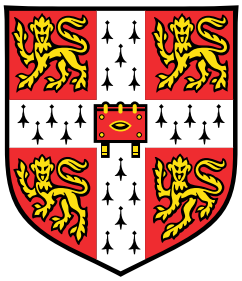
Efficient analysis and storage of large-scale genomic data



Marcus D. R. Klarqvist

Department of Genetics
University of Cambridge

This dissertation is submitted for the degree of
Doctor of Philosophy

Gonville and Caius

September 2019

I dedicate this thesis to my family: Karin, Dick, Lars, Isabell, and Alva

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Marcus D. R. Klarqvist
September 2019

Abstract

Efficient analysis and storage of large-scale genomic data

Marcus. D. R. Klarqvist

The impending advent of population-scaled sequencing cohorts involving tens of millions of individuals with matched phenotypic measurements will produce unprecedented volumes of genetic data. Storing and analysing such gargantuan datasets places computational performance at a pivotal position in medical genomics. In this thesis, I explore the potential for accelerating and parallelizing standard genetics workflows, file formats, and algorithms using both hardware-accelerated vectorization, parallel and distributed algorithms, and heterogenous computing.

First, I describe a novel bit-counting operation termed the *positional population-count*, which can be used together with succinct representations and standard efficient operations to accelerate many genetic calculations. In order to enable the use of this new operator and the canonical *population count* on any target machine I developed a unified low-level library using CPU dispatching to select the optimal method contingent on the available instruction set architecture and the given input size at run-time. As a proof-of-principle application, I apply the *positional population-count* operator to computing quality control-related summary statistics for terabyte-scaled sequencing readsets with >3,800-fold speed improvements. As another application, I describe a framework for efficiently computing the cardinality of set intersection using these operators and applied this framework to efficiently compute genome-wide linkage-disequilibrium in datasets with up to 67 million samples resulting in up to >60-fold improvements in speed for dense genotypic vectors and up to >250,000-fold savings in memory and >100,000-fold improvement in speed for sparse genotypic vectors. I next describe a framework for handling the terabytes of compressed output data and describe graphical routines for visualizing long-range linkage-disequilibrium blocks as seen over many human centromeres. Finally, I describe efficient algorithms for storing and querying very large genetic datasets and specialized algorithms for the genotype component of such datasets with >10,000-fold savings in memory compared to the current interchange format.

Acknowledgements

Completing this work has been a formative experience and have been influenced by many of my previous colleagues including Carlos Caldas, Oscar Rueda, John Todd, Chris Wallace, and Oliver Burren. I am grateful to James Bonfield for always being up for talking about file formats and algorithms. His positive spirit and genuine excitement about his work has always been an inspiration. I also thank members of the GA4GH Future of VCF and File Formats working groups for helpful comments and discussions. I am grateful for collaborating with Daniel Lemire and Wojciech Muła and for sharing their considerable technical expertise with me. This work would not have been possible without my supervisor Richard Durbin who allowed me to investigate the scientific questions I am passionate about. I am thankful to members of the Durbin lab for interesting discussions over the years.

Without the support of my family, I never would have been able to purse my interests in biology and computer science. I am forever grateful to Karin, Lars, Dick, Isabell, and Alva. Jag älskar er. My partner Vicki Barber and her family Valerie, John, Thomas, Emma, Laila-Faye, and Elden have been an important part of my life during this work. Weekends with the Barbers have always been a great joy and embodied the reason for finding a good work-life balance. I am grateful for all the friends I made at Gonville & Caius and in Cambridge over the years.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

The technology involved in determining the sequence of a human genome (sequencing) has observed dramatic progress in the 15 years since first being described in in 2003. Starting in the year 1990, the international Human Genome Project, a US$2.7 billion initiative, comprised of 13 years of uninterrupted work and involved hundreds of scientists to determine the sequence and annotate the first human genome. Today, it is possible to sequence a human genome for under US$1000 dollars in under 24 hours corresponding to a 4,745-fold reduction in time and a 2.7 million-fold reduction in price compared to the Human Genome Project. This continuous drop in cost coupled with an increasing amount of evidence supporting the use of genome sequencing in clinical care has motivated several large-scale national initiatives ranging from a few hundreds of thousands up to a hundred million individuals with planned completion times in the 5-10 year range. Such large-scale cohorts will involve sequencing, phenotypic, and other data at an unprecedented scale never before seen in biology. This deluge of data, classified as 'Big Data' owing to its gargantuan volume and velocity, has galvanized scientists to begin addressing performance problems in genetics and genomics at large and have given rise to a wave of high-performance computing-related efforts in this application space.

In this chapter I will describe the history and technical evolution of nucleic acid sequencing and the current and future technical problems in storing and querying this data. I will introduce several technical aspects of high-performance computing and then provide an introduction to the problems addressed in this thesis.

1.1 Historial overview of nucleic acid sequencing

Starting with the discovery of the structure of the molecule that is the basis for heredity, DNA, in 1953[132] there has been an incessantly growing interest and demand in

sequencing this molecule. The structure revealed that a double helix, of alternating phosphate and sugar groups, are connected by different combinations of four bases: adenine (A) with thymine (T), and guanine (G) with cytosine (C). Despite a strong academic interest in sequencing DNA, the first easily parallelizable and automatable method called dideoxy sequencing was proposed in 1977 by Fred Sanger[104] and became colloquially known as Sanger sequencing. In the same year, the first human gene was isolated and sequenced[108]. Fred Sanger was almost immediately awarded his second Nobel Prize in 1980 in recognition of this discovery. His method was based on the concept of sequencing-by-synthesis where the targeted sequence is randomly interrupted by the insertion of a modified nucleic acid, the so-called chain-terminating dideoxynucleotide triphosphates (ddNTPs). In 1986, Hood et al.[117] described an improvement in the Sanger sequencing method that involved conjugating flourophores to the ddNTPs. The four modified ddNTPs (ddATP, ddGTP, ddCTP, ddTTP) were attached to different fluorescent marker that emit light at different wavelengths when excited with a laser. This modification enabled the resulting sequences to be read by a computer. These two concepts, random chain-termination and fluorophores with different emission colours, are sufficient to deduce the exact sequence of a molecule by flowing the mixture of chain-terminated sequences (of different lengths) through a gel or capillary that separate the sequences by length. In other words, sequences terminated at different positions will flow through a given point in the gel/capillary in a time proportional to their lengths (sizes) and emit color according to its bound fluorophore (ddNTP). The instrument can then simply read out the sequence of colors that correspond to the correct nucleic acid sequence.

In April 1984, George Church published a short paper called "Genomic Sequencing"[18] that outlined a new technology that could enable the sequencing of the human genome. At the end of 1984, Church was invited to a meeting sponsored by the US Department of Energy that were interested in studying somatic mutations caused by nuclear fallout following their now infamous nuclear weapons programme the Manhattan Project. The following year, in May 1985 Robert Sinsheimer, then chancellor of the University of California-Santa Cruz (UCSC), invited several scientists to discuss the prospect of undertaking this endeavour in the UCSC environment [111]. This meeting is considered the first serious proposal to sequence the human genome and set in motion what would ultimately become the international US$2.7 billion Human Genome Project (HGP) with the main intent of sequencing the 3 billion bases in the human genome. Soon after, Sanger sequencing was automated and parallelized by the private company Applied Biosystems in 1987 by the development of a high-throughput DNA sequencer using the modifications

described by Hood et al. Automation and parallelization was a key practical prerequisite that needed to be addressed prior to attempting to sequence the human genome. The HGP formally launched in 1990[101, 15] in a collaboration between the US Department of Energy and the US National Institutes of Health with a 15-year and US$3 billion budget for completing the genome sequence. This project would span 13 years, involving multiple institutes in several countries, and culminate in the completion of the human genome sequence in 2003 with an initial draft published in 2001[51].

In 1997, James L. Weber and Eugene W. Myers proposed a plan for sequencing the human genome using parallel whole-genome shotgun sequencing[133] (WGSS). The WGSS paradigm involves sequencing many overlapping DNA fragment in parallel and then reconstruct the original sequence using computer software in a process known as assembly. Many small fragments are assembled into contiguous stretches of sequence (contigs) that are in turn connected together using paired reads[35] in a process known as scaffolding. Paired end sequences in this context refers to reading either end of a longer sequence and thereby get long-range information, especially when the insert size is large (the distance between the ends of the two reads). The final result is a large-scale map with the correct order, sequence, and orientation for each contig. This was not a new concept at the time but had already been used by Fred Sanger as early as 1982 to determine the sequence of the λ-virus[103]. Considering the size of the human genome compared to this virus, the application of WGSS to sequencing the human genome was dismissed as being too complicated and without existing software support.

In May 1998, the private company PE Biosystems developed an automated, high-throughput capillary DNA sequencer that subsequently came to be named the ABI PRISM 3700 DNA Analyzer. In the same month, the former US National Institute of Health (NIH) biologist J. Craig Venter announces a new company named Celera that declares that it will complete the sequence of the human genome within 3 years for one-tenth the cost (US$300 million) of the public HGP effort. In 2000, Celera and its academic collaborators Gerald Rubin and the Berkeley Drosophila Genome Project published the sequence of the model organism *Drosophila melanogaster* after less than one year of work[102, 89, 5] thereby demonstrating the viability of this sequencing paradigm on larger genomes. Using this approach, Celera generated an accurate sequence of the human genome in less than a year: starting on the 8th of September, 1999 and finishing on the 17th of June, 2000[128]. Both the public HGP and the private Celera effort finished sequencing the human genome three years before the expected deadline. Today, the principles behind WGSS and paired reads remains one of the primary methods for determining the sequence of new species.

1.2 Next-generation sequencing

Following the collective effort in sequencing the human genome, there has been an explosion of technical advancements in the sequencing instruments and their chemistry. Today, it is possible to sequence a human genome for <US$1000 in 24 hours corresponding to a 4,745-fold reduction in time and a 2.7 million-fold reduction in price compared to the Human Genome Project (Figure 1.1). Sanger sequencing (dideoxy chain-termination sequencing) was the principal workhorse for over 30 years from its first description in 1977[104] until around 2008. Around 2006, the private Cambridge-based company called Solexa described technology to massively parallelize sequencing[7] and eventually released their automated sequencer called the Genome Analyzer and published favorable results[10]. This ushered in a paradigm shift for sequencing instruments with dramatic improvements in throughput and concomitant decreases in price. This shift from Sanger sequencing to massively parallel sequencing methods became colloquially known as next-generation sequencing (NGS) or high-throughput sequencing (HTS). Solexa eventually became Illumina and is today the world's largest sequencing company.

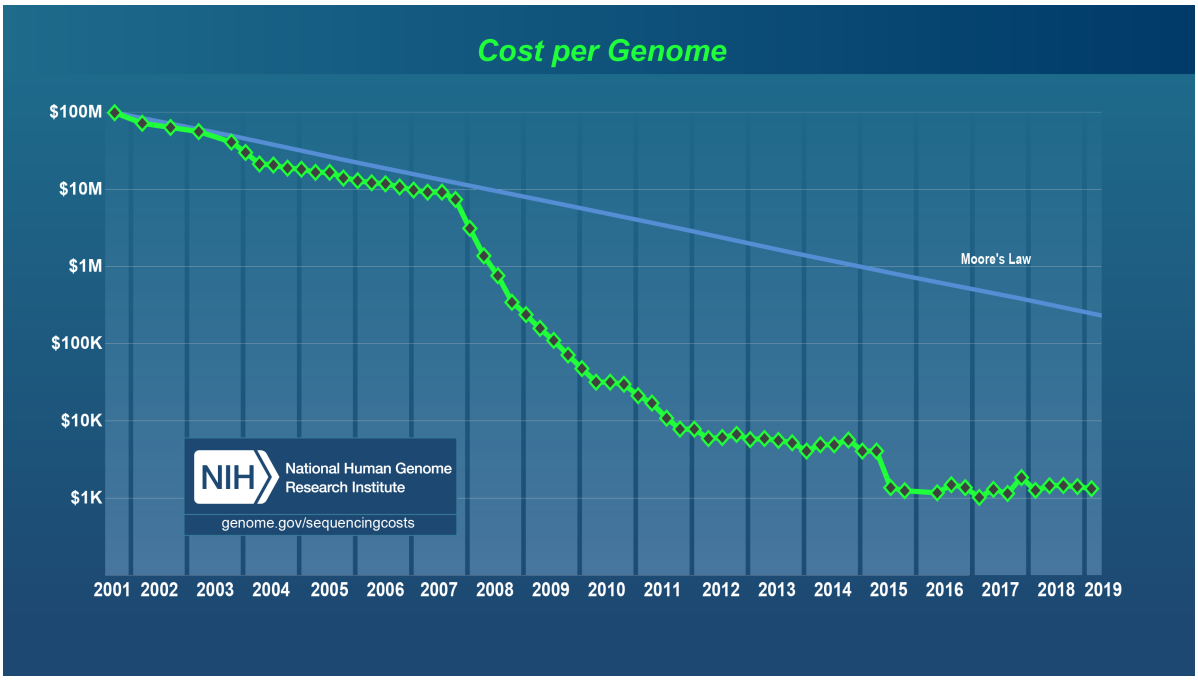


Figure 1.1 Decrease in sequencing cost of a human genome over time. The reductions in DNA sequencing cost is exceeding that of Moore's Law which predicts the doubling of transistors in compute hardware every two years. Sequencing costs are frequently compared to this prediction. The precipitous drop in sequencing cost at around the year 2008 corresponds to the transition from Fred Sanger's dideoxy chain-termination sequencing method to the so-called next-generation sequencing technologies. This graph depicts the total cost of sequencing a genome such that the total includes reagents, labor, consumables, amortization of the sequencing equipment, analysis cost related to sequence production, and other associated indirect costs. Reproduced from genome.gov.

There are two primary paradigms in next-generation sequencing technology: short-read sequencing and long-read sequencing (reviewed in [68, 43]). There are a large variety of short-read technologies but most offer very good per-base accuracy (>99%) over 100-500 base pair reads. In contrast, long reads tend to have considerably worse per-base accuracy (∼85-87% for the PacBio platform[99] and ∼90% Oxford Nanopore single-molecule platform[97]) but provide considerably longer reads with reports up to a million base pairs[92, 53]. Although long-read technologies generates much longer reads, it remains considerably more expensive and generate less total sequence information per run. Regardless of the platform used, there is an enormous amount of data generated. For example, the Illumina Hiseq-X platform generates 5.3-6 billion read pairs of 150 base pairs corresponding to 1.6-1.8 terrabases of information. This data is generated by each such system, every day.

1.3 'Big Data' in genomics

Throughout most of the life-history of genomic sequencing it has been restricted to the research space. This dramatic reduction in cost has enabled sequencing to be considered in routine and advanced clinical settings [4, 114] and there are now several emerging national initiatives that intend to sequence entire populations with around one million samples across Europe by 2022[106] (Figure 1.2 and Figure 1.3). There are estimates that genomic data for >60 million patients will have been generated before 2025[11, 114, 116]. These numbers are dwarfed by the behemothian Chinese national initiative, the China Precision Medicine Initiative, that aims to sequence 100 million genomes by 2030[21]. The Chinese government is outspending the U.S. Precision Medicine Initiative (PMI)[21] by 43-fold in terms of dollar-to-dollar spent[1]. Similarly, another large U.S. initiative, the Million Veterans Program (MVP)[41], aims to collect blood samples and health information from a million military veterans.

Currently, there are several emerging diagnostic applications in the clinic include sequencing the protein-coding part of the genome (the exome) in the course of complex disease or Mendelian disorders diagnosis with around 30% molecular diagnostic rate[115, 122, 8, 98, 93, 36, 139, 138, 110]. In one study[115], sequencing the exome in the early stages of clinical diagnosis tripled the diagnostic rate for one-third the cost per diagnosis. Owing to the continuing decrease in sequencing cost and improvement in the software for downstream analysis, whole-genome sequencing is beginning to be applied in the clinical setting. For example, the UK national initiative the 100,000 Genomes Project[127] is sequencing 100,000 genomes in a variety of clinical settings including rare disease and cancer.

[1]Estimated cost for the China Precision Medicine Initiative is US$9.2-billion over 15 years compared to the US$215-million U.S. Precision Medicine Initiative.

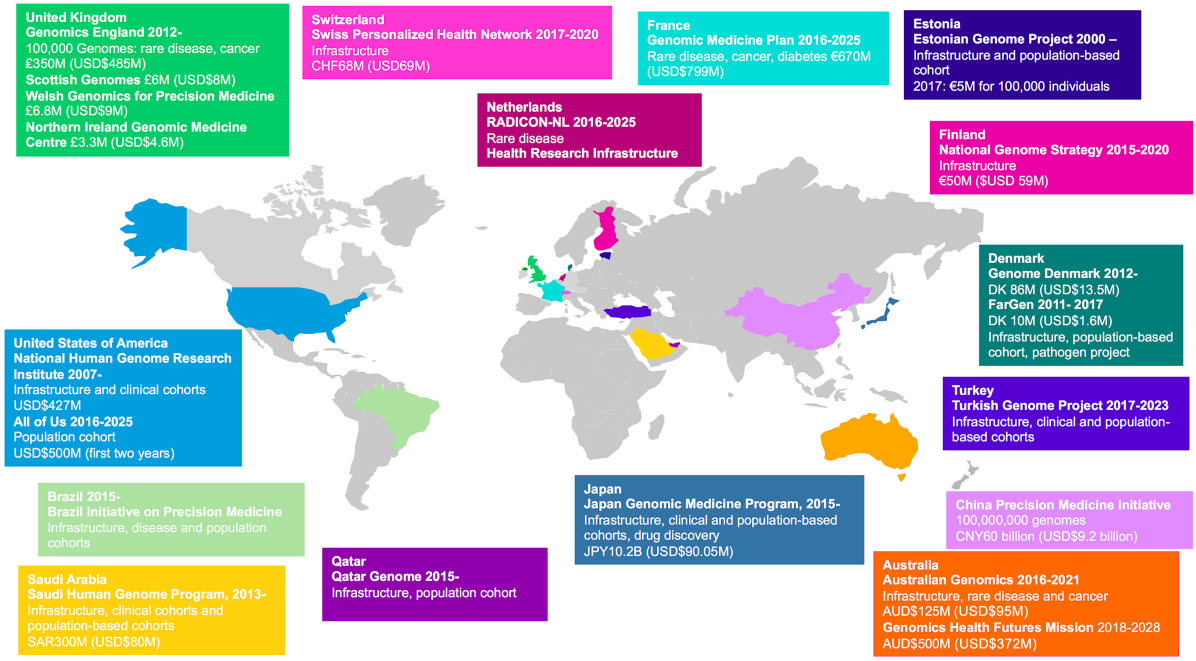


Figure 1.2 Map of Currently Active Government-Funded National Genomic-Medicine Initiatives. Reproduced from [114]

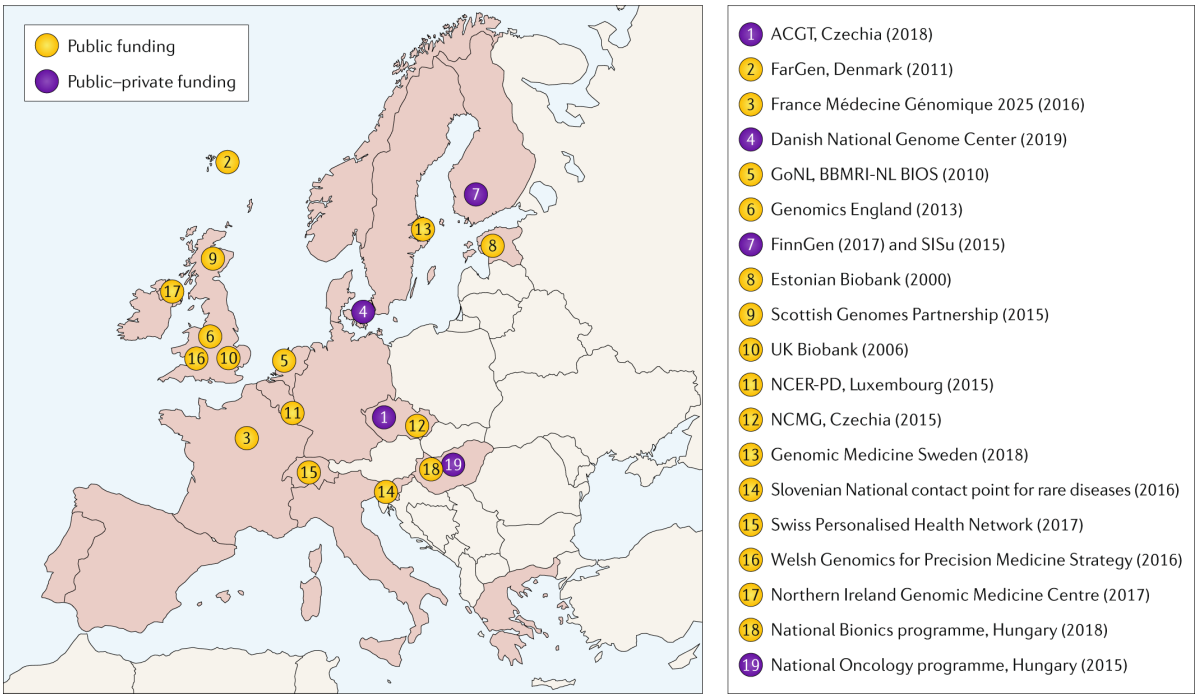


Figure 1.3 Examples of current health care-focused and genomics-based national initiative projects across ELIXIR members. Reproduced from [106]. Abbreviations: ACGT, Analysis of Czech Genome for Theranostics; BBMRI-NL, Biobanking and Biomolecular Resources Research Infrastructure - The Netherlands; BIOS, Biobank-based integrative omics study; FarGen, Faroe Genome Project; GoNL, Genome of the Netherlands; NCER-PD, National Centre of Excellence in Research on Parkinson's disease; NCMG, National Center for Medical Genomics; SISu, Sequencing Initiative Suomi.

These population-scaled sequencing datasets with matching phenotypic measurements can be classified as members of the 'Big Data' paradigm. Generally, 'Big Data' refers to datasets that are too large or complex to analyse with traditional methods and can be classified into three principal categories: volume, velocity and variety of data[38]:

- Volume: Large data volume in terms of storage space and/or number of records. As an example, the Sequence Read Archive (SRA) store >26 petabases from 100,000s of studies and the TCGA[90] and ICGC[23] have made >2.5 and 1.7 petabytes publicly available, respectively.

- Variety: Data was generated from a variety of sources and file formats and may contain multidimensional fields (tensors). If we limit our scope to the sequencing space of genomics, data is frequently generated from different types of instruments using

different techniques such as whole genome sequencing, whole exome sequencing, mutational panels, miRNA sequencing, mRNA sequencing, copy number arrays, low-pass whole genome sequencing, array-based expression, bisulphite sequencing, and many more. Clinical and phenotypic data is commonly collected in addition to the instrument data.

- Velocity: Data is generated with large frequency and/or is delivered with great frequency. Considerable advancements in sequencing instruments have pushed the total cost of sequencing down to under US$1000 in <24 hours. This can be compared to the many billions spent over 13 years of continuous work to sequence the first human genome[51]. As sequencing is beginning to move into routine healthcare, it will become increasingly important to rapidly analyse and diagnose patients for faster clinical interventions.

With the advent of population-scaled datasets and routine application of sequencing in a clinical setting, it is becoming increasingly important to maintain efficient data access and distribution. Several high-performance computing-based approaches have been suggested to address the considerable technical challenges imposed by the incessant deluge of this 'Big Data'. I will first describe how large-scale computational resources are being used now and in the future (Section 1.4.1). Next, I will describe how to maximize computing within a single processor (Section 1.4.2) and guarantee it will always be executable regardless of hardware (Section 1.4.3). Lastly, I will describe how to partition a large problem into many smaller problems (Section 1.4.4).

1.4 High-performance computing

High-performance computing (HPC) generally refers to massively parallel designs of interconnected computers with the sole intent of expediting computation of difficult tasks. Parallel computing generally implies some partitioning of a bigger, computationally intractable, problem into smaller more easily solved problems.

1.4.1 Cloud and cluster computing

Cluster computing involves linking together a large number of inexpensive commodity components such as processors (CPUs) or graphical processing units (GPUs) on a shared network. The primary intent is to improve computational throughput of parallelizable problems by using the collective compute of a multitude of relatively weak but cheap

components in lieu of fewer but more powerful but expensive components. Intrinsic to this modular paradigm is the ability to keep adding components as they become available. As a consequence of this, the resulting heterogenous cluster may contain different microarchitectures and instruction-set architectures (ISA) and/or other hardware incompatibilities.

Recently, there has been a trend for replacing in-house compute resources with cloud computing where computational resources are rented in a pay-as-you-go fashion (reviewed in [62]). This computing paradigm enables end-users to rent the exact desired resources without dealing with owning and maintaining those resources. Hardware resources are rented out as virtualized slices called instances that can be tailored to the exact resource requirements of a given task. This elasticity is a major advantage of cloud computing: from the short single job to tens of thousands of processor-intensive jobs on multiple nodes. Until recently, cloud computing was primarily a commercial enterprise with three principal providers: Amazon Web Services (AWS), Google Cloud Platform, and Microsoft Azure. Recently, several academic efforts have been made including EMBL-EBI Embassy Cloud and the Open Science Data Cloud. Although academic institutions are beginning to move into the cloud computing space, commercial cloud providers have several orders of magnitude more computers in their arsenal with the result of less waiting time for resource allocation and in turn faster results.

Potentially, the greatest contributions of cloud computing could be the immediate accessibility to archived data for reanalysis without first forcing end-users to download the target data to their local system. Securing storage space, allocating resources for downloading the target data, setting up software for the analysis, and then performing the analysis are all expensive steps that can be completely or partially circumvented in the cloud computing paradigm. This is especially empowering for smaller laboratories without significant computational resources. Realising this, the Sequence Read Archive (SRA) a hosting provider that store >26 petabases of data obtained from several 100,000s of studies is currently migrating to a cloud environment[2]. Frequently used datasets such as the 1000 Genomes Project (1KGP)[123], Genome Aggregation Database (gnomAD)[55], International Cancer Genome Consortium (ICGC)[23], The Cancer Genome Atlas (TCGA)[90], Trans-Omics for Precision Medicine (TOPMed)[121] can be copied to geographically diverse locations to enable efficient access irrespective of the physical location of the investigator.

[2] https://www.nlm.nih.gov/news/NLM_Moves_SRA_Cloud.html. Last accessed: 9th September, 2019.

Despite all the promises of cloud and cluster computing there are several current and forthcoming challenges. Firstly, heterogeneous compute clouds and clusters with mixed hardware introduce several important challenges: (1) different nodes/instances may have incompatible ISAs resulting in binary incompatibility for compiled code; (2) problems must be parallelizable at least at the single-node level but preferably the multi-node level; (3) heterogenous components could significantly skew load-balancing and result in inefficient computing. Secondly, the incessant deluge of genomics data ('Big Data') is posing both an immediate and future limitation to efficient analysis and storage of genomics data. I will discuss these challenges in the next sections.

1.4.2 Instruction level parallelism

Modern commodity processors rely heavily on a variety of parallelism paradigms to achieve peak performance. This include the well-known task parallelism where tasks are concurrently performed by having each processor executing a task on a different thread. Less known is the lower level parallelism referred to as instruction level parallelism where a large number of data is loaded into wide data registers and reduce the number of instructions required to complete a task by processing multiple machine words simultaneously (Figure 1.4). In many cases, it is possible to execute multiple instructions per CPU cycle (clocktick). This form of parallelism is called SIMD (single instruction, multiple data) and its instructions are available on most commodity processors. Currently, registers are 128, 256, or 512 bits wide and use one of the current instruction set architectures (ISAs) called `SSE`, `SSE2`, `SSE3`, `SSE4.1`, `SSE4.2`, `AVX`, `AVX2`, and a variety of `AVX512` instruction sets (Figure 1.5). Depending on the instruction set, any multiple of 8-bit, 16-bit, 32-bit, 64-bit, 128-bit, or 256-bit that adds to 128, 256, 512 bits is allowed but not all instructions are available in all size combinations. Although conceptually simple to understand, SIMD-based acceleration is generally difficult to program in practice. Throughout this thesis, SIMD-accelerated subroutines play a major role in achieving peak performance. I briefly describe the historical timeframes and major changes introduced in the most recent instruction sets starting with `SSE2` (ignoring the legacy `MMX` ISA and the `SSSE3` ISA):

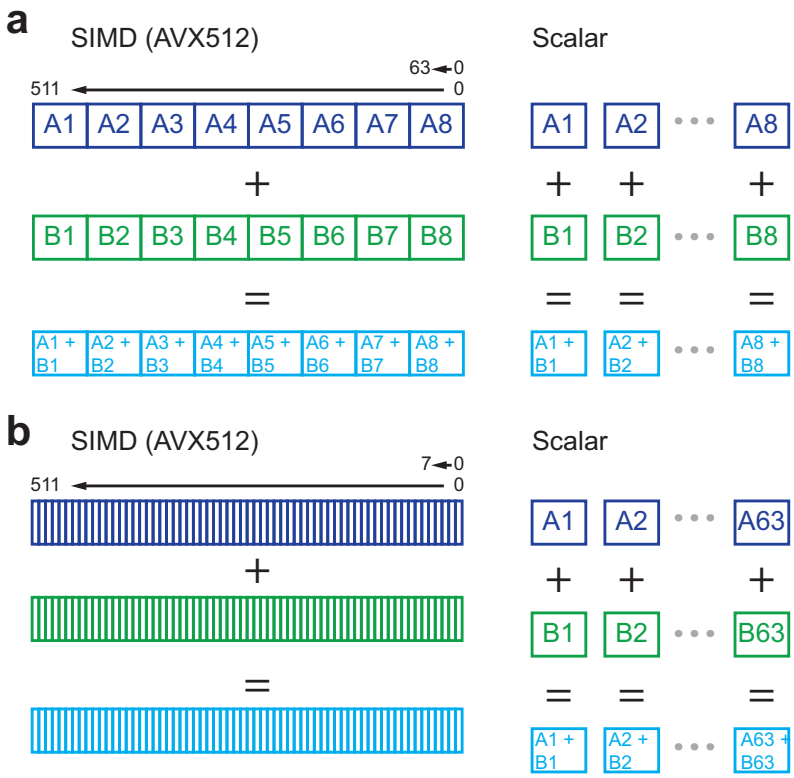


Figure 1.4 Scalar vs SIMD. a) SIMD accelerate computation by applying a single instruction on many packed machine words simultaneously. In this example, eight 64-bit words in two registers are added together. b) In this example, sixty-four 8-bit values are added together. These two operations have identical throughput.

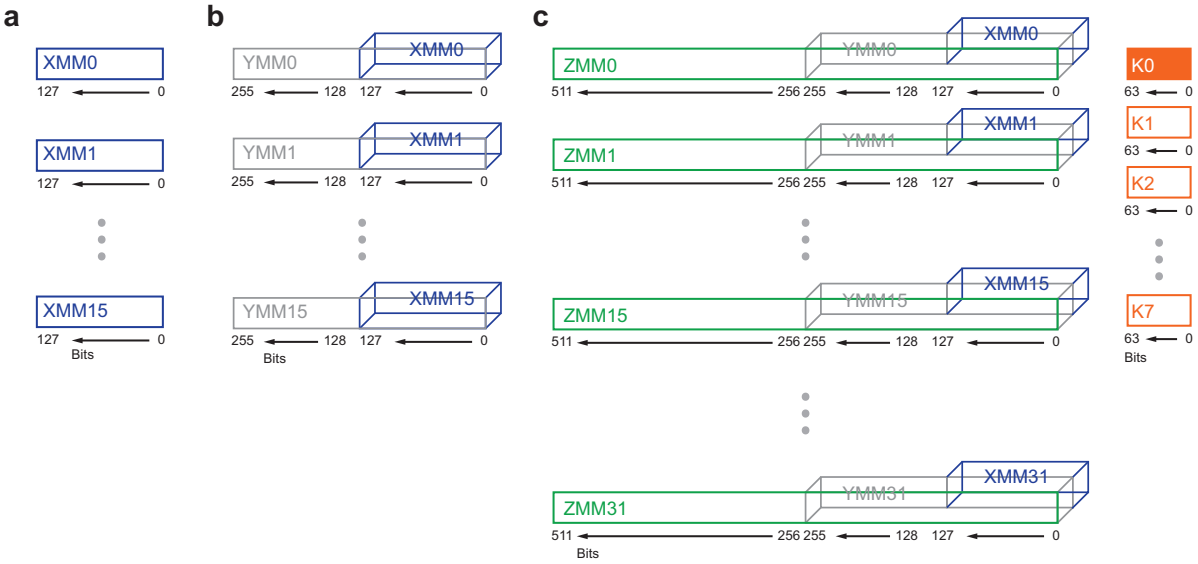


Figure 1.5 History of SIMD registers. a) 128-bit `XMM` registers used up until `AVX`. There were 8 registers in `x86` mode and 16 registers in `x86-64` mode. b) Starting with `AVX`, registers were widened to 256-bits and named `YMM` registers. The `YMM` registers alias the previous `XMM` registers and the `VEX` coding scheme was introduced to replace the most commonly used instruction prefix bytes in order to differentiate between `YMM`- and `XMM`-based instructions. c) In `AVX512`, registers were further widened to 512 bits and an additional 16 registers were added per core. Starting with `AVX512`, there are 8 opmask registers where one (k_0) is a hard-coded constant. `ZMM` registers are encoded using the `EVEX` coding scheme.

- The Streaming SIMD Extensions (`SSE`) instruction set, also called Katmai New Instructions (KNI) by its Intel project codename, was first released in 1999 and operates primarily on 32-bit single-precision floating point data. SSE originally had eight 128-bit registers called `XMM0`, `XMM1`, ..., `XMM7` but was later extended with another eight registers when operating in 64-bit mode (`XMM1`, ..., `XMM15`).

- `SSE2` (2000), also called Willamette New Instructions (WNI) by its Intel project codename, introduced with Pentium 4, provided major enhancements to SSE and introduced double-precision (64-bit) floating point for all SSE operations, and MMX integer operations on 128-bit `XMM` registers.

- `SSE3` (2004), also called Prescott New Instructions (PNI) by its Intel project codename, primarily introduced the capability to work horizontally in a register in contrast to the more limiting vertical operations of the previous SSE ISAs.

- SSE4 (generally referred to as SSE4.1; 2006), also called Penryn New Instructions (PNI) by its Intel project codename, was the first developed ISA without any explicit multimedia applications in mind. It features a number of instructions that takes a register and a constant field as arguments and implicitly take XMM0 as a third operand. Importantly, this ISA is the first to introduce the POPCNT and LZCNT instructions that compute the number of set bits in a word and the number of leading bits before the first bit in a machine word, respectively. I describe a software implementation of the POPCNT instruction using the AVX512BW/AVX512VBMI ISAs and a novel bitwise operation called the positional population count (POSPOPCNT) in Chapter 2.

- SSE4.2 (2008) introduced a collection of new instructions called STTNI (String and Text New Instructions) that perform string and character searches and comparisons on two operands 128 bits at a time. These string instructions are prefixed PCMP* and are used in Chapter 3 for accelerating non-string-related problems.

- Advanced Vector Extensions (AVX, described in 2008, first implementation in 2011), also called Gesher New Instructions (GNI) by its Intel project codename, is an advanced version of SSE that widened the data path from 128 bits to 256 bits and introduced fused multiply-accumulate (FMA) operations. Following the widening of the SIMD registers the XMM0-XMM15 registers were renamed to YMM0-YMM15. These 16 256-bit YMM registers are accompanied by a 32-bit control/status register called MXCSR that have bits indicating floating-point exceptions, invalid operations, divide-by-zero, overflow, underflow, and precision. Importantly, the 256-bit YMM registers alias the legacy 128-bit XMM registers allowing SSE instructions to be utilized through the VEX coding scheme (opcode) that operate on the lower half (128 bits) of the 256-bit YMM registers (Figure 1.5).

- Advanced Vector Extensions 2 (AVX2), released in 2011, also called Haswell New Instructions by its Intel project codename, expanded most vector AVX instructions from single float and double float to include signed and unsigned byte (B, 8-bit), word (W, 16-bit), doubleword (DW, 32-bit), quadword (QW, 64-bit), and doublequadword (DQ, 128-bit) lengths. Importantly, this instruction set include instructions for logical left and right shifts of vectors (VPSLLV*, VPSRLV*, and VPSRAVD) that is used extensively throughout this thesis (e.g. Chapter 2 and Chapter 3).

- AVX-512 (proposed in 2013, first implementation in 2015) extends the register width to 512-bit from the 256-bit in the AVX/AVX2 ISA. The `YMM` registers `YMM0-YMM15` were renamed to `ZMM` registers and extended by another 16 to a total of 32 (`ZMM0-ZMM31`). The `ZMM` registers alias the `YMM` registers enabling access to previous registers through the `EVEX` coding scheme (opcode). The `EVEX` coding scheme also enables access to the 16 additional registers `XMM16-XMM31` and `YMM16-YMM31`. Breaking convention, AVX-512 consists of separate extensions resulting in CPUs with different level of support: `F` (Foundation), `CD` (Conflict Detection Instructions), `ER` (Exponential and Reciprocal Instructions), `PF` (Prefetch Instructions), `4FMAPS` (Fused Multiply Accumulation Packed Single precision), `4VNNIW` (Vector Neural Network Instructions Word variable precision), `VPOPCNTDQ` (`POPCNT` instruction for DW and QW), `VL` (Vector Length Extensions), `DQ` (DW and QW Instructions), `BW` (Byte and Word Instructions), `IFMA` (Integer Fused Multiply Add), `VBMI` (Vector Byte Manipulation Instructions), `VNNI` (Vector Neural Network Instructions), `VBMI2` (Vector Byte Manipulation Instructions 2), `BITALG` (Bit Algorithms), `VPCLMULQDQ` (carry-less multiplication of QWs), `GFNI` (Galois Field New Instructions), and `VAES` (Vector AES instructions). Many instructions in AVX512 are simply `EVEX` versions of previous SSE/AVX instructions. The AVX512 ISA also adds 7 new opmask registers for masking most AVX-512 instructions (`k0-k7` where `k0` is a hardcoded constant). A bit-flag controls the behaviour of the opmasks: unset (zero) mask bits results in the corresponding word in the register to be zeroed out (unselected). Reciprocally, set (one) mask bits keeps the corresponding words untouched such that they can be merged. The opmask registers have their own extension of instructions that are `VEX`-encoded. Initially, opmask instructions were all word (16-bit) versions but was later extended in `AVX-512DQ` with 8-bit (B) support and `AVX-512BW` added 32-bit (DW) and 64-bit (QW) support. I will not further describe the opmask instruction set as it is not used in any great extent in this thesis. The extensions `F`, `BW`, and `VMBI` are used in Chapter 2 and Chapter 3. There are currently no consumer-level CPUs that support the AVX512 ISA. Intel are in the process of shipping their first consumer-level CPU with AVX512 support, the 10nm Ice Lake microprocessor, and is expected to release a desktop/workstation-based version during 2020.

It is becoming increasingly uncommon to develop SIMD-accelerated code that target ISAs prior to `SSE4.1` (released in 2006) as most commodity processors available have access to this ISA that also includes the important hardware instructions `POPCNT` and

`LZCNT` that compute the population count (count number of bits set to 1) and the leading zero count (number of zeros before the first set bit), respectively.

Another important aspect of maximizing SIMD-acceleration is memory alignment. Data is said to be memory aligned when the data operated upon is stored on a n-byte boundary. For example, when loading 128/256/512-bit data into a `*MM` register from a data source that is 128/256/512-bit aligned then the data is called aligned. Up until the AVX ISA, all SSE operations required memory alignment unless explicitly declared as unaligned. Starting in AVX, many memory alignment requirements were relaxed but generally operate faster when aligned. Compounding the difficulties in developing for multiple ISAs is that `XMM`, `YMM`, and `ZMM` registers have different optimal memory alignment requirements with 16-byte, 32-byte, and 64-byte boundaries, respectively.

Compiling against a target instruction set provides maximum performance but essentially binds the code to a particular subset of hardware. Addressing this shortcoming, it is possible to develop functions for multiple target architectures and select the supported function during run-time execution rather than during compilation time. This approach involves a technique called CPU dispatching and will be discussed next.

1.4.3 CPU dispatching and function multiversioning

Developing software optimized at the instruction level with support for multiple instruction sets involves the introduction of branching points in the code for sections that involve instruction sets on different subsets of hardware. For example, one subroutine can have code optimized for the latest Intel instruction set and another branch for the latest NEON instruction set. The software should then detect the availability of a given instruction set on the CPU it is running on during execution time and select the optimal binary path. This is known as CPU dispatching and is implemented as function multiversioning (FMV) in the GCC compiler. Throughout this thesis, I use CPU dispatching whenever possible with the intent of supporting CPUs without modern SIMD ISAs and provide targets for the SSE4.2, AVX2, and AVX512 ISAs.

1.4.4 Parallelization paradigms

Once instruction level optimizations have been exhausted the next natual step is to explore different parallelism paradigms where a large computational problem is addressed concurrently using as much compute resource as possible. The conventional way of querying and handling very large datasets is by divide-and-conquer-based approaches where a problem is partitioned into multiple smaller subproblems that can be tackled

in parallel on multiple cores and/or multiple compute nodes. In general, there are two primary paradigms for partitioning an application into concurrent parts to take advantage of parallel computing: data parallelism and task parallelism. Data parallelism involves the repeated and simultaneous execution of a single instruction or operation on different data. The simplest data parallel model is arguably the scatter-gather pattern for tasks that can be partitioned into independent subtasks without cross-dependency (embarrassingly parallel tasks). In this model, a problem is partitioned into subproblems (scatter) and solved independently on different CPUs or compute nodes followed by a final merging procedure (gather). In task parallelism, different processes simultaneously execute different set of instructions. This paradigm generally involves different execution schedules resulting in the need to have more complex synchronization such as mutex locks and semaphores.

There are a multitude of language extensions like OpenMP for shared-memory parallelism and libraries implementing the Message Passing Interface (MPI) for cluster-level parallelism and more heavy-duty frameworks like Apache Hadoop implementing MapReduce[29]. Fortunately, almost all target applications in this thesis involve problems that are embarrassingly parallel (Chapter 2,3,4,5) or involves easily chainable tasks that can be synchronized using spin-locks or simple mutex locks (Chapter 4,5).

1.5 Genetic matrices and storage of sequence variant data

An important goal in genetics is to compute pairwise similarities between either individuals or different locations in the genome given their genetic material. Self-similarity matrices are used as either input or internally for several popular within-species model-based clustering approaches (ADMIXTURE[6] and STRUCTURE[94]) and for standard data reduction approaches such as principal component analysis (PCA). Importantly, the output of PCA or ADMIXTURE/STRUCTURE are frequently used as input for other downstream analyses such as a parameter in the generalized linear models used in genome-wide association studies (GWAS) to correct for relatedness, cryptic relatedness, and population structure[13, 129, 130, 109, 148, 76, 145, 141, 79]. Although both general math frameworks (Intel Math Kernel Library, OpenBLAS, Eigen) and genetics-specific[83, 95, 126, 22, 39] matrix computations have garnered considerable interest recently, the current approaches are either too generic and do not exploit intrinsic properties in genetic data, or, have suboptimal scaling properties as genetic datasets are growing increasingly larger. A notable exception to addressing scalability of genetic

matrices was a recent effort using (the now discontinued) manycore layout of the Intel Knights Landing architecture using 256 threads[39].

