuechler, V. López-González, M. Martin, S. Rahmann, B. Schweiger, M. Splitt, B. Wollnik, H.-J. Lüdecke, M. Zeschnigk, and D. Wieczorek (2013). "Clinical and mutation data in 12 patients with the clinical diagnosis of Nager syndrome". In: *Human Genetics* 132.8, pp. 885–898.

Dehne, F. and K. Yogaratnam (2010). "Exploring the Limits of GPUs With Parallel Graph Algorithms". In: *CoRR* abs/1002.4482.

DePristo, M. A., E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. del Angel, M. A. Rivas, M. Hanna, A. McKenna, T. J. Fennell, A. M. Kernytsky, A. Y. Sivachenko, K. Cibulskis, S. B. Gabriel, D. Altshuler, and M. J. Daly (2011). "A framework for variation discovery and genotyping using next-generation DNA sequencing data". In: *Nature Genetics* 43.5, pp. 491–498.

Dobin, A., C. A. Davis, F. Schlesinger, J. Drenkow, C. Zaleski, S. Jha, P. Batut, M. Chaisson, and T. R. Gingeras (2013). "STAR: ultrafast universal RNA-seq aligner". In: *Bioinformatics* 29.1, pp. 15–21.

Durbin, R. (1998). *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press. 372 pp.

Ferragina, P. and G. Manzini (2000). "Opportunistic Data Structures with Applications". In: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. FOCS '00. IEEE Computer Society, pp. 390–398.

Fisher, R. A. (1922). "On the Interpretation of χ^2 from Contingency Tables, and the Calculation of P". In: *Journal of the Royal Statistical Society* 85.1, pp. 87–94.

Gansner, E. R. and S. C. North (2000). "An open graph visualization system and its applications to software engineering". In: *Software - Practice and Experience* 30.11, pp. 1203–1233.

Garrison, E. and G. Marth (2012). "Haplotype-based variant detection from short-read sequencing". In: *arXiv:1207.3907*.

Goecks, J., A. Nekrutenko, and J. Taylor (2010). "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences". In: *Genome Biology* 11.8, R86.

González, R., S. Grabowski, V. Mäkinen, and G. Navarro (2005). "Practical implementation of rank and select queries". In: *Proceedings of WEA*, pp. 27–38.

Goodstadt, L. (2010). "Ruffus: A Lightweight Python Library for Computational Pipelines". In: *Bioinformatics* 26.21, pp. 2778–2779.

Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

Halbritter, F., H. J. Vaidya, and S. R. Tomlinson (2011). "GeneProf: analysis of high-throughput sequencing experiments". In: *Nature Methods* 9.1, pp. 7–8.

Holtgrewe, M. (2010). *Mason – a read simulator for second generation sequencing data*.

Holtgrewe, M., A.-K. Emde, D. Weese, and K. Reinert (2011). "A novel and well-defined benchmarking method for second generation read mapping". In: *BMC Bioinformatics* 12.1, p. 210.

Hoon, S., K. K. Ratnapu, J.-M. Chia, B. Kumarasamy, X. Juguang, M. Clamp, A. Stabenau, S. Potter, L. Clarke, and E. Stupka (2003). "Biopipe: a flexible framework for protocol-based bioinformatics analysis". In: *Genome Research* 13.8, pp. 1904–1915.

Huffman, D. (1952). "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9, pp. 1098–1101.

Hunter, J. (2007). "Matplotlib: A 2D Graphics Environment". In: *Computing in Science Engineering* 9.3, pp. 90 –95.

Hyyrö, H. (2003). "A bit-vector algorithm for computing Levenshtein and Damerau edit distances". In: *Nordic Journal of Computing*, p. 2003.

IEEE (2008). "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008*, pp. 1–70.

Jacobson, G. J. (1988). "Succinct Static Data Structures". PhD thesis. Carnegie Mellon University.