In many cases we are only interested in, or can limit our interest to, biallelic sites. In this case, each allele can be encoded using either zero (0) or one (1) for the reference or alternative allele, respectively. Without loss of generality, given a binary matrix $X^{(n\times m)} \in [0,1]\mathbb{Z}$ for n haplotypes and m variant sites, we want to compute the self-similarity matrix XX^T resulting in a new matrix $Y^{(n\times n)}$ that is equivalent to the all-vs-all inner product of vectors in X. It is worthwhile highlighting that the inner product of vectors of binary values are equivalent to the cardinality of the set intersection of those binary vectors if encoded in integer space. For example, the two binary vectors $B_1 = (0,1,0,1)$ and $B_2 = (1,1,0,1)$ can be projected into the equivalent integer sets $Z_1 = \{2,4\}$ and $Z_2 = \{1,2,4\}$. Hence, the cardinality of the set intersection of $|Z_1 \cap Z_2| = 2$ is equivalent to the inner product of B_1 and B_2 such that $B_1B_2 = 0\times 1 + 1\times 1 + 0\times 0 + 1\times 1 = 2 = |\{2,4\}| = |Z_1 \cap Z_2|$. It therefore follows that the cardinality of the sets $Z_1 \setminus Z_2$ and $Z_2 \setminus Z_1$ can be described as $|Z_1 \notin (Z_1 \cap Z_2)| = |Z_1| - B_1B_2$ and $|Z_2 \notin (Z_1 \cap Z_2)| = |Z_2| - B_1B_2$ Lastly, it follows that given a fixed universe of n values that the cardinality of the set of unobserved values are $n - (B_1B_2 + (|Z_1| - B_1B_2) + (|Z_2| - B_1B_2))$

In Chapter 3 I will describe efficient algorithms and a framework for computing this self-similarity matrix for large cohorts and then apply it to the problem of computing linkage-disequilibrium. Without loss of generality let $A = XX^T$ and $B = (\mathbf{1}y^T) + (\mathbf{1}y^T)^T$, where $X^{(n\times m)} \in [0,1]\mathbb{Z}$ is a binary matrix, $y^{(1\times m)} \in [0,n]\mathbb{Z}$ is the alternative allele counts, $\mathbf{1}^{(1\times m)}$ is a vector of ones, n is the number of chromosomes (two per individual) and m is the number of variant sites. The matrix B corresponds to the total alternative allele count per pair of variant sites and the diagonal is equal to $2y$. The 2-locus haplotype counts can then be computed as f_{11}, f_{01}, f_{10}, and f_{00} for homozygous alternative, heterozygous either (reference-alternative and alternative-reference), and homozygous reference, respectively:

1. $f_{11} = A$
2. $f_{01} = (B - A) - (\mathbf{1}y^T)$
3. $f_{10} = (B - A) - (\mathbf{1}y^T)^T$
4. $f_{00} = 2n - f_{11} - f_{01} - f_{10}$

This information is sufficient to compute the coefficient of linkage disequilibrium (D): $D = f_{11}/n - f_{01}/nf_{10}/n$.

In Chapter 5 I describe efficient algorithms for computing the self-similarity matrix X^TX for individual genomes. The self-similarity matrix is a measure of genetic relatedness

between the input samples. A simple interpretation of genetic relatedness is the average genetic distance between pairs of individuals, such as identity-by-state, which compute the proportion of sites that are shared between two chromosomes[85]. If the input matrix X corresponds to a genetic matrix of n haplotypes and m biallelic sites then we can compute its self-similarity matrix as:

$$\frac{f_{11} + f_{00}}{f_{00} + f_{01} + f_{10} + f_{11}} = \frac{f_{11} + f_{00}}{2n} \tag{1.1}$$

Interestingly, removing SNPs in high linkage-disequilibrium prior to inferring population structure with PCA result in improved results[3]. This suggest that computing XX^T, or parts of it, can be beneficial when computing X^TX and ultimately improve population structure ascertainment.

These genetic matrices quickly grows in size with increased number of samples and variant sites. With the advent of whole-genome sequenced population-scaled cohorts we expect the number of samples to easily exceed a million and the number of sites a billion to create matrices with $\geqslant 10^{15}$ cells. There is a pressing need to develop algorithms to both store and query such matrices in reasonable time.

In many cases, it is preferable to jointly store a more generic genetic matrix $X^{(n\times m)} \in \mathbb{Z}^+$, supporting any number of alternative alleles, together with additional information such as where the variant maps to in the reference genome and other meta information such as alternative allele counts. Recognizing this emerging requirement during the early stages of the 1000 Genomes Project[123], the Variant Call Format (VCF) interchange format was proposed[26]. The human-readable VCF format uses textual encodings with simple indexing to enable reasonable data access for very small data files. It was designed for projects with millions of sites but at most a few thousand samples. Addressing the inefficient storage and poor random access, a binary representation of VCF was proposed (BCF) that simply stores information in machine words in a row-centric orientation followed by compression with the general purpose framework Gzip in blocks of 2^{16} bytes. Recently, much effort have been dedicated to compressing the genetic component X stored in VCF files with much success[30, 71, 34, 31, 63, 27]. In Chapter 4 I describe several algorithms to efficiently compress and query large genetic datasets. In Chapter 5 I further explore the potential of compressing the genetic matrix component X.

1.6 Dissertation Goals, Challenges, and Contributions

The goal of this work is to explore the potential for accelerating and parallelizing standard genetics workflows, format descriptions, and algorithms using both hardware-accelerated vectorization, parallel and distributed algorithms, and heterogenous computing. Undoubtedly, the historically fast-paced and organic co-evolution of the nascent bioinformatics field and sequencing field have provided a natural hindrance to the development of efficient algorithms as new technical advancements greatly outpaced the need for stability and performance. The bioinformatics field has now reached a more stable equilibrium with respect to technical advancements, while ever larger cohorts of individuals are being sequenced that require more and more efficient frameworks for downstream analysis and storage.

There are several nascent and persistent challenges that must be addressed:

- Develop algorithms that take advantage of the most recent ISA available during run-time execution rather than during compilation-time.
- Develop algorithms that can be parallelizable over multiple processor cores.
- Develop algorithms that can be parallelizable over multiple compute nodes on a heterogeneous compute farm.
- Support different use-cases such as optimizing for storage or optimizing for query speeds.
- Support immediate integration into the existing interchange ecosystem with minimal disruption.

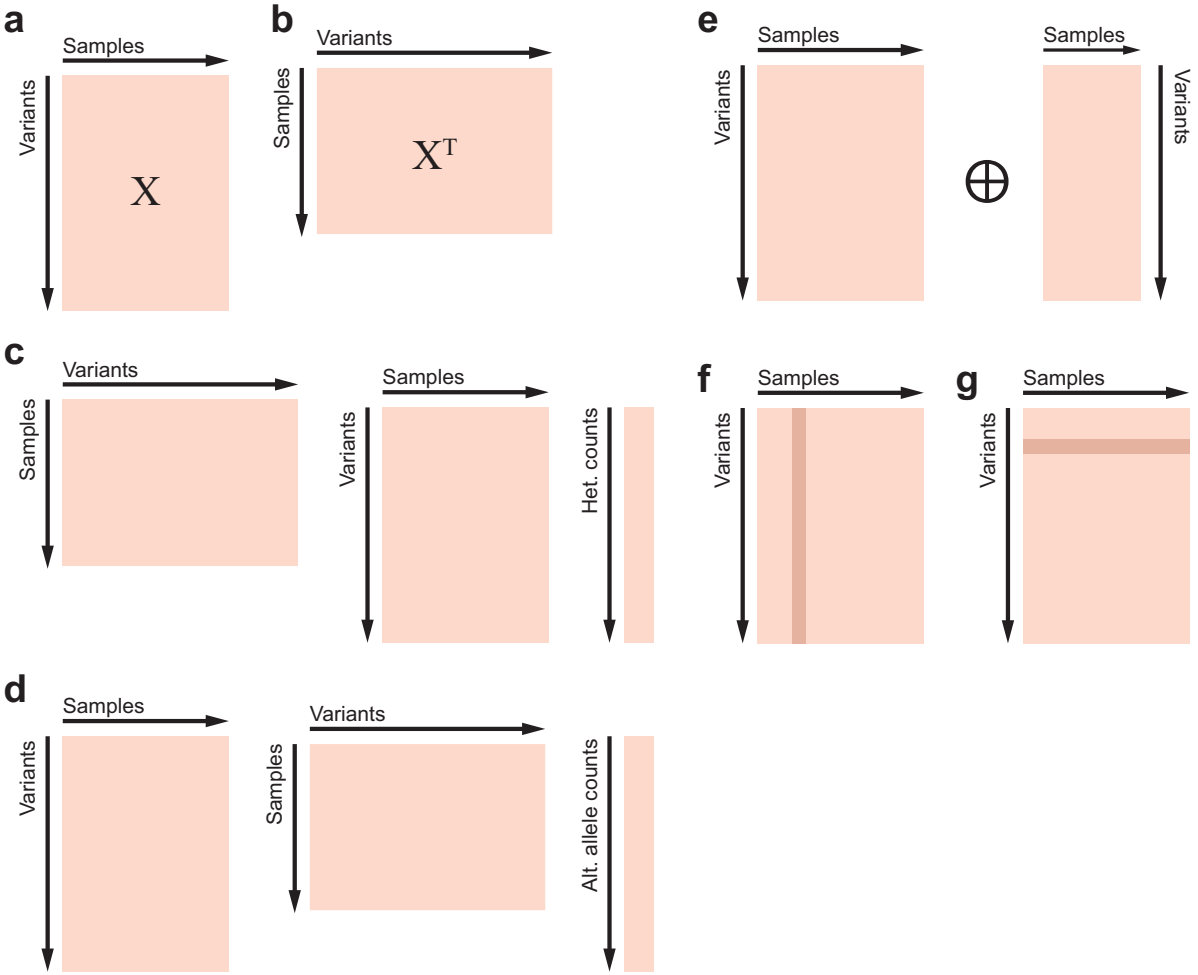


Figure 1.6 Overview of this work. a) Basic genetic matrix $X^{(n \times m)} \in \mathbb{Z}^+$ for n haplotypes/genotypes and m variant sites. b) X^T represent the transpose of the canonical representation in a). c) Computing $X^T X$ together with an auxiliary vector y of heterozygous counts per site is sufficient to compute most standard genetic similarity matrices. This is discussed in Chapter 5. d) Computing XX^T together with an auxilliary vector y of alternative allele counts per locus is sufficient to compute linkage-disequilibrium. This is discussed in Chapter 3. e) Concatenating two genetic matrices is a common performance-bounded problem and is discussed in Chapter 4 and Chapter 5. f) Efficient column projection is a core query in many applications and is discussed in Chapter 4 and Chapter 5. g) Efficient row-based slicing of genetic matrices is a core component in many queries and is discussed in Chapter 4 and Chapter 5.

In more detail, each chapter can be summarized as follows (Figure 1.6):

- In Chapter 2 I introduce the novel positional population count operator (`pospopcnt`) and develop hardware-limited implementations on most available Instruction Set

Architectures (ISA) from none to `SSE4.2` through `AVX512`. I describe a potential application by computing summary statistics for the `FLAG` field in the incumbent `SAM/BAM/CRAM` interchange format with >3,800-fold improvements in speed.

- In Chapter 3 I introduce several algorithms for efficiently compute XX^T for a binary input matrix $X^{n \times m}$ of n haplotypes and m biallelic variant sites using algorithms developed for the population count (`popcnt`) and (`pospopcnt`) library `libalgebra` described in Chapter 2. I develop a framework for applying these algorithms to the problem of computing chromosome-wide linkage-disequilibrium in large-scale cohorts up to 67 million haplotypes. This framework include a new hybrid bitmap approach (called `Storm` bitmaps) for computing the cardinality of set intersections. I further develop algorithms for storing and querying the hundreds of gigabytes of output data that is generated. Collectively, I named the software implementation `tomahawk` and its native R-bindings called `rtomahawk`. Tomahawk can use >300,000-fold less memory compared to the incumbent standard toolchain while computing identical results up to >10,000-fold faster. As in Chapter 2, all the presented algorithms can make use of the largest available ISA during run-time.

- In Chapter 4 I describe a bit-exact column-store representation for sequence variant data with powerful query capabilities called `tachyon`. I introduce a genotype-specific compression approach based on the linkage-disequilibrium-based Positional Burrows-Wheeler Transform (PBWT) and algorithms to query compressed data. I will demonstrate that the native column-projection capabilities of Tachyon can result in considerable savings in disk while supporting faster general queries compared to the incumbent VCF interchange format.

- In Chapter 5 I describe a series of algorithms for storing and querying sequence variant data for population-scaled cohorts building on ideas from Chapter 4. I describe algorithmic variations that optimize for different use-cases: (1) optimal storage, (2) balance between storage and query speed, and (3) optimal query speeds. I named the software implementation `djinn`. Djinn can achieve >300,000-fold compression compared to the incumbent VCF interchange format on large cohorts. I also describe a framework for querying the encoded data directly in compressed space in time proportional to the compressed size resulting in the considerable acceleration of general haplotype/genotype-based queries.

Chapter 2

The positional population count operation

In this chapter I will introduce several important algorithms for efficiently computing the number of set bits in a machine word in both the classic per-word orientation (`popcnt`) and the novel per-bit orientation over multiple words (`pospopcnt`). Efficient computation of the population count of a machine word is so important that most modern commodity processors have dedicated hardware instructions: `POPCNT` for x64 processors and `CNT` for the 64-bit ARM architecture. These embedded instructions are *extremely* efficient with a throughput of a single CPU cycle for a 64-bit word. A recently described family of algorithms have been shown to achieve a sustained performance of 0.5 CPU cycles / 64-bit word[88] for computing the standard population count. Building on this previous work I will describe a novel variation of the `popcnt` operation for computing the bit-wise populaton count across multiple machine words. These low-level algorithms achieve hardware limits on modern commodity processors. Methods, algorithms, and ideas developed in this chapter will be used extensively throughout the remaining chapters.

2.0.1 Collaboration Note and Author Contributions

The algorithms described in this chapter were developed in close collaboration with Daniel Lemire (D.L.) and Wojciech Muła (W.M.). M.D.R.K conceived the project and performed experiments. M.D.R.K., W.M., and D.L. co-authored the implementation. M.D.R.K. wrote this text with contributions from W.M. and D.L.

2.0.2 Summary

In several fields such as statistics, machine learning, and bioinformatics, categorical variables are frequently represented as one-hot encoded vectors. For example, given 8 distinct values, we map each value to a distinct bit in a 8-bit word. We are motivated to quickly compute statistics over such encodings.

Given a stream of k-bit words, we seek to sum the bit values at indexes 0, 1, 2, ..., $k-1$ across multiple words by computing k distinct sums. If the k-bit words are one-hot encoded then the sums corresponds to their frequencies.

This multiple-sum problem is a generalization of the population-count problem where we count the total number of set bits in independent machine words. We refer to this new problem as the *positional population-count problem*.

Using SIMD (Single Instruction, Multiple Data) instructions from recent Intel processors, we describe algorithms for computing the 16-bit position population count using about one eighth (0.125) of a CPU cycle per 16-bit word. Our best approach is about 140-fold faster than competitive code using only non-SIMD instructions in terms of CPU cycles.

As an example application in genomics, we compute a common summary statistic for >824 million sequencing reads >3,800-fold faster compared to current state-of-the-art tools.

2.1 Introduction

In many applications such as deep learning [25, 24, 144, 49], indexing [67], chemistry [44], cryptography [105], and bioinformatics [63, 27, 137, 95], it is desirable to compute the number of set bits in a computer word. This operation is referred to as the population count (`popcnt`), Hamming weight, or the sideways sum of the word. For example, the machine word 10010010 has a population count of three since there are three set bits. We have previously described efficient subroutines for computing the population count [88] of large arrays that takes advantage of SIMD (single instruction, multiple data) instructions available on most commodity processors. These instructions operate on wide data registers and reduce the number of instructions required by processing multiple machine words simultaneously.

It is common to represent categorical variables using one-hot (1-of-k) encoding [75, 87] where each categorical value maps to a corresponding bit and each word may only have a single bit set.

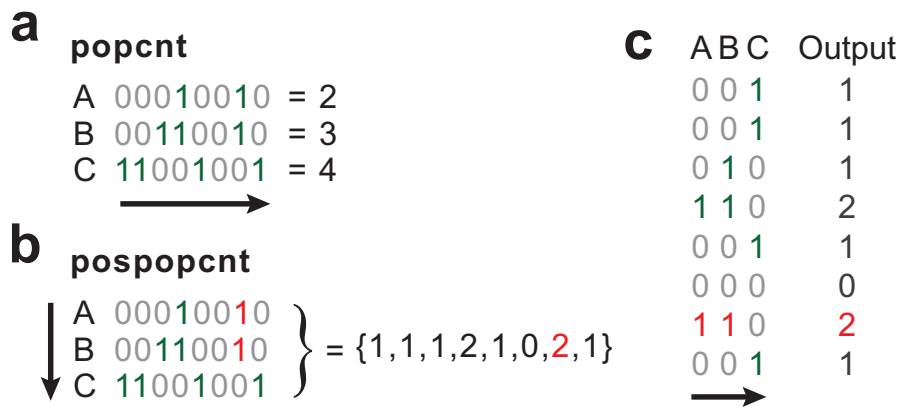


Figure 2.1 Comparing per-word population count (`popcnt`) with positional population count (`pospopcnt`). a) Set bits (ones) are counted for each of the machine words A, B, and C in a row-wise fashion and reported separately. b) In the positional popcount operation, set bits are counted in the columnar orientation over many machine words at a given position and reported as a fixed-width array equal to the length of the individual machine words. c) The positional popcount operation can be considered the implicit computation of popcounting the transposed input bit-matrix in b).

There are various generalizations such as zero one hot (up to one bit is set to one), one cold (a single zero must be present), and so forth. One of the motivations for such encodings is that many machine-learning algorithms cannot operate directly on categorical data. They require all variables to be numerical. These encodings also closely related to the concept of dummy variables in statistics where attributes (such as gender) are represented as 0s or 1s for computing purposes.

In this context, we would like to count the number of bits set at each position (first, second, ..., last). We describe a generalization of this population count operation to individual bits spanning multiple words in a columnar population count operation referred to as the *positional population count* (`pospopcnt`).

As an illustrative example, consider three given input words A, B, and C (Figure 2.1), the conventional per-word population count computes the number of set bits for each word independently (Figure 2.1a). In contrast, the positional population count operation computes the *vertical* population count over these words for independent bit positions (Figure 2.1b). Because bits are incremented independently, the positional population count operation depends on the width of the word. We describe several efficient subroutines for computing the positional population count of 16-bit words but can be applied to wider words by fragmenting them into consecutive 16-bit units.

In bioinformatics, the `FLAG` field in the incumbent SAM (Sequence Alignment/Map) interchange format [72] is defined as the bitwise union of twelve 1-of-k encoded states for a given sequencing read j ($k_0^j + k_1^j + \ldots + k_{11}^j$). In an example application, we demonstrate that a variation of our proposed positional population count operation can be applied to computing summary statistics for the `FLAG` field with >3,800-fold improvement in speed compared to current standard toolchain.

Code and reproducible results are available online at https://github.com/mklarqvist/positional-popcount for the positional popcount and at https://github.com/mklarqvist/libflagstats for the application on computing `FLAG` summary statistics.

2.2 Algorithms

2.2.1 Overview of the methods

Different input array sizes benefit from different algorithmic designs. In order to maximize performance given any input size, we describe algorithms for three different sizes of input streams: 1) For large inputs, the carry-save adder-based algorithms tend to operate the most efficiently [88]; 2) For medium inputs, we describe a variation of the trees-of-adders [60, 135] called a forest-of-adders; 3) For small inputs, we describe an algorithm based on blending the most-significant byte (MSB) and least-significant byte (LSB) into two new registers and then bits are iteratively peeled off and counted using the conventional popcount operation.

2.2.2 Scalar approach

The simplest and clearest way of computing the positional population count involves only four instructions using a simple mask-select-add step (Fig. 2.2):

1. Mask-selecting the target bit (`x bitand (1 « p)`), where x is the input word and $p \in \{0, 1, \ldots, 15\}$ is the bit index.
2. Shifting the selected bit to the least-significant bit (`x » p`), where x is the result from step 1.
3. Incrementing the counter at position p with the value from step 2.

Because of its simplicity, we expect this code to be compiled to efficient assembly and to have efficient baseline performance. This basic subroutine serve as reference for the other algorithms we describe.

```
void pospopcnt_u16_scalar_basic(uint16_t* data,
  uint32_t len, uint32_t* counters)
{
  // Every input word.
  for (int i = 0; i < len; ++i) {
    // Each bit in every input word.
    for (int j = 0; j < 16; ++j) {
      // Branchless mask-shift-add.
      counters[j] += ((data[i] bitand (1 << j)) >> j);
    }
  }
}
```

Figure 2.2 Reference algorithm for computing the positional population count. Given some input data we compute the total number of set bits at each position by using a branchless mask-shift-add update step.

2.2.3 Carry-save adder-based circuits

Given the performance of the circuit-based Harley-Seal (HS) algorithm [131, 88] described for the canonical population count problem we investigated possible variations of this algorithm for the positional population count problem. The original HS-algorithm was inspired by carry-save adder (CSA) circuits (Fig. 2.3) used to construct digital adders in microarchitectures and displays extremely good performance metrics on larger input streams.

Without loss of generality, given three distinct 1-bit input values ($a, b, c \in \{0, 1\}$) and their 2-bit sum $a + b + c \in \{0, 1, 2, 3\}$ it is possible to extract the least significant bit from the 2-bit sum using the bitwise expression $(a \oplus b) \oplus c$ and the most significant bit with $(a \wedge b) \vee ((a \oplus b) \wedge c)$, where $\oplus$ is the bitwise exclusive or operator. Note that no bitwise additions are actually used when computing the most and least significant bit from packed machine words. Also note that the most and least significant bit of $a + b + c$ are simultaneously computed for all bits in the input machine words using bitwise parallelism. We can generalize this workflow to any standard integer such that three input words produce two output words, h (high) and l (low), corresponding to the most and least significant bits, respectively (Fig. 2.4).

In our implementation, CSA operators are combined into a 16-input circuit using five different 5-bit accumulators referred to as `ones`, `twos`, `fours`, `eights` and `sixteens` such that the linear combination $1 \times \texttt{ones} + 2 \times \texttt{two} + 4 \times \texttt{fours} + 8 \times \texttt{eights} + 16 \times \texttt{sixteens}$ correspond to the final population count. All our Harley-Seal-based implementations

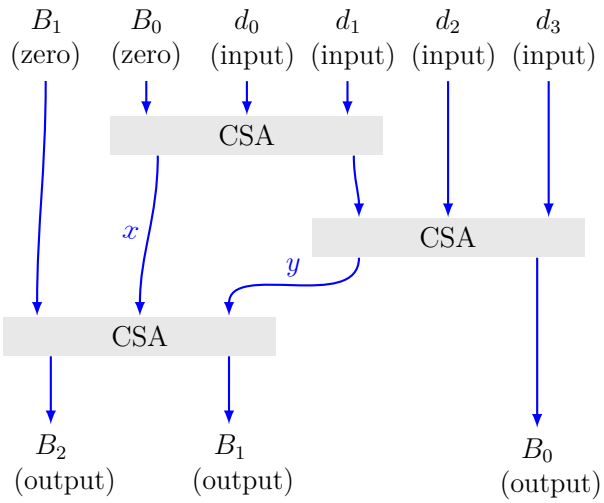


Figure 2.3 Schematic overview of the circuit-based carry-save adder (CSA) network for computing population counts. CSA circuit algorithm aggregating four inputs (d_0, d_1, d_2, d_3), producing three outputs B_0, B_1, B_2 corresponding to the least significant bit, second most significant bit and most significant bit of the sum.

```
void CSA(uint64_t* h, uint64_t* l,
  uint64_t a, uint64_t b, uint64_t c)
{
  uint64_t u = a xor b;
  *h = (a bitand b) bitor (u bitand c);
  *l = u xor c;
}
```

Figure 2.4 Generalization of the carry-save adder update step. Given three input values a, b, and c we compute the most significant bit h and least significant bit l. These output values are used in the next layer of the circuit.

operate on blocks of 16 vectors, requiring at least $16r$ input bits where r is the register width in bits, and proceeds as follows:

1. Compute the high and low bits (CSA) from a zero-vector, and data words d_0 and d_1, and store the most significant bit (MSB_b) in `twosA` and least significant bit (LSB_b) to `ones`.
2. Repeat step 1 for d_2 and d_3 and store the MSB_b in `twosB`. At this point we have two bitvectors of MSB_bs from four different input words $\{d_0, d_1, d_2, d_3\}$.
3. To proceed, we compute the high and low bits using the CSA subroutine with the arguments `twosA` and `twosB` and store the MSB_b in `foursA`.
4. Start over and repeat step 1-3 with new input words $\{d_4, d_5, d_6, d_7\}$. As in step 3, we now have a pair of bitvectors that we compute the high and low bits for `foursA` and `foursB` into `eightsA`.
5. Repeat steps 1-4 with words $\{d_8, d_9, \ldots, d_{15}\}$ to get `eightsB`. Again, we can now compute the high and low bits using `eightsA` and `eightsB` to get `sixteens`.
6. The number of set bits in `sixteens` corresponds to 1/16 of the number of set bits in the last 16 input words. We compute the number of set bits using a population count and increment the partial sum accumulator.
7. Repeat steps 1-6 until no more data is available. The residual counts in the partial 5-bit accumulators `ones`, `twos`, `fours`, `eights` are multiplied with {1, 2, 4, 8} and added to the total count.

The CSA subroutine can be further simplified down to two instructions on machines with the AVX512F instruction set available by using the new three-operand instruction, `VPTERNLOGD` (`_mm512_ternarylogic_epi32`, Fig. 2.5). This new instruction operates on three input bits x, y, z and a control sequence i and return the bit at index $x + 2y + 4z$ of i. For example, computing the exclusive set bits (`XOR`) of the inputs requires the bits at index {1, 2, 4, 7} of i to be set. Similarly, the most significant bit can be computed by setting the bits at index {3, 5, 6, 7} of i.

Building on this idea of using circuits for counting bits, we derived a natural extension of the HS-based algorithm by replacing the population count of `sixteens` in step 5 with 16 separate shift-add updates of partial sum accumulators corresponding to bit position $\{0, 1, \ldots, 15\}$. Care must be taken to ensure that the relatively small 16-bit accumulators do not overflow. To address this, we block the inner loop into steps of 65,535 iterations and update the output counters with the partial results following each blocked iteration.

To analyze how the change from a population count to multiple partial accumulators affects performance between the two HS-based algorithms we focus on their AVX512

```
void CSA_AVX512(__m512i* h, __m512i* l,
   __m512i b, __m512i c)
{
  *h = _mm512_ternarylogic_epi32(c, b, *l, 0b11101000);
  *l = _mm512_ternarylogic_epi32(c, b, *l, 0b10010110);
}
```

Figure 2.5 The carry-save adder update step using AVX512-based instructions. Note that the parameter a is dropped in this representation as it is invariantly equal to l in our application.

implementations using the performance of the Cannon Lake architecture as reference. The `avx512_popcount` function [88] involves 10 instructions with a latency of 16 cycles whereas the partial accumulator update step involves 3 instructions with a latency of 3 cycles for a total of 48 instructions and 48 latency. We therefore expect the `pospopcnt` HS-algorithm to run slightly slower compared to its `popcnt` counterpart.

2.2.4 Byte-blending

When the input data is very small (e.g. <1024 input words) we cannot use the HS-based algorithm as it operates on large blocks of 16 vectors at a time. Addressing this, we describe an efficient algorithm for small input data with much lower input limit. Without loss of generality, given two 2-byte input words d_0 and d_1 we reshuffle their bytes such that the most signiicant bytes from both words are placed adjacent to one another and similarly for the least siginficant byte. For example, given four input bytes b_0, b_1, b_2, b_3 corresponding to d_0 and d_1 we reorder the byte sequence into b_0, b_2, b_1, b_3. This memory layout enables us to use the `VPMOVB2M` (Packed MOVe Byte to Mask, `_mmNNN_movemask_epi8`, where `NNN` is empty, 256, or 512) instruction to extract the most-significant bit from each byte of b_0 and b_1 into a target word. In other words, the output of this instruction on the four example bytes b_0, b_2, b_1, b_3 above is equal to the four bits (b_0 bitand $0b1000000$) >> 4 bitor (b_2 bitand $0b1000000$) >> 5 bitor (b_1 bitand $0b1000000$) >> 6 bitor (b_3 bitand $0b1000000$) >> 7 packed into a new word. The target bit accumulator is then incremented with the population count of this resulting machine word. Next, we use the add instruction to shift b_0 and b_1 one bit to the left ($b_0 + b_0$ or $b_1 + b_1$) instead of using the actual shift instruction. Although conceptually identical, the add operator has better throughput (0.33 clocks-per-instruction (CPI)) compared to an actual shift operation (1 CPI) on most modern commodity processors.

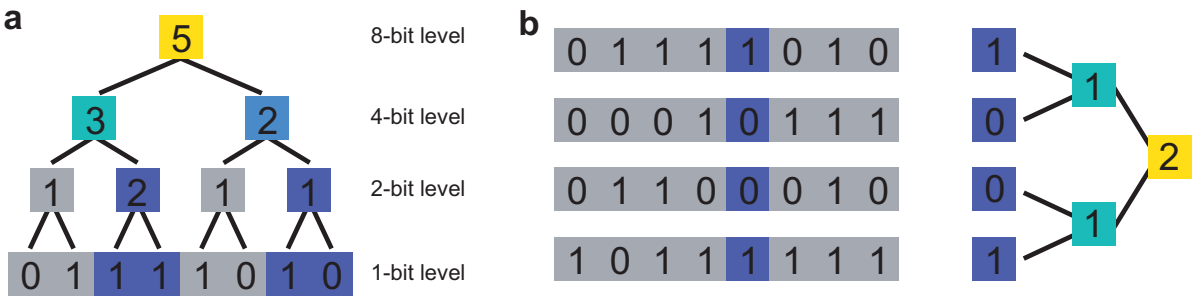


Figure 2.6 Overview of the difference between trees-of-adders and the proposed forest-of-adders. a) In the standard population count it is possible to compute the number of set bits in $\log_2 w$-time by first summing neibhouring bits, then pairs of those sums, and so on. b) Our proposed forest-of-adders operate in a similar fashion but is applied on the transpose of multiple input words on a per-bit basis such that for w-words we compute a forest of w trees.

As an example implementation using 256-bit registers, given sixteen 32-bit words $\{w_1, w_2, \ldots w_{16}\}$ from two registers we extract out the MSB from two vectors into a new register of thirty-two 8-byte words. This is similarly done for the LSB. Next, the most-significant bit for each byte is extracted using the `VPMOVB2M` instruction resulting in a single 32-bit masked word. The destination counters for bit 16 and 8 are updated with the population count of the masked word for the MSB and LSB components, respectively. The next bit is shifted in to the most-significant bit position using an add instruction. This prodecure is repeated seven more times for bits $15 \rightarrow 9$ and $7 \rightarrow 1$.

2.2.5 Tree-of-adders and forest-of-adders

Next we describe an algorithm based on trees-of-adders [60, 135] (Fig. 2.6) that performs well on moderately sized inputs (e.g. between 512 and 4096 words). The key insight to using a tree-of-adders for the conventional population count problem is to recognize that the sum of two n-bit numbers is never wider than $2n$ bits. By exploiting this property in an iterative fashion we can compute the total sum in time proportional to the logarithm of the final sum bit-width. For example, starting with 1-bit numbers, their sum require two bits of storage, and pairs of 2-bit numbers require four bits of storage, and so on. This property combined with large computer words (currently 64, 128, 256 or 512 bits in commodity hardware) allows us to perform several such pairwise additions simultaneously using bit-level parallelism.

It is possible to describe the same algorithmic approach for the positional population count problem by maintaining k separate trees-of-adders for bits $[0, 1, \ldots, k)$ (Fig. 2.6).

Table 2.1 Host machines used for testing. Code was compiled using GCC 8 on all hosts.

Processor	Microarchitecture	Cache (KB)		
		L1	L2	L3
Intel i3-8121U (10nm)	Cannon Lake (x64)	32+32	256	4096
Intel Xeon E5-2697 v3 (22 nm)	Haswell (x64)	32+32	256	35840

We call a collection of trees-of-adders an *adder forest*. So for a forest of 16 trees, we conceptually process batches of 16 input values $b_0, b_1, \ldots, b_{15}$ as follows:

1. Add the target bit k of b_0 and b_1 by shifting it to the most significant bit position ($b_0 >> k$ or $b_1 >> k$, for b_0 and b_1, respectively) to calculate the 2-bit sum $s^2_{0,1}$.
2. Repeat step 1 seven times with the words $\{2, 3, \ldots, 15\}$ to get $s^2_{2,3}$, $s^2_{4,5}$, $s^2_{6,7}$, $s^2_{8,9}$, $s^2_{10,11}$, $s^2_{12,13}$, and $s^2_{14,15}$.
3. Sum together $s^2_{0,1}$ with $s^2_{2,3}$ to get $s^4_{0,3}$. Repeat three times to get $s^4_{4,7}$, $s^4_{8,11}$, and $s^4_{12,15}$.
4. Sum together $s^4_{0,3}$ with $s^4_{4,7}$ to get $s^8_{0,7}$. Repeat this to get $s^8_{8,15}$.
5. Sum together $s^8_{0,7}$ and $s^8_{8,15}$ to get the final count for the target bit.
6. Repeat steps 1-5 for the remaining 15 bits in the forest.

In practice we process trees in a memory-friendlier fashion by updating all 16 trees for the current word pair before proceeding.

2.3 Experiments

Algorithms are implemented in C99 and are available online at https://github.com/mklarqvist/positional-popcount under the Apache 2.0 license. Code was compiled with GCC 8.3 using the optimization flags "-O3 -march=native" to restrict optimizations to the host-machine architecture. All tests were performed using a host machine with a Cannon Lake microarchitecture (Table 2.1) unless otherwise specified. Performance was measured using the Linux `perf` subsystem with the virtualized `PERF_COUNT_HW_CPU_CYCLES` counts as processor cycles (clockticks).

2.3.1 Simulated datasets

For all experiments, we generated random data from a uniform distribution, $\mathcal{U}(0, M)$, of input bits with positions generated using the Mersenne Twister pseudorandom number generator implemented in C++11. However, we find that performance is independent of

the generated data and decided to run all benchmarks with $M = 65535$. To guarantee reliability, we repeated each test 10,000 times and ascertained that the average cycle counts are all within a 5% deviation threshold.

2.3.2 Microbenchmarking

We performed microbenchmarks of the different subroutines to assess various performance metrics. First, we investigated the effect on performance with different input sizes (Fig. 2.7). For very short input streams an unrolled SSE4-based implementation of the byte-blend algorithm (`sse_blend_popcnt_unroll8`, see section 2.2.5) outperforms all AVX-512-based algorithms. This is unsurprising as the implementations using large register consume up to 512 values per iteration. At around 256 input values the AVX-512-methods display superior performance and continue to increase in throughput until around 65,536 input values. After this point no further gains are observed either between the SIMD-based methods themselves or compared to the reference scalar approach. The Harley-Seal-based population count algorithm performs well up until exhausting low-level instruction and data cache (L1 cache, 32+32 KiB on our Cannon Lake machine) whereafter performance degrades slightly. We observe another drop in performance when exhausting L2 cache (256 KiB). Remarkably, at this point the pospopcnt Harley-Seal-based algorithm has achieved parity in speed with the conventional Harley-Seal population count (Fig. 2.7) with very similar number of used CPU cycles despite executing 60% more instructions. We performed the same experiments using Haswell-based architecture (Table 2.1) with similar results (Fig. 2.7 and Table 2.2).

To quantify the relative speedup, we compared the pospopcnt algorithms to the reference scalar method (Fig. 2.2) with auto-vectorization disabled resulting in scalar computation (Table 2.3). Our proposed pospopcnt algorithms runs faster at all input sizes and achieves almost 140-fold improvement in speed compared to the non-SIMD version on larger inputs.

Taken together, we have demonstrated a significant acceleration of the novel pospopcnt operation using SIMD instructions for different sizes of input streams. Furthermore, we have shown that the considerably more complex pospopcnt operation can achieve parity in speed with the fastest known implementation of the canonical popcount-operation on large input streams [88] despite executing more instructions.

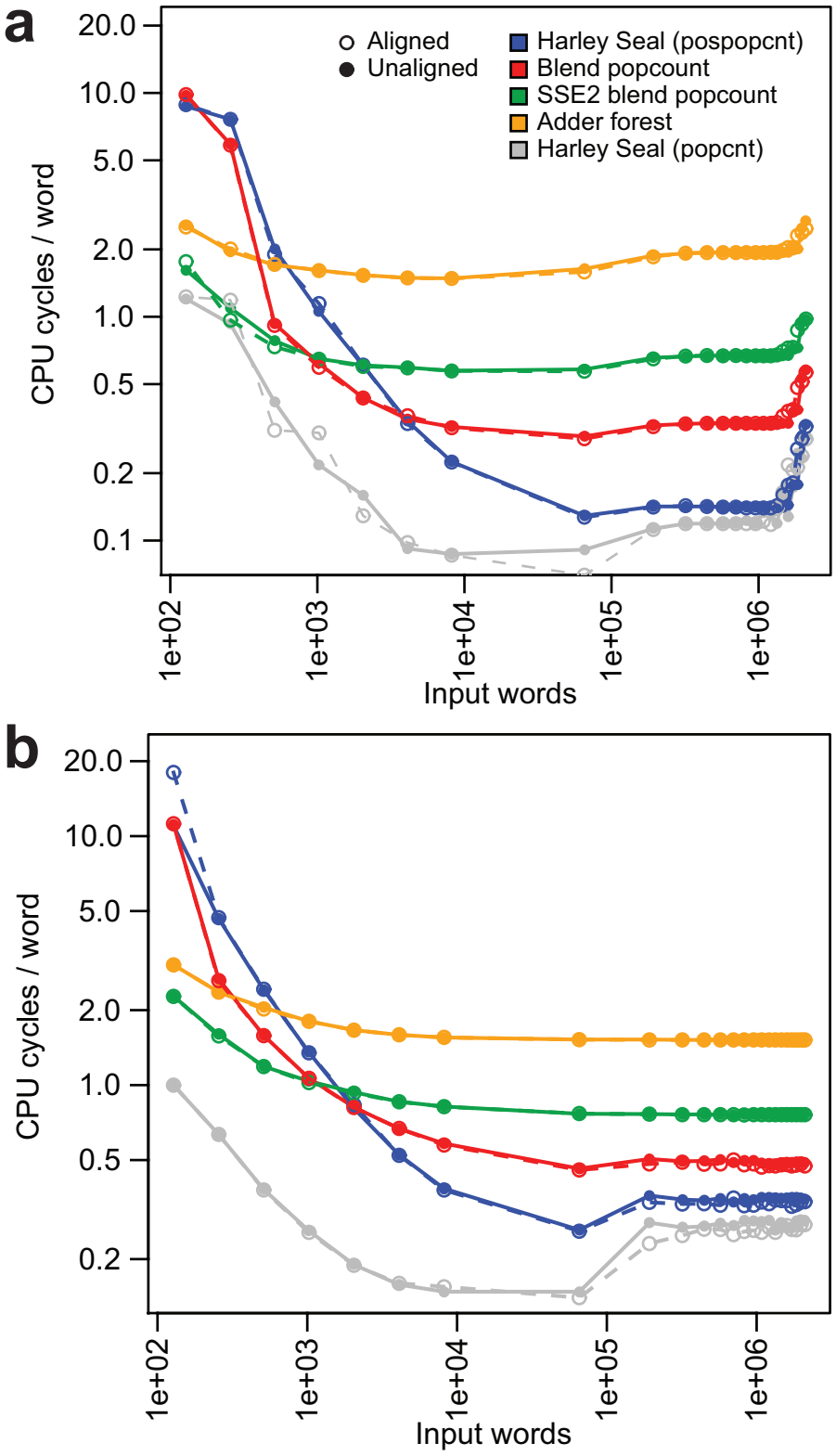


Figure 2.7 Number of CPU cycles / 16-bit word as a function of number of input values. a) Cannon Lake and b) Haswell. Machines used are listed in Table 2.1.

Table 2.2 Performance metrics of the `pospopcnt` operation at 262,144 input values for Cannon Lake and Haswell host machines (Table 2.1). For reference, the `popcount` (AVX2 and AVX512) functions compute the conventional population count.

Algorithm	**Architecture: Cannon Lake**		
	Inst. / cycle	**Cycles / 16-bit word**	**Inst. / 16-bit word**
popcount (AVX512BW)	1.47	0.12	0.17
Byte-blend (AVX2, unrolled-8)	3.99	0.75	2.98
Byte-blend (AVX512BW, unrolled-8)	3.14	0.55	1.71
Adder-forest (AVX512BW)	2.25	0.27	0.61
Harley-Seal (AVX512BW)	2.19	0.12	0.26
Scalar (no SIMD)	3.82	17.52	67.00
Algorithm	**Architecture: Haswell**		
	Inst. / cycle	**Cycles / 16-bit word**	**Inst. / 16-bit word**
popcount (AVX2)	1.28	0.27	0.41
Byte-blend (SSE4, unrolled-8)	3.85	1.49	5.84
Byte-blend (AVX2, unrolled-8)	3.84	0.75	2.92
Adder-forest (AVX2)	2.94	0.49	1.46
Harley-Seal (AVX2)	1.92	0.36	0.69
Scalar (no SIMD)	3.82	17.52	67.00

Table 2.3 Relative throughput compared to the scalar algorithm (Fig. 2.2) with vectorization explicitly disabled. For each count of input values, the best relative throughput is in bold. No further improvements in relative throughput is observed after 65,536 input values.

Algorithm	**Input values**							
	128	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**65536**
Byte-blend (SSE4, unrolled-8)	**8.3**	9.8	10.6	11.0	11.6	11.9	12.1	12.3
Byte-blend (AVX512BW, unrolled-8)	7.1	**11.2**	**16.2**	2	25.5	27.9	29.7	31.6
Adder-forest (AVX512BW)	3.1	2.8	14.5	**23.1**	**34.4**	44.9	52.8	61.7
Harley-Seal (AVX512BW)	2.1	2.3	8.2	15.1	28.2	**49.1**	**76.1**	**138.7**

2.3.3 Quality control of next-generation sequencing data

Over the last decade, large-scale international collaborative efforts have amassed increasingly large and diverse sequencing datasets that are now culminating in population-scaled national initiatives such as the UK 100,000 Genomes Project[127], and the US All of Us Research Program[52] that will encompass up to a million individuals with sequenced genomes. In these projects, genomic sequencing with the use of massively parallel next-generation sequencing technologies generates hundreds of millions of genomic fragments (reads) per sample. Sequencing reads that have been mapped (aligned) to a reference sequence are frequently stored in the incumbent SAM interchange format [72] or its binary representations BAM and CRAM. Twelve distinct 1-hot encoded read states (Table 2.4) are stored as their union ($s_0 + s_1 + \ldots + s_{11}$ where s_i is the encoding for category i) in the `FLAG` field. These states encode information such as the orientation of the read and if the read was successfully mapped to the reference. Computing summary statistics for this field is frequently used as an important quality control step. The current standard toolchain, SAMtools [72], implements this analysis using heavily branched and scalar code. We investigated the potential application of the positional popcount operation as a method for accelerating this scalar code.

Table 2.4 State description for the FLAG field. Read states are classified into 12 distinct states ranging from its pairing information to failing quality control checks such as being a likely PCR or optical duplicate.

Bit	One-hot	Description	Field name
1	00000000 0000000**1**	Read paired	`BAM_FPAIRED`
2	00000000 000000**1**0	Read mapped in proper pair	`BAM_FPROPER_PAIR`
3	00000000 00000**1**00	Read unmapped	`BAM_FUNMAP`
4	00000000 0000**1**000	Mate unmapped	`BAM_FMUNMAP`
5	00000000 000**1**0000	Read reverse strand	`BAM_FREVERSE`
6	00000000 00**1**00000	Mate reverse strand	`BAM_FMREVERSE`
7	00000000 0**1**000000	First in pair	`BAM_FREAD1`
8	00000000 **1**0000000	Second in pair	`BAM_FREAD2`
9	0000000**1** 00000000	Not primary alignment	`BAM_FSECONDARY`
10	000000**1**0 00000000	Read fails platform/vendor quality checks	`BAM_FQCFAIL`
11	00000**1**00 00000000	Read is PCR or optical duplicate	`BAM_FDUP`
12	0000**1**000 00000000	Supplementary alignment	`BAM_FSUPPLEMENTARY`

The standard approach for computing summary statistics for the `FLAG` field involves several bit-wise dependencies that introduce unavoidable branch conditions (Figure 2.8). In addition, this statistic is computed jointly for reads that pass quality control and those that do not. Despite these limitation, the heavily branched and bit-dependent SAMtools code can be rewritten using a mask-select propagate-carry approach that feeds into Harley-Seal-based carry-save adder networks using SIMD instructions in four steps (Figure 2.9):

1. The bit-fields `BAM_FUNMAP` and `BAM_FDUP` are always counted.
2. Rule 1: If the read has the bit-field `BAM_FSECONDARY` set then mask all bits.
3. Rule 2: If the read has the bit-field `BAM_FSUPPLEMENTARY` set mask all bits.
4. Count the remaining bit-fields (possible masked out by rule 1 or 2).

```
for c in 1..n # Loop over FLAGs
  # Selector for passing QC or failing QC
  QC = 1 if (c & BAM_FQCFAIL) else 0
  if c & BAM_FSECONDARY: # Secondary read
    ++out[QC]["secondary"]
  elif c & BAM_FSUPPLEMENTARY: # Supplementary read
    ++out[QC]["supplementary"]
  elif c & BAM_FPAIRED: # Read is paired
    if (c & BAM_FPROPER_PAIR) && not (c & BAM_FUNMAP):
      ++out[QC]["n_pair_good"]
    if c & BAM_FREAD1: ++out[QC]["n_read1"]
    if c & BAM_FREAD2: ++out[QC]["n_read2"]
    if (c & BAM_FMUNMAP) && not (c & BAM_FUNMAP):
      ++out[QC]["n_sgltn"]
    if not (c & BAM_FUNMAP) && not (c & BAM_FMUNMAP):
      ++out[QC]["pair_map"]
  if not (c & BAM_FUNMAP): ++out[QC]["mapped"] # Is mapped
  if not (c & BAM_FDUP):   ++out[QC]["dup"] # Is not duplicated
```

Figure 2.8 Psuedo-code for algorithm used in SAMtools for computing summary statistics for `FLAG` values. Field names are described in Table 2.4. The original macro subroutine can be found at https://github.com/SAMtools/SAMtools/blob/master/bam_stat.c#L47 (Last accessed: September 4, 2019)

```
# Masks (rules)
mask1 = BAM_FSECONDARY + BAM_FSUPPLEMENTARY
mask2 = BAM_FUNMAP + BAM_FSECONDARY + BAM_FDUP + BAM_FSUPPLEMENTARY
mask3 = BAM_FQCFAIL

for c in 1..n # Loop over FLAGs
  # Mask operation: the assumption here is that predicate
  # evaluations result in either a one-mask (11...1) or a zero
  # mask (00...0) as they would for SIMD instructions prior to the
  # AVX512 ISA.
  dat = c & (((c & mask1) > 0) | mask2)
  # Branchless mask-select: if QC is not set then L is kept and LU
  # is zeroed out. Otherwise LU is kept and L is zeroed out.
  L  = dat & ((dat & mask3) == 0)
  LU = dat & ((dat & mask3) == mask3)
  out[0] = pospopcnt(L)  # count positional bits for not BAM_FQCFAIL
  out[1] = pospopcnt(LU) # count positional bits for BAM_FQCFAIL
```

Figure 2.9 Psuedo-code for SIMD-based algorithm for a single update for a `FLAG` value.

We benchmarked the difference between the positional popcount-based algorithm and the reference implementation in SAMtools (`samtools flagstats`) on a deeply sequenced human dataset with >824 million reads[1]. In order to simulate a more realistic situation where column projections are stored in contiguous blocks of memory (as in CRAM) and require decompression prior to computation, we compressed `FLAG` fields into contiguous blocks of 512 kb (8192 16-bit words) using the general purpose compressors Lz4 (https://lz4.github.io/lz4/) or Zstd (https://facebook.github.io/zstd/).

On this readset, the pospopcnt-based subroutine is up to 2562-fold faster compared to the binary representation of SAM when using LZ4 and up to 3802-fold faster when the input data is uncompressed (Table 2.5). The pospopcnt function was bounded by both disk I/O and by decompression performance as around 80% of CPU time is spent retrieving and inflating data from disk. Arguably, this comparison is unfair as the legacy SAM/BAM format have limited random access and no column projection capabilities compared to the newer CRAM format [48]. When compared to CRAM, our proposed pospopcnt-based subroutine is still 402.6-fold faster. Lastly, we investigated the potential speedup of the original scalar subroutine in SAMtools by using efficient column projection

[1]Downloaded from https://dnanexus-rnd.s3.amazonaws.com/NA12878-xten/mappings/NA12878D_HiSeqX_R1.bam. Last accessed: 29 July, 2019.

and changing the compression method to Lz4 for efficient decompression. This code refactoring resulted in a massive 390.3-fold improvement in speed but remain 6.6-fold slower compared to the pospopcnt-based subroutine (Table 2.6-2.7).

Table 2.5 Completion time for SAMtools and pospopcnt. The pospopcnt-based algorithm is considerable faster compared to SAMtools using either the standard binary interchange format BAM or the more modern column-projection-capable format CRAM.

Approach	**Time**	**Speedup**
SAMtools – BAM	30m 50.06s	1
SAMtools – CRAM	4m 50.68s	6.36
pospopcnt-LZ4	0.72s	2569.53
pospopcnt-raw	0.48s	3802.90

Table 2.6 Completion time for refactored SAMtools and pospopcnt. After refactoring SAMtools to support column-projection and using the general compression engine LZ4, the pospopcnt-based approach is still considerably faster.

Approach	**Time**	**Speedup**
SAMtools-rewrite+LZ4	4.74 s	1
pospopcnt	0.72s	6.58

Table 2.7 Completion time for computing FLAG summary statistics. `FLAG` fields were compressed in blocks of 512 kB (8192 16-bit words) using the general compression libraries LZ4 or Zstd. LZ4 can be used in two different modes: the standard mode that compress faster but at worse fold compresion (LZ4) and in a mode that compress better at slower speeds (LZ4-HC). All compression methods were evaluated over their parameter space for compression: LZ4 (1-9) and Zstd (1-20). Times are listed in milliseconds. Decompression times are included in either algorithm for computing the `FLAG` summary statistic. The SAM-branchless subroutine is described online at https://github.com/mklarqvist/FlagStats/. Abbreviations: Comp. Method: Compression Method; Decomp: Decompression.

Comp. Method	**Decomp.**	**SAM-branchless**	**SAM**	**pospopcnt**
LZ4-HC-c1	988	8924	4991	1107
LZ4-HC-c2	993	8848	5076	1132
LZ4-HC-c3	938	8686	4930	1049
LZ4-HC-c4	846	8803	4876	933
LZ4-HC-c5	824	8525	5117	974

LZ4-HC-c6	770	8536	4774	851
LZ4-HC-c7	680	8404	4748	837
LZ4-HC-c8	644	8453	**4662**	755
LZ4-HC-c9	**580**	**8434**	4740	**722**
LZ4-c2	814	8658	4886	990
LZ4-c3	837	8576	4840	941
LZ4-c4	889	8627	4861	1026
LZ4-c5	826	8590	4885	1037
LZ4-c6	823	8629	5034	951
LZ4-c7	837	8606	4999	985
LZ4-c8	834	8604	4920	962
LZ4-c9	853	8615	4944	951
Zstd-c1	3435	11438	7798	3630
Zstd-c2	3577	11231	8110	3767
Zstd-c3	3403	11250	7922	3553
Zstd-c4	3562	11223	7949	3649
Zstd-c5	2919	10584	7263	2986
Zstd-c6	2964	10680	7545	3015
Zstd-c7	2681	10591	7067	2715
Zstd-c8	2641	10523	7103	2850
Zstd-c9	2352	10453	6669	2463
Zstd-c10	2309	10094	6756	2509
Zstd-c11	2344	10018	6430	2467
Zstd-c12	2116	9916	6242	2252
Zstd-c13	2107	9844	6183	2236
Zstd-c14	1955	9616	5969	2044
Zstd-c15	1716	9562	5807	1808
Zstd-c16	1286	9208	5592	1448
Zstd-c17	1278	8996	5592	1396
Zstd-c18	1192	8907	5294	1306
Zstd-c19	1181	8931	5362	1293
Zstd-c20	1175	8982	5369	1303

This example application demonstrates the efficiency and inherent flexibility of the pospopcnt-based algorithms. Using the pospopcnt operation as a replacement for heavily

branched code resulted in over >3,800-fold improvement in speed. We expect these subroutines to enhance existing methods and tools for the exploration of large genomics datasets in a variety of ways.

2.4 Conclusion

We can update sixteen counters approximately every 0.12 CPU cycles / 16-bit word when large registers are available. Notably, the positional population count operation display remarkable throughput: the `pospopcnt` operation can achieve parity in speed with the traditional population count while performing 16 separate updates and executing 60% more instructions (Table 2.2). The Harley-Seal-based algorithm is almost 140-fold faster compared to the non-vectorized approach (Fig. 2.2). Next, we demonstrated that the positional population count can be used as for computing summary statistics from unions of 1-hot-encoded vectors in bioinformatics with >3,800-fold improvement in speed.

Given the considerable efficiency and ease-of-use of the pospopcnt-operator, we envision it will be useful in a wide-range of applications involving 1-of-k-encoded vectors and their bitwise union. Future work should generalize our results from 16-bit words to wider machine words. We expect that our techniques could be ported to other architectures such as ARM or POWER.

2.5 Other approaches

In the course of this work we and members of the open-source community described several sub-optimal algorithms compared to the proposed algorithms above but are worthwile presenting as they contain interesting concepts.

2.5.1 Shift-pack popcount accumulator

In this approach, data from sixteen words of bits $(0, 1, \ldots, 15)$ are first reshuffled into words of bits from each different bit-position $\{(0_0, 0_1, \ldots, 0_{15}), (1_1, 1_1, \ldots, 1_{15}), \ldots (15_0, 15_1, \ldots, 15_{15})$ by shifting in 16 one-hot values into new 16-bit primitives. The target bit counter is incremented with the popcount of the machine word. Psuedo-code for the conceptual model (Figure 2.10):

```
# Prepare machine words.
for c in 1..n, c+=16 # 1->n with stride 16
  for i in 1..16 # FLAG 1->16 in range
```

```
    for j in 1..16 # Each 1-hot vector state
      y[j] |= (((x[c+i] & (1 << j)) >> j) << i)

  # Count bits in words.
  for i in 1..16 # 1->16 packed element
    out[i] += popcnt(y[i]) # popcount
    y[j] = 0 # Reset
```

Figure 2.10 In this model, sixteen 16-bit words have bits from each position shuffled into a sixteen new machine words where each word is the target bit k from each word.

2.5.2 Register accumulator and aggregator

In a varition of the shift-pack popcount accumulator approach (Section 2.5.1) we accumulate up to 16×2^{16} partial sums of a 1-hot in a single register followed by a horizontal sum update. By using 16-bit partial sum accumulators we must perform a secondary accumulation step every 2^{16} iterations to prevent overflowing the 16-bit primitives. Psuedo-code for the conceptual model (Figure 2.11):

```
for i in 1..n, i+=65536 # 1->n with stride 65536
  for c in 1..65536 # Section of 65536 iterations to prevent overflow
    # In practice this is a horizontal register sum (VPADDW or PADDW)
    # for 16x 16-bit integers instead of a loop
    for j in 1..16 # Each 1-hot vector state
      y[j] |= ((x[c+i] & (1 << j)) >> j)


  for j in 1..16 # 1->16 packed element
    out[j] += y[j] # Accumulate
    y[j] = 0 # Reset
```

Figure 2.11 In this model, registers of 16 packed 16-bit values are horizontally added to a register of 16 packed 16-bit aggregators. Every 2^{16} iterations we reset the aggregators and accumulate into the final counters.

This algorithm can be written to make use of the AVX-512 ISA by computing 16×2^{32} partial sums. The AVX512 instruction set do not provide native instructions to perform 16-bit-wise sums of registers. By being restricted to 32-bit accumulators while consuming

16-bit primitives we must performed a second 16-bit shift-add operation to mimic 32-bit behavior. Unlike the AVX2 algortihm, the 32-bit accumulators in this version do not require blocking under the expectation that the total count in either slot do not exceed 2^{32}.

2.5.3 Interlaced register accumulator and aggregator

Instead of having 16 registers of 16 values of partial sums for each 1-hot state we have a single register with 16 partial sums for the different 1-hot states. We achieve this by broadcasting a single integer to all slots in a register and performing a 16-way comparison. The target value v is broadcasted to all the slots in a target register $r = [v, v, \ldots, v]$.

Psuedo-code for the conceptual model:

```
for i in 1..n, i+=4096 # 1->n with stride 4096
  for c in 1..4096 # Block of 4096 iterations to prevent overflow
    f = {x[c], x[c], x[c], x[c], ..., x[c]} # 16 copies of x[c]
    for j in 1..16 # Each 1-hot vector state
      y[j] += (((f[j] & (1 << j)) == (1 << j)) & 1)

  for j in 1..16 # 1->16 packed element
    out[j] += y[j] # accumulate
    y[j] = 0 # reset
```

Figure 2.12 A value is broadcast to a register and use in a 16-way comparison.

2.5.4 Interlaced register accumulator and aggregator

The AVX512VL/AVC512BW instruction set comes with the new `VPCMPW/VPCMPUW` (`_mm512_cmpeq_epu16_mask`) instruction that performs a SIMD compare of the packed 16-bit words between two operands and returns the equality predicate as a packed 32-bit integer mask (`__mask32`). The result of each word comparison is packed into a single mask bit such that zero encode for a comparison evaluating to false or one when a comparison evaluated to true. This algorithm combines bit-packed mask with a 32-bit `POPCNT` operation. In a second approach, we pack two 32-bit masks into a 64-bit primitive before performing a 64-bit `POPCNT` operation on the packed mask.

2.6 Acknowledgements

I am grateful to J. Bonfield (Wellcome Sanger Institute) for helpful discussions and suggestions regarding the implementation of computing `FLAG` summary statistics. I am grateful to J. D. McCalpin (University of Texas at Austin) for benchmarking advice. I am grateful to members of the open-source community for helping improving the implementation.

Chapter 3

Efficient computation of genome-wide linkage-disequilibrium

In this chapter I will introduce several algorithms for efficiently for computing XX^T for binary matrices and will apply this to computing various common linkage-disequilibrium statistics from two vectors of haplotypes/genotypes. Several of these algorithms utilize or directly extend the population count-based algorithms introduced in Chapter 2. Many of these low-level algorithms presented here achieve hardware limits on modern commodity processors. In Chapter 5 I will demonstrate how to compute the self-similarity matrix X^TX using no additional overhead using the methods presented here. I will describe a generalized hybrid bitmap compression approach that can be used in indexing approaches for high-performance application in 'big data' and serves as the foundational idea for later chapters in compression and analysis of population-scaled sequence variant data.

In addition to the generalizable algorithms, I describe several file formats and associated algorithms capable of handling the huge data files produced for computing chromosome-wide linkage-disequilibrium. Lastly, I describe a R-package with native C++-bindings with a special focus on data visualization. Collectively, these methods and algorithms are packaged into the software project `tomahawk` and associated R-binding called `rtomahawk`. A standalone library using functional multiversioning (FMV) supporting all modern instruction-set architectures (ISAs) was released separately as `Storm bitmaps` that depends on the header-only library `libalgebra` that was developed for Chapter 2. Software is available online at:

- `https://github.com/mklarqvist/tomahawk`
- `https://github.com/mklarqvist/rtomahawk`
- `https://github.com/mklarqvist/StormBitmaps`

- `https://github.com/mklarqvist/libalgebra`

3.1 Introduction

Linkage disequilibrium (LD), the non-random pairwise association between variants at distinct genomic loci, is a key descriptor of genetic structure in a population and is essential for many analyses investigating human genome variation. When two genomic regions are in spatial proximity and the rate of ancestral recombination between them is low, then the pair of alleles from the two loci are frequently coinherited in a unit (**Figure 3.1**). If this allelic coinheritance occurs at a rate that significantly deviates from random expectation then this phenomenon is referred to as gametic association or linkage disequilibrium. Historically, the first rigorous approach to mathematically modelling this random assortment of alleles was performed in 1944 by Hilda Geiringer[42]. Two decades later, the term linkage disequilibrium was coined in 1960 by Lewontin and Kojima[70].

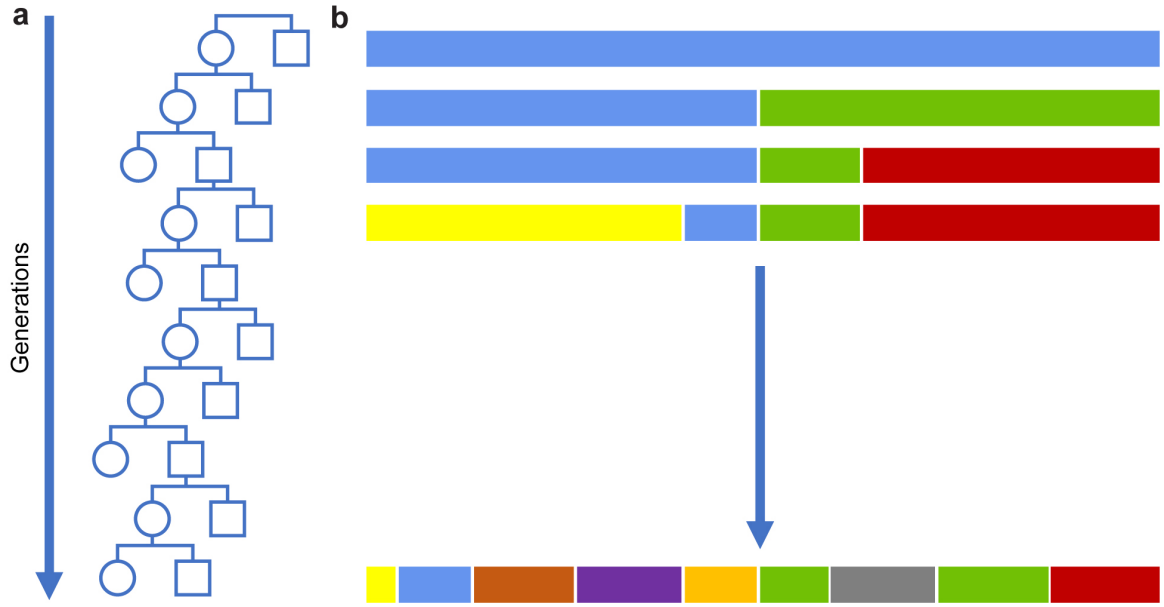


Figure 3.1 Overview of the origin of linkage-disequilibrium. **a)** Within a family, genetic linkage occurs between two genetic markers on the same chromosome by remaining unbroken by recombination. **b)** Starting with a founder chromosome (single color), recombination will break and fuse the genetic material of both parents each generation at some position. Following many generations, a target chromosome represent a mosaic of the partial chromosomes of all its ancestors (different colors). If two genetic markers map to a haplotype block (single color) then they are said to be in linkage disequilibrium as they will be coinherited in the population at a frequency deviation from random chance.

Several measurements of LD summary statistics have been proposed[45], including Pearson's (product moment) correlation coefficient R[47] and D[69] but all depend on the difference between the observed joint frequency of alleles co-occurring on the same haplotype compared to what is expected by random chance (**Figure 3.2**). For example, the squared Pearson's correlation coefficient (R^2) is defined in the range $[0, 1]$ such that $R^2 = 0$ when two alleles do not co-occur more frequently then expected by random sampling and $R^2 = 1$ when the two alleles are always co-ocurring. This correlation coefficient is directly related to statistical power[32]. Another frequently used measure of LD is the standardized difference D' that can be estimated by the Pearson correlation between the counts of the minor alleles for two SNPs.

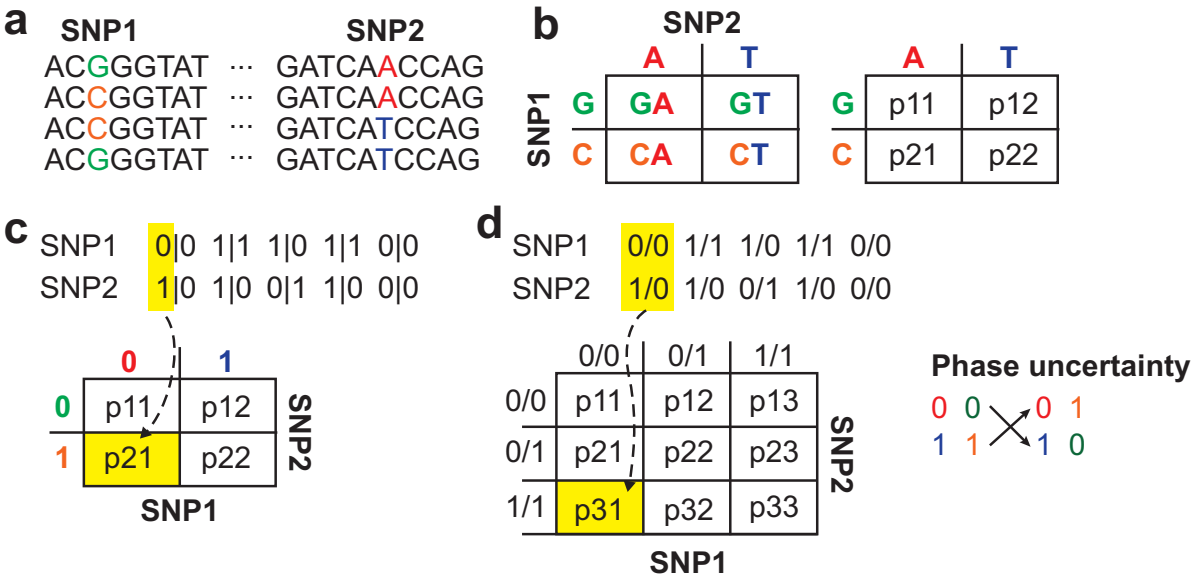


Figure 3.2 Overview calculating linkage-disequilibrium. **a)** Most methods used to calculate linkage-disequilibrium requires a pair of distinct diallelic sites (SNP1 and SNP2) **b)** The phased haplotype, the two-locus in-order sequence, counts can be accumulated into a 2-by-2 contingency table summing up to $2N$ if there are no missing values. **c)** In the most naive way possible, the two-locus haplotype counts are computed by looking at pairs of alleles and incrementing that corresponding cell. Phased genotypes are represented as allele 1 | allele 2. **d)** Unphased haplotypes have unknown order with respect to the two chromosomes of an individual and require additional consideration. All 2-locus haplotypes can be unambiguously inferred with the exception of the haplotype that is heterozygous at both loci. This phase uncertainty problem has to be statistically inferred given the observed data. This inference is most frequently performed with iterative expectation-maximization algorithms until converging on a solution[46, 113]. A fundamental problem with iterative methods is that they are not guaranteed to converge on the global optimum solution in presence of local maximums. To accommodate this uncertainty, unphased haplotype frequencies have to be formulated as a 3-by-3 contingency table. The central cell in this matrix corresponds to haplotypes with unknown phase. In a naive algorithm, the cell counts can are incremented while iterating over pairs of genotypic vectors (as in **c)**. Unphased genotypes are represented as allele 1 / allele2.

Computing LD summary statistics between pairs of loci is not computationally challenging but grows rapidly with increasing dataset size. For example, computing pairwise LD across all pairs of the 1.7 million variants on chromosome 20 for the 2,504 samples in the 1000 Genomes Project (1000GP)[123] panel involves 1.5×10^{12} variant and 7.2×10^{15} allelic comparisons. With the advent of national cohorts extending towards a million samples, such as the UK 100,000 Genomes Project[14], Million Veteran

Program, and the All of Us Research Program[52], there is an urgent need for highly performant and scalable methods capable of computing LD-based statistics when working with large-scale datasets.

3.1.1 Applications in fine-mapping

Quantitative assessment of LD is an important procedure to conduct fine-mapping of casual variants identified by genome-wide association studies (reviewed in [107]). Genome-wide association studies have been widely used in the last decade to identify the chromosomal locations that harbour genetic risk loci of disease for a large number of diseases and traits, including insomnia[54], height and body mass index[140], educational attainment[64], anorexia nervosa[33], cancers[86, 118], type-2 diabetes mellitus[146], coronary artery disease[124, 146], schizophrenia[73], inflammatory bowel disease[28], and major depressive disorder[50] among others. GWAS studies involve the interrogation of hundreds of thousands of variants across a genome in large cohorts of individuals for potential associations for variants to the trait of interest by investigating discrepancies in the haplotype structure of two populations using various statistical tests. Reflecting its success, there are 143,963 reported associations of single-nucleotide polymorphisms (SNPs)[1] in the US National Human Genome Research Institute (NHGRI)–European Bioinformatics Institute (EBI) GWAS Catalog[81] with a P-value $< 1 \times 10^{-5}$.

Since the first GWAS for age-related macular degeneration was published in 2005 encompassing 96 cases and 50 controls[59] there are now beginning to emerge studies in excess of a million samples[64, 77, 54]. For-profit companies, such as 23andMe and Ancestry, have amassed tens of millions of genotyped samples that may be used in both commercial and private efforts in studying health and disease in the near future.

Undoubtedly, the immensely successful rise of GWAS studies was primarily driven by the cost-effectiveness of genotype microarrays. The cost of these microarrays increases with the number of typed SNPs and in extension the total cost of a study. Therefore, target loci, called tag-SNPs, are selected for typing because they have large amount of linkage disequilibrium with neighbouring variants in the same haplotype segment and thereby enable accurate inference of unobserved (untyped) variants. These surrogate markers are used as proxies for unmeasured SNPs in relatively large genomic regions. As a consequence of interrogating only tag-SNPs, the measured variant is rarely directly associated with the target trait[80].

[1] `gwas_catalog_v1.0-associations_e96_r2019-07-12.tsv` downloaded from `https://www.ebi.ac.uk/gwas/docs/file-downloads`. Last accessed: 18 July, 2019

Fine-mapping of SNP-trait associations refers to the additional statistical approaches used to identify the causal variants responsible for the observed GWAS signals. Regions of interest for fine-mapping is determined following the identification of statistically significant SNPs. These regions are typically partitioned into non-overlapping subsets according to local linkage disequilibrium in order to reduce the computational burden of fine-mapping. The statistical models used in fine-mapping is reviewed in [107].

Using linkage disequilibrium in fine-mapping is based on the observation that ancestral meiotic recombinations results in a decreasing frequency of coinheritance of alleles over genetic distance. This can suggest that the causal variant is the one with the strongest association with the trait of interest.

3.2 Methods

3.2.1 Overview of the strategies

To summarize, I describe two bitmap-based approaches for computing the cardinality of the set intersection between pairs of genotypic vectors (**Figure 3.3**). In the general case, I describe a time-efficient approach that use both uncompressed bitmaps and scalars. For datasets with a large number of individuals, I describe a variation that directly operates on compressed bitmaps for considerable savings in both compute time and memory cost. These algorithms exploit the large memory registers on modern processors and horizontally compares pairs of bit vectors using vectorized instructions. In the case of sparse-dense or spare-sparse comparisons, I describe efficient algorithms that provide additional acceleration. All the performance critical algorithms have been optimized across most of the currently available SIMD instructions sets (SSE4.2, AVX2, and AVX512BW) using `libalgebra` (https://github.com/mklarqvist/libalgebra).

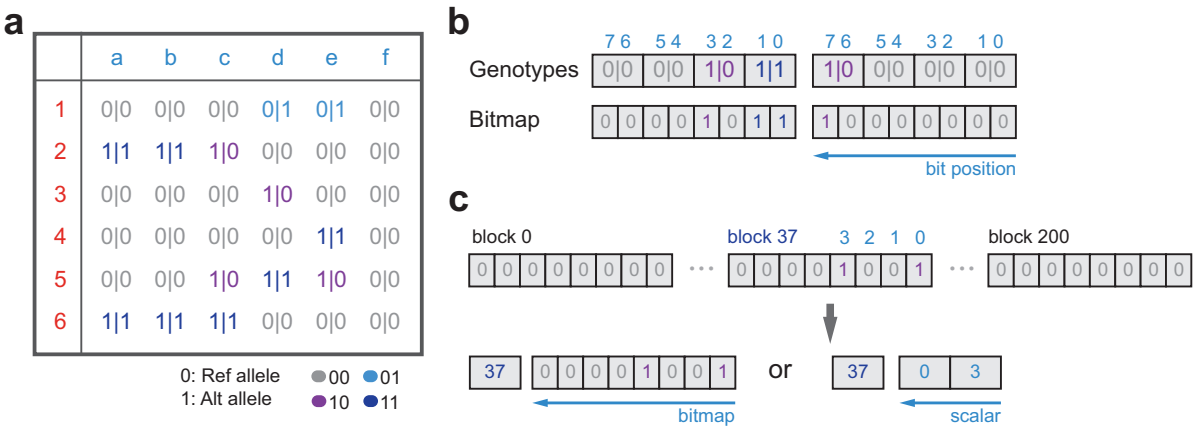


Figure 3.3 Overview of the encodings in Tomahawk. a) Genetic variant data is most frequently stored as haplotypes/genotypes in a matrix $X^{n \times m}$ for n samples (columns: $a - f$) and m loci (rows: $1 - 6$) where 0 encodes for the reference allele at that position and 1 for the alternative allele. b) This binary representation of reference/alternative allele for a given position is encoded in a binary string (bitvector) of width pn where p is the ploidy. By default, p is fixed to 2 in Tomahawk. c) As the number of samples are increasing, most rows (sites) in a binary genetic matrix $X^{n \times m}$ will be sparse because of the limited haplotypic diversity in humans. It is possible to further exploit this sparsity in this case by partitioning the universe of samples $[0, 1, \ldots,]$ into non-overlapping, but contiguous, subsection (slice) of this universe: $[[0 \times 2^{16}, 1 \times 2^{16}), [1 \times 2^{16}, 2 \times 2^{16}), \ldots]$. Data is stored only for non-empty slices as either (1) a bitvector, or (2) a list of scalars, depending on the density of the slice. This approach reduces the fixed storage cost from pn bits in the dense case to the variable range $[2^{16}, pn)$ at a small performance cost.

3.2.2 General considerations

Computing LD summary statistics such as R^2 and D' depend on the difference in the observed joint allele frequency compared to that expected by random sampling. To calculate the expected joint frequencies, the studied population is assumed to be random mating and to be in Hardy-Weinberg equilibrium at each locus. Assuming diploid genomes and biallelic sites: At locus L_A there are two alleles A and a with frequencies p and $1-p$ and at locus L_B there are two alleles B and b with frequencies q and $1-q$. The 2-locus haplotype frequencies AB, Ab, aB, and ab are f_{11}, f_{12}, f_{21} and f_{22}, respectively.

Tomahawk is designed for diploid species (specifically humans) and as such filters out all non-diploid variants and sites with mixed ploidy. By default, variant sites that are either invariantly the reference allele (encoded as the zero vector $(00\ldots0)$) or the alternative allele (encoded as the one-vector $(11\ldots1)$) with respect to non-missing data or have >20% missing values are filtered out and not included for further analysis.

3.2.3 Calculating linkage-disequilibrium for phased samples

Let a_i and A_i be the alleles for some variant i and a_j and A_j the alleles for some variant j where $i \neq j$. Let $f(a_i)$ be the allele frequency of a_i and let $h(a_i a_j)$ be the frequency of the 2-variant haplotype. The alleles a_i and a_j are said to be in LD if $D = h(a_i a_j) - f(a_i)f(a_j) > 0$. The LD measures are calculated as $R^2 = D^2/(f(a_i)(1 - f(a_i))f(a_j)(1 - f(a_j)))$ and $D' = D/D_{\max}$, where $D_{\max} = \min(f(a_i)f(a_j), f(A_i)f(A_j))$ if $D < 0$ and $D_{\max} = \min(f(a_i)f(A_j), f(A_i)f(a_j))$ if $D \geq 0$. Let $D'(f(a_i), f(a_j))$ and $R^2(f(a_i), f(a_j))$ be the pairwise LD between two variants v_i and v_j.

3.2.4 Inferring haplotype frequencies for unphased samples

In the case of unphased data, computing LD is more difficult because of the phase-uncertainty problem where quantifying jointly heterozygous alleles is ambiguous because of two possible haplotype combinations (**Figure 3.2**). In a 3×3 matrix for a pair of biallelic markers, all possible phases can be unambiguously determined with the exception of individuals heterozygous at both loci (the central cell). This is frequently addressed by employing statistical inference techniques such as expectation-maximization[46, 113]. In this process, an initial estimate of haplotype frequencies is substituted in the equation resulting in a new estimate. This is then fed back into the equation and this procedure continues until convergence. Unfortunately, these methods are comparatively slow because multiple iterations are required to converge to a solution. I address this by using an iteration-free closed form solution[91, 40] where the estimated frequency $\widehat{f_{11}}$ of haplotype AB can be described as a cubic function[40] adapted from (equation 4 in [46]):

$$a\widehat{f_{11}^3} + b\widehat{f_{11}^2} + c\widehat{f_{11}} + d = 0 \tag{3.1}$$

I briefly describe the parameters and equations[143] here for clarity.

$$\begin{aligned} g &= 2n_{11} + n_{12} + n_{21} \\ m &= 2n \\ a &= 4n = 2m \\ b &= m(1 - 2p - 2q) - 2g - n_{22} \\ c &= mpq - g - n_{22}(1 - p - q) \\ d &= -gpq \end{aligned}$$

where n is the number of subjects, p common allele frequency for locus 1, q common allele frequency for locus 2, and the number of subjects homozygous at both loci n_{11}, homozygous at locus 1 and heterozygous at locus 2 n_{12}, heterozygous at locus 1 and homozygous at locus 2 n_{21}, and heterozygous at both loci n_{22}. Finally, the essential cubic parameters are calculated using the constants a-d:

$$
\begin{aligned}
x_N &= \frac{-b}{3a} \\
\delta^2 &= \frac{b^2 - 3ac}{9a^2} \\
h^2 &= 4a^2\delta^6 \\
y_N &= ax_N^3 + bx_N^2 + cx_N + d
\end{aligned}
$$

where x_N and y_N are the x and y coordinates on a polynomial curve $f(x)$ of degree n. The geometric discriminant of the cubic $\Delta_3 = y_{T_1} y_{T_2} = y_N^2 - h^2$ can now be used to find the outcome in real roots with three potential solutions: (1) If $y_N^2 > h^2$ there is only one possible root (α) defined as:

$$
\alpha = x_N + \sqrt[3]{\frac{1}{2a}(-y_N + \sqrt{y_N^2 - h^2})} + \sqrt[3]{\frac{1}{2a}(-y_N - \sqrt{y_N^2 - h^2})}
$$

(2) If $y_N^2 = h^2$ there are three possible roots (α, β, γ) defined for $\mu = \sqrt[3]{\frac{y_N}{2a}}$:

$$
\begin{aligned}
\alpha &= x_N + \mu = \beta \\
\gamma &= x_N - 2\mu
\end{aligned}
$$

In the special case when $y_N = h = 0$ meaning that $\mu = 0$ there are three equal roots $\alpha = \beta = \gamma = x_N$. (3) In the last case when $y_N^2 < h^2$ there are three possible roots (α, β, γ):

$$
\begin{aligned}
\theta &= \frac{\arccos(\frac{-y_N}{h})}{3} \\
\alpha &= x_N + 2\delta\cos\theta \\
\beta &= x_N + 2\delta\cos(\theta + 2\pi/3) \\
\gamma &= x_N + 2\delta\cos(\theta + 4\pi/3)
\end{aligned}
$$

As previously suggested[40], roots are only considered biologically possible when $\widehat{f} \in [0-\epsilon, 1+\epsilon]$, where ϵ is the allowed floating point error, and when $\widehat{f_{11}} + \widehat{f_{12}} + \widehat{f_{21}} + \widehat{f_{22}} = 1$. After estimating $\widehat{h_{11}}$ the inference of $\widehat{h_{12}}, \widehat{h_{21}}$, and $\widehat{h_{22}}$ is straightforward:

$$\begin{aligned}\widehat{h_{12}} &= p - \widehat{h_{11}} \\ \widehat{h_{21}} &= q - \widehat{h_{11}} \\ \widehat{h_{22}} &= 1 - (\widehat{h_{11}} + \widehat{h_{12}} + \widehat{h_{21}})\end{aligned} \tag{3.2}$$

The most likely root is then heuristically selected using the smallest χ^2 critical value compared to the expected model.

3.2.5 Technical problem statement

Following the mathematical description of the coefficient of linkage disequilibrium, D, from Section 3.2.3 it is possible to generalize this formulation using set logic. Given two integer sets L_0 and L_1 such that $L \in [0, M)$ we want to compute the set intersection $L_{01} = L_0 \cap L_1$ and the cardinalities $|L_0|$ and $|L_1|$. In this form, the 2-variant haplotype count can be expressed as $f_{22} = |L_{01}|/M$ and the heterozygous counts as $f_{12} = |L_0|/M - f_{22}$ and $f_{21} = |L_1|/M - f_{22}$ and the homozygous alternative count as $f_{11} = 1 - f_{12} - f_{21} - f_{22}$. This information is sufficient to compute linkage disequilibrium (Equation 3.3):

$$D_{AB} = f_{22} - f_{12}f_{21} = M^{-1}|L_{01}| - M^{-2}(|L_0| - |L_{01}|)(|L_1| - |L_{01}|) \tag{3.3}$$

when the data is complete. Missing values can be considered members of the separate integers sets P_0 and P_1 that mask out the positions such that $P_{01} = P_0 \cup P_1$ (Equation 3.4):

$$D_{AB} = M_m^{-1}|L_{01} \setminus P_{01}| - M_m^{-2}(|L_0 \setminus P_{01}| - |L_{01} \setminus P_{01}|)(|L_1 \setminus M_0 \cup M_1| - |L_{01} \setminus P_{01}|) \tag{3.4}$$

where $M_m = M - |P_{01}|$ is the number of jointly non-missing values.

Given a list of integers sets $(L_0, L_1, \ldots, L_N)$ we want to compute D_{AB} for all upper-triangular pairs of integer sets as efficiently as possible. In almost all situations, the number of distinct loci N far exceeds that of the number of haplotypes M. Thus, the computational complexity is $O(M\binom{N}{2})$.

3.2.6 Representing genotypes using bitmaps

Biallelic variants in the incumbent Vcf interchange format are stored as dictionary-encoded values such that zero (0) maps to reference allele and one (1) to the alternative

allele at that site. This simple representation allows us to interpret this encoding as a bitmap index mapping the alternative allele to haplotypes. A bitmap is a data structure that pack multiple distinct values into a single machine word. For example, the vector $(0, 0, 1, 0, 1)$ can be encoded into the machine word [00010100] (reading in the right to left orientation)(see Algorithm 1). Given an input table of n individuals and m biallelic sites each bitmap is n bits wide, or $\lceil n/w \rceil$ machine words, where $w \in \{32, 64\}$ on modern commodity processors.

Bitmaps often have superior performance compared to scalar approaches when the cardinality of a set S is large compared to the universe size n. In our application, this occurs when $|S| > n/64$ and when $n < 2^{18}$, where n is the number of haplotypes and S is the number of alternative alleles.

In addition to improve storage compared to scalars, the bitvector encoding facilitates several efficient operations by using optimized bitwise logical operations available on modern commodity processors. For example, computing the intersection or union between two equal-length bitmaps can be performed with a single bitwise `AND` operation or a single `OR` operation, respectively. Similarly, computing the resulting set cardinality can be performed by counting the number of set bits. This operation, also called the population count, is generally available as a CPU-intrinsic instruction such as `POPCNT` on Intel processors or as a highly optimized software implementation (e.g. GNU `__builtin_popcount`) (also see Chapter 2).

Algorithm 1 Packing symbols into a bitmap

function PackBitmap(I, n, σ, W)
    Let n be the length of input I
    Let $M = \lceil \log 2(\sigma) \rceil$ be the bit-width of a symbol
    Let $\Sigma = \left\lfloor \frac{W}{M} \right\rfloor$ be the number of symbols per machine word
    Let $O[0 \ldots n]$ be the output vector
    for $i = 1$ to n **do**
        $O[i/\Sigma] = I[i] << M * (i \bmod \Sigma)$
    end for
    return O
end function

3.2.7 Computing LD using bitmaps

Encouraged by the ability to compute set operations efficiently using bitmaps, I describe efficient algorithms for applying this logic to the problem of computing linkage-

disequilibrium. We can derive bitwise operations such that a bit is set when a given 2-locus haplotype condition is satisfied and zero when not. The sum of the population bit-count of these bitmaps is equivalent to the inner product of the target 2-locus haplotype. Notably, bitmap-based algorithms have a guaranteed fixed cost proportional to the ratio between the number of samples and the register width, independent of allele frequency. For phased data, we want to compute the set intersection h_{00}, h_{01}, h_{10} and h_{11} using two genotypic vectors $\mathbf{G}_j$ and $\mathbf{G}_k$ from two distinct biallelic sites j and k such that $j \neq k$. The set intersection h can be described in terms of conditioned positional index vectors for a given 2-locus haplotype (Equation 3.5):

$$\begin{cases} h_{00}(\mathbf{G}_j, \mathbf{G}_k) = \{i | G_{ji} = 0, G_{ki} = 0\} \\ h_{01}(\mathbf{G}_j, \mathbf{G}_k) = \{i | G_{ji} = 0, G_{ki} = 1\} \\ h_{10}(\mathbf{G}_j, \mathbf{G}_k) = \{i | G_{ji} = 1, G_{ki} = 0\} \\ h_{11}(\mathbf{G}_j, \mathbf{G}_k) = \{i | G_{ji} = 1, G_{ki} = 1\} \end{cases} \tag{3.5}$$

For vectors without missing data, we note that $h_{00}(\mathbf{G}_j, \mathbf{G}_k) \cup h_{01}(\mathbf{G}_j, \mathbf{G}_k) \cup h_{10}(\mathbf{G}_j, \mathbf{G}_k) \cup h_{11}(\mathbf{G}_j, \mathbf{G}_k) = \{1, 2, \ldots, 2N\}$. This definition also demonstrates that there is mutual exclusivity between the sets. We denote the cardinality of the set intersections $|h_{00}|$, $|h_{01}|$, $|h_{10}|$, and $|h_{11}|$ as f_{00}, f_{01}, f_{10}, and f_{11}, respectively. Similarly, we denote the genotype vector $\mathbf{G}_j$ and $\mathbf{G}_k$ in bitmap encoding as $\mathbf{B}_j$ and $\mathbf{B}_k$, respectively. In the phased case for diploid samples, we can compute f_{11} as $\texttt{POPCNT}(\mathbf{B}_j \texttt{ AND } \mathbf{B}_k)$. Naturally, the inverse property must be true such that f_{00} equals $\texttt{POPCNT}(\sim \mathbf{B}_j \texttt{ AND } \sim \mathbf{B}_k)$, where $\sim$ is the bitwise `NOT` operator. Heterozygous counts, f_{01} and f_{10}, must evaluate to true when there is bitwise exclusivity at position i between two vectors: $\mathbf{B}_j^i \neq \mathbf{B}_k^i$. This inequality criteria is trivially computed as $\texttt{POPCNT}(\mathbf{B}_j \texttt{ AND } \sim \mathbf{B}_k)$ and $\texttt{POPCNT}(\sim \mathbf{B}_j \texttt{ AND } \mathbf{B}_k)$ for f_{01} and f_{10}, respectively. All of these functions combined use no more then five bitwise instructions (example implementation using the AVX2 instruction set: **Figure 3.4a**, **Figure 3.5a**, **Table 3.1**, and **Table 3.2**).