Jokinen, P. and E. Ukkonen (1991). "Two algorithms for approxmate string matching in static texts". In: *Mathematical Foundations of Computer Science 1991*. Ed. by A. Tarlecki. Lecture Notes in Computer Science 520. Springer Berlin Heidelberg, pp. 240–248.

Kellerer, H., U. Pferschy, and D. Pisinger (2004). *Knapsack Problems*. Springer. 572 pp.

Kircher, M. and J. Kelso (2010). "High-throughput DNA sequencing – concepts and limitations". In: *BioEssays* 32.6, pp. 524–536.

Klöckner, A., N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih (2012). "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation". In: *Parallel Computing* 38.3, pp. 157–174.

Klus, P., S. Lam, D. Lyberg, M. S. Cheung, G. Pullan, I. McFarlane, G. S. Yeo, and B. Y. Lam (2012). "BarraCUDA - a fast short read sequence aligner using graphics processing units". In: *BMC Research Notes* 5.1, p. 27.

Krauter, K., R. Buyya, and M. Maheswaran (2002). "A taxonomy and survey of grid resource management systems for distributed computing". In: *Software: Practice and Experience* 32.2, pp. 135–164.

Köster, J. and S. Rahmann (2012a). "Building and Documenting Workflows with Python-Based Snakemake". In: *German Conference on Bioinformatics 2012*. Ed. by S. Böcker, F. Hufsky, K. Scheubert, J. Schleicher, and S. Schuster. Vol. 26. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, pp. 49–56.

Köster, J. and S. Rahmann (2012b). "Snakemake - a scalable bioinformatics workflow engine". In: *Bioinformatics* 28.19, pp. 2520–2522.

Köster, J. and S. Rahmann (2014). "Massively parallel read mapping on GPUs with the q -group index and PEANUT". In: *PeerJ* 2:e606.

Köster, J., E. Zamir, and S. Rahmann (2012). "Efficiently mining protein interaction dependencies from large text corpora". In: *Integrative Biology* 4.7, pp. 805–812.

Lam, H. Y. K., M. J. Clark, R. Chen, R. Chen, G. Natsoulis, M. O'Huallachain, F. E. Dewey, L. Habegger, E. A. Ashley, M. B. Gerstein, A. J. Butte, H. P. Ji, and M. Snyder (2012). "Performance comparison of whole-genome sequencing platforms". In: *Nature Biotechnology* 30.1, pp. 78–82.

Lander, E. S. et al. (2001). "Initial sequencing and analysis of the human genome". In: *Nature* 409.6822, pp. 860–921.

Langmead, B. and S. L. Salzberg (2012). "Fast gapped-read alignment with Bowtie 2". In: *Nature Methods* 9.4, pp. 357–359.

Levenshtein, V. I. (1966). "Binary Codes Capable of Correcting Deletions, Insertions and Reversals". In: *Soviet Physics Doklady* 10, p. 707.

Li, H. (2011). "Improving SNP discovery by base alignment quality". In: *Bioinformatics* 27.8, pp. 1157–1158.

Li, H. (2010). *Mathematical Notes on SAMtools Algorithms*. URL: `http://lh3lh3.users.sourceforge.net/download/samtools.pdf` (visited on 04/01/2014).

Li, H. (2013). "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM". In: *arXiv:1303.3997*.

Li, H. and R. Durbin (2009). "Fast and accurate short read alignment with Burrows-Wheeler transform". In: *Bioinformatics* 25.14, pp. 1754–1760.

Li, H. and R. Durbin (2010). "Fast and accurate long-read alignment with Burrows-Wheeler transform". In: *Bioinformatics* 26.5, pp. 589–595.

Li, H., J. Ruan, and R. Durbin (2008). "Mapping short DNA sequencing reads and calling variants using mapping quality scores". In: *Genome Research* 18.11, pp. 1851–1858.