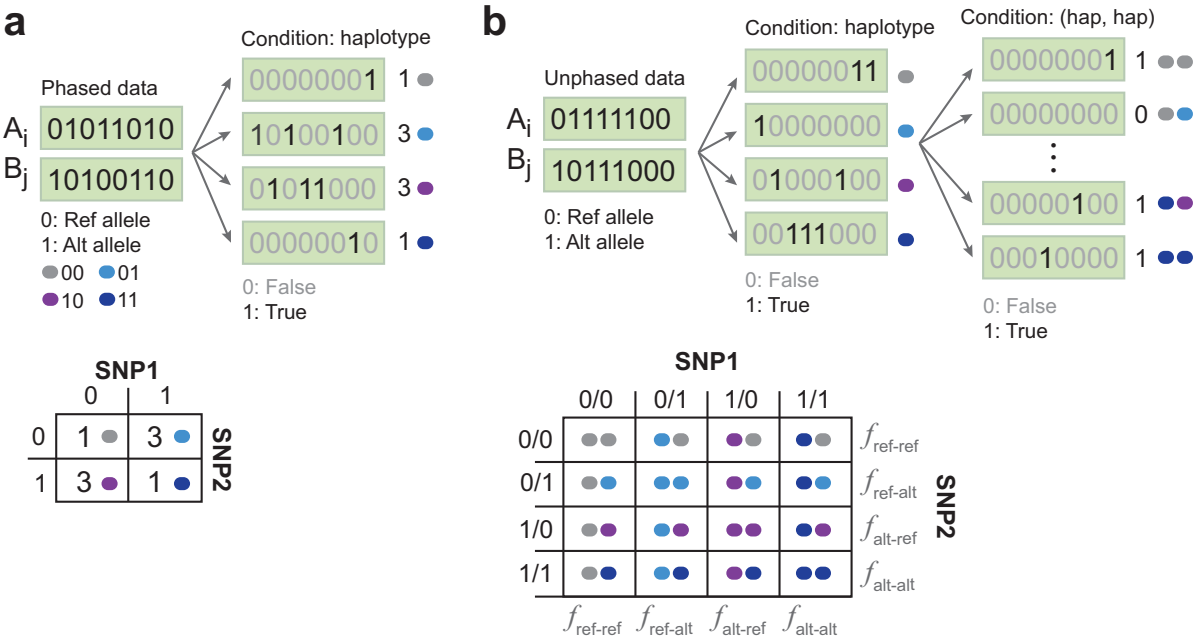


Figure 3.4 Relationship between the conditioned bitvectors. a) Given two target vectors A and B at some position $j \in [0, k)$, where $k \in [0, \lceil pm/w \rceil)$ for ploidy $p = 2$ and number of samples n and machine-word width in its w, we want to compute f_{00}, f_{01}, f_{10}, and f_{11}, color-coded as grey, light blue, purple, and dark blue. Using simple bitwise operations, we can update the counts in a 2×2 contingency table (Punnett square) of alleles for two sites, SNP1 and SNP2. Various linkage-disequilibrium-related statistics are then computed directly from this matrix. Observe that $|f_{00} \cup f_{01} \cup f_{10} \cup f_{11}| = pn$ when no data is missing. **b)** In the unphased case, the four haplotype conditions $f(\cdot)$ are first computed as in **a)** and then used as input to compute a 4×4 contingency table of genotypes for two sites, SNP1 and SNP2. Frequently, this 4×4 contingency table is represented as the simpler 3×3 matrix where all heterozygous genotypes (0/1 and 1/0) are collapsed to 1/0. Collapsing all genotypes into this representation will result in incorrect results in datasets with mixed phasing as phased-phased computations will be incorrect.

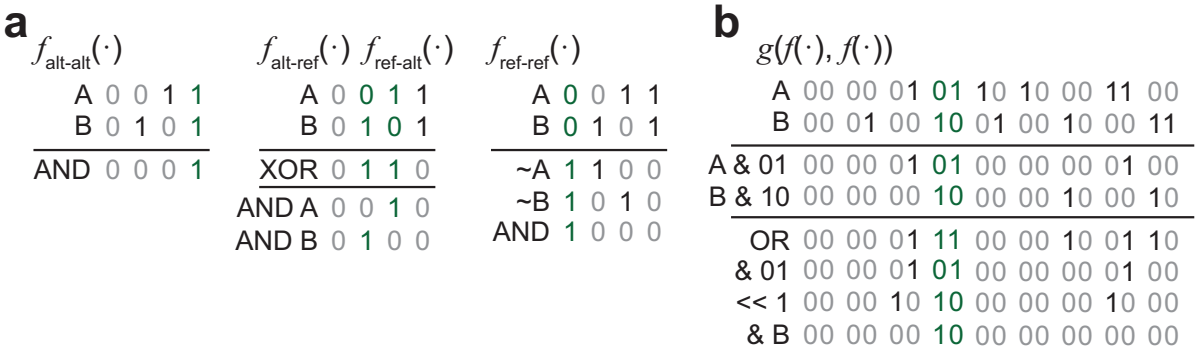


Figure 3.5 Worked example for bitwise operations conditioning for a 2-locus haplotype or 2-locus joint genotype. a) For the phased case, given two bitvectors (machine words), $A = [0, 0, 1, 1]$ and $B = [0, 1, 0, 1]$, we can compute all possible 2-locus haplotype combination using one of the four conditioning functions f_{00}, f_{01}, f_{10}, and f_{11} with the result that a bit is set (1) if the target 2-locus haplotype is observed. The haplotype case of joint alternative alleles (f_{11}) is the simplest as this condition correspond to the default encoding. **b)** In the unphased case, additional inference is generally required because of the phased uncertainty problem caused by jointly heterozygous genotypes. As in **a)**, the goal is to have a bit set (1) when a pair of conditions are true. In this case, the target condition are pairs of conditioned outputs from $f(\cdot)$. If $A \neq B$ then we can guarantee that $|A \wedge B| = 0$ resulting in the reduced universe of pairwise 2-bit combinations: $\{[00, 00], [00, 01], [01, 00], [01, 10], [10, 01], [10, 00], [00, 10], [11, 00], [00, 11]\}$. The function $g(\cdot)$ takes as input two pairs of 2-bit combinations as computed with $f(\cdot)$ from **a)**. The series of bitwise operation always require 6 operations (not counting the final `POPCNT` operation) regardless of the input parameters $f_1(\cdot)$ and $f_2(\cdot)$. Abbreviations: AND, bitwise AND; XOR, bitwise exclusive OR; $\sim$, bitwise NOT; &, bitwise AND; $<<$, bitwise left shift.

Table 3.1 Example implementation for conditioning a pair of bitmaps given a 2-locus haplotype using the AVX2 Instruction Set Architecture (ISA). Most modern compilers will generate identical assembly for implicit and explicit bitwise operations on vectors. For example, `_mm256_and_si256(A, B)` and `A & B`, will both result in a `VPAND` instruction being used. For sake of clarity, I describe these conditioning functions using explicit functions using the AVX2 ISA as an example. All listed functions have two arguments, `A` and `B`, that each takes as input a vector of bitmaps. If data is missing, another pair of masks are merged $M = M_A$ bitor M_B, where M is equal in length to the input vectors A and B. The special vector `ONE_MASK` is a vector with all bits set (1111...1).

Function	**Operations**
ALT_ALT(A, B)	`_mm256_and_si256(A, B)`
REF_REF(A, B)	`_mm256_and_si256(_mm256_xor_si256(A, ONE_MASK), _mm256_xor_si256(B, ONE_MASK))`
ALT_REF(A, B)	`_mm256_and_si256(_mm256_xor_si256(A, B), B)`
REF_ALT(A, B)	`_mm256_and_si256(_mm256_xor_si256(A, B), A)`
ALT_ALT_MISS(A, B, M)	`_mm256_and_si256(ALTALT(A, B), M)`
REF_REF_MISS(A, B, M)	`_mm256_and_si256(REFREF(A, B), M)`
ALTREF_MISS(A, B, M)	`_mm256_and_si256(ALTREF(A, B), M)`
REFALT_MISS(A, B, M)	`_mm256_and_si256(REFALT(A, B), M)`

Table 3.2 Latency and throughput for the different instructions used to condition two bitmaps given 2-locus haplotype state (**Table 3.1**). Data is presented for two different Intel processor microarchitectures: Skylake and Broadwell. Skylake have the AVX-512 ISA and Broadwell have the AVX2 ISA. Data from https://software.intel.com/sites/landingpage/IntrinsicsGuide/ Lat: Latency; Tput: Throughput (CPI), CPI: Cycles per instruction.

Function	**Instruction**	**Skylake**		**Broadwell**	
		Lat.	**Tput.**	**Lat.**	**Tput.**
_mm256_and_si256	VPAND	1	0.33	1	0.33
_mm256_xor_si256	VPXOR	1	0.33	1	0.33
_mm256_or_si256	VPOR	1	0.33	1	0.33
_mm256_slli_si256	VPSLLQ	1	1	1	1
_mm256_srli_si256	VPSRLDQ	1	1	1	1

For unphased data, it is possible to describe set intersections given the extended 2-symbol alphabet $\sigma_2 \in \{h_{00}, h_{01}, h_{10}, h_{11}\}$. In this situation, (h_A, h_B)-tuples can be described in terms of conditional positional index vectors for genotypes (Equation 3.6):

$$\begin{pmatrix} g(f_{00}(\cdot), f_{00}(\cdot)) & g(f_{01}(\cdot), f_{00}(\cdot)) & g(f_{10}(\cdot), f_{00}(\cdot)) & g(f_{11}(\cdot), f_{00}(\cdot)) \\ g(f_{00}(\cdot), f_{01}(\cdot)) & g(f_{01}(\cdot), f_{01}(\cdot)) & g(f_{10}(\cdot), f_{01}(\cdot)) & g(f_{11}(\cdot), f_{01}(\cdot)) \\ g(f_{00}(\cdot), f_{10}(\cdot)) & g(f_{01}(\cdot), f_{10}(\cdot)) & g(f_{10}(\cdot), f_{10}(\cdot)) & g(f_{11}(\cdot), f_{10}(\cdot)) \\ g(f_{00}(\cdot), f_{11}(\cdot)) & g(f_{01}(\cdot), f_{11}(\cdot)) & g(f_{10}(\cdot), f_{11}(\cdot)) & g(f_{11}(\cdot), f_{11}(\cdot)) \end{pmatrix} \tag{3.6}$$

These counts are sufficient to compute $\widehat{f_{11}}$ using the analytic solution described above (**Figure 3.4b** and **Figure 3.5b**). Since these tuples are limited to the universe of f, the universe of possible 2-tuple states are limited to the search space $|\sigma_2|^2$. This state space can be refactored by noting that all jointly heterozygous (central cells) and all symmetric genotypes can be collapsed (Equation 3.7 and **Figure 3.6**):

$$\begin{cases} g(f_{00}(\cdot), f_{00}(\cdot)) \\ g(f_{01}(\cdot), f_{00}(\cdot)) + g(f_{10}(\cdot), f_{00}(\cdot)) \\ g(f_{11}(\cdot), f_{00}(\cdot)) \\ g(f_{00}(\cdot), f_{01}(\cdot)) + g(f_{00}(\cdot), f_{10}(\cdot)) \\ g(f_{01}(\cdot), f_{01}(\cdot)) + g(f_{10}(\cdot), f_{01}(\cdot)) + g(f_{01}(\cdot), f_{10}(\cdot)) + g(f_{10}(\cdot), f_{10}(\cdot)) \\ g(f_{11}(\cdot), f_{01}(\cdot)) + g(f_{11}(\cdot), f_{10}(\cdot)) \\ g(f_{00}(\cdot), f_{11}(\cdot)) \\ g(f_{01}(\cdot), f_{11}(\cdot)) + g(f_{10}(\cdot), f_{11}(\cdot)) \\ g(f_{11}(\cdot), f_{11}(\cdot)) \end{cases} \tag{3.7}$$

This reduced representation (9 instead of 16 states) is cheaper, and therefore faster, to calculate as we do not need to distinguish between certain pairs of genotypes when the input data is unphased (**Table 3.3**).

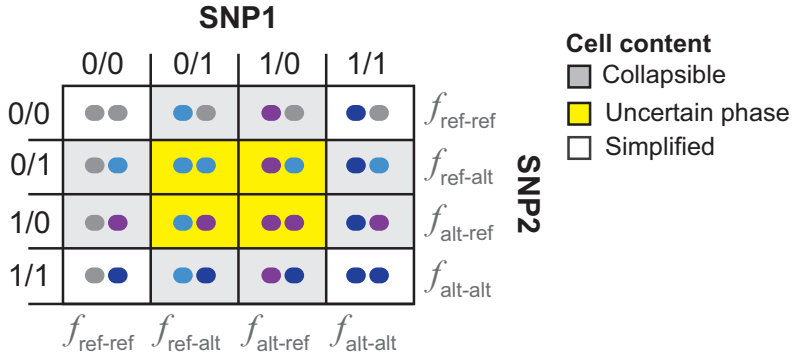


Figure 3.6 State space reduction for unphased genotypes. Tomahawk must represent a 2-locus genotype matrix as a 4×4 matrix because heterozygous genotypes are not collapsed from 0/1 and 1/0 into 1/0 in order to support datasets with mixed phasing. Although this larger matrix require an additional 7 pairwise states ($9 \rightarrow 16$) compared to the frequently used 3×3 matrix, it is possible to reduce computation by noting that all possible genotype pairs that involve a heterozygous genotype can be collapsed. Four genotype pairs can have their results collapsed (summed) into a single cell in a smaller matrix (grey). All jointly heterozygous genotypes can be jointly computed without disregard for their individual values (yellow). These insights result in moderate savings in compute.

In certain edge cases, we note that there are several states that cannot exist and would require unnecessary compute. For example, the state space can be reduced

to $f_{01}(\mathbf{G}_j, \mathbf{G}_k) \cup f_{11}(\mathbf{G}_j, \mathbf{G}_k)$ or $f_{10}(\mathbf{G}_j, \mathbf{G}_k) \cup f_{11}(\mathbf{G}_j, \mathbf{G}_k)$ when $\mathbf{G}_j = \mathbf{0}$ or $\mathbf{G}_k = \mathbf{0}$, respectively. Similarily, when both genotypic vectors $\mathbf{G}_j$ and $\mathbf{G}_k$ are $\mathbf{0}$ or $\mathbf{1}$ then the state space is exclusively either $f_{00}(\mathbf{G}_j, \mathbf{G}_k)$ or $f_{11}(\mathbf{G}_j, \mathbf{G}_k)$.

Table 3.3 Example implementation for conditioning a pair of bitmaps given a 2-locus genotype using the AVX2 Instruction Set Architecture (ISA). These functions takes as input the output from the functions **Table 3.1**. All the listed functions have two or more arguments, (`A`, `B`, ...), that each takes as input a conditioned output vector of bitmaps. The special vectors `ML` and `MU` encode for the two-bit sequence (0101...01) and (1010...10), respectively.

Function	Operations

UNPHASED	`_mm256_and_si256(_mm256_slli_epi64(`
	`_mm256_slli_epi64(`
	`_mm256_and_si256(_mm256_or_si256(_mm256_and_si256(A, MH),`
	`_mm256_and_si256(B, ML)), ML),`
	`1), A)`

UNPHASED_PAIR	`_mm256_or_si256(_mm256_srli_epi64(UNPHASED(A, B), 1), UNPHASED(C, D))`
UNPHASED_SPECIAL	`_mm256_and_si256(_mm256_and_si256(_mm256_srli_epi64(A, 1), A), ML)`

3.2.8 Scalar-bitmap intersections

In case of sites with small allele count (sparse vectors), it is possible to further accelerate the computation of the cardinality of set intersections by additionally storing the alternative allele positions in an array of scalars. Using these scalars, we count the set bits in the bitmap with the largest allele count given the allele positions in the other (**Algorithm 2** and **Figure 3.7**). This approach runs in $O(\min(|\mathbf{G}_i|, |\mathbf{G}_j|)$-time and can result in considerable speedups (**Table 3.16** and **Table 3.17**)

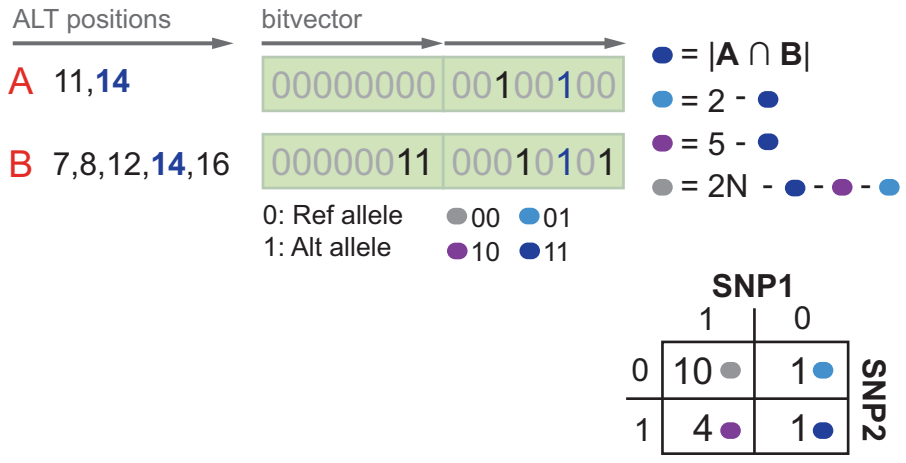


Figure 3.7 Algorithm for scalar-bitmap intersections. If either, or both, bitvectors A and B have a small cardinality (generally $< m/200$) and have no missing values then it possible to accelerate the computation of the cardinality of set intersections by storing an additional list of scalars for each vector and perform scalar-bitmap intersections. This list of scalars values, L_A and L_B, maps to the positions with a set bit (alternative allele bit is set). The cardinality of the set intersection for f_{11} (dark blue) can then be computed as $|L_B \in A|$ or $|L_A \in B|$ (see Algorithm 2). In this case, the remaining cell counts (light blue, purple, and grey) are then known. The resulting 2×2 matrix is shown for this example.

Algorithm 2 Scalar-bitmap intersection

function INTERECTCOUNTBITMAPSCALAR(A, B, S_A, S_B, n_A, n_B)
    Let A and B be the uncompressed 64-bit bitmaps
    Let S_A and S_B be vector of scalars and n_A and n_B their lengths
    if $n_A < n_B$ **then**
        for $i = 0 \ldots n_A$ **do**
            c += ($B[S_A[i]/64]$ bitand $(1 << (S_A[i] \bmod 64))) \neq 0$
        end for
    else
        for $i = 0 \ldots n_B$ **do**
            c += ($A[S_B[i]/64]$ bitand $(1 << (S_B[i] \bmod 64))) \neq 0$
        end for
    end if
    return c
end function

3.2.9 Sparse vector compression

To address the inefficient memory usage of sparse bitmaps when the number of samples $>2^{18}$, I implement a hybrid bitmap compression scheme (**Figure 3.3**)). Instead of storing uncompressed bitmaps, we partition the universe $[0, n)$ into non-overlapping chunks of size 2^{16} (8 kb) that store dense and sparse chunks separately. This approach was first described in the Roaring bitmap compression scheme[67, 66]. Our approach follows this approach very closely with a few key differences. Dense chunks are defined as having >4096 set bits and are stored as standard uncompressed bitmaps. Sparse chunks store 32-bit integers by first storing their shared 16-bit prefix as a scalar and their 16-bit lowest significant bits in an array. This is sufficient information to restore their original 32-bit

values while saving 2 bytes of storage per scalar and also enables vectorized algorithms to operate on a larger number of values per register. The sparse-to-dense transition threshold of 4096 values was selected because $[0, 4096)$ 16-bit values require $\leq 2^{16}$ bits while bitmaps invariantly require 2^{16} bits independent of density. Because of the limited haplotypic diversity in humans[123] (**Figure 3.8**), I expect this compression scheme to work well on future large human datasets.

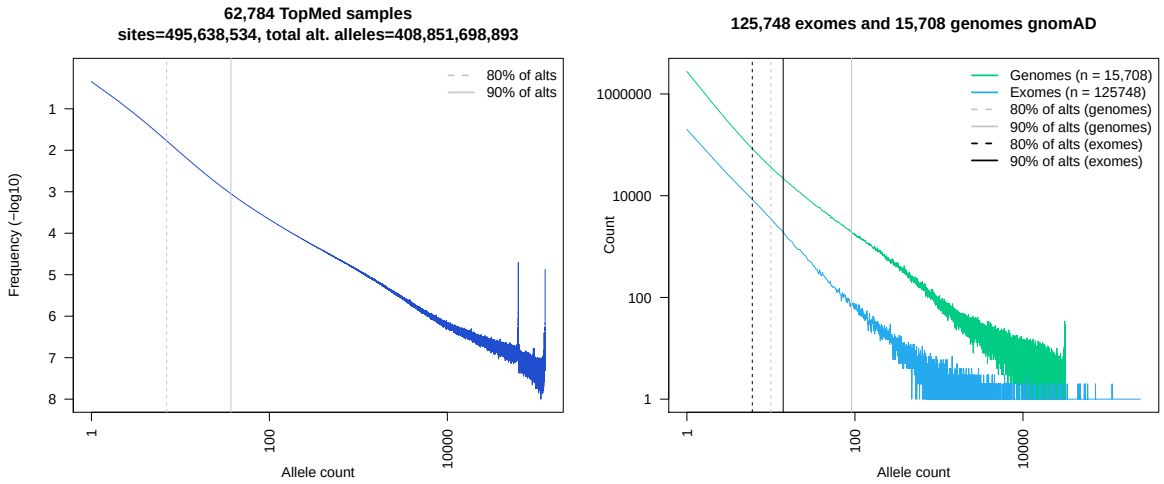


Figure 3.8 Allele count distribution on the Trans-Omics for Precision Medicine (TOPMed) Program and The Genome Aggregation Database (gnomAD) datasets. These large datasets demonstrate that increasing the number of samples result in an increasingly sparse genetic matrix because of the limited haplotypic diversity in humans. (Left panel) Whole-genome sequenced data for 62,784 individuals (data freeze 5) reveal that 80% of variant sites have <7 alternative alleles (alts, dashed grey line) and 90% have <37 alternative (solid grey line) genome-wide. Unsurprisingly, 223 million sites (45%) have an allele count of one. (Right panel) Allele frequency distribution for 125,748 samples with exome sequencing data and 15,708 samples with whole-genome sequencing data for chromosome 20.

To query for the presence of a 32-bit integer x using the hybrid approach we first search for the target container having the prefix $x/2^{16}$ using a vectorized binary search in $O(n_1 + n_2)$-time[67], where n_1 and n_2 are the respective length of prefixes in either container. If the target container exist and is a bitmap container, then we evaluate the predicate $(b[x/64]$ bitand $1 << (x \bmod 64)) \neq 0$ in $O(1)$-time. If the target container is an array container then we perform a binary search by querying x against the array of 16-bit suffixes.

We implement the set intersection count operation using a two-stage approach (Algorithm 3): (1) we compute the set intersection between the 16-bit prefix values from two containers to identify chunks with potential overlaps; (2) if no container prefixes overlap then the resulting set intersection is empty. Otherwise, we compare integers in each container using either bitmap-bitmap, bitmap-scalar, or scalar-scalar algorithms. Bitmap-bitmap and bitmap-scalar intersections are performed as described for contiguous-memory bitmaps. Scalar-scalar intersections use a modified vectorized binary search such that it returns a count instead of the actual resulting set.

Algorithm 3 Computing set intersection using hybrid compression

function IntersectContainers(C_A, C_B) ▷ C - hybrid containers A and B
  Let A_p and B_p be arrays of 16-bit prefixes for two containes
  Let P be a vector of (j,k)-tuples indexing where $A_p = B_p$
  if $|P| = 0$ **then return** 0
  else
    for $i \in [0, 1, \ldots, |P|)$ **do**
      Let $|A[P[i]_1]|$ and $|B[P[i]_2]|$ be the cardinality in a container
      if $|A[P[i]_1]| > 4096$ and $|B[P[i]_2]| > 4096$ **then**
        Let C_a and C_b be an indexed array of 1024 64-bit integers
        for $p \in [0, 1, \ldots, 1024)$ **do**
          $c \leftarrow c + \text{popcnt}(C_{ap}$ AND $C_{bp})$
        end for
        return c
      else if $|A[P[i]_1]| > 4096$ and $|B[P[i]_2]| < 4096$ **then**
        return IntersectBitmapScalar($\cdot$)
      else if $|A[P[i]_1]| < 4096$ and $|B[P[i]_2]| > 4096$ **then**
        return IntersectBitmapScalar($\cdot$)
      else
        return IntersectScalarScalar($\cdot$)
      end if
    end for
  end if
end function

This hybrid compression approach is not without downsides as illustrated by the case when the input data is extremely sparse. For example, the input dataset $(1 \times 2^{16}, 2 \times 2^{16}, \ldots, k \times 2^{16})$ will store a single scalar per container in addition to the container

overhead cost. Under these circumstances the meta information in the chunk container require more space than the stored integers themselves. Although we recognize this inefficiency in terms of compression for extremely sparse set in terms of compression we do not address this shortcoming in this work.

Both these sparse-vector intersection algorithms and the standard SIMD-accelerated bitmap-bitmap intersection algortihms were released as the standalone library Storm bitmaps using functional multiversioning (FMV) that supports all modern instruction-set architectures (ISAs). Storm bitmaps are available online at `Storm bitmaps`. For the remainder of this chapter I refer to the sparse bitmap representation as Storm and the standard contiguous memory implementation as Storm-contig.

3.2.10 Heuristic decision tree

Our proposed algorithms display optimal performance at different allele frequencies and cohort sizes. In order to maximize performance, we apply a heuristic-based decision tree that automatically selects the best method given the properties of the input pairs of genotype vectors.

3.2.11 Memory-aware load balancing

It is generally not possible to load all the required data into memory when handling hundreds of thousands to many millions of samples. In this situation we would have to load only the current data blocks into memory from disk and then release the memory in order to load the next block when finished. This approach impose considerable overhead costs and disk-based latency resulting in slower overall compute. To illustrate this problem, consider the situation with a set of three blocks $\{1, 2, 3\}$ that should be compared pairwise. Each block can be compared pairwise using a simple iterative algorithm:

1. Load blocks 1 and 2 into memory and compare $\{1, 2\}$
2. Release block 2 from memory
3. Load block 3 and compare $\{1, 3\}$
4. Release block 1 from memory
5. Load block 2 from memory and compare $\{2, 3\}$

Block 2 have now been loaded and released twice. This undesired memory- and I/O-overhead grows square to the number of blocks, $O((B-1)^2)$, where B is the number of blocks. For example, using standard import parameters, chromosome 20 for the 1000

Genomes Project data has 1,696 blocks. Without addressing this problem, there would be 2,873,025 overhead loads.

If there are B number of blocks in a file and N number of variants, then let N_i be the number of variants in block i. All possible block combinations has to be computed and this is equivalent to the upper triangular of a N^2 square matrix for a total of $\binom{N}{2}$ pairwise comparisons. We can factorize out the number of comparisons in these blocks that either border the diagonal (first term) and those that do not (second term):

$$\binom{N}{2} = \sum_{i=1}^{i \leq B} \binom{N_i}{2} + \sum_{i=1}^{i \leq B} \sum_{j=i+1}^{j \leq B} N_i N_j$$

If we naively balance the workload into linear slices of $\mathbf{B}$ we will use unnecessary amounts of memory. Instead, we partition the subproblems such that blocks iteratively choose their available k-nearest neighbours. Because of the blockwise memory layout of Tomahawk, this approach is trivially approximated by slicing the B^2 matrix into $\binom{P}{2} + P$ sections. This partitioning scheme is always guaranteed to use smaller amounts of memory when $P > 1$ and $B > 1$. Next, the workload within each subproblem is partitioned according to $\lceil t/N_P \rceil$, where t is the desired number of threads and N_P the number of variants in sub-problem P. Sub-problems bordering the diagonal P_d will always perform less work ($\binom{P_d}{2}$ comparisons, see equation 3.8) compared to non-diagonal P_o sub-problems ($P_o^a P_o^b$ comparisons, see equation 3.9).

The index of block members of the diagonal set are images of the function:

$$d(i) = 0 + \sum_{j=0}^{j<i} P - j \tag{3.8}$$

Similarly, indices for blocks not anchored to the diagonal are members of the function:

$$s(i) = \sum_{i=0}^{i<P} \sum_{j=i+1}^{j<P} j \tag{3.9}$$

This partitioning of workload assumes that the comparison throughput is constant in each sub-problem.

3.2.12 Cache-blocking, out-of-order execution, and parallel computing

By performing $m(m-1)/2$ pairwise operations between vectors of bitmaps we recognize that if n is small then several such vectors will fit into low-level cache such as L1 through

L3 (**Table 3.4**). The primary goal of the cache system is to ensure that the processor have the next required data in memory by the time it is needed (a cache hit). Memory access times in these cache regions are faster by over an order of magnitude compared to main memory (DRAM). It is therefore desirable to keep computing on data in these low-level cache levels as long as possible.

Table 3.4 Reproduced in part from Intel (https://software.intel.com/en-us/articles/memory-performance-in-a-nutshell). Abbrevations: L1 (Level one), L1 (Level two), L1 (Level three), Multi-channel DRAM (MCDRAM), dual inline-memory modules (DIMMs), double data rate (DDR). *MCDRAM is a 3D-stacked DRAM that was used in Intel Xeon Phi processors (Knights Landing).

Memory	Size	Latency	Bandwidth
L1 cache	32 KB	1 nanosecond	1 TB/second
L2 cache	256 KB	4 nanoseconds	1 TB/second
L3 cache	8 MB or more	10x slower than L2	≥400 GB/second
MCDRAM*		2x slower than L3	400 GB/second
Main memory on DDR DIMMs	4 GB-1 TB	2x slower than L3	100 GB/second

An important algorithmic change involves reorganizing data memory access such that cache is loaded with a small subset of the data. By reusing cached memory we can reduce the need to retrieve new data from memory and thereby reduce memory bandwidth pressure.

Transforming the unblocked version for diagnonal components $d(i)$ (Equation 3.8 and square components $s(i)$ (Equation 3.9) from naive loops (**Fig. 3.9** and **Fig. 3.10**) into blocked versions (**Fig. 3.11** and **Fig. 3.12**) result in considerable perormance gains (data not shown).

```
for (uint32_t i = 0; i < n_data; ++i) {
  for (uint32_t j = i + 1; j < n_data; ++j) {
    count += intersect(data[i], data[j], length);
  }
}
```

Figure 3.9 No blocking for a diagonal component.

```
for (uint32_t i = 0; i < n_data; ++i) {
  for (uint32_t j = 0; j < n_data; ++j) {
    count += intersect(data1[i], data2[j], length);
```

```
  }
}
```

Figure 3.10 No blocking for a square component

```
for (/**/; i + bsize <= n_data; i += bsize) {
  // Diagonal component
  for (uint32_t j = 0; j < bsize; ++j) {
    for (uint32_t jj = j + 1; jj < bsize; ++jj) {
      count += intersect(data[i+j], data[i+jj], length);
    }
  }

  // Square component
  uint32_t curi = i;
  uint32_t j = curi + bsize;
  for (/**/; j + bsize <= n_data; j += bsize) {
    for (uint32_t ii = 0; ii < bsize; ++ii) {
      for (uint32_t jj = 0; jj < bsize; ++jj) {
        count += intersect(data[curi+ii], data[j+jj], length);
      }
    }
  }

  // Residual
  for (/**/; j < n_data; ++j) {
    for (uint32_t jj = 0; jj < bsize; ++jj) {
      count += intersect(data[curi+jj], data[j], length);
    }
  }
}
// Residual tail
for (/**/; i < n_data; ++i) {
  for (uint32_t j = i + 1; j < n_data; ++j) {
    count += intersect(data[i], data[j], length);
  }
}
```

Figure 3.11 Cache-blocked version of Figure 3.9 using a step size of `bsize` in a diagonal component.

```
for (/**/; i + bsize <= n_data1; i += bsize) { // block1
  j = 0;
  for (/**/; j + bsize <= n_data2; j += bsize) { // block2
    for (uint32_t ii = 0; ii < bsize; ++ii) {
      for (uint32_t jj = 0; jj < bsize; ++jj) {
        count += intersect(data1[i+ii], data2[j+jj], length);
      }
    }
  }
  // Residual
  for (/**/; j < n_data2; ++j) { // block2
    for (uint32_t jj = 0; jj < bsize; ++jj) {
      count += intersect(data1[i+jj], data2[j], length);
    }
  }
}

// Residual tail
for (/**/; i < n_data1; ++i) {
  for (j = 0; j < n_data2; ++j) {
    count += intersect(data1[i], data2[j], length);
  }
}
```

Figure 3.12 Cache-blocked version of Figure 3.10 using a step size of `bsize` in a square component.

Using these functions we can trivially implement the load balancing approach from Section 3.2.11 (**Figure 3.13**). Although this pairwise comparison of blocks of variants enables embarrassingly parallel computing, there is a moderate overhead associated with storing memory in non-contiguous locations. As an illustrative example, computing the cardinality of set intersection of incremental slices $(0, [10, 200] \times 10^3)$ of variants from chromosome 6 from the 1000 Genomes Project dataset demonstrates as ∼50% loss of performance (**Table 3.5**). This decrease in performance is attributed to the additional >2-fold branch misses and >7-fold more memory accesses for the variant-blocked pattern

(although >95% are cache hits) compared to a contiguous memory access model (data not shown). Using this approach, compute decrease linearly with increased number of physical threads (**Figure 3.14**).

```
for (int i = 0; i < n_blocks; ++i) {
  total += intersect_block_diagonal(block[i]);
  for (int j = i + 1; j < n_blocks; ++j) {
    total += intersect_block_square(block[i], block[j]);
  }
}
```

Figure 3.13 Computing XX^T can then be computed using the subroutines detailed in Figure 3.11 and Figure 3.12.

Table 3.5 Loss of performance due to memory access and branch misprediction overhead when using a variant-blocked model compared to a pure contiguous memory model. Using data from 1000 Genomes Project chromosome 6[123] slices from $(0, [10, 200] \times 10^3)$ in steps of 10^3 variants we compared a contiguous model (all variants in aligned and contiguous memory) or a variant-blocked approach with block size set to 50×10^3. Both models used internal memory blocking using 400 variants. Chit.: Cache hits; Cmiss.: Cache misses; CCW: CPU cycles / 64-bit word

Variants	Method	Chit. (%)	Cmiss. (%)	Time (s)	Fold diff	CCW	MB/s
10,000	Cont.	1.00	0.00	0.4	1.72	0.17	163,323
	Blocked	0.96	0.04	0.6		0.27	94,907
20,000	Cont.	0.94	0.06	1.8	1.53	0.19	134,906
	Blocked	0.93	0.07	2.7		0.30	88,016
30,000	Cont.	0.95	0.05	3.2	1.48	0.16	170,095
	Blocked	0.98	0.02	4.7		0.25	115,190
40,000	Cont.	0.95	0.05	6.1	1.71	0.17	158,061
	Blocked	0.94	0.06	10.4		0.29	92,680
50,000	Cont.	0.95	0.05	8.7	1.51	0.17	173,952
	Blocked	0.98	0.02	13.1		0.25	115,435
60,000	Cont.	0.95	0.05	12.6	1.53	0.16	172,203
	Blocked	0.97	0.03	19.3		0.25	112,196
70,000	Cont.	0.94	0.06	17.1	1.47	0.16	172,697
	Blocked	0.98	0.02	25.2		0.25	117,395
80,000	Cont.	0.95	0.05	21.9	1.51	0.16	176,313
	Blocked	0.98	0.02	33.0		0.25	116,908
90,000	Cont.	0.96	0.04	28.3	1.46	0.17	172,332
	Blocked	0.98	0.02	41.3		0.24	118,248

100,000	Cont.	0.96	0.04	34.3	1.49	0.17	175,637
	Blocked	0.99	0.01	51.2		0.25	117,670
110,000	Cont.	0.95	0.05	41.9	1.46	0.16	174,154
	Blocked	0.98	0.02	61.3		0.24	118,974
120,000	Cont.	0.95	0.05	49.5	1.49	0.16	175,201
	Blocked	0.98	0.02	73.6		0.25	117,955
130,000	Cont.	0.96	0.04	58.7	1.47	0.17	173,481
	Blocked	0.98	0.02	86.4		0.25	117,831
140,000	Cont.	0.96	0.04	67.7	1.49	0.17	174,435
	Blocked	0.99	0.01	101.2		0.25	116,694
150,000	Cont.	0.95	0.05	78.8	1.46	0.17	172,000
	Blocked	0.99	0.01	115.1		0.25	117,823
160,000	Cont.	0.94	0.06	88.8	1.47	0.17	173,849
	Blocked	0.99	0.01	130.6		0.25	118,155
170,000	Cont.	0.95	0.05	100.2	1.51	0.17	173,883
	Blocked	0.98	0.02	150.9		0.25	115,394
180,000	Cont.	0.96	0.04	113.1	1.49	0.17	172,606
	Blocked	0.98	0.02	168.7		0.25	115,756
190,000	Cont.	0.96	0.04	126.6	1.49	0.17	171,918
	Blocked	0.98	0.02	188.6		0.25	115,387
200,000	Cont.	0.96	0.04	138.8	1.8	0.17	173,704
	Blocked	0.95	0.05	250.1		0.28	96,387

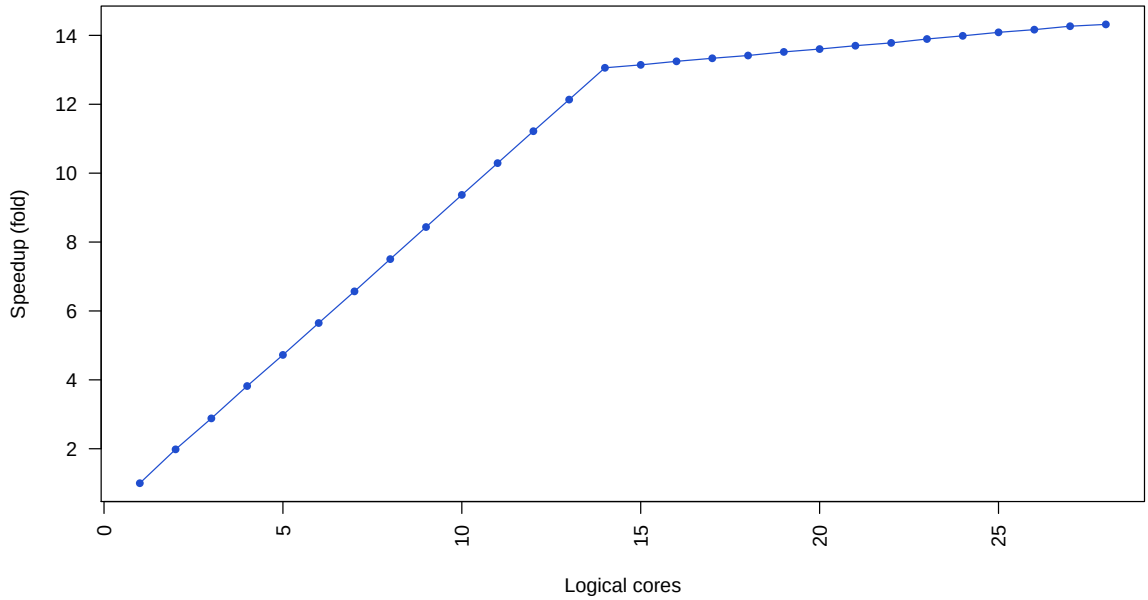


Figure 3.14 Improved performance with increasing number of logical threads. Computing growths linearly ($s = 0.9255t$, $R^2_{adj} = 0.9993$, where s is speedup for t physical threads) with increasing number of physical threads (14 cores). After 14 physical threads, additional hyper-threaded logical threads add around ∼1% per thread. This stark drop-off in additional performance gain is expected given the hardware-limited performance of the algorithms. Actual times are listed in **Table 3.6**. The host architecture used is a 14-core 22 nm Haswell Xeon E5-2697 v3 with the AVX2 ISA and hyper-threading enabled.

Table 3.6 Completion time for computing chromosome-wide linkage-disequilibrium as a function of threads. Performance increase linearly over all 14 physical threads when computing LD for chromosome 22 from the 1000 Genomes Phase 3 dataset (n = 5,008, **Figure 3.14**). After this point no further performance improvements are observed and all additionally expended energy is wasted. Also shown is the number of 2-locus haplotype comparisons per second as this measurement is directly comparable across different cohort sizes. The host architecture used is a 14-core 22 nm Haswell Xeon E5-2697 v3 with the AVX2 ISA and hyper-threading enabled. Abbreviations: 2L-cmps/s, 2-locus comparisons/second.

Threads	**Time**	**2L-cmps/s** (10^{12})
28	06m10,510s	8.14
27	06m11,867s	8.11

26	06m14,487s	8.05
25	06m16,566s	8.01
24	06m19,300s	7.95
23	06m21,808s	7.90
22	06m24,941s	7.83
21	06m27,235s	7.78
20	06m29,991s	7.73
19	06m32,397s	7.68
18	06m35,437s	7.62
17	06m37,837s	7.58
16	06m40,486s	7.53
15	06m43,620s	7.47
14	06m46,239s	7.42
13	07m17,141s	6.90
12	07m52,876s	6.37
11	08m35,479s	5.85
10	09m26,253s	5.32
9	10m28,853s	4.79
8	11m46,832s	4.26
7	13m27,627s	3.73
6	15m38,974s	3.21
5	18m42,992s	2.68
4	23m08,504s	2.17
3	30m41,014s	1.64
2	44m36,01s	1.13
1	88m25,01s	0.57

3.2.13 Sorting output data

Tomahawk can produce hundreds of gigabytes of compressed output data for genome-wide linkage-disequilibrium queries. As a design consequence, Tomahawk computes pairwise correlations between two sites using a block-wise out-of-order execution pattern that results in unsorted output. This volume of data can generally not be sorted in memory as the uncompressed file-sizes exceeds the available memory of most workstations by several orders of magnitude. In order to overcome this restriction, we implemented

algorithms belonging to the external sort family. This class of algorithms espouse the divide-and-conquer paradigm to reduce the computational challenge of sorting extremely large datasets using bounded resource requirements. Without losing generality, external sort algorithms comprises of three principal steps: (1) partitioning a input dataset $\mathbf{D}$ into non-overlapping blocks, called *pages*; (2) sorting each page in main memory and writing out partially sorted sub-files called *runs*; and (3) performing a k-way merge of the runs $\mathbf{d} \in \mathbf{D}$. The page size, k, is chosen such that uncompressed (raw) data can fit in main memory. The algorithm implemented in Tomahawk utilize a min-priority queue in the merge step to reduce complexity to $O(n \log n)$. For the sake of illustration, we assume that the cardinality of $\mathbf{D}$ (denoted as $|\mathbf{D}|$) is divisible by the number of runs, k (Algorithm 4).

Although efficient in memory and time complexity, this algorithm can be slow in practice on datasets involving hundreds of billions of records. This source of inefficiency comes from the single-threaded merging operation using the priority queue. In the naive implementation, every record has to be compared to the smallest entry in the priority queue before being either immediately written to a output buffer or possibly inserted into the queue. Fortunately, as a consequence of our dynamic execution order of blocks and cache-blocking of records, we can model a simple stochastic pattern of partial order given the distance from any given block. First we make some simplifying assumptions: (1) assume that the distributed workload is balanced; (2) every sub-problem (pairwise comparison of blocks) is finished equally fast; and (3) every processed block pair produce equal number of output records. Under these idealized conditions, we expect output records to be locally sorted, allowing us to skip many portions of the k-way merge procedure.

Algorithm 4 Outline of external sort of Tomahawk output (`TWO`) data

function SortTWO(D, k, T, O)▷ Where D - input TWO file, k - number of runs, T - temporary output file, O - output file
  Let $m = |\mathbf{D}|/k$ be the number of records per page **d**
  for $i = 1$ to k **do**
    d = **D**$[im, (i+1)m)$ ▷ Slice of dataset into page k
    Sort(**d**) ▷ Sort data in each page
    Write(**d**) ▷ Write runs to T
  end for

  Let $Q[1..k]$ be a vector of cyclic queues of r
  Let r be a TWO record
  Let $F[1..k]$ be a vector of file handles to runs
  Let M be a min-priority queue of (k, r)-tuples, where k is the run offset
  for $i = 1$ to k **do**
    $F[i]$.**open**(T).**seek**(im) ▷ Open temporary file and seek to the virtual offset for run i
    for $j = 1$ to n **do** ▷ Load n records into queue
      $Q[i]$.**push**($F[i]$.**read**())
    end for
    M.**push**($k, Q[i][0]$) ▷ Push the first record into the min-queue
  end for

  while M.**empty**() == FALSE **do** ▷ k-way merge of k runs
    $x = M$.**top**().k ▷ The run this record came from
    Write(M.**top**().r)
    M.**pop**() ▷ Remove this record from the min-queue
    $Q[x]$.**push**($F[x]$.**read**()) ▷ Read a new record from stream if possible
    if Q[x].empty() == FALSE **then**
      M.**push**($Q[x]$)
    end if
  end while
end function

3.2.14 Comparing performance

I compared performance for computing all-vs-all linkage-disequilibrium against PLINK[95] v1.90p 64-bit (17 Jun 2019). PLINK is most likely the most common program used to compute LD because of its simplicity and good overall performance owing to its heavy use of SIMD-accelerated bitmap operations. PLINK was run using the following command:

```
plink --r2 inter-chr --ld-window-r2 1 --bfile ${plink_in}
```

where we limit the output associations to $R^2 = 1$ to restrict the performance impact that disk input/output operations have on the benchmark.

As another reference point, I compared against the recently described Emerald[96] program that makes use of both bitmap-bitmap intersections and scalar-bitmap intersections together with informed subsampling. I used the following command when using informed subsampling:

```
emeraLD -i ${emerald_in}.vcf.gz
    -o STDOUT --threshold 1
    --phased --nmax 1000
    --window 1000000000 > /dev/null
```

otherwise additionally add the parameter `-nmax ${samples}`.

3.2.15 Experimental data sets

Individual chromosome BCF files were retrieved for each cohort and combined into a single dataset using the bcftools `concat` command.

- 1000 Genomes phase 3 (hg19)[123]: This dataset comprises variants and small insertions/deletions (indels) for 2,504 individuals of various ancestry.
- Haplotype Reference Consortium (hg19)[125]: This dataset comprises of a mixed cohort of 32,488 whole-genome sequenced individuals. There are >39 million SNPs with an allele count ≥ 5 and all indels have been removed. All samples from the 1000 genomes phase 3 project are included in this dataset.

3.2.16 Simulated data sets

Genotypes from idealized datasets were simulated using msprime version 0.5d[56] with the mutation rate and recombination rate per site per 4N generations set to 0.001 and effective population size to 10,000. Region and number of samples varies according to

the experiments detailed above. Because msprime simulates individual haplotypes, we simulated 2N haplotypes for each case and greedily combined adjacent haplotypes into a single diploid genome.

3.2.17 Computing environment

Code was compiled with GCC 8.3 using the optimization flags "-O3 -march=native" to restrict optimizations to the host-machine architecture. All tests were performed using a host machine with a Intel i3-8121U (10nm) Cannon Lake (x64) microarchitecture and a pair of NVMe solid-state hard drives operating in RAID-0 unless otherwise specified. Performance was measured using the Linux `perf` subsystem with the virtualized `PERF_COUNT_HW_CPU_CYCLES` counts as processor cycles (clockticks). Memory usage was measured using Linux `time` subroutines.

Computing chromosome-wide linkage-disequilibrium was evaluated using the heterogeneous compute cluster available at the Wellcome Sanger Institute by splitting each job into 45 parts onto different compute nodes. Each node was allowed to use 8 CPU cores (hyper-threading disabled).

3.2.18 Format descriptions

We outline several simple data formats used in the `tomahawk` ecosystem that enables efficient storage and analysis of output data:

1. `twk`: We store pre-compressed haplotype/genotype data a simple structure (`twk1_t`) in the array-of-structure orientation. For each variant site, we store: (1) the number of alternative alleles, (2) allele count, (3) allele number, (4) number of heterozygous alleles, (5) number of homozygous alleles, (6) Hardy-Weinberg equilibrium probability, (7) a set of bitpacked binary states including phasing and if any missing values are present, (8) the chromosome identifier and position, and (8) compressed haplotype/genotype data in array form. We partition the space $[0, m)$ into non-overlapping chunks such that a block never contain > 1 chromosomes, where m is the number of variant sites. Blocks are indexed using segmental statistics (storing $(\min(p_i), \max(p_i))$-tuples), where p_i are the list of positions in block i. Block are compressed using the general purpose compressor Zstd (https://facebook.github.io/zstd/).

2. `two`: Output associations are stored in a similar fashion using the simple data structure `twk1_two_t` instead of `twk1_t` in the `twk` format. We store (1) the start

and stop chromosome identifier, (2) start and stop position, (3) phasing information and missing information for both start and stop positions, (4) R and R^2, (5) D and D', (6) Fisher's exact P value (**Table 3.8**), (7) χ^2 critical value for the 3×3 or 2×2 contingency tables (**Table 3.9**), and (8) the 2-locus haplotype counts f_{00}, f_{01}, f_{10}, and f_{11}. Lastly, we store the union of 12 different 1-of-k encoded flags describing various states (**Table 3.7**). Block are compressed using the general purpose compressor Zstd (`https://facebook.github.io/zstd/`).

3. `ld`: Uncompressed and human-readable `two` data is stored in the `ld` format (**Table 3.10**). This format is immutable and cannot be directly queried.

Table 3.7 Description of 1-of-k encodings in the `two` format. A series of Boolean states (yes/no) are 1-of-k-encoded in a single 16-bit machine word for a record. All fourteen encoded states are stored as their union $k_0 + k_1 + \ldots + k_{13}$.

Bit	Num.	One-hot	Description
1	1	000000000000000**1**	Used phased math.
2	2	000000000000000**1**0	Acceptor and donor variants are on the same contig.
3	4	00000000000000**1**00	Acceptor and donor variants are far apart on the same contig.
4	8	0000000000000**1**000	There are ≥ 1 empty cell (referred to as complete).
5	16	000000000000**1**0000	Output correlation coefficient is perfect (1.0).
6	32	00000000000**1**00000	There are ≥ 1 possible solutions.
7	64	0000000000**1**000000	Output data was generated in 'fast mode'.
8	128	000000000**1**0000000	Output data is estimated from a subsampling of the total pool of genotypes.
9	256	000000000**1**00000000	Donor vector has missing value(s).
10	512	000000000**1**000000000	Acceptor vector has missing value(s).
11	1024	000000**1**0000000000	Donor vector has low allele count (<5).
12	2048	000000**1**00000000000	Acceptor vector has low allele count (<5).
13	4096	000**1**000000000000	Acceptor vector has a HWE-P value <1e-4.
14	8192	00**1**0000000000000	Donor vector has a HWE-P value <1e-4.

Table 3.8 Contingency table for 2-locus haplotype frequencies used to test for non-random associations between two binary variables. We statistically determine this using a Fisher's exact test.

	REF-A	**REF-B**
REF-B	f_{00}	f_{01}
ALT-B	f_{10}	f_{11}

Table 3.9 Contingency table for genotype-genotype frequencies used to statistically test for goodness-of-fit between the observed frequencies and the expected one under our model. Note that this matrix is usually presented as a 3×3 matrix where heterozygous genotypes have been collapsed. For technical reasons, we compute frequencies for a 4×4 matrix and then collapse down to a 3×3 matrix.

	0/0	**0/1**	**1/1**
0/0	$f_{00,00}$	$f_{01,00}$	$f_{11,00}$
0/1	$f_{00,01}$	$f_{01,01}$	$f_{11,01}$
1/1	$f_{00,11}$	$f_{01,11}$	$f_{11,11}$

Table 3.10 Column description for the human-readable `LD` format.

Column	**Description**
`FLAG`	Bit-packed boolean flags (see below)
`CHROM_A`	Chromosome for marker A
`POS_A`	Position for marker A
`CHROM_B`	Chromosome for marker B
`POS_B`	Position for marker B
`REF_REF`	Count of (0,0) haplotypes (h_{00})
`REF_ALT`	Count of (0,1) haplotypes (h_{01})
`ALT_REF`	Count of (1,0) haplotypes (h_{10})
`ALT_ALT`	Count of (1,1) haplotypes (h_{11})
`D`	Coefficient of linkage disequilibrium
`D'`	Normalized coefficient of linkage disequilibrium (scaled to $[-1, 1]$)
R	Pearson correlation coefficient
R^2	Squared pearson correlation coefficient
`P`	Fisher's exact test P-value of the 2x2 haplotype contingency table
`ChiSqModel`	χ^2 critical value of the 3x3 unphased table of the selected cubic root
`ChiSqTable`	χ^2 critical value of table (useful comparator when P = 0)

3.2.19 Visualization functions

Tomahawk can generate hundreds of gigabytes of compressed data. This scale of data would prevent the application of existing packages and solutions for graphically visualising and operating on this data. There are not only hardware limitations such as loading all the data into memory, or directly rendering billions of data points, but also more

practical considerations such as displaying such a vast number of data points in a finite number of pixels would result in an uninformative image.

Addressing these challenges, I developed a compressed data aggregation format and a package with native R-bindings for interfacing with Tomahawk libraries and provides additional graphical functionality called `rtomahawk`. This means exposing most of the features and flexibility of the C++ API while not sacrificing the usability that R provides. `rtomahawk` can be accessed at `https://github.com/mklarqvist/rtomahawk`.

Example visualization functions include: (1) plotting LD for a region (**Figure 3.15**); (2) upper- or lower-triangular plots of a LD matrix (**Figure 3.16**); (3) orienting plots to enable intergration with other graphical units (**Figure 3.17**); (4) calculating and plotting LocusZoom-like plots in close-to-real-time (**Figure 3.18**); (5) adding LD plots as a track to LocusZoom-like plots (**Figure 3.19**); (6) adding LD plots and gene tracks to LocusZoom-like plots (**Figure 3.20**).

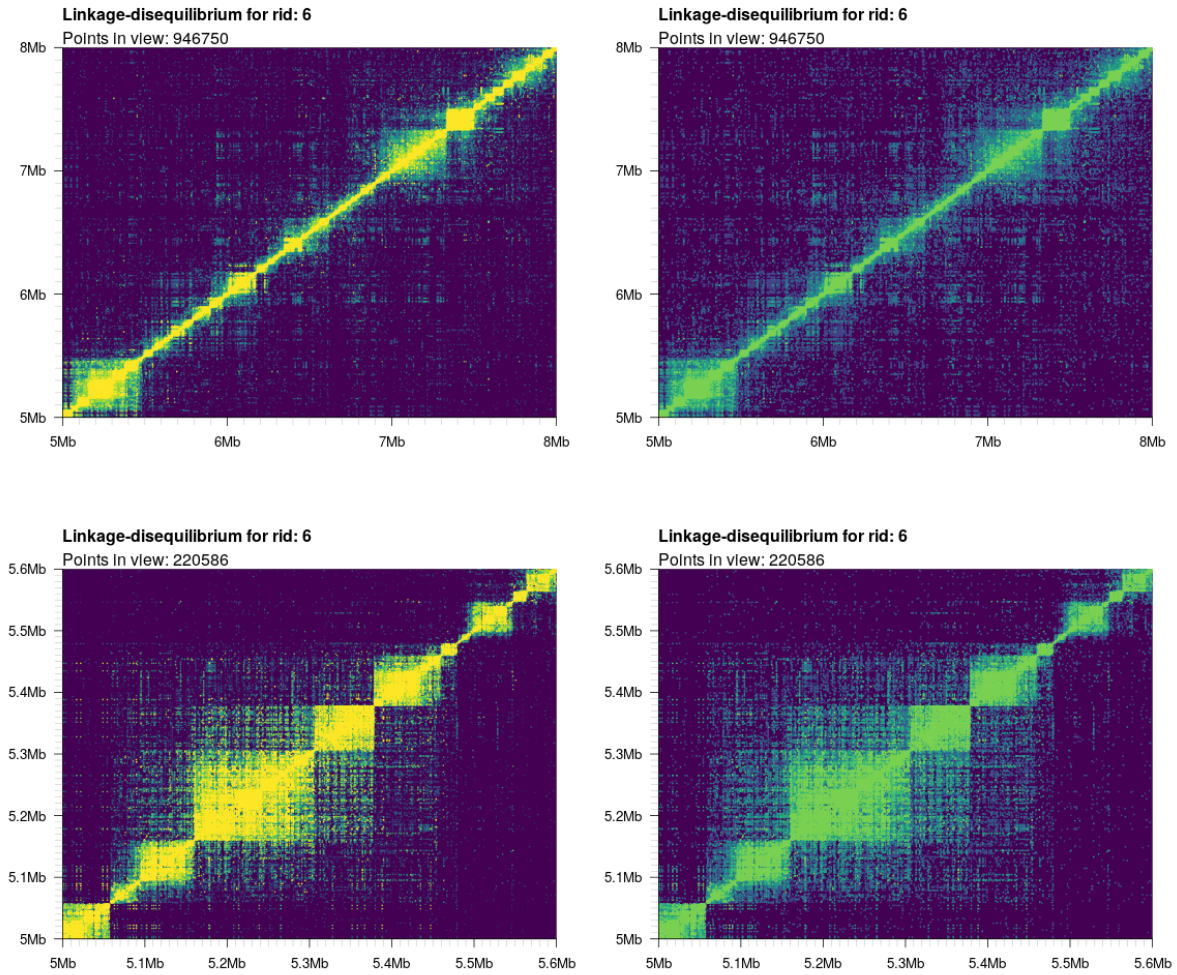


Figure 3.15 Example square LD plot using `plotLD` in rtomahawk. Because of the vast number of data points rendered and the finite amount of pixels available, we render data points with an opacity gradient scaled according to its R^2 value from $[0.1, 1]$. This allows for mixing of both colors and opacities to more clearly represent the distribution of the underlying data (left column). It is possible to disable this functionality by setting the optional argument opacity to `FALSE` (right column). In the following examples, we render both a large region (5-8 Mb, top row) and a small region (5.0-5.6 Mb, bottom row) with and without the opacity flag set.

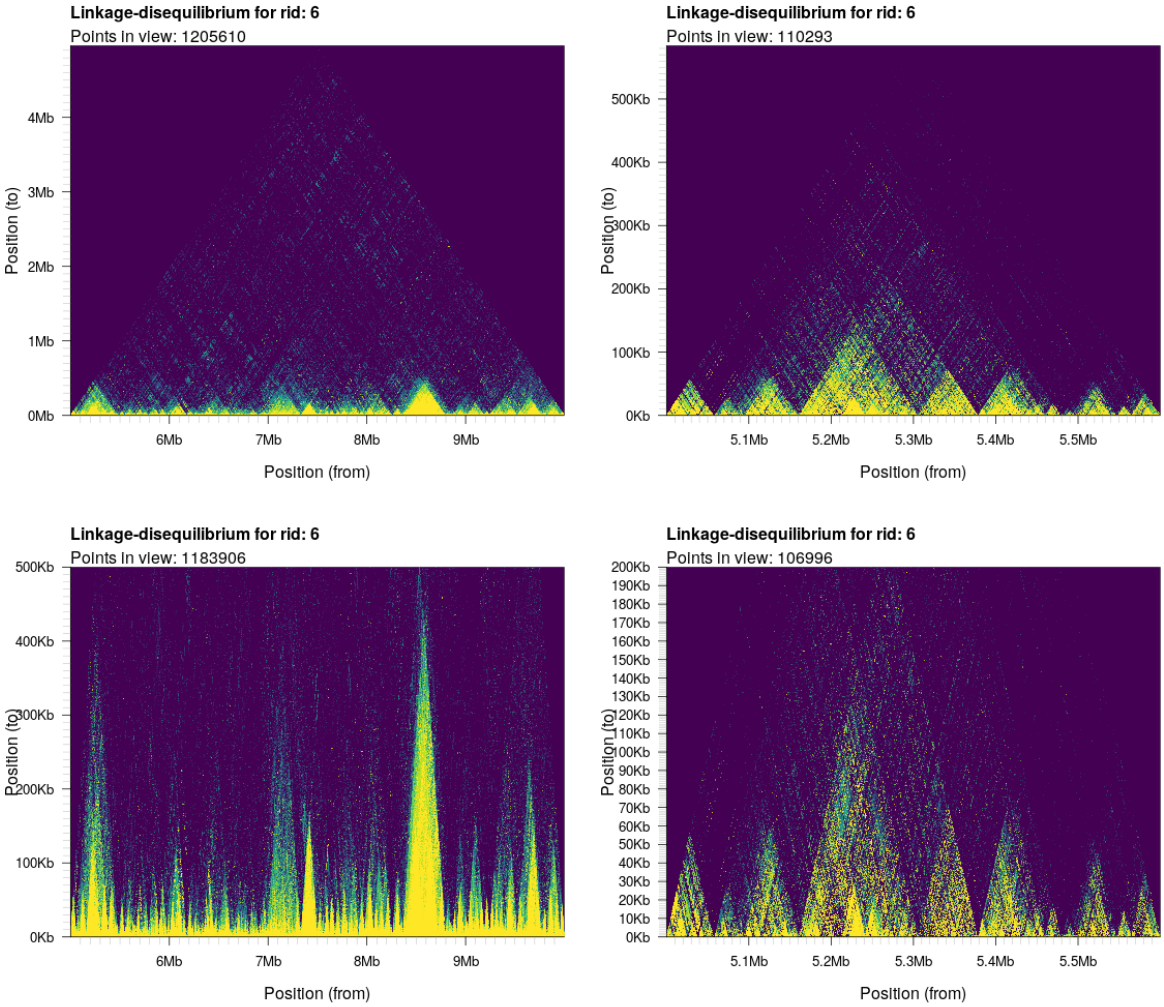


Figure 3.16 Example triangular LD plot using `plotLDTriangular` in rtomahawk. In many cases, there is generally no need to graphically represent the entire symmetric square (or rectangular) symmetric matrix of associations. This is especially true when combining multiple graphs together to create a more comprehensive picture of a particular region or feature. All of the examples here involves the subroutine `plotLDTriangular`. As a design choice, we decided to restrict the rendered y-axis data such that it is always bounded by the x-axis limits. For this reason, these plots will always be triangular will partially "missing" (omitted) values even if they are technically present in the dataset. Because of the smallish haplotype block size in humans, most of these visible triangular structures will have a limited span in the y-axis. We can truncate the y-axis to zoom into the local neighbourhood and more accurately display the local haplotype structure (bottom row).

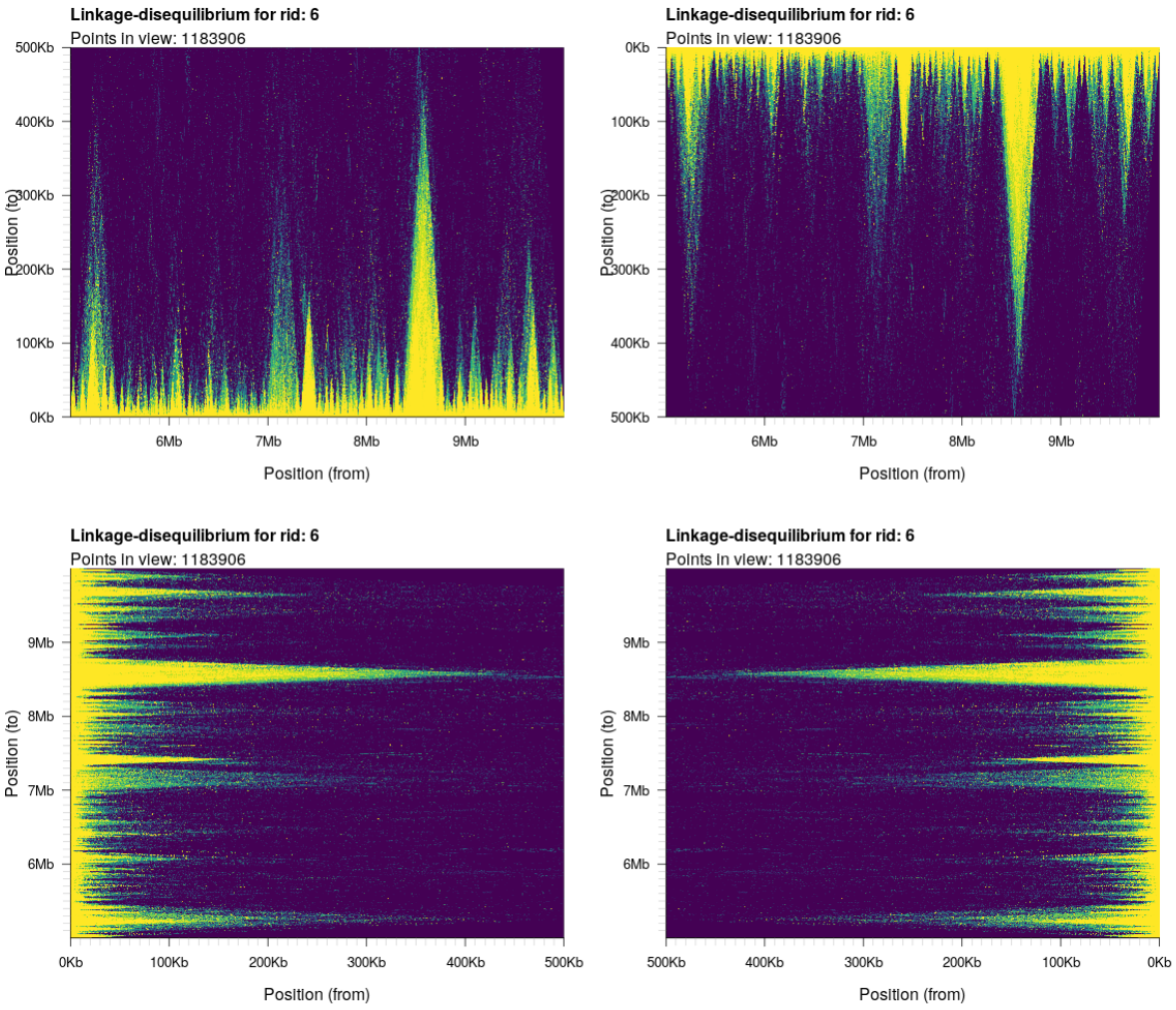


Figure 3.17 Flipping orientations of triangular plots. It is possible to control the orientation (rotation) of the output graph by specifying the orientation parameter. The numerical encodings are: 1) standard; 2) upside down; 3) left-right flipped; and 4) right-left flipped.

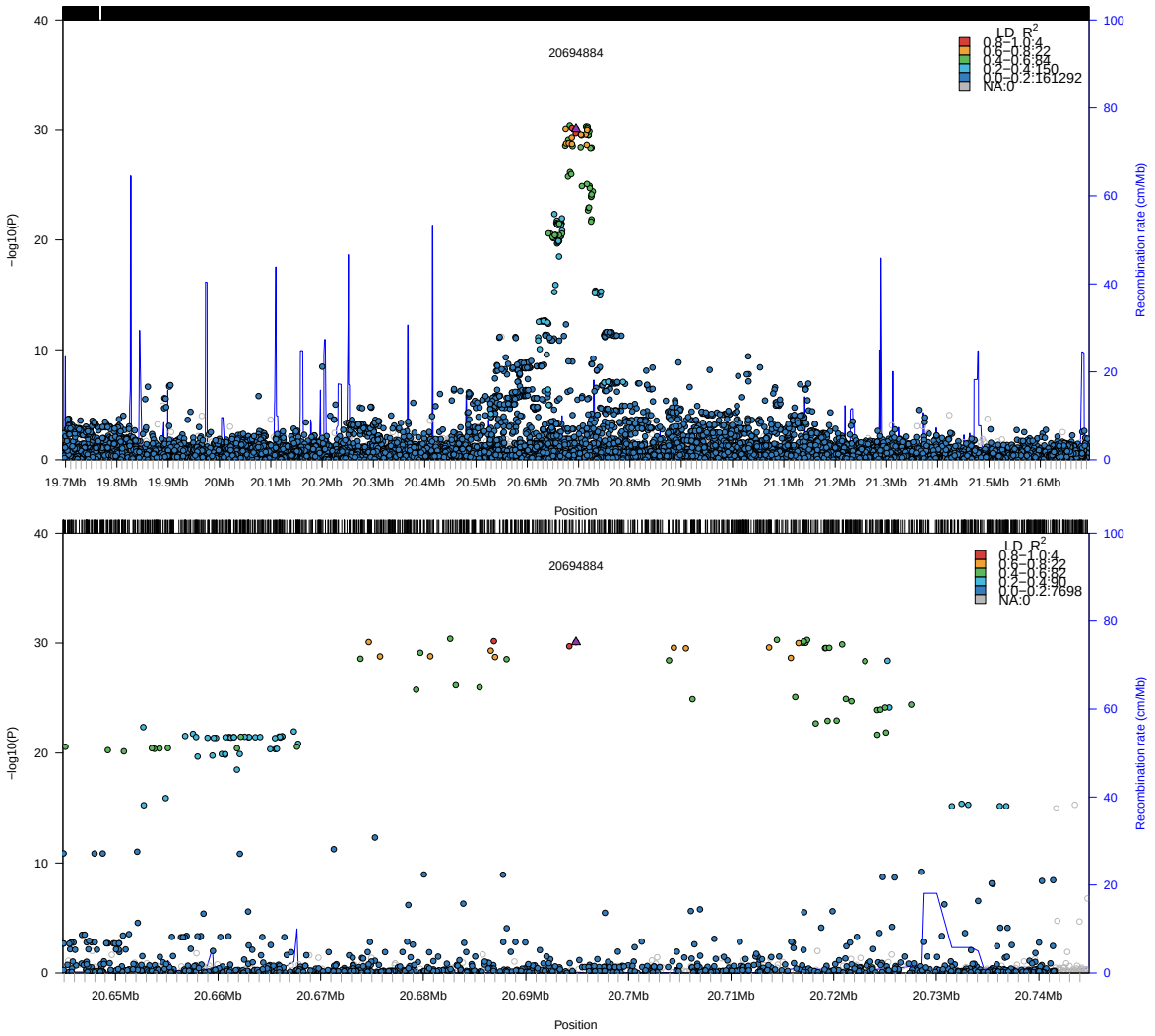


Figure 3.18 Computing LocusZoom-like plots for GWAS data. In the following examples we will investigate the association of genotypes at chromosome 6 and diabetes in the imputed UK BioBank cohort. The `plotLZ` function will internally compute linkage-disequilibrium for a target SNV and its surrounding genomic region. Given a target SNP of interest, purple triangle, we compute the LD with all neighbouring SNPs and fill points according to its R^2 value. The Y-axis is the log10-transformed P-value for an association with diabetes. The background computation of LD is stored in a temporary file and then loaded back into memory and returned as a new twk class instance. Shown is the SNV at `6:20694884` in a 1 Mb surrounding window (top) or 50 kb (bottom).

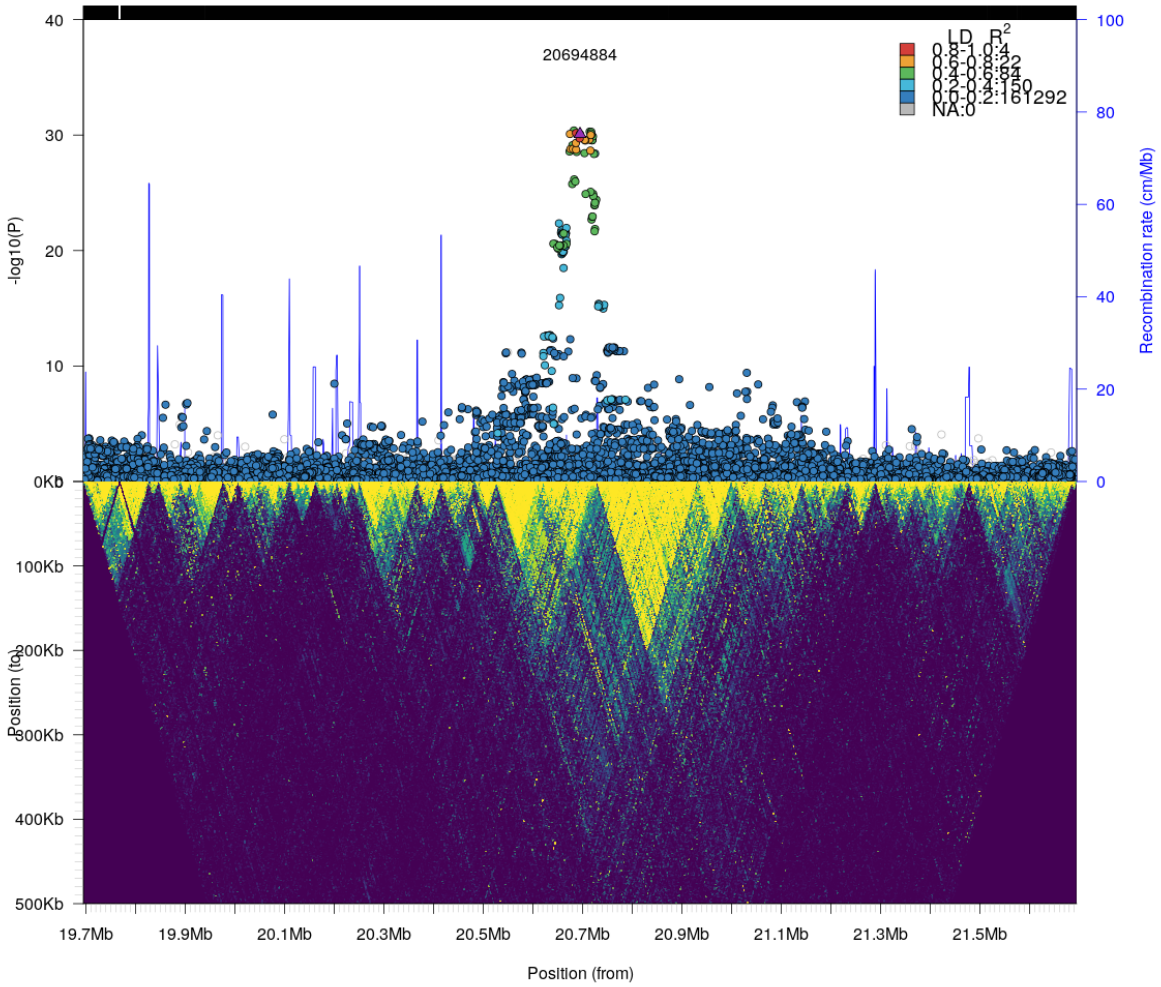


Figure 3.19 Mixing plot types in `rtomahawk`. If your visualizations require additional data layers it is possible to combine these into a single plot as `rtomahawk` renders plots using base-R. In this example we will combine two plots: the GWAS P-value and its single-site LD together with the all-vs-all pairwise LD for the same region in the upside-down orientation.

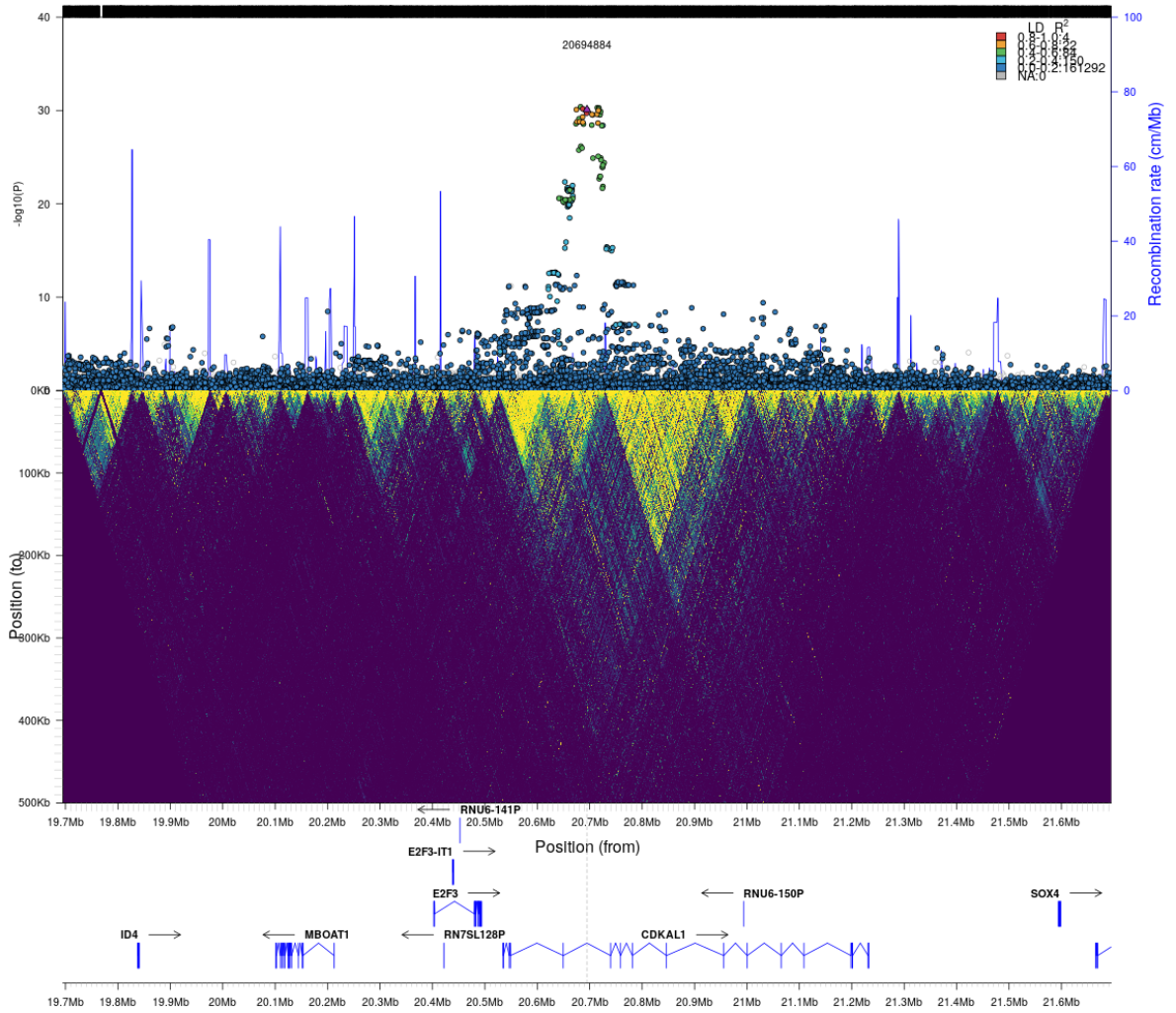


Figure 3.20 Mixing plot types in `rtomahawk`: a more advanced example. It is possible to add more advanced data layers using external packages. In this example we will add a gene track using genetic information extracted from biomaRt and drawn using Sushi, both third-party packages.

3.2.20 Aggregating and reducing data

In order to describe the scale that image generation for this volume of data would involve, take for example the small chromosome 20 with data from the 1000 Genomes Project Phase 3[123] that comprises of 1,733,484 diploid SNVs. Assuming we can plot LD data for a pair of SNVs in a single pixel, a monitor would have to be 400×400 meters wide to display this data if we assume a standard monitor with 1920×1080 pixel resolution and

20 inches of physical size. Equally cumbersome, the memory requirement for rendering this image would be ∼400 GB. Here we describe methods to overcome these restrictions.

Aggregation is the process of reducing larger datasets to smaller ones for the purposes of displaying more data than can fit on the screen while maintaining the primary features of the original dataset. Tomahawk performs aggregation into regular grids (2D partitions) and then applies summary statistics functions in these bins (**Table 3.11**).

Table 3.11 Currently available aggregation functions in Tomahawk. These aggregators are applied after partitioning a matrix $M^{m \times m}$ into non-overlapping bins of size b. This can be considered a k-nearest-neighbour summary statistics for m/b blocks.

Function	**Action**
Summation	Sum total of the desired property
Summation squared	Sum total of squares of the desired property
Mean	Mean of the desired property
Standard deviation	Standard deviation of the desired property
Minimum	Smallest value observed of the desired property
Maximum	Largest value oserved of the desired property
Count	Number of times a non-zero value is observed of the desired property

Here I give a worked example. Imagine we start out with a 4×4 matrix of observations and we want to plot 4 pixels (2×2):

Table 3.12 As an illustrative example we have the following assymetric square matrix with C rows and R rows.

	C1	**C2**	**C3**	**C4**
R1	1	2	3	4
R2	5	6	7	8
R3	9	10	11	12
R4	13	14	15	16

For example, aggregation by summation would result in the following matrix (**Table 3.13**):

Table 3.13 Aggregating by summation the matrix in Table 3.12 to a smaller 2×2 destination matrix. First we compute total sum for the four values in the top-left 2×2 sub-matrix in Table 3.12: $1 + 2 + 5 + 6 = 14$. Next for the 2×2 top-right matrix $3 + 4 + 7 + 8 = 22$. Finally, the last two sub-matrices are computed.

	C1-2	**C3-4**
R1-2	14	22
R3-4	46	54

Aggregation by means would yield the following matrix (**Table 3.14**):

Table 3.14 Aggregating by mean. In an identical approach as in Table 3.13 but with a different operator we compute the mean of the top-left 2×2 sub-matrix $(1+2+5+6)/4 = 3.5$. This procedure is repeated for the remaining sub-matrices.

	C1-2	**C3-4**
R1-2	3.5	5.5
R3-4	11.5	13.5

3.3 Results

We address the computational scaling requirements by representing genotypes using uncompressed bitmaps, which permits us to efficiently compare alleles by taking advantage of SIMD (single instruction, multiple data) instructions available on standard processors. These machine instructions operate on wide data registers and accelerate computation by processing multiple machine words simultaneously. To take advantage of these instructions, we perform bitwise transformations such that a two-locus haplotype is set to one (1) when true or zero (0) otherwise (see Methods). Set bits are simultaneously transformed and counted using a modified carry-save adder network[88] (also see Chapter 2). Using this approach and by leveraging multiple cores and SIMD-instructions we achieve a sustained and hardware-limited peak performance of 456 billion allelic comparisons per second ($\sim$0.2 CPU cycles/64-bit word) on a single core and >11.5 trillion comparisons per second on 28 cores (**Figure 3.21** and **Table 3.15**). Because this approach is independent of allele frequency (data density) it displays good performance across most sample sizes. We can further accelerate analysis for sparse vectors by supplementing the bitmaps with scalar lists storing the positions of set bits in the bitmaps (**Algorithm 2** and **Table 3.16,3.17**). This additional information reduces the search space by limiting the number of comparisons to the set positions in the vector with the lowest allele frequency by

performing scalar-bitmap comparisons. In order to maximize performance, we apply a heuristic-based decision tree that automatically selects the best method given the allele frequencies of the input pairs of genotype vectors.