Li, H., B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin (2009). "The Sequence Alignment/Map Format and SAMtools". In: *Bioinformatics* 25.16, pp. 2078–2079.

Lin, E. Y.-H. (1998). "A bibliographical survey on some well-known non-standard knapsack problems". In: *INFOR* 36.4, pp. 274–317.

Liu, C.-M., T. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, R. Li, and T.-W. Lam (2012). "SOAP3: ultra-fast GPU-based parallel alignment tool for short reads". In: *Bioinformatics* 28.6, pp. 878–879.

Liu, Y., B. Popp, and B. Schmidt (2014). "CUSHAW3: Sensitive and Accurate Base-Space and Color-Space Short-Read Alignment with Hybrid Seeding". In: *PLoS ONE* 9.1, e86869.

Liu, Y. and B. Schmidt (2014). "CUSHAW2-GPU: Empowering Faster Gapped Short-Read Alignment Using GPU Computing". In: *IEEE Design Test* 31.1, pp. 31–39.

Liu, Y. and B. Schmidt (2012). "Long read alignment based on maximal exact match seeds". In: *Bioinformatics* 28.18, pp. i318–i324.

Luo, R., T. Wong, J. Zhu, C.-M. Liu, X. Zhu, E. Wu, L.-K. Lee, H. Lin, W. Zhu, D. W. Cheung, H.-F. Ting, S.-M. Yiu, S. Peng, C. Yu, Y. Li, R. Li, and T.-W. Lam (2013). "SOAP3-dp: Fast, Accurate and Sensitive GPU-Based Short Read Aligner". In: *PLoS ONE* 8.5, e65632.

Marschall, T. and A. Schönhuth (2013). *Sensitive Long-Indel-Aware Alignment of Sequencing Reads*. arXiv e-print 1303.3520.

Marschall, T., I. G. Costa, S. Canzar, M. Bauer, G. W. Klau, A. Schliep, and A. Schönhuth (2012). "CLEVER: clique-enumerating variant finder". en. In: *Bioinformatics* 28.22, pp. 2875–2882.

Martin, M. (2014). "Algorithms and tools for the analysis of high throughput DNA sequencing data". PhD thesis. TU Dortmund.

Martin, M., L. Maßhöfer, P. Temming, S. Rahmann, C. Metz, N. Bornfeld, J. van de Nes, L. Klein-Hitpass, A. G. Hinnebusch, B. Horsthemke, D. R. Lohmann, and M. Zeschnigk (2013). "Exome sequencing identifies recurrent somatic mutations in EIF1AX and SF3B1 in uveal melanoma with disomy 3". In: *Nature Genetics* 45.8, pp. 933–936.

McLaren, W., B. Pritchard, D. Rios, Y. Chen, P. Flicek, and F. Cunningham (2010). "Deriving the consequences of genomic variants with the Ensembl API and SNP Effect Predictor". In: *Bioinformatics* 26.16, pp. 2069–2070.

Mealy, G. H. (1955). "A Method for Synthesizing Sequential Circuits". In: *Bell System Technical Journal* 34.5, pp. 1045–1079.

Mesirov, J. P. (2010). "Accessible Reproducible Research". In: *Science* 327.5964, pp. 415–416.

Mounie, G., C. Rapine, and D. Trystram (1999). "Efficient Approximation Algorithms for Scheduling Malleable Tasks". In: *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '99. ACM, pp. 23–32.

Mullis, K. B., F. Ferre, and R. A. Gibbs (1994). *The Polymerase Chain Reaction*. Birkhäuser Verlag. 550 pp.

Myers, G. (1999). "A fast bit-vector algorithm for approximate string matching based on dynamic programming". In: *J. ACM* 46.3, pp. 395–415.

Müller, P., G. Parmigiani, and K. Rice (2006). *FDR and Bayesian Multiple Comparisons Rules*. Working Paper 115. Johns Hopkins University, Dept. of Biostatistics Working Papers.