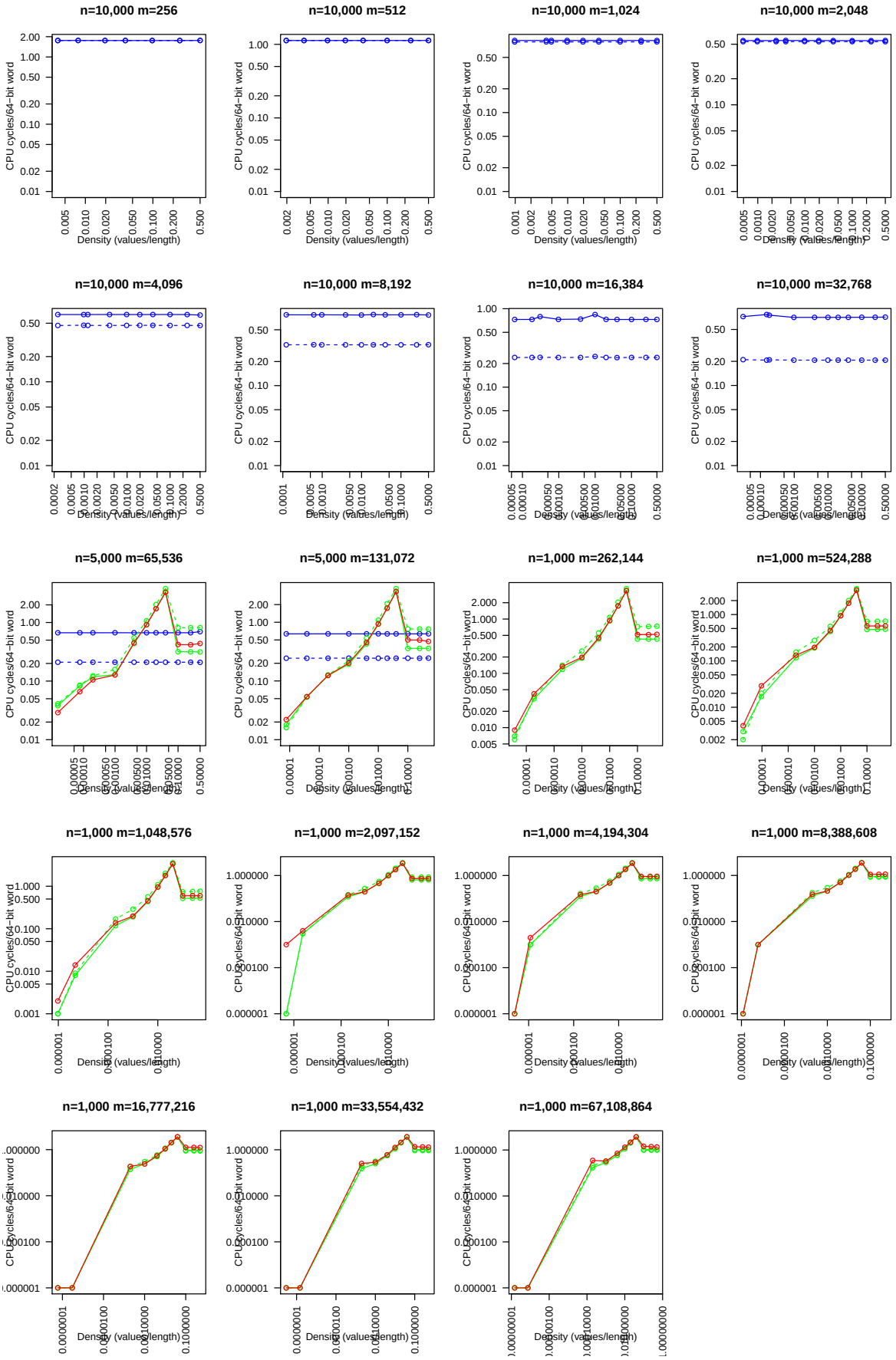

n=10,000 m=256
n=10,000 m=512
n=10,000 m=1,024
n=10,000 m=2,048
n=10,000 m=4,096
n=10,000 m=8,192
n=10,000 m=16,384
n=10,000 m=32,768
n=5,000 m=65,536
n=5,000 m=131,072
n=1,000 m=262,144
n=1,000 m=524,288
n=1,000 m=1,048,576
n=1,000 m=2,097,152
n=1,000 m=4,194,304
n=1,000 m=8,388,608
n=1,000 m=16,777,216
n=1,000 m=33,554,432
n=1,000 m=67,108,864
CPU cycles/64−bit word
Density (values/length)

Figure 3.21 Performance for different number of samples and allele frequencies. Data for n alleles and m sites was generated by drawing positions from the uniform distribution $\mathcal{U}(0, n)$ until the target alternative allele count ρ is reached. The y-axis corresponds to the number of CPU cycles / pair of 64-bit words equivalent in bitmap space in order to compare the throughput of both sparse and dense algorithms. Legend: blue solid, unblocked dense vector; blue dashed, blocked dense vector; red solid, unblocked sparse vector; green solid, blocked sparse vector.

Table 3.15 Performance for computing the cardinality of the set intersection between a pair vectors of bitmaps using the AVX512BW Instruction Set Architecture (ISA) with number of samples from 2^8 to 2^{26} with a fixed 10,000 rows (sites). After 262,144 samples (marked with *) we switch algorithm from uncompressed bitmaps to a hybrid-compressed approach that dramatically saves memory at the cost of performance.

Haplotypes	**CPU cycles / 64-bit word**	**MB/s**
256	1.754	13,870
512	1.127	21,565
1,024	0.785	30,900
2,048	0.535	45,290
4,096	0.47	51,610
8,192	0.324	74,882
16,384	0.238	101,556
32,768	**0.207**	**116,453**
65,536	0.215	111,968
131,072	0.248	95,535
262,144*	0.427	51,337
524,288	0.483	45,359
1,048,576	0.527	41,524
2,097,152	0.645	32,810
4,194,304	0.732	24,915
8,388,608	0.855	20,423
16,777,216	0.875	17,252
33,554,432	0.933	16,572
67,108,864	0.987	13,883

Table 3.16 Performance difference between using bitmap-bitmap (SIMD) or bitmap-scalar intersections as a function of allele count for a varying number of haplotypes from 2^{10} to 2^{17} over 10,000 rows (variant sites). Varying number of allele counts are set in random positions for each row (Alts column). This approach makes use of the largest ISA available at execution time as a function of the largest input array size. The chosen algorithm is shown in the Instruction set column. The cut-off between scalar-bitmap and bitmap-bitmap is set at $N/200$ where N is the number of haplotypes.

Haps.	Instruction set	Alts	SIMD (MB/s)	Scalar-bitmap (MB/s equivalent)	Speedup
1,024	SSE4.2	1	24,229	86,133	3.6
		10	24,231	29,610	1.2
		100	24,233	29,593	1.2
2,048	SSE4.2	1	34,970	172,071	4.9
		10	35,026	40,784	1.2
		100	35,064	44,157	1.3
4,096	AVX2	1	51,202	342,790	6.7
		10	51,085	82,772	1.6
		100	51,231	50,963	1
8,192	SSE512BW	1	76,483	678,720	8.9
		10	75,875	164,812	2.2
		100	74,482	73,528	1
16,384	SSE512BW	1	102,963	1,343,250	13
		10	103,198	328,654	3.2
		100	100,970	100,155	1
32,768	SSE512BW	1	117,859	2,550,386	21.6
		10	118,683	643,469	5.4
		100	117,047	117,681	1
65,536	SSE512BW	1	114,474	4,608,914	40.3
		10	115,290	1,270,198	11
		100	115,217	122,229	1.1
131,072	SSE512BW	1	100,047	8,137,207	81.3
		10	99,272	2,396,232	24.1
		100	98,745	289,108	2.9

Table 3.17 Time and memory savings for using scalar-bitmap intersections. This table demonstrates the extreme case where there are 10,000 variant sites with a single alt set for different number of haplotypes spaced in base-2 steps from 2^8 to 2^{26}. In the worst case for the hybrid bitmaps approach and an allele count of one, we will use 8kb of memory per site regardless of the universe size of haplotypes. Storm-contig use uncompressed bitmaps in aligned and contiguous memory whereas Storm use a hybrid approach to save memory. Memory in Storm is always aligned but not guaranteed to be contiguous. Note that memory requirements for Storm-contig are estimated for ≥262,144 haplotypes but measured for Storm.

Haps.	**Storm-contig.**		**Storm**		**Speedup**	**Mem. saving (fold)**
	Time (ms)	**Mem. (MB)**	**Time (ms)**	**Mem. (MB)**		
256	322	0.3	220	0.3	1.5	1
512	403	0.6	142	0.6	2.8	1
1,024	503	1.2	141	1.2	3.6	1
2,048	705	2.4	142	2.4	5	1
4,096	956	4.9	143	4.9	6.7	1
8,192	1,302	9.8	145	9.8	9	1
16,384	1,931	19.5	152	19.5	12.7	1
32,768	3,350	39.1	165	39.1	20.3	1
65,536	6,770	78.1	182	78.1	37.2	1
131,072	15,654	156.3	210	156.3	74.5	1
262,144	N/A	312.5	1,082	0.3	N/A	1,042
524,288	N/A	625	889	0.3	N/A	2,083
1,048,576	N/A	1,250	764	0.3	N/A	4,167
2,097,152	N/A	2,500	687	0.3	N/A	8,333
4,194,304	N/A	5,000	669	0.3	N/A	16,667
8,388,608	N/A	10,000	650	0.3	N/A	33,333
16,777,216	N/A	20,000	643	0.3	N/A	66,667
33,554,432	N/A	40,000	640	0.3	N/A	133,333
67,108,864	N/A	80,000	635	0.3	N/A	266,667

Lastly, we can sustain high performance in the unphased case by leveraging an exact analytic solution[91, 40] in place of the more standard iterative procedure[112] (see Methods).

To analyse computational performance, we benchmarked Tomahawk against existing tools[95, 96]. First, we compared run times and memory usage by simulating a fixed 1 Mb region with increasing number of samples from 256 to 16 million (**Figure 3.22**, and **Table 3.19**). In this situation, Tomahawk is substantially faster compared to PLINK and emeraLD at all tested numbers of samples (**Table 3.18**). Notably, our highly optimized algorithms display >13-fold faster single-thread performance compared to PLINK while operating on the same data structure when using the `AVX512` ISA (data not shown).

Table 3.18 Performance for simulated 1 Mb with samples from 2^8 to 2^{24}. Because of the very small region simulated, a unrealistically proportion of sites will have large allele frequency (common variants) resulting in Tomahawk using dense matrix computations. Despite of this, Tomahawk is generally an order of magnitude faster then PLINK.

Haplotypes	Variants	Time		Speedup
		Tomahawk	PLINK	
512	5623	00.196s	0.30s	1.5
1,024	5859	00.251s	0.36s	1.4
2,048	6596	00.384s	0.63s	1.6
4,096	7250	00.544s	1.0s	1.8
8,192	7812	00.827s	1.93s	2.3
16,384	8210	00.597s	4.07s	6.8
32,768	8895	00.549s	9.32s	17.0
65,536	9272	00.598s	20.29s	33.9
131,072	9909	01.261s	45.62s	36.2
262,144	10703	05.582s	1m46.98s	19.2
524,288	10743	12.974s	3m36.30s	16.7
1,048,576	11789	32.830s	8m41.71s	15.9
2,097,152	11778	59.055s	15m59.82s	16.3
4,194,304	12495	02m05.673s	33m49.37s	19.2
8,388,608	12920	04m02.044s	01h35m51s	23.8
16,777,216	13708	09m26.16s	04h16m22s	27.2

Table 3.19 Statistics for simulated 1 Mb regions with $(2^8, 2^9, \ldots, 2^{20})$ haplotypes. File sizes for the BCF, BED, and VCF.gz formats are included for reference. The `bcftools` plugin `+fill-tags` was used to compute allele counts and failed to run (memory segmentation fault) after 524,288 samples. AC: allele count.

Haplotypes	Sites	BCF	BED	VCF.gz	AC = 1	AC <5	AC <10	AC <100
256	4,884	0.1	0.1	0.1	718	1,638	2,260	4,158
512	5,623	0.2	0.3	0.3	848	1,656	2,253	4,172
1,024	5,859	0.4	0.7	0.5	843	1,706	2,319	4,014
2,048	6,596	0.8	1.6	1	789	1,601	2,255	4,163
4,096	7,250	1.8	3.5	2.1	784	1,674	2,307	4,254
8,192	7,770	3.4	7.6	4	801	1,652	2,322	4,230
16,384	8,210	7	16	8.3	810	1,654	2,225	4,029

32,768	8,895	13.6	34.7	15.7	802	1,657	2,332	4,163
65,536	9,272	25.5	72.4	29.6	834	1,735	2,322	4,153
131,072	9,909	48.8	154.8	57.2	817	1,714	2,331	4,228
262,144	10,703	106.4	334.5	125.6	816	1,694	2,260	4,133
524,288	10,743	180	671.4	213.9	793	1,657	2,235	4,279
1,048,576	11,789	418.6	1473.6	497.4	ERROR	ERROR	ERROR	ERROR

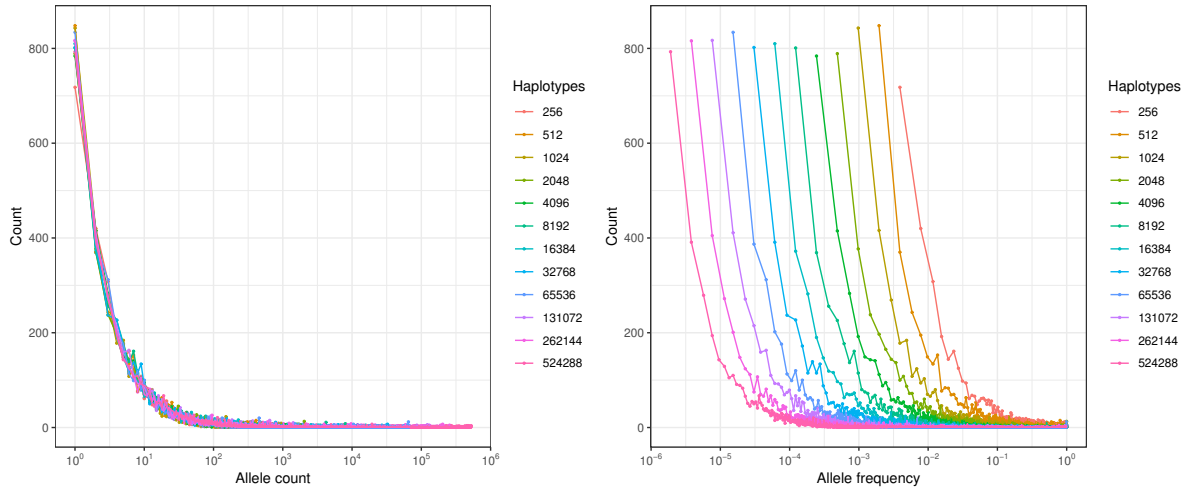


Figure 3.22 Allele count (left) and allele frequency (right) distribution plots for simulated data in a 1 Mb range for $(2^8, 2^9, \ldots, 2^{20})$ haplotypes. No data is available for 2^{20} as allele counts could not be extracted using `bcftools`.

Because uncompressed bitmaps are less useful for large numbers of individuals because of their memory requirements we implemented a simplified variation of well-established hybrid bitmaps [67, 66, 16] that both reduce the memory footprint and improve performance for large datasets (see Methods). This approach partitions the data interval into non-overlapping chunks and stores information only in chunks with data available as either bitmaps or scalars. Using this approach, Tomahawk can achieve up to >100,000-fold speedups compared to PLINK depending on allele frequency (**Table. 3.20**).

Table 3.20 Time and memory usage for computing all-vs-all LD for PLINK and Tomahawk on large datasets with low allele count. We simulated number of haplotypes from 1 million to 16 million for 10,000 variant sites. Each site have an allele count of 1. No larger cohorts were tested as PLINK require too much memory to run. Memory listed in MB and time in seconds.

Haplotypes	PLINK		Tomahawk		Speedup	Memory saving
	Time	Memory	Time	Memory		
1,048,576	2167.4	5438.5	0.764	0.3	2,837	18,128
2,097,152	6428.9	7805	0.687	0.3	9,358	26,017
4,194,304	12672.7	7926	0.669	0.3	18,943	26,420
8,388,608	30356.1	7954.2	0.650	0.3	46,702	26,514
16,777,216	66783.4	7954.2	0.643	0.3	103,862	26,514

Lastly, we investigated performance on real-world data by computing LD for the 1000GP dataset (n = 2,504, **Table.** 3.21). All tested methods grow linearly with increasing number of pairs of sites. Using chromosome 22 from the 1000GP cohort as an example, Tomahawk is >61-fold faster (t = 6.8 min) compared to PLINK (t = 248.5 min). Similarly for the HRC dataset, Tomahawk is 18.37-fold faster (t = 6.04 min) compared to PLINK (t = 111.02 min) and 133.84-fold faster compared to emeraLD (t = 808.89 min). On the basis of these simulated datasets, we expect that Tomahawk will continue to improve over other methods as sample sizes increase.

Table 3.21 Run-time for 2,504 samples in the 1000 Genomes Phase 3 dataset for Tomahawk. The host architecture used is a 14-core 22 nm Haswell Xeon E5-2697 v3 with the AVX2 ISA and hyper-threading enabled (but unused). Times are reported as elapsed wall time using 14 cores.

Chromosome	Variants	Time
1	6,419,532	03h46m05s
2	7,026,684	04h27m05s
3	5,787,493	03h03m30s
4	5,685,496	03h01m35s
5	5,224,039	02h29m30s
6	4,983,194	02h21m05s
7	4,680,137	02h02m02s
8	4,560,663	01h53m22s
9	3,533,035	01h08m36s
10	3,961,685	01h27m38s

11	4,014,414	01h28m10s
12	3,837,554	01h21m23s
13	2,835,142	44m57s
14	2,634,138	38m15s
15	2,405,829	31m45s
16	2,675,201	38m17s
17	2,311,055	28m55s
18	2,249,409	28m09s
19	1,817,889	18m36s
20	1,798,559	17m41s
21	1,096,282	06m50s
22	1,094,014	06m47s

Next, we computed chromosome-wide LD for the 1000GP pan-population panel[123] for a total of 181.6×10^{12} pairwise variant comparisons (**Figure 3.23**). This analysis finished in <33 hours on a single 14-core machine and resulted in >47.4 billion output associations with an R^2-correlation over 0.1 involving variants with total non-reference two-locus haplotype count >5 compressed into 663 gigabytes (~7.3 terabytes of uncompressed binary data, See Methods). We similarly computed chromosome-wide LD for the Haplotype Reference Consortium[125] (HRC, n = 32,470) panel and **Table. 3.22**). This analysis finished in <98 hours. I limited additional analyses to the 1000GP dataset.

Table 3.22 Run-time for 32,470 samples in the HRC dataset. The host architecture used is a 14-core 22 nm Haswell Xeon E5-2697 v3 with the AVX2 ISA and hyper-threading enabled (but unused). Times are reported as elapsed wall time using 14 cores.

Chromosome	**Variants**	**Time**
1	3,069,931	10h47m35s
2	3,990,910	13h44m4s
3	2,821,894	9h06m55s
4	2,787,581	9h04m31s
5	2,588,168	7h35m56s
6	2,460,111	7h02m44s
7	2,289,305	6h01m42s
8	2,242,705	5h44m54s
9	1,686,471	3h19m34s

10	1,937,230	4h23m25s
11	1,936,990	4h25m11s
12	1,848,117	4h03m28s
13	1,385,433	2h19m04s
14	1,270,436	1h56m37s
15	1,139,215	1h33m34s
16	1,281,297	1h56m08s
17	1,090,072	1h24m05s
18	1,104,755	1h28m35s
19	868,554	54m57s
20	884,983	56m40s
21	531,276	21m38s
22	524,544	21m11s

In order to handle this volume of data, I developed a comprehensive framework that enables efficient storage, queries and visualizations called `rtomahawk`. Exploring data using these visualizations, we observe large LD blocks spanning the centromere that frequently encompass several megabases (**Figure 3.24** and **Figure 1-21**), corresponding to the recently described cenhaps[61]. Chromosomes 2-8, 10-12, and 16-20 display a strong pan-centromeric pattern while chromosomes 1,9,13,14,15,21 show little to no signal. These centromere-spanning LD blocks are present in all super-populations and populations, consistent with the centromeric regions being recombination-quiescent[9, 120] and that the observed haplotype structure preceded the separation of continental populations.

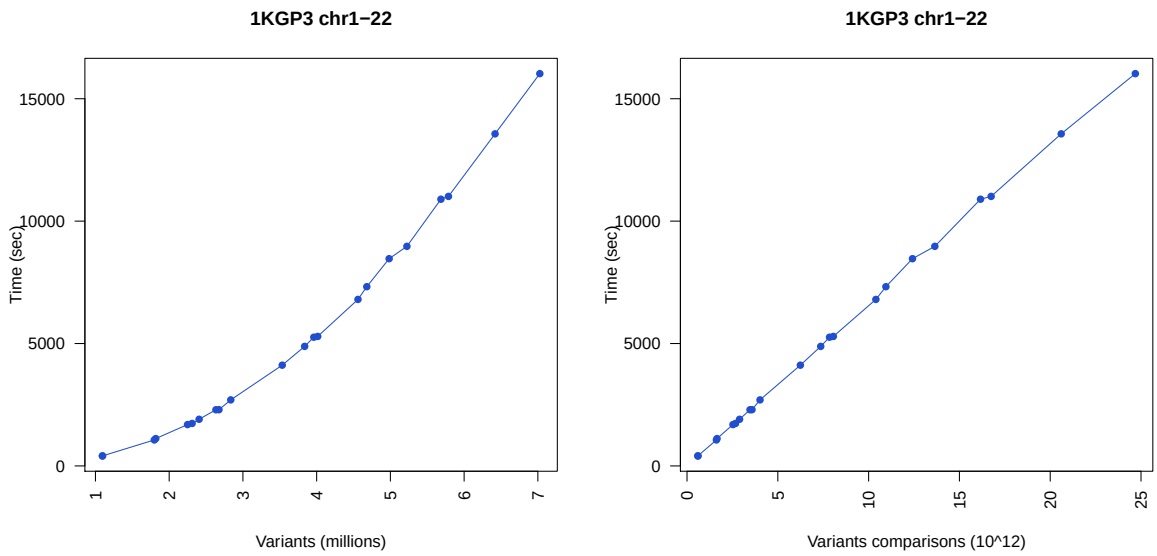


Figure 3.23 Completion times for computing chromosome-wide LD for 1KGP3 data. Left panel: Completion times grows in $O(2N(V-1)(V)/2)$-time where $2N$ is 5,008 haplotypes and V is the number of bi-allelic variants ranging from 1,094,014 for chr22 to 7,026,684 for chr2. Computation for chr2 finished in 4h27min on 14 cores and chr22 finished in 6m47s. Right panel: X-axis is $(V-1)(V)/2$-transformed to demonstrate a perfect linear relationship in this space. The host architecture used is a 14-core 22 nm Haswell Xeon E5-2697 v3 with the AVX2 ISA and hyper-threading enabled (but unused).

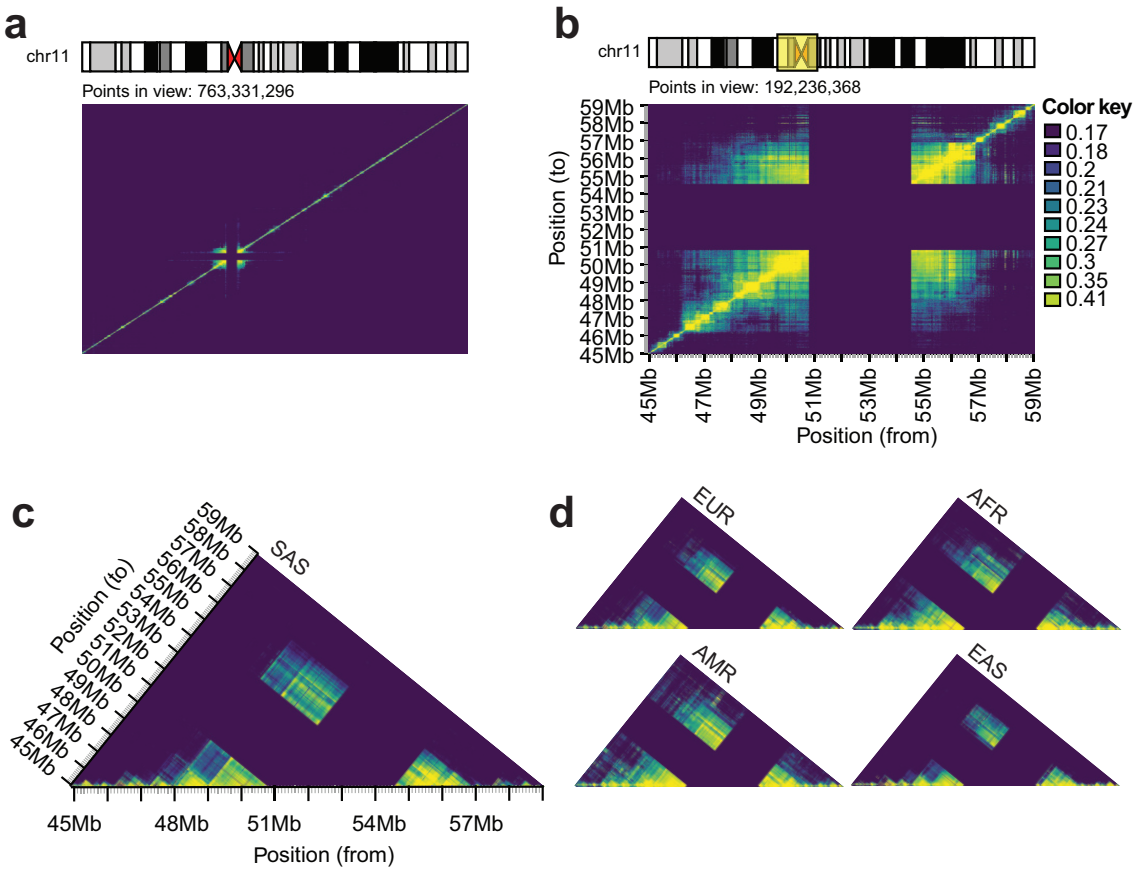


Figure 3.24 Example pan-centromeric linkage-disequilibrium block. **a)** Chromosome-wide linkage-disequilibrium (LD) for 1000 Genomes Phase 3 (1KGP3) chromosome 11 demonstrating a long-range, pan-centromeric LD block. **b)** Zoomed in region of **a)** reveal a pan-centromeric LD block from around 46 Mb to 57 Mb. The missing information in the middle correspond to the repetitive centromeric region that is masked out during the mapping procedure. **c)** Upper triangular LD block for the South Asian (SAS) super-population of the 1KGP3 dataset with genomic coordinates shown. **d)** Upper triangular blocks for admixed American (AMR), East Asian (EAS), European (EUR), and African (AFR) super-populations demonstrating that these long-range pan-centromeric LD blocks are present in all of them.

3.4 Discussion

Together, these algorithms ensure completeness and correctness while reducing computational cost and time. I applied them to the general problem of computing genome-wide linkage-disequilibrium statistics for both real and simulated population-scaled datasets ranging from 1,000 samples to 67 million samples. Given the efficiency and flexibility of

these algorithms owing to their generalizability, I expect that they will enhance both existing and novel methods exploring population-scaled datasets.

3.5 Additional approaches

In the following subsections I describe functionality and algorithms that used to be part of Tomahawk but were eventually removed when better performing approaches were discovered. Although not optimal in this application space, these subroutines and algorithms have interesting properties making them worthwhile to mention.

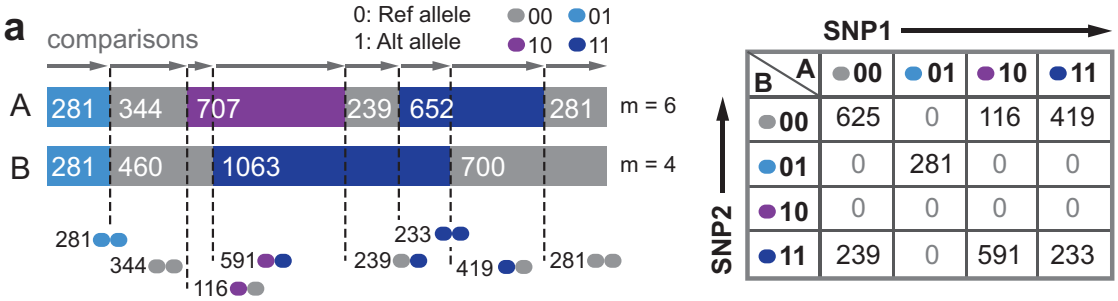
3.5.1 Representing genotypes using run-length encoding

As the haplotypic diversity is low in humans[123], most bitmaps will be predominantly empty (mostly zeros). For example, >80% of variants in the 1000 genomes project (1000GP)[123] have an allele count <5. As cohort size increases, rare alleles are beginning to dominate and heavily polarize the variant dataset to extreme sparsity: Topmed[2] (n = 62,784)[121] and gnomAD[3] (n = 125,748 exomes and n = 15,708 genomes)[55].

Run-length encoding is a well-established encoding algorithm that represents repetitive symbols as a (symbol, copies)-tuple. For example, the sequence 0000000000 is stored as (0,10). As mentioned, most variable sites have low allele frequency and will compress into few RLE objects (data not shown). The following algorithms exploit this property of sparsity by first run-length encoding (RLE) variants (Algorithm 5) at a locus and then directly compare pair of RLE objects from two different loci (**Figure 3.25**, Algorithm 6 and 7). Although these algorithms are no longer part of tomahawk, a variation of them is used in Chapter 5.

[2]Freeze5 (GRCh38), MD5 checksum `773e9e97759a4a5b4555c5d7e1e14313`. Last accessed: 22 July, 2019

[3]version 2.1.1, WGS file: `gnomad.genomes.r2.1.1.sites.vcf.bgz`, MD5 checksum `e6eadf5ac7b2821b40f350da6e1279a2`. WES file: `gnomad.exomes.r2.1.1.sites.vcf.bgz`, MD5 checksum `f034173bf6e57fbb5e8ce680e95134f2`. Last accessed: 22 July, 2019


a
comparisons
0: Ref allele
1: Alt allele
00 01
10 11
A 281 344 707 239 652 281 m = 6
B 281 460 1063 700 m = 4
281
344
116
591
239
233
419
281
SNP1
SNP2
B \ A | 00 | 01 | 10 | 11
00 | 625 | 0 | 116 | 419
01 | 0 | 281 | 0 | 0
10 | 0 | 0 | 0 | 0
11 | 239 | 0 | 591 | 233

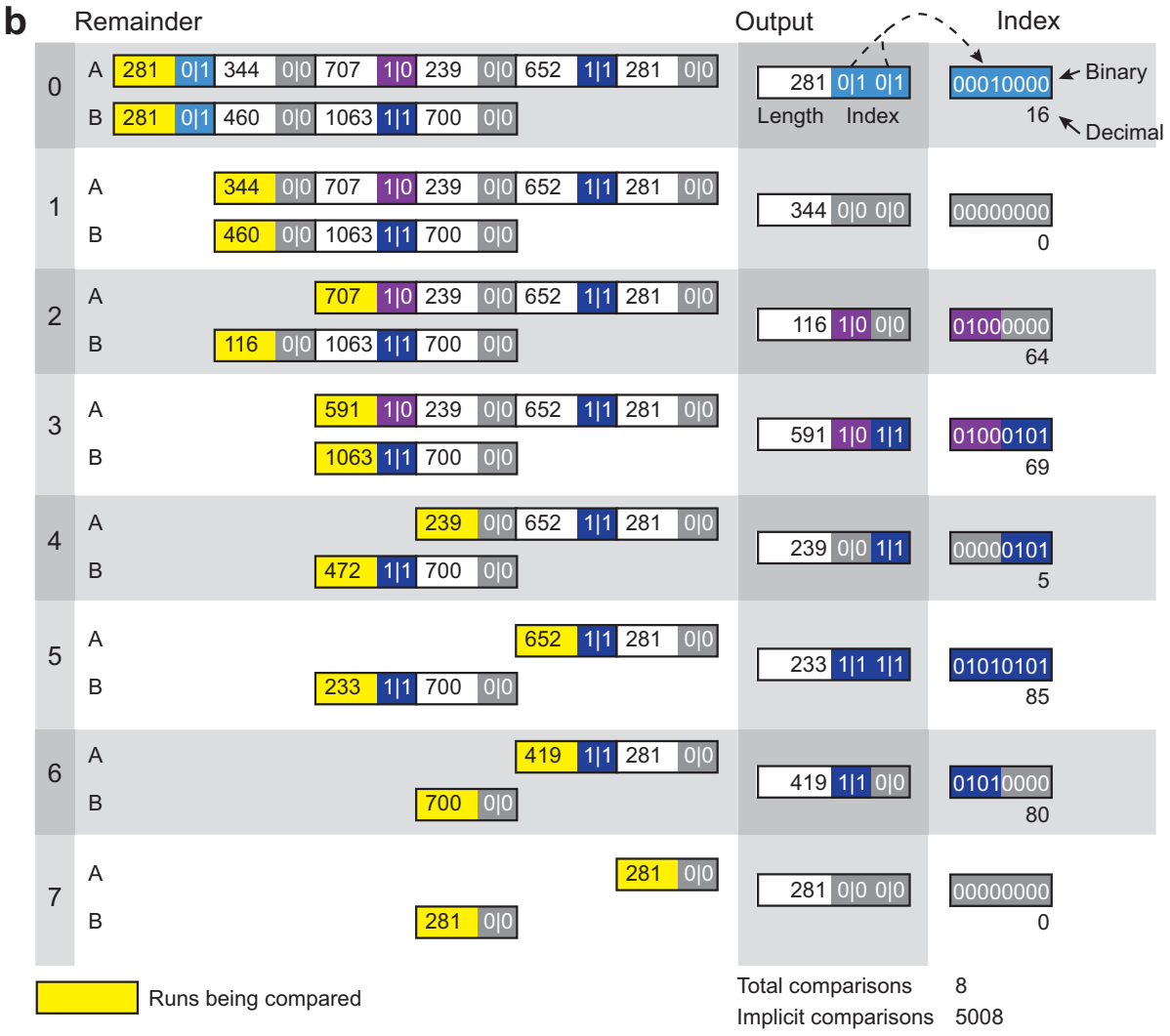

b
Remainder
Output
Index
Length Index
Binary
Decimal
0 A 281 0|1 344 0|0 707 1|0 239 0|0 652 1|1 281 0|0
B 281 0|1 460 0|0 1063 1|1 700 0|0
281 0|1 0|1
00010000
16
1 A 344 0|0 707 1|0 239 0|0 652 1|1 281 0|0
B 460 0|0 1063 1|1 700 0|0
344 0|0 0|0
00000000
0
2 A 707 1|0 239 0|0 652 1|1 281 0|0
B 116 0|0 1063 1|1 700 0|0
116 1|0 0|0
01000000
64
3 A 591 1|0 239 0|0 652 1|1 281 0|0
B 1063 1|1 700 0|0
591 1|0 1|1
01000101
69
4 A 239 0|0 652 1|1 281 0|0
B 472 1|1 700 0|0
239 0|0 1|1
00000101
5
5 A 652 1|1 281 0|0
B 233 1|1 700 0|0
233 1|1 1|1
01010101
85
6 A 419 1|1 281 0|0
B 700 0|0
419 1|1 0|0
01010000
80
7 A 281 0|0
B 281 0|0
281 0|0 0|0
00000000
0
Runs being compared
Total comparisons 8
Implicit comparisons 5008

Figure 3.25 Algorithm for computing set intersections using run-length encoding of genotypes. a) Run-length encoding (RLE) genotypes as (template, length)-tuples enables direct comparison of sparse vectors in theoretically faster time compared to naive comparisons. Given two vectors of RLE elements, A and B, we compare each element pairwise and report the joint genotype/haplotype of either (1) the smaller or (2) both and advance the pointer in either or both, respectively, resulting in the output matrix shown. b) Worked example of the vectors in a).

Algorithm 5 Run-length encode biallelic diploid variants

function RLEGENOTYPES(G, n) ▷ Where G - vector of biallelic genotypes, n - number of samples
  Let $L = 1$ be the current run length
  Let E = (allele A, allele B, phase) $\mapsto$ $(\{0,1,2\},\{0,1,2\},\{0,1\})$ be a 3-tuple genotype
  RLE = (E, run length)
  $R = \{\}$ be an empty vector of RLE
  $E_{\text{ref}} = (G_0^A, G_0^B, G_0^P)$
  for $i = 1$ to n **do**
    $E_{\text{comp}} = (G_i^A, G_i^B, G_i^P)$
    if $E_{\text{comp}} \neq E_{\text{ref}}$ **then**
      Push RLE(E_{ref}, L) into **R**
      $E_{\text{ref}} = E_{\text{comp}}$
      $L = 0$
    end if
    L += 1
  end for
  return R
end function

By exploiting the run-length representation of genotypes instead of their explicit values we can reduce the number of operations required to compute the inner product of two vectors in the average case. We do this by noting that the number of comparisons required to compare two random multi-sets R_A and R_B of RLE objects can be computed in worst-time proportional to the total set sizes of the two sets: $O(|R_A| + |R_B| + 1)$. Algorithm 6 and 7 are functionally identical but operate on two different template sizes: two bits for phased data and four bits for unphased data.

Algorithm 6 Constructing implicit 3-by-3 contingency table for unphased genotypes directly from run-length compressed data

function BUILDUNPHASEDTABLEALG1(R_A, R_B)▷ R - run-length encoded vector A and B
  Let $O[0 \dots 256)$ be an empty array
  Let $R^E[i]$ be the 4-bit encoded genotype at position i
  Let $R^L[i]$ be the run-length at i
  Let **pack**$(R_a^E[i], R_b^E[j])$ be the joint 8-bit genotype combination
  Let $i = 0, j = 0$ such that $i \in [0 \dots |R_A|)$, $j \in [0 \dots |R_B|)$
  Let $c_A = R_A^L[0]$ and $c_B = R_B^L[0]$
  while $TRUE$ **do**
    $X =$ **pack**$(R_A^E[i], R_B^E[j])$
    if $c_A < c_B$ **then** ▷ Current length of A $<$ current length of B
      $c_A \mathrel{-}= c_B$
      $O[X] \mathrel{+}= c_B$
      $j \mathrel{+}= 1$
      $c_B = R_B^L[j]$
    else if $c_A > c_B$ **then** ▷ Current length of A $>$ current length of B
      $c_B \mathrel{-}= c_A$
      $O[X] \mathrel{+}= c_A$
      $i \mathrel{+}= 1$
      $c_A = R_A^L[i]$
    else ▷ Current length of A is equal to the current length of B
      $O[X] \mathrel{+}= c_A$
      $i \mathrel{+}= 1$, $j \mathrel{+}= 1$
      $c_A = R_A^L[i]$, $c_B = R_B^L[j]$
    end if
    if $i == |R_A|$ or $j == |R_B|$ **then**
      break
    end if
  end while
  return CalculateLDUnphased(O)
end function

Algorithm 7 Constructing implicit 2-by-2 contingency table for phased genotypes directly from run-length compressed data

function BUILDPHASEDTABLEALG1(R_A, R_B) ▷ R - run-length encoded vector A and B
  Let $O[0 \dots 16)$ be an empty array
  Let $R^E[i]$ be the 2-bit encoded genotype at some array position i
  Let $R^L[i]$ be the run-length at i
  Let **pack**$(R_a^E[i], R_b^E[j])$ be the joint 4-bit genotype combination
  Let $i = 0, j = 0$ such that $i \in [0 \dots |R_A|)$, $j \in [0 \dots |R_B|)$
  Let $c_A = R_A^L[0]$ and $c_B = R_B^L[0]$
  while $TRUE$ **do**
    $X =$ **pack**$(R_A^E[i], R_B^E[j])$
    if $c_A < c_B$ **then**
      $c_A \mathrel{-}= c_B$
      $O[X] \mathrel{+}= c_B$
      $j \mathrel{+}= 1$
      $c_B = R_B^L[j]$
    else if $c_A > c_B$ **then**
      $c_B \mathrel{-}= c_A$
      $O[X] \mathrel{+}= c_A$
      $i \mathrel{+}= 1$
      $c_A = R_A^L[i]$
    else
      $O[X] \mathrel{+}= c_A$
      $i \mathrel{+}= 1$, $j \mathrel{+}= 1$
      $c_A = R_A^L[i]$, $c_B = R_B^L[j]$
    end if
    if $i == |R_A|$ or $j == |R_B|$ **then**
      break
    end if
  end while
  return CalculateLDPhased(O)
end function

Both of these algorithms can be rewritten into branchless form to avoid costly prediction errors.

3.6 Acknowledgements

I am grateful to C. Wallace, J. Todd, P. Daněček, J. Bonfield, J. Barrett, E. Garrison, E. Dawson for helpful discussions and suggestions. C.W. introduced M.D.R.K. to non-iterative approaches to solving the phase-uncertainty problem. The basis for this work was partly conceived when M.D.R.K. was a member of J.T. group. Discussion with J.Bonfield resulted in the development of self-contained indices. Discussion with J.Barrett resulted in additional functionality. P.D. provided assistance with htslib. I am grateful to members of the open-source community for helping to improve the implementation.

Chapter 4

Analysing population-scaled sequence variant data

In this chapter I will introduce several algorithms for efficiently storing and querying large-scale genetic datasets that can interchange to `htslib`-compliant file formats in a bit-exact fashion. I will demonstrate how it is possible to reduce the file size while simultaneously accelerating query speeds by using a column-centric storage model (column store) compared to the row-centric model used in most incumbent genomics file formats. I will introduce a novel supportive data structure that allows for summary statistics queries to be performed directly from the compressed representation. Collectively, these methods and algorithms are packaged into the software project `tachyon`. Software is available online at: `https://github.com/mklarqvist/tachyon`.

4.1 Introduction

Over the last decade, large-scale international and national collaborative efforts have created increasingly larger genetic variation datasets. Recent population-scaled national initiatives such as the UK Biobank[14], the UK 100,000 Genomes Project[127], and the US All of Us Research Program[52], will encompass up to a million people. These datasets will contain trillions of data points for genotypes and, if stored using the incumbent Vcf interchange format[26], would require prohibitive amounts of storage and would be inexpedient to query.

The concept of compressive genomics [78] involves leveraging inherent redundancies in data to achieve sublinear storage and analyses and has been widely successful in the storage of sequencing data [17, 12, 142] and genetic variant data [34, 71, 30]. Large genetic datasets of humans have a highly redundant structure primarily because of low

haplotypic diversity [123]. Notably, data from between 250 and 1,000 individuals are required to double the genetic information content compared to a single genome as humans differ by around 0.1-0.4% [123]. As additional samples of similar ancestry are added, the majority of newly observed variants will be either of very low frequency or be *de novo* mutations [121, 55, 123]. This limited ancestral diversity and highly structured data can be exploited to improve compression and accelerate analyses by employing encoding schemes that do not require decompression. Genetic archives compressed in this fashion can be queried in time proportional to its compressed size resulting in considerable performance gains.

4.2 Methods

4.2.1 Column-centric representation

Traditional databases store information in a per-record (row-centric) orientation such that three records $r_1 = (A_1, B_1, C_1)$, $r_2 = (A_2, B_2, C_2)$, and $r_3 = (A_3, B_3, C_3)$ are stored end-to-end in a table (r_1, r_2, r_3). In contrast, column-centric (column store) systems vertically partition records into collections of individual columns for each data field. Using the same example data, three columns c_A, c_B, and c_C store all data for the three data fields A, B and C such that $c_A = (A_1, A_2, A_3)$, $c_B = (B_1, B_2, B_3)$, and $c_C = (C_1, C_2, C_3)$. These columns can then be compressed and stored separately on disk.

Database performance is directly related to the efficiency of moving target data into CPU registers for processing. By storing each column separately on disk, column stores can perform queries involving only the target field of interest without first loading entire records into memory and then discarding unwanted information. Individually stored columns contain data of a shared primitive type resulting in improved compression and faster loading time from disk[1, 150]. Operating directly on compressed data, when possible, heavily improves utilization of memory bandwidth by reducing the amount of data in CPU registers[2].

Maintaining the relationship (map) between fields and record requires additional effort in a column store for genetic variant data. Data in the Vcf interchange format require storage in a hierarchical (nested) model where only the root level has a defined schema. The root level has the defined schema chromosome, position, name, reference allele, alternative alleles, quality, flags, and additional per-site information (`INFO`, **Figure 4.1a**). In contrast, the `INFO`-field is a tensor of fields with the only requirement that its primitive type is predefined in the archive header. This hybrid schemaless representation of fields

below the root level requires a function that maps a list of target field identifiers back to a given configuration (**Figure 4.1d**). Using a simple hash function $H(x), f : \mathbb{Z} \to \mathbb{Z}$ that deterministically maps any input bit-string into a fixed-width integer, I first hash the global identifiers (dictionary-compressed identifiers mapping to a field) of a field into local identifiers and thereby guarantee they map to the range $[0, \cdot)\mathbb{Z}$. This list of local identifiers corresponds to a pattern, representing the local schema for a given record, that can in turn be hashed to create a pattern identifier. In order to reduce space and improve computational efficiency, the list of identifiers in a pattern is stored in a bitmap. Storing the pattern identifiers together with the list of local identifiers it contains is sufficient to index the observed schemas in a nested field. Collectively, this approach enables Tachyon to interchange between `htslib`-compliant file formats and the `yon` file format in a bit-exact fashion.