Navarro, G. and M. Raffinot (2008). *Flexible Pattern Matching Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press.

Needleman, S. B. and C. D. Wunsch (1970). "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of Molecular Biology* 48.3, pp. 443–453.

Oinn, T., M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li (2004). "Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows". In: *Bioinformatics* 20.17, pp. 3045–3054.

Park, P. J. (2009). "ChIP–seq: advantages and challenges of a maturing technology". In: *Nature Reviews Genetics* 10.10, pp. 669–680.

Patterson, M., T. Marschall, N. Pisanti, L. v. Iersel, L. Stougie, G. W. Klau, and A. Schönhuth (2014). "WhatsHap: Haplotype Assembly for Future-Generation Sequencing Reads". In: *Research in Computational Molecular Biology*. Ed. by R. Sharan. Lecture Notes in Computer Science 8394. Springer International Publishing, pp. 237–249.

Rahmann, S., M. Martin, J. H. Schulte, J. Köster, T. Marschall, and A. Schramm (2013). "Identifying transcriptional miRNA biomarkers by integrating high-throughput sequencing and real-time PCR data". In: *Methods* 59.1, pp. 154–163.

Rasmussen, K. R., J. Stoye, and E. W. Myers (2006). "Efficient q-gram filters for finding all epsilon-matches over a given length". In: *Journal of computational biology: a journal of computational molecular cell biology* 13.2, pp. 296–308.

Reif, J. H. (1993). *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers In. 1011 pp.

Rescheneder, P., A. von Haeseler, and F. J. Sedlazeck (2012). "MASon: Million Alignments in Seconds - A Platform Independent Pairwise Sequence Alignment Library for next Generation Sequencing Data." In: *BIOINFORMATICS*. Ed. by J. Schier, C. M. B. A. Correia, A. L. N. Fred, and H. Gamboa. SciTePress, pp. 195–201.

Roberts, A. and L. Pachter (2013). "Streaming fragment assignment for real-time analysis of sequencing experiments". In: *Nature Methods* 10.1, pp. 71–73.

Ross, M. G., C. Russ, M. Costello, A. Hollinger, N. J. Lennon, R. Hegarty, C. Nusbaum, and D. B. Jaffe (2013). "Characterizing and measuring bias in sequence data". In: *Genome Biology* 14.5, R51.

Sadedin, S. P., B. Pope, and A. Oshlack (2012). "Bpipe: a tool for running and managing bioinformatics pipelines". In: *Bioinformatics* 28.11, pp. 1525–1526.

Salomon, D. (1998). *Data compression: the complete reference*. Springer. 427 pp.

Sanger, F., S. Nicklen, and A. R. Coulson (1977). "DNA sequencing with chain-terminating inhibitors". In: *Proceedings of the National Academy of Sciences of the United States of America* 74.12, pp. 5463–5467.

Saunders, C. T., W. S. W. Wong, S. Swamy, J. Becq, L. J. Murray, and R. K. Cheetham (2012). "Strelka: accurate somatic small-variant calling from sequenced tumor-normal sample pairs". In: *Bioinformatics* 28.14, pp. 1811–1817.

Schramm, A., B. Schowe, K. Fielitz, M. Heilmann, M. Martin, T. Marschall, J. Köster, J. Vandesompele, J. Vermeulen, K. d. Preter, J. Koster, R. Versteeg, R. Noguera, F. Speleman, S. Rahmann, A. Eggert, K. Morik, and J. H. Schulte (2012). "Exon-level expression analyses identify MYCN and NTRK1 as major determinants of alternative exon usage and robustly predict primary neuroblastoma outcome". In: *British Journal of Cancer* 107.8, pp. 1409–1417.