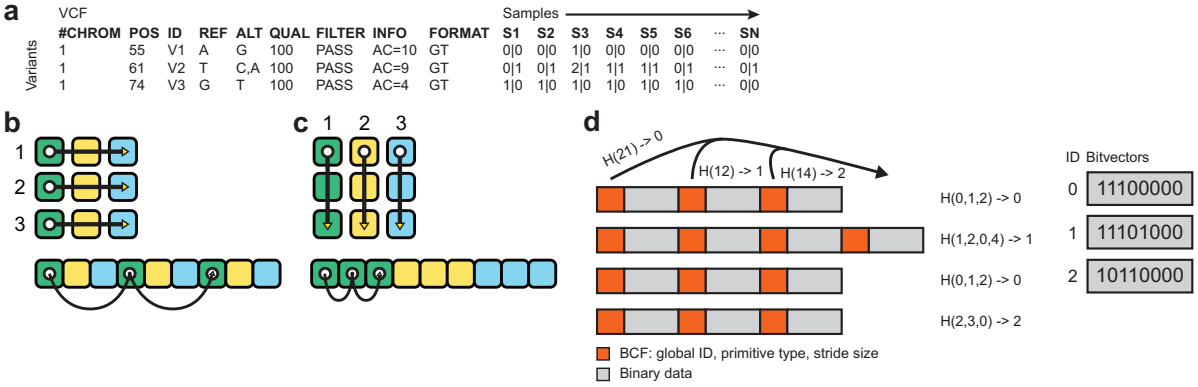


Figure 4.1 Overview of the storage model. a) Genetic datasets stored in the widely used Vcf interchange format are stored as per-site records comprising of a variety of site-descriptive fields, such as position and reference allele, and optionally per-sample information such as genotype information. b) In the row-centric (record) memory layout used by Vcf, and the binary representation Bcf, fields in a record are stored end-to-end. This traditional memory layout is efficient when querying many, or all, fields in a few records. Random-access to fields is generally inefficient as unwanted fields needs to be read from disk, uncompressed, and then discarded. c) In contrast, the vertical partitioning of a table into a column-centric memory layout (also called a column store) allows the same field of multiple records to be stored into a single, contiguous, memory address. Querying columnar data is generally faster because of better data locality and have better random-access performance because of the fixed stride sizes for most data types. d) Additional effort is required to maintain the relationship between a record and the fields it contains. I address this mapping problem by first hashing the global identifier of the fields in a record to its block-wise (subset of dataset in terms of variants), local, identifiers. This list of local identifiers are in turn hashed to create a local identifier that maps to a given pattern of identifiers. The pattern itself is stored as a packed bitvector. Storing the local identifier to a pattern is sufficient information to retrieve back data from the correct columns.

4.2.2 Trade-offs in the choice of storage model

The choice of data storage model, either row-centric or column-centric, depends on the target downstream application. Accessing a single record, or multiple fields in a single record, is generally faster in the row-centric orientation as only a single disk seek is required to retrieve the target data. In this case, a column store would have to perform multiple seeks, decompress multiple columns, slice out a single value from each, and then reconstitute the output record. In contrast, if the target queries are required to access multiple records then the seek costs of a column store is amortized by the reduced transfer time, lower memory bandwidth, and more efficient vectorized computations. For these reasons, column stores are generally most useful where performance critical queries involve few fields from many records. This is generally the case in common genetics workflows where only a single, or a handful, of fields are required to compute the statistics or metrics of interest.

4.2.3 Compressing genotypes using run-length encoding

In the incumbent interchange format, Vcf/Bcf[26], genotypes are stored in a implicit $X^{n \times m}$-matrix with 0 encoding the reference allele and $1, 2, \ldots, k$ encoding the first, second, up to k alternative alleles (**Figure 4.1a**). Most of this information is highly repetitive (sparse) because of low haplotype diversity in humans[123]. This property can be exploited using well-established and light-weight compression paradigms such as run-length encoding where long repetitions (runs) of a symbol (template) are stored as (template, length)-tuples. In most applications for compressing genetic data[34, 71], run-length encoding is applied on the single symbol level as (allele, length)-tuples using either variable-length integers[34] or fixed-width words. In Tachyon, run-length encoding is applied on the genotype level resulting in (genotype, length)-tuples. Tuples are partitioned into different data streams according to the longest observed run-length at a site such that $[0, 2^{8-\sigma})$, $[2^{8-\sigma}, 2^{16-\sigma})$, $[2^{16-\sigma}, 2^{32-\sigma})$, and $[2^{32-\sigma}, 2^{64-\sigma})$ correspond to the four possible target bins where σ is the number of bits used to store the template. Furthermore, we expect that most sites in human datasets will be biallelic and thereby require at most $\log_2(2) = 1$ bit of storage for the template component. In contrast, the much less frequent multiallelic sites, require a variable $\lceil \log_2(\sigma) \rceil$ bits of storage for the template and have greater data entropy and in consequence worse compression performance. Because of this, biallelic and multiallelic genotypes are stored separately. Bialleic sites are compressed using Algorithm 5 and n-allelic sites are compressed using Algorithm 8.

Algorithm 8 Run-length encode n-allelic diploid variants

function EncodeGenotypesSimple(G, n) ▷ Where G - vector of n-allelic genotypes, n - number of samples
  Let m be the number of unique alleles
  Let $E =$ (allele A, allele B, phase) $\mapsto (\mathbb{Z}^+, \mathbb{Z}^+, \{0, 1\})$ be a 3-tuple genotype
  Let ceil$((2 \times \text{ceil}(\log_2(m)) + 1)/8) \in \{1, 2\}$ be the word-size of E
  $R = \{\}$ be an empty vector of E
  for $i = 0$ to n **do**
    Push (G_i^A, G_i^B, G_i^p) into R
  end for
  return R
end function

4.2.4 Permutation-based preprocessing of genotypes

Directly compressing (run, template)-tuples results in moderate memory savings while maintaining the ability to be queried directly. It is possible to further reduce the storage cost of genotypes by exploiting the limited haplotypic diversity by conditionally sorting the collection of genotypes up to, but not including, the current position l resulting in a prefix-sort in the range $[0, l)$. This preprocessing paradigm was popularized in the genomics community with the description of the Positional Burrows-Wheeler Transform (PBWT)[34] and its successive implementation in BGT[71]. Unlike previous implementations, Tachyon permutes genotypes, of any ploidy, rather than individual haplotypes. This representation enables genotype-based summary statistics to be queried directly from compressed representation in contrast to the otherwise more practically limiting haplotype-based queries.

Permuting genotypes using the PBWT has the consequence that each site has a pseudo-random permutation order in respect to the original sample order. This unpredictable pattern inherently disallows random-access lookups of samples with the consequence that random-access first requires the iterative unpermuting of stretches of sites in the genetic matrix from the last checkpoint to the current position. I address this using a hybrid approach where a given input genetic matrix is first partitioned into non-overlapping blocks, by either number of variants or number of bases covered, followed by computing the final PBWT-based permutation order for each given block. Instead of permuting each genotypic vector at a site given the suffix-sorted position up to that point I permute all records in a block using the same permutation order. In this approach, random access can

be maintained to any record in a block without positional dependency between records. The block-wise permutation order is stored in order to map back to the original sample order. Because of these difference to the original PBWT method, I call this approach the gtPBWT.

Storing the block-wise permutation order can be prohibitive compared to the genotypes themselves for large datasets. Fortunately, this permutation of $0, 1, \ldots, s-1$ for s samples using w-bit machine words can be compressed by exploiting the skewed per-bit distribution. Without loss of generality, s words of w bits represented as $\{(0, 1, \ldots, w-1), (0, 1, \ldots, w-1), \ldots, (0, 1, \ldots, w-1)\}$ can be reversibly transformed into the form $\{(0, 0, \ldots, 0), (1, 1, \ldots, 1), \ldots, (n-1, n-1, \ldots, n-1)\}$ with the result of reduced data entropy and better compression. The relative storage cost of the permutation array compared to the genotypic data can be balanced by modifying the block size. Large block sizes will compress the genotypic component worse because linkage-disequilibrium is generally short in humans. Small block sizes compress genotypes better but the savings is generally outweighed by the storage cost of the permutation array.

The conditional PBWT permutation can be implemented in linear time as a partial radix sort (Algorithm 9) by maintaining an index mapping the permuted order, the positional prefix array (PPA), back to the original order. The PBWT can then be constructed (Algorithm 10) and similarly reversed in linear time, with or without the permutation array, in either the forward or reverse orientation (Algorithm 11 and Algorithm 13).

Algorithm 9 Updating the Positional Prefix Array

```
1:  function UpdatePrefixArray(I, P) ▷ I - input vector, P - the current positional
    prefix array
2:      Let Q_0 and Q_1 be queues
3:      for i = 1 to n do                          ▷ Radix sort on binary alphabet
4:          if I[P[i]] == 0 then
5:              Q_0.enqueue(P[i])
6:          else
7:              Q_1.enqueue(P[i])
8:          end if
9:      end for
10:     P = {Q_0, Q_1}                             ▷ P is now in sorted order
11: end function
```

Algorithm 10 Constructing PBWT

```
1: function BuildPBWT(I)                                   ▷ Where I - input vector
2:     Let O[1...n, 1...m] be the output matrix
3:     Let P be the prefix array initialized to 1...n
4:     for k = 1 to m do
5:         for i = 1 to n do                               ▷ Output I[k] permuted by P
6:             O[k, i] = I[k, P[i]]
7:         end for
8:         UpdatePrefixArray(I[k], P)                      ▷ Update P
9:     end for
10:    return O                          ▷ Optional storage of P for right-left decoding
11: end function
```

Algorithm 11 Reversing the PBWT in Left-Right Orientation

```
1: function ReversePBWTForwards(I)
2:     Let O[1...n, 1...m] be the output matrix
3:     Let P be the prefix array initialized to 1...n
4:     for k = 1 to m do
5:         for i = 1 to n do        ▷ Input I[k] is permuted by P: inverse permutation
6:             O[k, P[i]] = I[k, i]
7:         end for
8:         UpdatePrefixArray(O[k], P)                      ▷ Update P
9:     end for
10:    return O
11: end function
```

Algorithm 12 Reversing Position Prefix Array in Right-Left Orientation

```
1: function ReversePrefixArrayBackwards(I, P)
2:     ▷ Where I - input haplotype array at position i, P - positional prefix array at position i + 1
3:     Let Q_0 and Q_1 be queues
4:     Let R[1...n] be an empty array
5:     for i = 1 to n do                                   ▷ Loop over permuted order
6:         if I[P[i]] == 0 then
7:             Q_0 += P[i]
8:         else
9:             Q_1 += P[i]
10:        end if
11:    end for
12:    for i = 1 to n do                                   ▷ Loop over linear order
13:        if I[i] == 0 then
14:            R[i] = Q_0.dequeue()
15:        else
16:            R[i] = Q_1.dequeue()
17:        end if
18:    end for
19:    P = R                                               ▷ Update P
20: end function
```

Algorithm 13 Reversing PBWT Right-Left

```
1: function ReversePBWTBackwards(I, P)
2:     Let O[1...n, 1...m] be the output matrix
3:     for k = m to 1 do
4:         ReversePrefixArrayBackwards(I[k], P)
5:         for i = 1 to n do        ▷ Input I[k] is permuted by P: inverse permutation
6:             O[k, P[i]] = I[k, i]
7:         end for
8:     end for
9:     return O
10: end function
```

4.2.5 Efficient summary statistics from compressed genotypes

Encoded, and possibly permuted, genotypes packed into succinct (template, length)-tuples result in considerable memory savings but is less useful in the context of efficient genotype-based queries as the decompression overhead would outweigh the benefits of the memory savings. Addressing this, I describe a supportive data structure called **gtOcc** that enables the summarization of genotypes between any two points in the cumulative mass function for a collection of individuals (a grouping) in compressed representation (**Figure 4.2**). In other words, a grouping G is a subset of individuals $G \subseteq I$ of the total set of available individuals, I. The proposed **gtOcc** data structure is similar to the **Occ** matrix used in FM-mapping[37] but has to my knowledge not been used in this context previously.

Formally, the function $\mathbf{gtOcc}(c, k)$ is the number of occurrences from $S[1 \dots k]$ for samples belonging to group $c \in C$. This function makes it possible to count the occurrences between any two points i and j in $O(1)$-time through the relationship $\mathbf{gtOcc}(c, j) - \mathbf{gtOcc}(c, i)$ where $\mathbf{gtOcc}(\cdot, 0)$ is defined to 0. We exploit this property in terms of run-length encoding as $\mathbf{gtOcc}(c, p + R^L[i]) - \mathbf{gtOcc}(c, p)$ where p is the cumulative positional offset in $[0, n]$ observed in $S[1 \dots p]$. Constructing $\mathbf{gtOcc}(\cdot)$ is described in Algorithm 14 and the application of $\mathbf{gtOcc}(\cdot)$ in the context of run-length encoded data is described in Algorithm 15. $\mathbf{gtOcc}(c, k)$ returns the total count for all groupings c in a linear pass of the data in worst case $O(n|C|)$-time. It is worthwhile to mention that two or more groups may contain the same sample and there is no restriction such that all samples must be included.

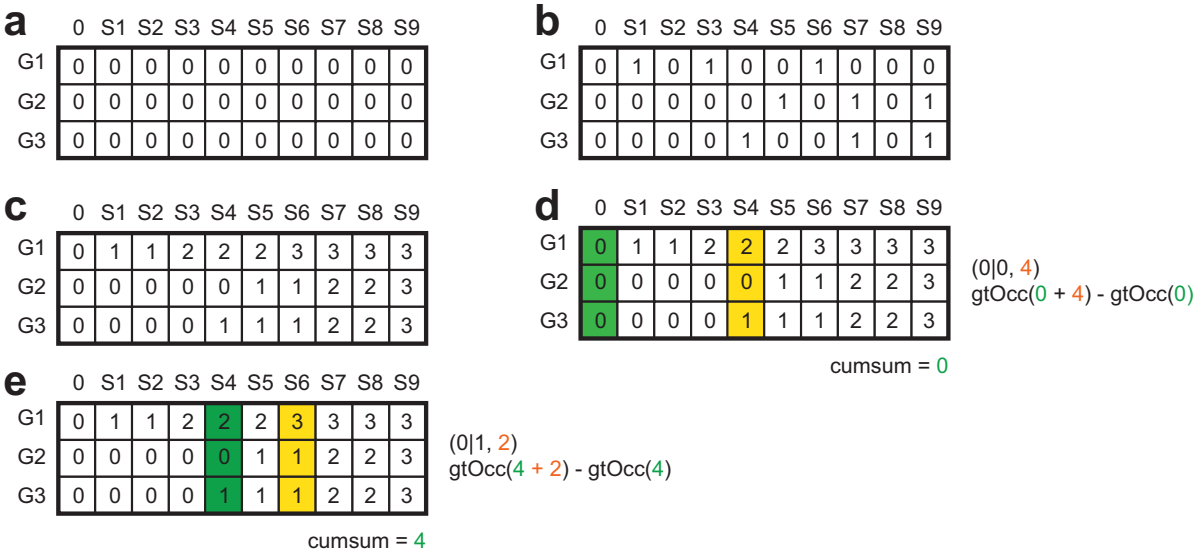


Figure 4.2 Worked example of gtOcc(·) over three groupings. a) Given nine samples labelled $S_1, S_2, \ldots, S_9$ and three groupings labelled G_1, G_2, and G_3 the gtOcc structure can be described as $O^{|S|+1\times|G|}$ matrix initialized to zero. By definition, O_0 is defined to **0**. b) In this case, our groupings are binary (member or not member). If a sample belongs to a target group then we set that cell to one (1). For example, S_7 is a member of groups G_2 and G_3. c) Next, the cumulative sum across each grouping is computed such that the value in any cell correspond to the number of samples belonging to this group up to that point. d) This matrix is sufficient to enable summary statistics between any two points. In this example, the first observed run-length encoded object is a (0|0, 4)-tuple. By using the gtOcc table it is possible to answer how many individuals belong to each group in the range $[0, 4)$ and thereby how many copies of 0|0 belong to each group. This process involves simple arithmetic operations: $\mathbf{gtOcc}(\sum_{i=0}^{i<j} R_i + R_j) - \mathbf{gtOcc}(\sum_{i=0}^{i<j} R_i)$ where R_i is the run-length of object i. The next step with the tuple (0|1, 2) is depicted in e).

Algorithm 14 Construct sample-group lookup matrix

function ConstructgtOcc(S, G, n) ▷ Where S - sample vector, G - grouping vector, n - number of samples
  Let g be the unique members of G
  Let X be a map from G to $g \in [0, |g|)\mathbb{Z}$ ▷ Implemented as a hash table
  Let $A[0 \ldots n, 0 \ldots |g|]$ be the output matrix
  $A[0] = \{0, 0, 0, \ldots |g|\}$
  for $k = 1$ to n **do** ▷ Complexity $O(ng) \leq 2n, g \leq n$
    for $i = 0$ to $g - 1$ **do** ▷ Initialise $A[k]$ equal to $A[k-1]$
      $A[i, k] = A[i, k-1]$
    end for
    $A[X[G[i]], k]$ += 1
  end for
  return A
end function

Algorithm 15 Summarise group-wise genotype frequencies from RLE data

function UPDATEGROUPRLE(R, A)▷ Where R - run-length compressed vector, A - lookup matrix
  Let g be the number of groups
  Let $X[0 \dots g-1, 0 \dots 15]$ be the output matrix
  Let $P = A[0]$
  Let $p = 0$ be the cumulative position
  for $i = 0$ to $|R| - 1$ **do** ▷ Complexity $O(|R|g) \leq O(n^2), g \leq n$
    $p \mathrel{+}= R^L[i]$
    for $j = 0$ to $g - 1$ **do**
      $X[j, R^E[i]] \mathrel{+}= A[p+1, j] - P[j]$
    end for
    $P = A[p+1]$
  end for
  return X
end function

4.2.6 Population genetic statistics

Many common statistical measures in genetics can be accelerated by directly operating on compressed run-length encoded data by using the **gtOcc** table. This strategy well suited to expedite many common statistical and population genetics measures by reducing run-time to $O(|R|)$-time, where $|R| \ll n$ on average. To demonstrate the efficiency of this approach, I describe how to compute a series of population genetics statistics using this representation.

The fixation index (F_{ST})[19, 20, 134] is a common statistics that attempt to quantify the population differentiation because of drift or selection. F_{ST} is calculated on a per-site basis and requires the genotype frequencies per population of interest. Genotype frequencies can be efficiently computed over many populations (groups) using the **gtOcc** function (Algorithm 15).

In contrast to statistics that involve partitioning the input data into groups, several measurements can be computed directly from the genotypic vector itself. For example, measures of nucleotide diversity such as θ_S (Equation 4.1) and π (Equation 4.2) can be computed from the genotype frequencies at a site:

$$\theta_S = \frac{S_n}{\sum_{i=1}^{n-1} i^{-1}} \tag{4.1}$$

$$\pi = \frac{n}{n-1} \sum_{i=1}^{S_n} 2p_i q_i \tag{4.2}$$

where n is the number of chromosomes, S_n is the number of polymorphic sites, p_i is the derived (nonancestral) allele frequency of the i-th site, and q_i is the ancestral allele frequency of the i-th site. Accumulating genotype counts from a run-length compressed genotypic vector can be done in $O(|R|)$-time. These measurements can in turn be used to quantitatively evaluate departures from the expected patterns of neutral variation. Tajima's D[119] (Equation 4.3) is a well-established and frequently used measurement for investigating this:

$$D = \frac{\pi - \theta_S}{\sqrt{Var(\pi - \theta_s)}} \tag{4.3}$$

4.2.7 Comparing performance

Compressing genotypes/haplotype stored in the incumbent Vcf/Bcf interchange format was converted to a valid Tachyon archive using the base command:

```
tachyon import -i ${input_bcf} -o ${output_yon}
```

Interconvertion between Vcf, Vcf.gz, uBcf, and Bcf was performed using `bcftools`.

```
bcftools -i ${input_file} -O ${file_type} -o ${file_name}
```

4.2.8 Simulated data sets

Genotypes from idealized datasets were simulated using msprime version 0.5d[56] with the mutation rate and recombination rate per site per $4N$ generations set to 0.001 and effective population size to 10,000. Genomic region was set 100 megabases and the number of samples varied according to the experiments. Because msprime simulates individual haplotypes, we simulated $2N$ haplotypes for each case and greedily combined adjacent haplotypes into a single diploid genome.

4.2.9 Experimental data sets

Individual chromosome Bcf files were retrieved for each cohort and combined into a single dataset using the Bcftools `concat` command.

- 1000 Genomes phase 3 (hg19)[123]: This dataset comprises variants and small insertions/deletions (indels) for 2,504 individuals of various ancestry.

- Haplotype Reference Consortium (hg19)[125]: This dataset comprises of a mixed cohort of 32,488 whole-genome sequenced individuals. There are >39 million SNPs with an allele count ≥ 5 and no indels. All samples from the 1000 Genomes phase 3 project are included in this dataset.

4.2.10 Computing environment

Code was compiled with GCC 8.3 using the optimization flags "-O3 -march=native" to restrict optimizations to the host-machine architecture. All tests were performed using a host machine with a 14-core 22 nm Haswell (x64) Xeon E5-2697 v3 with the AVX2 microarchitecture and a pair of NVMe solid-state hard drives operating in RAID-0 unless otherwise specified. Memory usage and execution times were measured using Linux `time` subroutines.

4.3 Results

With the advent of population-scaled datasets, methods for efficiently storing and analyzing large genetic variant datasets have been of considerable interest recently[74, 147, 27, 31, 34, 71, 30, 63]. Unlike many of these studies that limit their focus to compressing genotypes[27, 31, 34, 71, 30, 63], we address the storage and analysis of entire genetic variant datasets.

In a similar fashion to CRAM[48] for sequencing data, row-centric genetic variant records (**Figure 4.1a**) are vertically transposed and stored in separate data streams according to the target field (**Figure 4.1b**). This memory layout allows for field-specific compression algorithms to be used resulting in greater compression. The relationship mapping the set of columns to the set of records is maintained using a simple hash-based approach (**Figure 4.1d** and see Methods). In Tachyon, genotypes from a consecutive series of variant sites are partially sorted according to their prefix[34] and then greedily compressed using variable-length run-length encoding.

To explore the performance gains of representing genetic variant datasets this way, I compared Tachyon to a series of frequently used compression engines across their respective parameter spaces (**Figure 4.3**). Using data for chromosome 20 of the 2,504 individuals in the 1000 Genomes project[123](1KGP3), Tachyon is 9-fold smaller compared to the frequently used BGZF-compressed Vcf archives (vcf.gz) and 5-fold smaller compared to Bcf (**Figure 4.3a**). In addition, uncompressed Bcf (uBcf) was externally compressed using Zstd in order to demonstrate that any file size savings are not exclu-

sively caused by using a better compression engine compared to BGZF used in htslib. Despite using Zstd, Tachyon remains 1.2-3-fold time smaller while maintaining random-access to data compared to the complete loss of random-access when using an external compressor. Additional gains in compression can be seen by preprocessing genotypes using a permutation-based approach (**Figure 4.3b**) at the cost of additional compute (**Figure 4.3c**). Similar numbers are seen when using data for chromosome 11 for the 32,470 whole-genome sequenced individuals in the Haplotype Reference Consortium (HRC) project (**Figure 4.3d-f**).

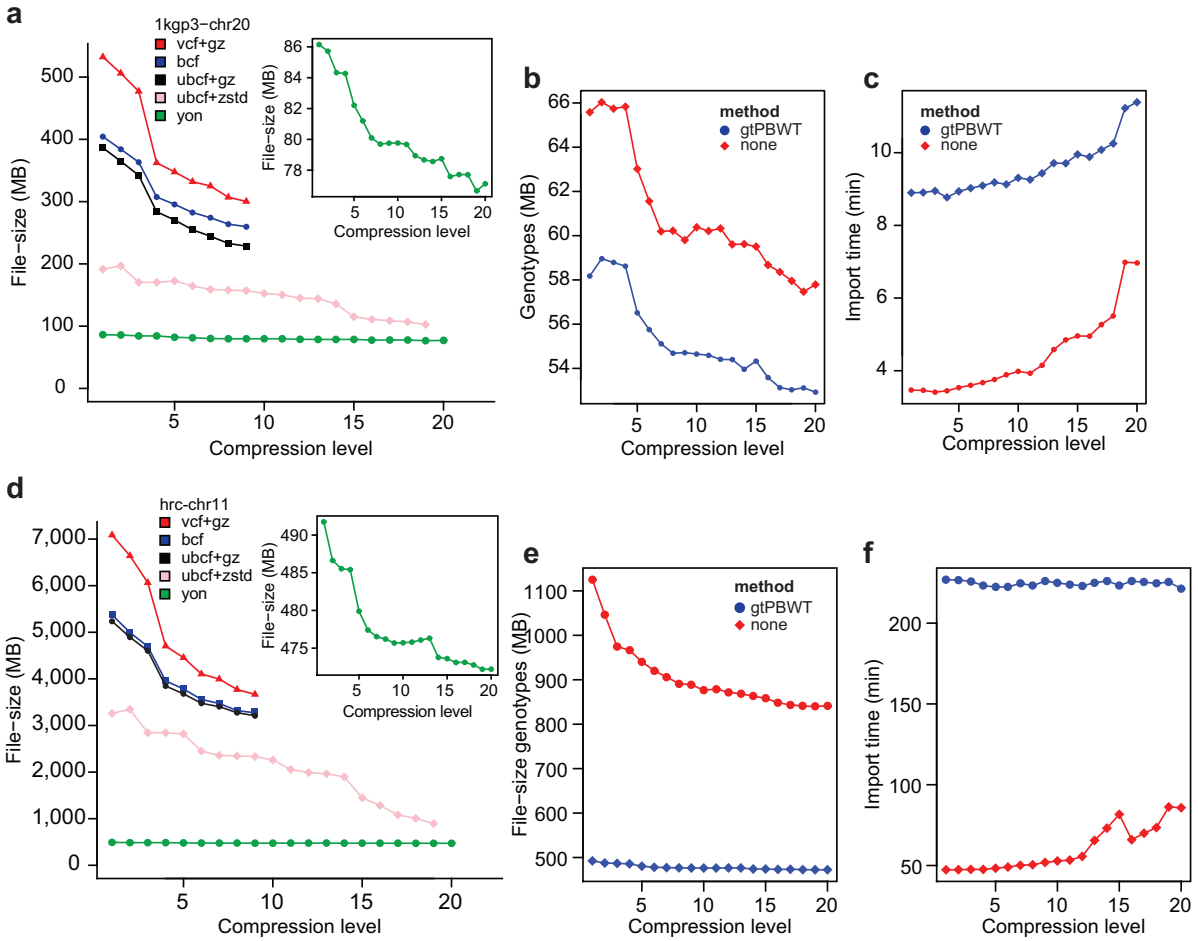


Figure 4.3 Compression performance on real data. a) Data for chromosome 20 for the 2,504 individuals in the 1KGP3 dataset was compressed over different compression levels using `htslib` with either gzip, in the bcf format, or uncompressed bcf using gzip as an external compressor, uncompressed bcf using zstd as an external compressor, or using tachyon. Tachyon offer considerably better compression performance. b) First permuting genotypes with the gtPBWT method result in considerable savings in memory but require additional import time c). d) Same as a) for chromosome 11 for the 32,470 individuals in the HRC dataset. e-f) same as b-d) for HRC-chr11.

Next, we compared compression performance over the entire 1KGP3 dataset (**Figure 4.4a,c**) and HRC dataset (**Figure 4.4b,d**) demonstrating that Tachyon has better growth properties compared to Bcf. As a consequence of the succinct representations of genotypes, and some columnar data, uncompressed (succinctly compressed) Tachyon data approximately the same size as compressed Bcf for the 1KGP3 dataset (**Figure 4.4c**) and ∼60% smaller for the HRC dataset (**Figure 4.4d**). Next, I examined if this property continues to scale with future datasets encompassing hundreds of thousands of samples by simulating haplotypes from 10,000 to 1 million diploid chromosomes. Unsurprisingly, compression in Tachyon scales considerably better with increased number of samples (**Figure 4.4e**) and the succinct representation of genotypes result in dramatic savings in space (**Figure 4.4f**).

Storing column projections of a table in separate byte streams enables the selective retrieval of the target fields of interest in time proportional to its compressed size. To demonstrate the efficiency of column stores compared to row-centric formats, I benchmarked the completion time of retrieving only the chromosome and position from these large datasets. At 1 million samples Tachyon demonstrate a >233-fold speed improvements compared to Bcftools (**Figure 4.4g**). As another example, vertical partitioning of the gnomad chromosome 1[55] data result in a 4-fold reducing in size (36 Gb to 8.6 Gb, see **Table 1**) while also enabling the efficint retrieval of individual columns >17-fold faster (47 min for bcftools vs 2.6 min).

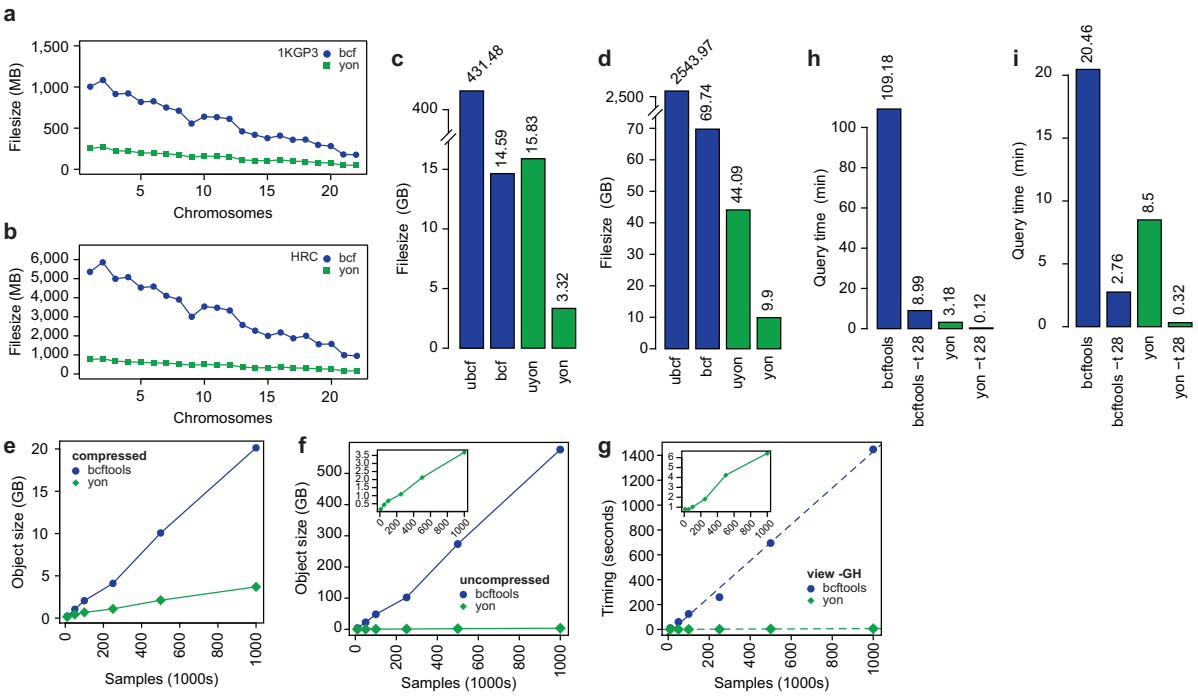


Figure 4.4 Performance on real and simulated data. a) File sizes for Bcf (blue) and tachyon (green) for 2,504 diploid individuals of various genetic ancestries for each chromosome in the 1KGP3 dataset. b) File sizes for Bcf and tachyon for 32,470 diploid individuals of mostly European ancestry for each chromosome in the HRC dataset. c) File sizes for uncompressed (ubcf and uyon) and compressed (bcf and yon) archives for the entire 1KGP3 dataset demonstrating that the uncompressed yon file format is comparable in size to compressed bcf. d) File sizes for uncompressed and compressed archives for the HRC dataset. e) Haplotypes were generated in the range from 1,000 to 1,000,000 and compressed with bcf and yon. f) Uncompressed data for e) demonstrate that the succinct run-length encoding representation in tachyon result in dramatic savings in memory for uncompressed genotypes. g) Response times when querying for meta information only (all fields excluding all the per-sample `FORMAT` fields). The columnar representation of data in tachyon result in query times proportional to the individual columns. In contrast, the row-centric orientation of Bcf scale in proportion to all available data. h) Response times for querying for meta information only (as in g)) for the 1KGP3 dataset with either a single thread or 28 threads. i) Same as h) for the HRC dataset.

Once genotypes are in a compressed representation it is highly desirable to perform queries in compressed space to avoid costly decompression back into literal space. This additional decompression overhead could outweigh the benefits of the memory savings. To address this, I developed a supportive data structure that enables genotype-specific queries in the run-length encoded space using simple arithmetic (see Methods). In order to demonstrate potential applications, I computed a large number of common summary statistics and population genetics statistics (**Table 4.1**) for several distinct subsets of samples (groups) using the 1KGP3 chromosome 20 dataset (**Table 4.2**). Notably, this approach is very flexible, allowing an individual to be a member of zero or more groups. Computing multiple statistics grows sublinear with increasing number of groupings. Calculating >554 million statistics for 36 overlapping groups using the 1KGP3 chromosome 20 completes in under 10 minutes on a single CPU core.

Table 4.1 Example of implemented statistics and population genetic statistics. Lengths correspond to the number of possible values in the output vector where "A" corresponds to an array of length $[0,)$.

Field	Length	Type	Description
`FS_A`	A	Float	PHRED-scaled Fisher's exact test P-value for allelic strand bias
`AN`	1	Integer	Total number of alleles in called genotypes
`NM`	1	Integer	Total number of missing alleles in called genotypes
`NPM`	1	Integer	Total number of samples with non-reference (compared to largest) ploidy
`AC`	A	Integer	Total number of alleles
`AC_P`	A	Integer	Total number of alleles each strand

AF	A	Float	Allele frequency of allele
HWE_P	1	Float	Hardy-Weinberg equilibrium P-value
VT	A	String	Variant classification (SNP, MNP, INDEL, CLUMPED, SV, UNKNOWN)
MULTI_ALLELIC	0	Flag	Indicates if a site is multi-allelic (number of alternative alleles >1)
F_PIC	1	Float	Population inbreeding coefficient (F-statistic)

Table 4.2 Annotation time for 2,504 individuals using 1KGP3 chromosome 20. Given an annotation file mapping sample names to some group(s), including across all samples, Tachyon can compute the properties listed in Table 4.1 for each group and variant site. Query used: `tachyon view -i <file.yon> -GHX -b <groupings>`. All refers to a single grouping encompassing all samples. Abbreviations: super-pop, 1000 Genomes super-populations (AFR, AMR, EAS, EUR, and SAS); pop, 1000 Genomes populations.

Query	Time	Groups	Values
Output only	15.21s	0	0
Annotate all	51.65s	1	9
Annotate super-pop	2m32.74s	6	54
Annotate pop	07m28.69s	27	243
Annotate super-pop + pop	08m43.75s	32	288
Annotate super-pop + pop + gender	09m42.00s	34	306

4.4 Discussion

Recognizing current and impending scaling challenges posed by large-scale sequence variant datasets, my motivation was to explore the application of column-centric storage solutions to improve compression and decrease downstream analysis time. I have shown that storing large genetic variant datasets in this proposed format results in considerable savings in both the compressed and uncompressed space. In order to simultaneously offer efficient compression and query performance I described a distinct variation of the PBWT while still maintaining random-access. Additional query performance can be achieved by operating directly on the succinct representation of genotypes. To support summary statistics-based queries on compressed genotypes I developed the gtOcc data structure. I demonstrated how it is possible to expedite a range of statistics and population genetic statistics by using this data structure.

Together this file format and associated algorithms can reproduce a target input file with bit-exact correctness while reducing computational cost and time. I expect that they will enhance both existing and novel methods exploring population-scaled datasets.

After this work was completed, I recognized that there are other compression methods that provide superior performance on phased haplotypes[27, 31, 34, 71, 30]. This will be explored in Chapter 5.

4.5 Acknowledgements

I am grateful to J. Bonfield for discussions regarding self-contained indices. I am grateful to members of the The Global Alliance for Genomics and Health (GA4GH) Future of Vcf and File Formats working groups for fruitful discussions. I am grateful to members of the open-source community for helping improving the implementation.

Chapter 5

Efficient compression and analysis of population-scaled genetic variation datasets

In this chapter I will introduce several algorithms for efficiently storing and querying large-scale genetic matrices ($X^{(n \times m)} \in \mathbb{Z}^{+}$) of haplotypes/genotypes that builds on ideas from Chapters 2, 3, 4. I will also extend the computation of XX^T for binary matrices ($X^{(n \times m)} \in [0, 1]\mathbb{Z}$) to the problem of computing genetic similarity matrices ($X^T X$). Collectively, these methods and algorithms are packaged into the software project `djinn` and can be accessed at `https://github.com/mklarqvist/djinn`.

5.1 Introduction

In the Chapter 4 I discussed efficient algorithms for storing and querying population-scaled sequence variant datasets including all non-genetic fields such as per-site and per-sample information and other meta information. In many situations, the genetic variants stored in sequence variant datasets are of principal interest. Recognizing the need for both efficient compression methods and efficient algorithms for querying genotypes for downstream analysis, I explore different computationally efficient representations of genotype datasets.

5.2 Methods

5.2.1 Overview of the strategies

There are two basic kinds of memory layout approaches to storing sequence variant data. (1) In a variant-centric record layout, each row is a vector of genotypes across all samples at a site. This kind of layout is best suited for queries involving all samples in few records. (2) In a sample-centric layout, each row is a vector of genotypes for a single individual. Storing genotypes this way enables efficient queries involving a small number of individuals in an archive. For example, individual-centric layouts are efficient in computing pairwise differences between pairs of individuals.

Genotypic vectors, in either orientation, are encoded using a mixture of run-length encoding (RLE) and bitvectors. Representing genotypes using either of these encodings permit queries without first inflating data back to the literal genotype vector. Operating directly on encoded data generally results in considerable speed improvements. For storage on disk, encoded genotypes are ultimately compressed using either general-purpose compressors or a series of tailored statistical models.

In order to support multi-allelic genotypes and missing data without compromising on compression we decided to partition input data into three non-overlapping categories: (1) bi-allelic sites, (2) bi-allelic sites with missingness and/or with mixed ploidy, and (3) all other cases. Data is encoded and compressed separately and their order relationship is maintained with a simple map. This approach requires an additional processing step but result in considerable gains in compression.

5.2.2 Permuting sample order to reduce data entropy

The positional Burrows-Wheeler transform (PBWT)[34] is a reversible preprocessing routine that takes a set of aligned haplotype sequences as input. It sorts a set of equal length strings at each character position in the lexicographic order of the reverse prefixes up to that position. Sorting strings in this fashion results in considerably reduced data entropy and improved downstream compression (**Figure 5.1**) in most cases as similar symbols are placed closed together because of linkage disequilibrium (haplotype structure). Originally, the PBWT was described for binary alphabets[34] but it can be extended to support arbitrary alphabet sizes[82] with minimal effort.

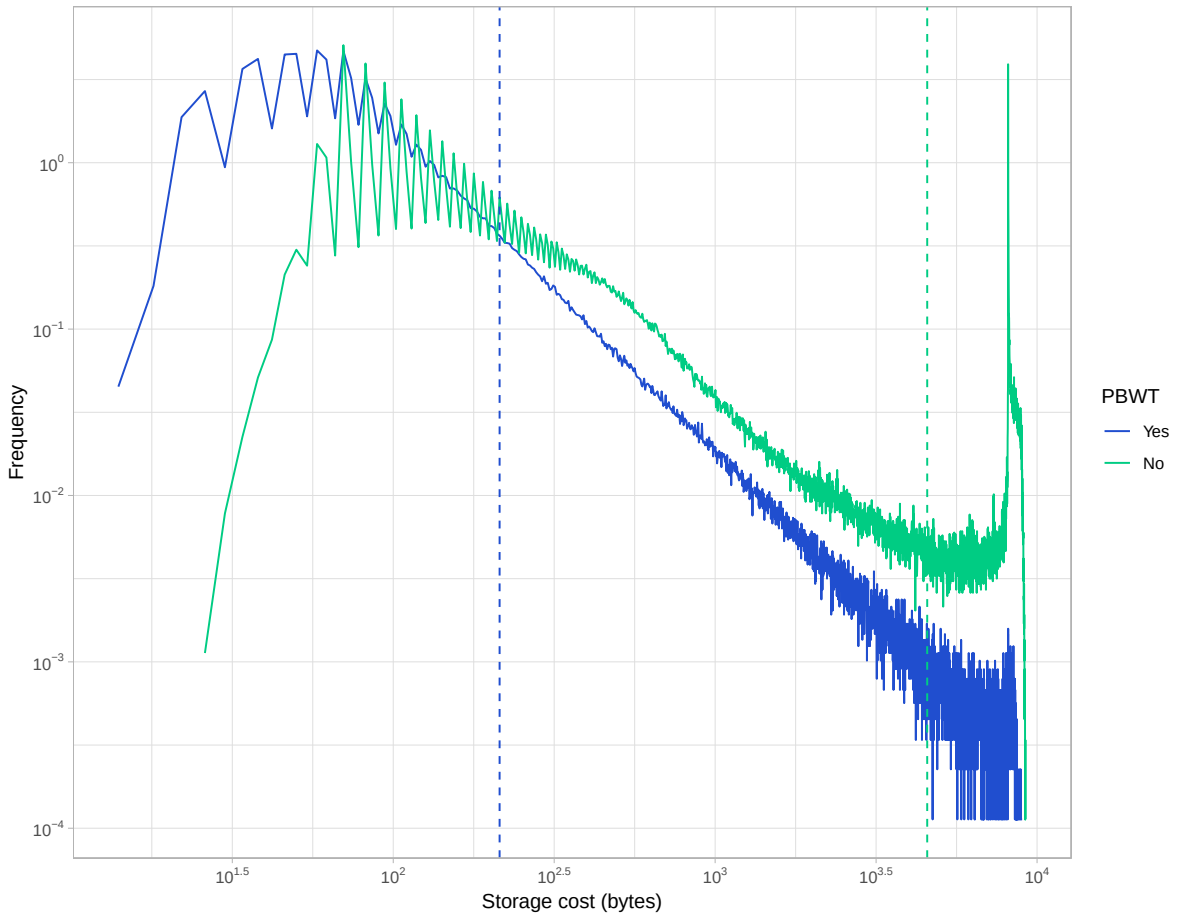


Figure 5.1 Size distribution for EWAH object for PBWT-permuted and unpermuted variants. Variants from HRC chromosome 20 for 32,470 diploid individuals were either permuted or not and analysed for its uncompressed storage cost per variant. Permuting with PBWT result in a dramatic shift of high storage-cost variant sites into low storage-cost representations. Vertical dashed lines correspond to the average storage cost for 90% of the data.