Schramm, A., J. Köster, T. Marschall, M. Martin, M. Schwermer, K. Fielitz, G. Büchel, M. Barann, D. Esser, P. Rosenstiel, S. Rahmann, A. Eggert, and J. H. Schulte (2013). "Next-generation RNA sequencing reveals differential expression of MYCN target genes and suggests the mTOR pathway as a promising therapy target in

MYCN-amplified neuroblastoma". In: *International Journal of Cancer* 132.3, E106–E115.

Sedlazeck, F. J., P. Rescheneder, and A. von Haeseler (2013). "NextGenMap: fast and accurate read mapping in highly polymorphic genomes". In: *Bioinformatics* 29.21, pp. 2790–2791.

Shah, S. P., D. Y. He, J. N. Sawkins, J. C. Druce, G. Quon, D. Lett, G. X. Zheng, T. Xu, and B. F. Ouellette (2004). "Pegasys: software for executing and integrating analyses of biological sequences". In: *BMC Bioinformatics* 5.1, p. 40.

Shendure, J. and H. Ji (2008). "Next-generation DNA sequencing". In: *Nature Biotechnology* 26.10, pp. 1135–1145.

Siragusa, E., D. Weese, and K. Reinert (2013). "Fast and accurate read mapping with approximate seeds and multiple backtracking". In: *Nucleic Acids Research* 41.7, e78–e78.

Smith, T. and M. Waterman (1981). "Identification of common molecular subsequences". In: *Journal of Molecular Biology* 147.1, pp. 195–197.

Tanaka, M. and O. Tatebe (2010). "Pwrake: a parallel and distributed flexible workflow management tool for wide-area data intensive computing". In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. HPDC '10. ACM, pp. 356–359.

Taura, K., T. Matsuzaki, M. Miwa, Y. Kamoshida, D. Yokoyama, N. Dun, T. Shibata, C. S. Jun, and J. Tsujii (2013). "Design and Implementation of GXP Make - A Workflow System Based on Make". In: *Future Generation Compututer Systems* 29.2, pp. 662–672.

The 1000 Genomes Project Consortium (2012). "An integrated map of genetic variation from 1,092 human genomes". In: *Nature* 491.7422, pp. 56–65.

Trapnell, C., L. Pachter, and S. L. Salzberg (2009). "TopHat: discovering splice junctions with RNA-Seq". In: *Bioinformatics* 25.9, pp. 1105–1111.

Trapnell, C., B. A. Williams, G. Pertea, A. Mortazavi, G. Kwan, M. J. van Baren, S. L. Salzberg, B. J. Wold, and L. Pachter (2010). "Transcript assembly and quantification by RNA-Seq reveals unannotated transcripts and isoform switching during cell differentiation". In: *Nature Biotechnology* 28.5, pp. 511–515.

Vanderster, D., N. Dimopoulos, and R. Sobie (2006). "Metascheduling Multiple Resource Types Using the MMKP". In: *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*. GRID '06. IEEE Computer Society, pp. 231–237.

Wakeley, J. (2008). *Coalescent Theory: An Introduction*. Roberts & Co. 328 pp.

Wang, Z., M. Gerstein, and M. Snyder (2009). "RNA-Seq: a revolutionary tool for transcriptomics". In: *Nature Reviews Genetics* 10.1, pp. 57–63.

Wasserman, L. (2004). *All of Statistics: A Concise Course in Statistical Inference*. Springer. 442 pp.

Weese, D., M. Holtgrewe, and K. Reinert (2012). "RazerS 3: Faster, fully sensitive read mapping". In: *Bioinformatics* 28.20, pp. 2592–2599.

Ziv, J. and A. Lempel (1977). "A universal algorithm for sequential data compression". In: *IEEE Transactions on Information Theory* 23.3, pp. 337–343.

Zook, J. M., B. Chapman, J. Wang, D. Mittelman, O. Hofmann, W. Hide, and M. Salit (2014). "Integrating human sequence data sets provides a resource of benchmark SNP and indel genotype calls". In: *Nature Biotechnology* 32.3, pp. 246–251.