During the PBWT procedure, we maintain a positional array of the previous permutation order. Storing this positional array is not required as it is possible to reverse the PBWT in the front-to-back orientation (e.g. position 1, position 2, position 3) without it. However, if the array is stored, it is possible to reverse the PBWT in the back-to-front orientation (e.g. position 3, position 2, position 1).

Unlike BGT[71], we do not intermittently store (checkpoint) the permutation array but instead reset the permutation order for each successive block of variants. Not storing the permutation array results in smaller file sizes but is expected to reduce random access speed times by a factor of two as reverse searching (back-to-front) is not available.

Restarting the permutation order works well in practice when the block size is not small (**Figure 5.2**).

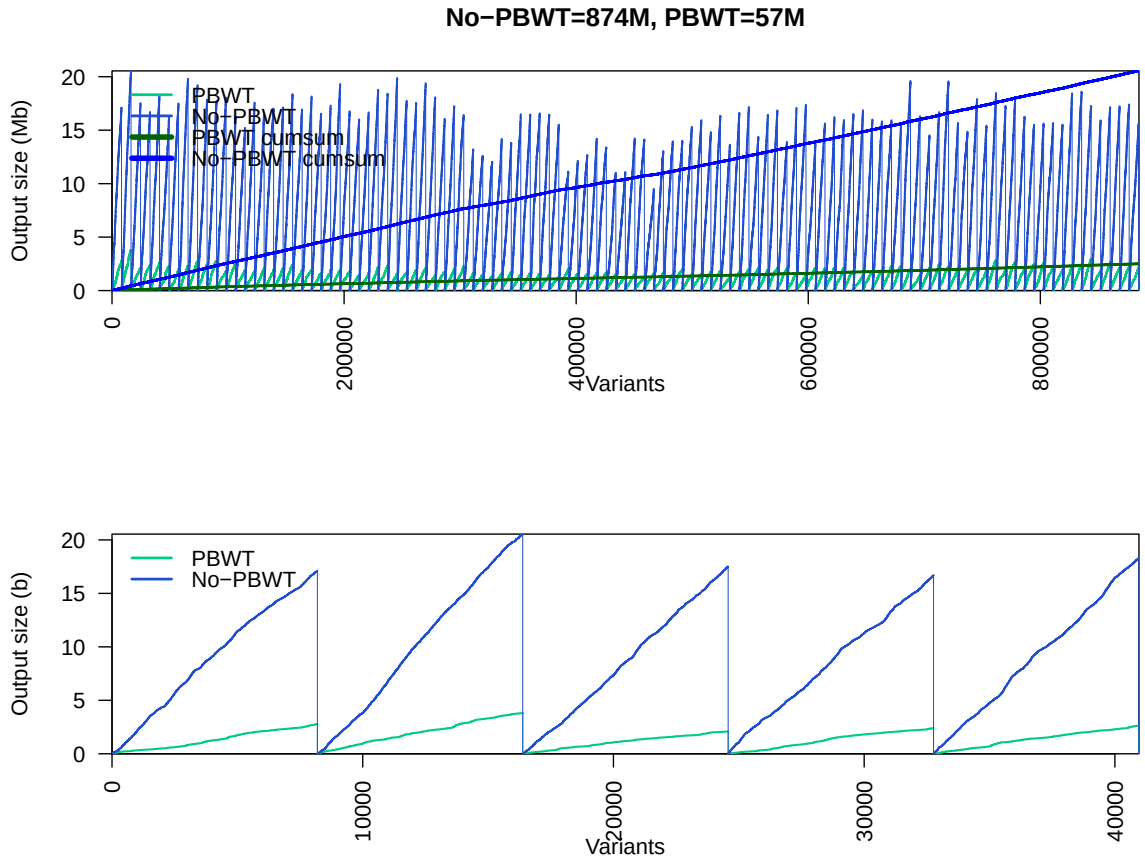


Figure 5.2 Overview of block-based compression. Blocks of variants are compressed and stored in independent data blocks. This enables partial random access to PBWT-permuted data blocks without having to checkpoint the permutation array (current haplotypic sort order). This approach gives rise to a characteristic wave-like periodic pattern for both PBWT-based (green) and unpermuted (blue). Storage keeps increasing until the target number of variants have been stored and a new block is started. Cumulative file size is shown for both approaches. The bottom panel is restricted to the first five variant blocks.

5.2.3 Representing alleles with bitmaps and run-lengths

In the incumbent Vcf format[26], alleles at a locus are dictionary encoded such that "0" maps to the reference allele, and "1" to the first alternative allele, and "2" maps to the second alternative allele, and so on. Missing values are encoded with the special symbol

"." . An uncompressed bitmap index (bitmap) is an efficient encoding strategy that stores values using bits in a bit array. For example, biallelic alleles can be packed into machine words using a single bit each. Similarly, in multi-allelic sites and/or sites with missingness, each log_2(number of alleles) bits correspond to the allele at that position. For example, the vector {(2,0),(1,1),(1,0),(1,1)}, would be encoded as the bitmap `0000101` (ordered in reverse order, $8 \rightarrow 1$). Storing haplotypes using a bitmap has a fixed cost independent of allele frequency or haplotype diversity. Because of this, storing alleles in bitmaps result in at most 8-fold compression (1 bit per allele in a 8-bit machine word) for biallelic data and progressively worsen with increased number of alternative alleles. In order to achieve higher compression, we must employ additional methods.

The overall sparsity of non-reference alleles and the shared haplotype structure in humans[123] makes variant data highly amenable to run-length encoding (RLE). Repeated stretches of identical symbols (called runs) are packed in a (number of copies, symbol)-tuple: For example, the run 00000000 is stored as the tuple (8,0). Regions of high entropy, usually occurring in medium-to-high allele frequency variants, can result in an undesirably large number of consecutive short runs. For example, run-length encoding the example vector above would require more memory than storing the reference values in a vector of literal symbols $(0, 0, 1, 0, 1)$. In the worst case, RLE result in an increase in file size and poorer performance.

5.2.4 Extended word-aligned hybrid (EWAH) encoding of alleles

By mixing RLE with bitmaps in high entropy regions we can achieve better compression and frequently observe a concomitant gain in decompression performance. This hybrid encoding scheme was initially described as word-aligned hybrid (WAH)[136] encoding. More formally, WAH partitions a bitmap of n bits into $\lceil \frac{n}{w-1} \rceil$ words of $w-1$ bits, where w is a convenient machine word width in bits. Partitions are stored as either "fill" words comprising of only ones $(111\ldots1)$ or zeros $(000\ldots0)$ or partitions comprising of mixtures of ones and zeros called "literals". Runs of fill words are stored in a single machine word with the most significant bit set to one (1) to signal that this is a run-length encoding of fill words. The second most significant bit is set to the fill value (either 1 or 0) and the remaining $w-2$ bits encode the run-length itself. Literal words have their most significant bit set to 0 to indicate that it should not be interpreted as a run-length encoding of fills. Because one bit in every word is reserved for distinguishing between

literals and fills, this encoding occupies $1/w$ (e.g. 3% for a 32-bit word) additional space compared to uncompressed bitmaps.

This problem was addressed with a modified approach called the extended WAH (EWAH)[65](Figure 5.3). Starting with a 32-bit marker word, the most significant bit indicates what type of fill words (0 or 1) will follow. The next 16 bits represent the run-length of the fill words, and the remaining 15 bits give the run-length of the literal words directly following the fill words. Data is repeatedly processed into (marker, literal)-tuples until no more data is available. In contrast to WAH, the extended WAH will very rarely construct compressed bitmaps that are larger than the uncompressed original. We build on EWAH encoding to support arbitrary alphabet sizes used to encode multi-allelic sites and missingness by allocating additional bits for the fill word type. Similarly, by extending the alphabet from binary (0 and 1) to an arbitrary σ alphabet size we use $\log_2(\sigma)$ bits for each allele in the bitmaps.

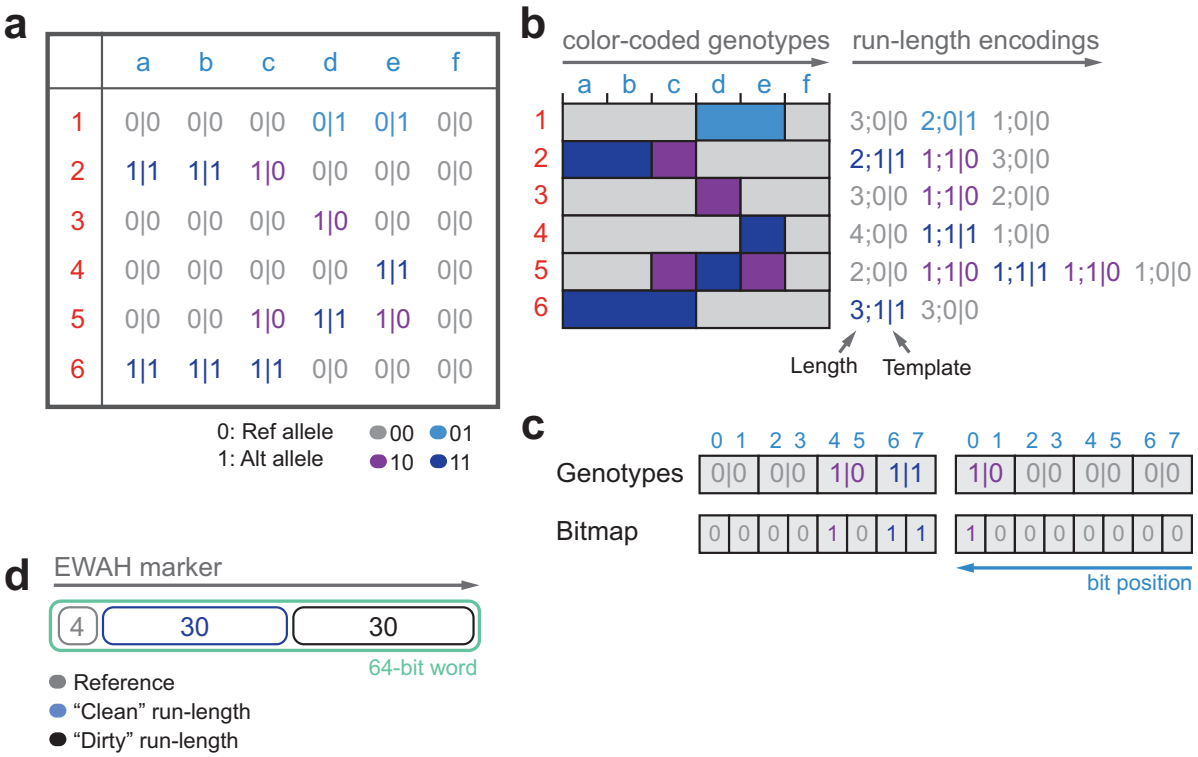


Figure 5.3 Overview of compression genotypes/haplotypes. **a)** Genetic variant data is most frequently stored as haplotypes/genotypes in a matrix $X^{n \times m}$ for n samples (columns: $a - f$) and m loci (rows: $1 - 6$) where 0 encodes for the reference allele at that position and 1 for the first alternative allele, and 2 for the second and so on. **b)** Repeated symbols can be succinctly represented as (run length, template)-tuples in a process known as run-length encoding. The relatively low haplotypic diversity in humans cause most rows in X to be sparse and therefore compress well using run-length encoding. However, many sites will compress into a larger object because of short runs (frequent template switches). **c)** Storing genotypes/haplotypes in a k-bit string (bitmap or bitvector) is an extremely computationally efficient approach for sites with few alternative alleles. Bitmaps enables the computation of many set operations such as intersection, union, and difference using a single CPU instruction. Unfortunately, bitmaps have a fixed cost of k-bits per row (record) making this approach inefficient on larger datasets. **d)** The extended word-aligned hybrid (EWAH) compression method combines run-length encoding (b) and bitmaps (c) such that long runs of a template (clean words) are run-length encoded and high-entropy regions are stored as bitmaps (dirty words). In this application, the machine-word size k is 64-bits and 4 bits are reserved for describing the template, 30 bits for the number of clean words with the defined template, and 30 bits describing the number of immediately following dirty words of size 64-bits.

After EWAH-encoding genotypic data, we support a range of final compressors with different intended use-cases. Encoded genotypic data is then compressed using

a general-purpose compressor such as Zstd (http://www.zstd.net/) or Lz4 (http://www.lz4.org/) The best choice of compressor depends on the downstream use-case: (1) Zstd produce smaller archives at the expense of decompression speed, or (2) LZ4 produce larger archives with faster decompression for query performance. Both of these compress and decompress faster into smaller archives compared to the prevalent gzip (http://www.gzip.org/) compressor. We generally suggest using Lz4 over Zstd as the former has considerably faster decompression speeds with only marginally worse compression performance. For better compression at the expense of query speeds I next describe a statistical compression method.

5.2.5 Arithmetic encoding and context modelling

Arithmetic encoding[100] and range encoding[84] are entropy encoding schemes for lossless data compression that generally offers superior compression levels at the expense of additional computation time. Given a string of symbols, entropy encoding assigns bits according to a symbol frequency distribution such that symbols that can be accurately predicted (high frequency) is stored using fewer bits. Conceptually, compression is achieved by coding the string of symbols into a single floating-point number in the range [0, 1] enabling the storage of information in fractions of a bit. Formally, arithmetic encoding predicts the conditional probability $P(x_i = t|c_{i-n}, c_{i-n+1}, \ldots, c_0)$ of a new symbol x_i given its so-called context c of size n. As the name implies, the context refers to the previous symbolic context that symbol was observed in. In many cases, the context is the preceding n symbols forming an order-n context model. In order to encode a symbol, the current interval is updated based on the predicted probability of that symbol.

First, the input haplotype string is permuted according to the sorted reverse prefix up to this position. Next, the permuted haplotype string is encoded using EWAH with the run-length and bitmap components modelled separately as follows:

1. Encode a symbol $\in \{0, 1\}$ describing that the next word is an implicit (0) EWAH marker word or (1) a bitmap.
2. If the current word is a bitmap then it is partitioned into individual bytes that are encoded individually using a shared order-0 model. Go to step 1).
3. If the current word is a (run length, reference symbol)-tuple then it is partitioned into its components and modelled separately:
4. The run-length template reference is stored in a simple order-10 context model.
5. We compute and store $\log_2$(run length), where the run length is in the target machine word units.

6. Using the $\log_2$(run length) we store either 1, 2, 4, or 8 bytes in the most significant byte order in separate order-10 context models.
7. Go to step 1).

Predicted probabilities from these context models are compressed using a shared range coder.

5.2.6 Efficient computations using compressed data

Compressing genotypes with EWAH encoding has benefits beyond reducing memory usage and storage space. Both run-length encoded and bitmap encoded data can be directly interrogated without first inflating the compressed EWAH representation. Bitmap-encoding facilitates efficient comparisons by using optimized bitwise logical operations on modern CPUs. As an example for biallelic sites, counting the number of set bits in a bitmap corresponds to the allele count at that site. Enumerating the set bits in a machine word (fixed-size unit of bits used by the CPU) is also referred to as its population count (popcount). This operation is available as a CPU instruction on most recent commodity processors and require approximately one CPU cycle / 64-bit word[88]. For biallelic data in bitmap form, it is therefore possible to count 64 alleles / CPU cycle. For non-biallelic sites, we must additionally preprocess bitmaps using a few basic bitwise operations before invoking the popcount instruction on the resulting word (see Chapter 2). Similarly, bitmaps expedite set membership, intersect, union, and difference queries through bitwise parallelism (see Chapter 3). Intersecting two bitmaps, B_1 and B_2, can be answered with a single bitwise AND operation ($B_1 \wedge B_2$). Similarly, $B_1 \vee B_2$ and $B_1 - B_2$ is equivalent to B_1 OR B_2 and B_1 NOR B_2, respectively. All of these bitwise operations require a single instruction and $\leq$1 CPU cycle on most commodity processors (https://software.intel.com/sites/landingpage/IntrinsicsGuide/).

Similar to bitmaps, when processing RLE-compressed objects, it is possible to perform a variety of useful operations without decoding. For example, counting the number of alternative alleles in a vector of RLE object R takes $O(|R|)$-time in a linear pass over the data. Given that run lengths are generally very long, this could be equivalent to counting many thousands of alleles per CPU cycle. Comparing a pair of RLE-encoded vectors, R_1 and R_2, can be performed in $O(|R_1| + |R_2|)$-time. Although RLE-compressed bitmaps permit several efficient operations it is considerably more difficult to compute in-place set operations compared to either RLE or bitmaps alone. For example, assessing set membership could in the worst case require a complete scan of a RLE-bitmap in O(|R|)-time. These scans can be hundreds to thousands of times slower compared to

using uncompressed bitmaps which provide random access in O(1)-time. Because of this considerable implication on performance, we make exclusive use of uncompressed bitmaps when performance is critical.

5.2.7 Comparison of file formats

Using the incumbent Vcf interchange format as the reference, this format stores several required meta-information fields that are not of interest in this work. These include variant name, filtering flags, and other per-site and per-individual information such as allele count and genotype quality scores. To make comparisons fair, all fields except the genotype (`FORMAT:GT`) field was stripped out and the essential contig and position fields were set to 1 and $\{1, 2, \ldots\}$, respectively. The overhead compared to storing only the genotypes is negligible even for large datasets. Trimmed Vcf files were converted into Bcf or uBcf files using Bcftools [26].

Vcf files are frequently externally compressed with gzip that use the DEFLATE[1] algorithm that combines Huffman and LZ77 encoding [149]. Similarly, the internal compression of Bcf files use a block-wise implementation of DEFLATE called BGZF that operates on blocks of 65,535 bytes (2^{16}). BGZF essentially concatenates multiple blocks back-to-back to allow random access to different sections of the archive. Partial random access at the variant-level is maintained by storing virtual offsets to the end of the current record.

In EWAH mode, Djinn stores haplotypes for N consecutive variants in three different streams according to their classification: (1) bi-allelic, no missing, and no special end-of-value symbols (mixed ploidy), (2) bi-allelic, (3) all other cases. Additional meta-information that is required to decode data is stored for each block together with a 1-of-k encoded machine word describing its content and what compressor was used. Optionally, virtual offsets to the start of each variant and to each EWAH word are stored together with the meta-information. In context modelling mode, Djinn stores data as in EWAH mode but cannot store any virtual offsets as the range coder disallows this.

5.2.8 Efficient summary statistics from compressed haplotypes

PBWT-permuted and EWAH-encoded result in considerable memory savings but is less useful in the context of efficient queries as the decompression overhead would outweigh the benefits of the memory savings. Addressing this, I describe a variation of the supportive data structure called **gtOcc** (see Chapter 4) that enables the summarization of haplotypes

[1] https://www.rfc-editor.org/info/rfc1951. Last accessed 30th July, 2019

between any two points in compressed representation for predefined groups of individuals using simple arithmetic operations.

Formally, the function **gtOcc**(c, k) is the number of occurrences from $S[1 \dots k]$ for samples belonging to group $c \in C$. This function makes it possible to count the occurrences between any two points i and j in $O(1)$-time through the relationship **gtOcc**(c, j) − **gtOcc**(c, i) where **gtOcc**$(\cdot, 0)$ is defined to 0. Two versions of **gtOcc** must be used in order to be used in conjunction with EWAH-encoded data: (1) a scalar matrix version for use with run-length encoded data and (2) a bitmap version for efficient intersection with bitmap encoded data.

5.2.9 Estimating kinship coefficients in a homogeneous population

Computing all pairwise kinship coefficients is an example of a sample-centric query that is performance critical. This query involves $\binom{N}{2}M$ genotype comparisons for N samples and M variant sites. For example, 5×10^{16} genotype comparisons is required for a million individuals over 100,000 variant sites. Fortunately, computing $N_{AA,aa}$ and $N_{Aa,Aa}$ for a pair of individuals can be calculated as the cardinality of the set-intersection between a pair of conditioned uncompressed bitmaps where $N_{AA,aa}$, $N_{Aa,Aa}$, N^i_{Aa}, and N^j_{Aa} are the total number of variants where both individuals are homozygous different, both heterozygous, and individually heterozygous for individual i or j, respectively. In order to maximize these computations we extended the vectorized carry-save add algorithm[88] for computing the popcount to jointly condition the genotype bitmaps. Several variations of this algorithm was developed to support most modern CPU-intrinsic instruction sets: SSE4.1, AVX2, and AVX-512 (see Chapters 2,3).

To estimate the kinship coefficient ϕ_{ij} between two individuals i and j using genotypes we use the genetic relationship estimator KING-homo[83]. We briefly outline the method here for completeness:

$$\hat{\phi_{ij}} = \frac{N_{Aa,Aa} - 2N_{AA,aa}}{2\hat{H}_{ij}} + \frac{1}{2} - \frac{N^i_{Aa} + N^j_{Aa}}{4\hat{H}_{ij}} \tag{5.1}$$

A variant site is rejected if there is missingness in either individual. The normalization scalar $\hat{H}_{ij}$ is defined as:

$$\hat{H}_{ij} = \sum_m 2\hat{p}_m(1 - \hat{p_m}) \tag{5.2}$$

where m indexes SNPs excluding those with missing genotypes in either individual of the pair, and allele frequency $\hat{p}_m$ at the m-th SNP is estimated from the genotype frequencies in the entire sample as

$$\hat{p}_m = \frac{\#AA + \#Aa/2}{\#AA + \#Aa + \#aa} \tag{5.3}$$

5.2.10 Comparing performance

All file size comparisons use uncompressed Vcf (uVcf) as a baseline reference. To make comparisons fair across tools, Vcf files were stripped to contain the absolute minimum amount of meta information while being legal (see 5.2.7). Interconvertion between Vcf, Vcf.gz, uBcf, and Bcf was performed using `bcftools` (version 1.9-209-gf9984ee, using htslib 1.9-271-g6738132):

```
bcftools -i ${input_file} -O ${file_type} -o ${file_name}
```

`PLINK` (v1.90p) stores the genetic component in its own binary format (BED) and the sample information in a separate BIM file.

```
plink --bcf ${file.bcf} --out ${file}
```

Stripped Vcf files were imported into the succinct tree representation[58, 57] (`.trees`) using `tskit` (version 0.2.0a4) using a modifed version of the provided `convert_1kg.py` helper script. Vcf/Bcf was converted into valid Djinn format using the base command:

```
djinn -ci ${input_bcf} -o ${output_djn}
```

The flag "-z", "-l" and "-m" are used to indicate if Zstd, LZ4, or a custom range coder-based approach is used for compression. For linkage disequilibrium-based preprocessing (PBWT), the flags "-p" and "-P" are used to indicate if preprocessing should be performed or not, respectively. Compression level for Zstd and Lz4 can be modified using the "-#" flag.

5.2.11 Experimental data sets

Individual chromosome Bcf files were retrieved for each cohort and combined into a single dataset using the Bcftools `concat` command.

- 1000 Genomes phase 3 (hg19)[123]: This dataset comprises variants and small insertions/deletions (indels) for 2,504 individuals of various ancestry.

- Haplotype Reference Consortium (hg19)[125]: This dataset comprises of a mixed cohort of 32,488 whole-genome sequenced individuals. There are >39 million SNPs with an allele count ≥ 5 and no indels. All samples from the 1000 Genomes phase 3 project are included in this dataset.

5.2.12 Simulated data sets

Haplotypes were simulated using msprime[56] version 0.7.0 with both mutation rate and recombination rate per site per 4N generations set to 2×10^{-8} and effective population size (N_e) to 10,000. Region size was fixed at 100 megabases and number of simulated haplotypes ranging from 10 and 10,000,000. Since msprime simulates haplotypes, we concatenated random pairs of haplotypes into diploid genomes. It is computationally prohibitive to store the actual Vcf files for these datasets so we piped the output Vcf from msprime into Djinn and computed the file sizes for Vcf and uBcf internally during run-time.

5.3 Results

Storing and analyzing genetic variant data has garnered considerable interest with proposals in both the variant-centric [27, 31, 34, 71, 30], sample-centric [63] orientations. Unlike previous efforts that mostly focused on developing monolithic applications or novel file formats, we restrict our attention to developing plug-and-play algorithms for integration into the existing ecosystem with minimal disruption. Here we present three algorithms that balance the trade-off between achieving optimal data compression and query speed: (1) maximizing query speed with limited compression potential; (2) balancing query speeds and compressibility; and (3) optimizing compressibility for transmission or long-term archival storage with poor query performance.

Our proposed variant-centric methods are based on an efficient encoding scheme involving the Extended Word-Aligned Hybrid (EWAH)[65] that combines run-length encoding with bit-packed vectors (bitmaps, **Figure 5.3**). This hybrid encoding performs well as most low allele frequency variants encodes into long runs of identical symbols (genotypes) and higher allele frequency variants encodes efficiently into bit-packed vectors (bitmaps) (**Figure 5.4**). On simulated datasets (see Methods), this codec alone achieves moderate gains in compression over Vcf and Bcf while maintaining the ability to be efficiently queried without first decompressing.

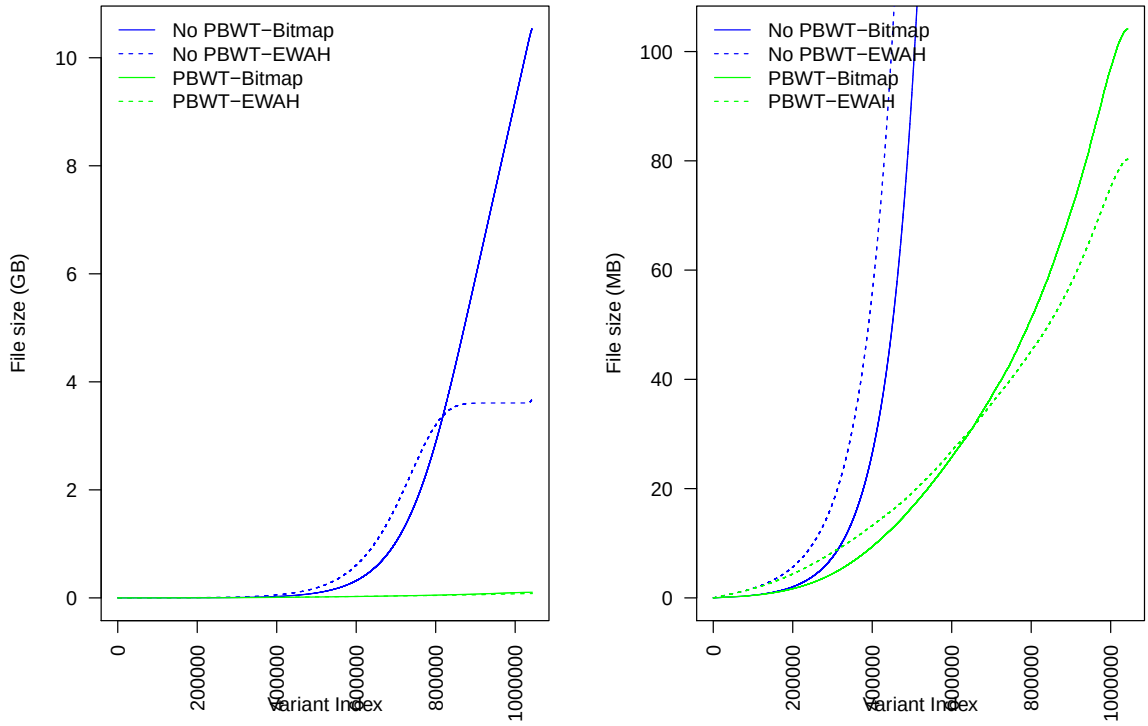


Figure 5.4 Size distribution for the EWAH components (EWAH struct and bitmap) for PBWT-permuted and unpermuted data. 1,043,341 biallelic sites for 262,144 samples simulated using `msprime` was either PBWT-permuted or left unpermuted and the resulting size distribution for the two components of EWAH-encoding are shown. For unpermuted data the bitmap component require 10535.76 MB and the EWAH component 3690.2 MB. In contrast, for PBWT-permuted data the corresponding sizes are 104.1 MB and 80.3 MB, respectively. This correspond to a 45-fold saving for the EWAH structure component and a 101-fold saving for the bitmap component. Overall, the uncompressed data is 14226 MB whereas the PBWT-permuted data is 184.5 MB (77-fold difference). This divide in compressibility keeps growing with larger cohorts. The right panel have the y-axis limited to the bounds of the PBWT-permuted data range. Also note the difference in y-axis scaling on the two panels: GB and MB, respectively.

Additional gains in compressibility can be achieved by applying the linkage-disequilibrium-based PBWT [34] pre-processor at the expense of additional computation. This reversible transformation improves compression by permuting the sample order at a given position to its reverse suffix order. First, in order to generate large datasets, we simulated haplotypes for 10 to 10 million haplotypes over a 100 Mb region, a length equivalent to that of human chromosome 15, and evaluated compression performance. Next, we

pre-processed genotypes with PBWT followed by applying a final statistical context model and compared to similar tools [34, 71, 57] (**Figure 5.5** and see Methods). Motivating this work, both Vcf and Bcf grow rapidly with an increasing number of haplotypes with Vcf exceeding a terabyte of disk storage for a million haplotypes. The compressibility of our proposed context-based method exceeds that of the recently proposed succinct tree structures[57] by up to an order of magnitude even if additionally packed using an efficient external compression engine.

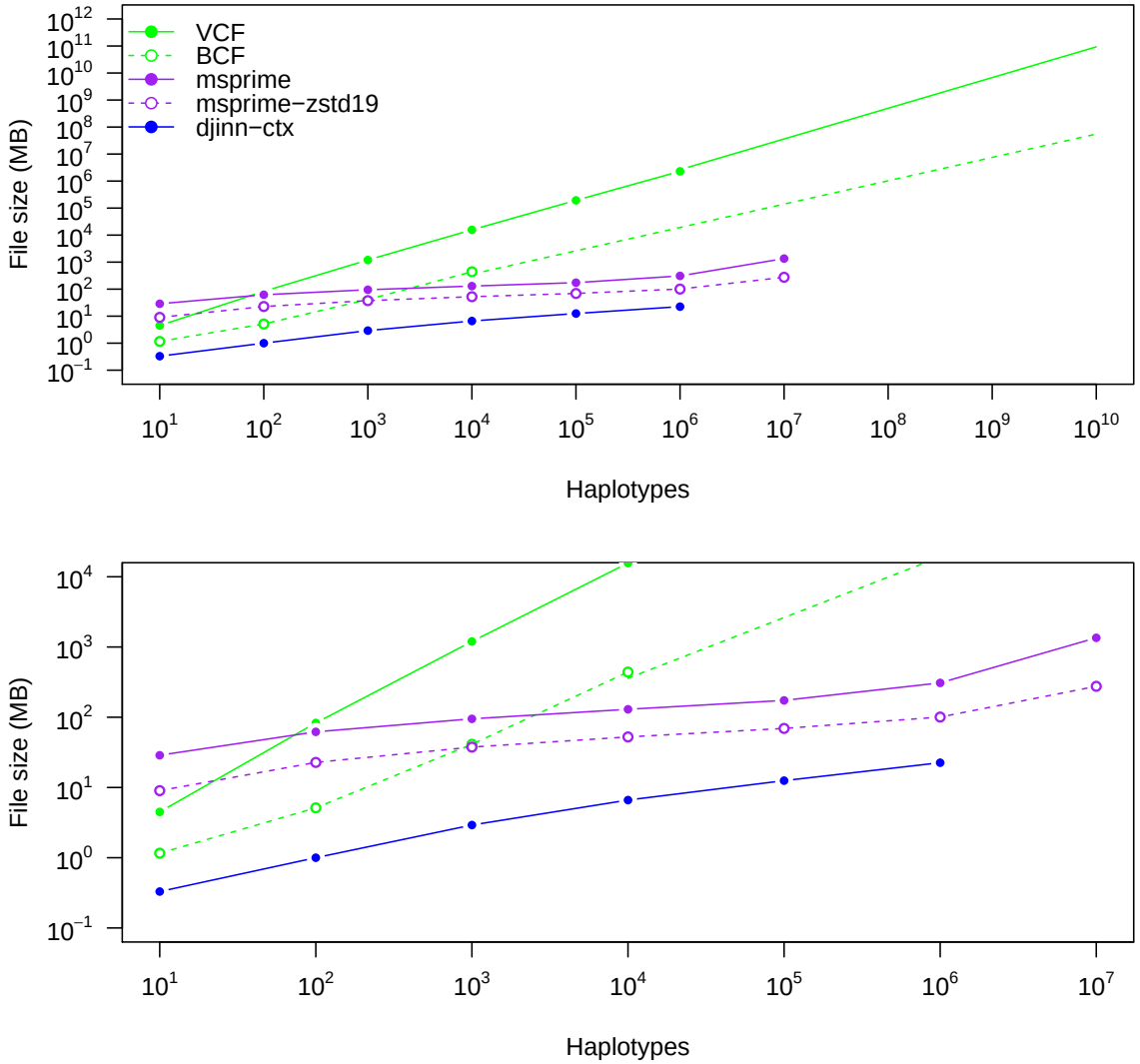


Figure 5.5 Compression performance of simulated data

Simulating haplotypes from an idealized population is a useful way of generating data to investigate the relativistic growth properties of large datasets. Unfortunately, these

datasets have unrealistic haplotype distributions and are misleading when estimating the projected cost of future large-scale datasets. Investigating this, we evaluated compression performance on real data from the 1000 Genomes Project (1KGP3, **Figure 5.6**) and the Haplotype Reference Cosortium (HRC, **Figure 5.7**) datasets. All our proposed compression methods (EWAH-encoding followed by either context modelling, LZ4, or Zstd) display a considerably lower memory footprint compared to Vcf, Bcf, and msprime (tskit) on the 1KGP3 dataset. There is generally an additional 2-fold gain in compression when exchanging the general-purpose compressors with a statistical, and context-specific, compression model. This additional gain in compression comes with a larger overhead in retrieving and decompressing data. As expected from the simulated data (**Figure 5.5**), msprime provides approximately equal compression as Bcf. Notably, the converting procedure into the succinct representation used in msprime is extremely slow even for these small chromosomal datasets. For example, importing chromosome 22 into msprime takes around 44 hours compared to under 2 minutes for djinn representing a 1,320-fold difference. Proportional performance metrics are seen for the considerably larger HRC dataset (**Figure 5.7**). msprime was not included in this analysis as this dataset could not be imported in <48 hours.

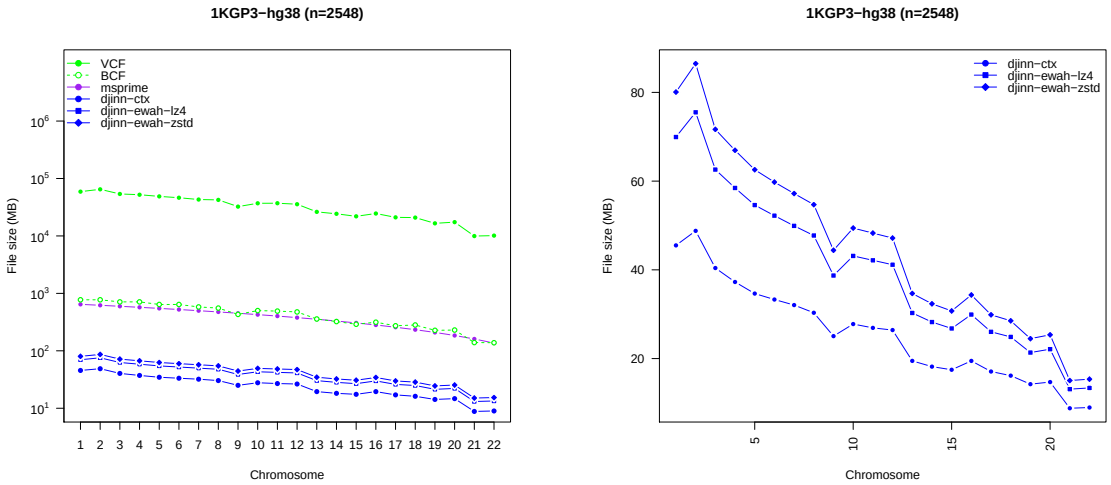


Figure 5.6 Compression performance for Djinn on 1KGP3 datasets. Sequence variant data called from 1000 Genomes Project data for 2,548 diploid individuals whole-genome sequenced to >30-fold coverage. Final archive size per chromosome is shown for different compression methods: Vcf, Bcf, msprime, and different Djinn algorithms. Djinn-ctx: context modelling on EWAH objects; Djinn-EWAH-LZ4: direct compression of EWAH objects using the general-purpose compressor LZ4; Djinn-EWAH-ZSTD: direction compression of EWAH objects using the general-purpose compressor Zstd. A zoomed in figure (right panel) is shown for the different Djinn models to illustrate the significant file-size differences between the context model and the general-purpose compressors.

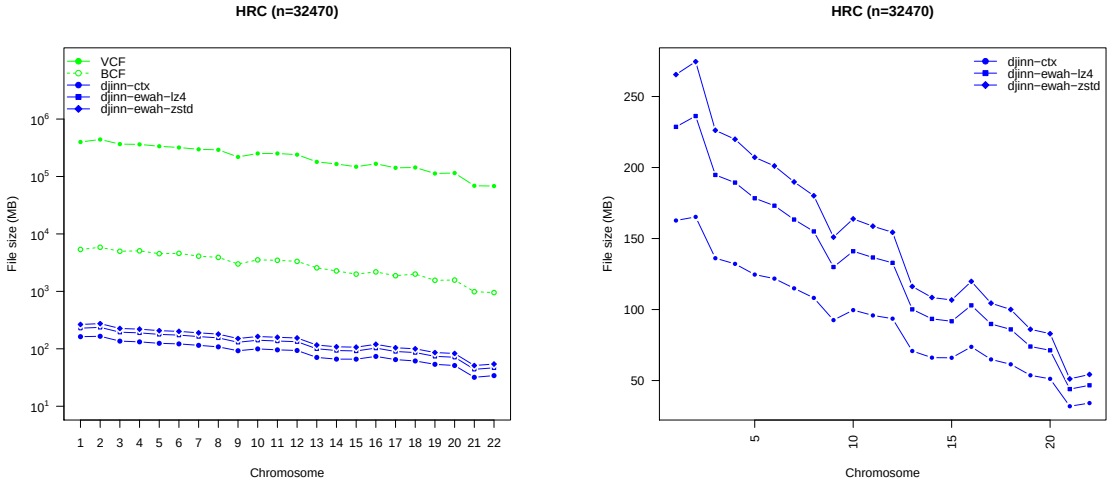


Figure 5.7 Compression performance for Djinn on HRC datasets. Sequence variant data for 32,470 whole-genome sequenced individuals from a variety of datasets where most individuals have European ancestry. Final archive size per chromosome is shown for different compression methods: Vcf, Bcf, and different Djinn algorithms. `msprime/tsinfer` was excluded from this analysis as the import procedure for any chromosome failed to complete in 24 hours on 28 CPU cores. Djinn-ctx: context modelling on EWAH objects; Djinn-EWAH-LZ4: direct compression of EWAH objects using the general-purpose compressor LZ4; Djinn-EWAH-ZSTD: direction compression of EWAH objects using the general-purpose compressor Zstd. A zoomed in figure (right panel) is shown for the different Djinn models to illustrate the significant file-size differences between the context model and the general-purpose compressors.

5.4 Discussion

With the advent of population-scaled datasets[11, 114, 116, 21, 41, 127] and emerging diagnostic applications in the clinical setting there is a pressing need to efficiently store and query the genetic component of a sequence variant dataset. Computational performance will be of critical importance in the clinical setting with rapid-reporting approaches in for example the intensive care setting, early diagnosis of complex disease, or for influencing medical decision making in cancer care. Recognizing this impending scaling challenge, my motivation was to explore algorithms for simultaneously reducing storage cost and improving analysis time while balancing the different trade-offs associated with a particular approach. I proposed a unified compression approach of genotypes using a variation of the succinct extended word-aligned hybrid (EWAH)[65] encoding approach that combines run-length encoding with uncompressed bitmaps. I showed that

additional reductions in the memory footprint can be achieved by applying the reversible pre-processing function PBWT[34]. Further extensions to the PBWT-based approach such as allowing an arbitrary large alphabet and a heuristic to avoid storing the sample permutation order enabled additional savings. Notably, all our proposed approaches for storing genetic matrices are embarrassingly parallelizable such that most queries can be executed in parallel on a machine and/or across multiple compute nodes in a compute cluster or cloud.

Querying EWAH-encoded genotypes directly can be achieved with an extension of the gtOcc approach described in Chapter 4. Additionally, I demonstrate that it is possible to compute genetic similarity matrices ($X^T X$) using the algortihms described in Chapter 2 and Chapter 3. Together, these algorithms ensure correctness while reducing computational cost and time. Owing to their efficiency, I expect that these algorithms will enhance both existing and novel methods exploring population-scaled datasets.

5.5 Acknowledgements

I am grateful to members of the The Global Alliance for Genomics and Health (GA4GH) Future of Vcf and File Formats working groups for fruitful discussions. I am grateful to members of the open-source community for helping improving the implementation.

Chapter 6

Conclusions

The future of genomics will undoubtedly involve considerable volumes of data and the transitioning of genome sequencing into a routine clinical tool will demand considerable improvements in performance over current standard methods. This transformative bench-to-bedside transition will usher in a new era of high-performance computing and algorithms. In this work I addressed these impending scaling requirements in a variety of settings. Next-generation algorithms in this space will require highly performant basic operations including counting the number of set bits in a large number of machine words. Addressing this, I developed a variation of the classic population count operation called the positional population count by building on previous work involving carry save-adder networks. To enable its application on any target machine, I developed a single header library including these operators that selects the optimal method during run-time. Using these basic operators, I described a variety of applications including computing linkage-disequilibrium, genetic similarity matrices, slicing out populations from large genetic datasets, and to efficiently computing summary statistics from compressed genotypes. Given that the performance of these functions operates at close to hardware limits, I envision that these functions, or variations thereof, will continue to be useful in this space for the foreseeable future. Upcoming hardware architectures will provide hardware instructions (e.g. `VPOPCNT`) that will provide similar performance to the AVX512-based population count used in this work (personal communication). Irrespective of this impending performance boost, the library used in this work could be useful for a long period of time considering the general latency period between releasing a new hardware architecture and seeing large community adoption. For example, many bioinformatics tools are still principally developed to target the SSE2 or SSE4 instruction set architectures despite being released in 2000 and 2007, respectively.

In two chapters, I described how it is possible to use both, or either, row-centric and column-centric data storage approaches to efficiently store and query large sequence variant datasets with different trade-offs. Recently, column stores have enjoyed renewed interest in big data domains because of its superior capabilities of querying large number of rows efficiently. The future application of either memory layout in genomics will be application-specific but will most likely be primarily column-centric with the exception of genotypes/haplotypes that will remain as succinctly represented row-centric records. It is more natural to operate on all samples at a given locus (site-centric) and discard target samples that are not of interest. This orientation is also superior in compression performance and in extension its query performance. General storage of non-genotypic sequence variant data will benefit greatly from being completely pivoted into a column store as field with different primitive types will be packed together for greater compression. In addition, many queries involve a few, or even a single, target column out of a dozen or more resulting in faster querying times.

Future sequence and sequence variant formats must include high-performance techniques in column stores that are already used in the broader scientific community such as vectorized processing, late materialization, column-specific compression, efficient join implementations, operating directly on compressed data, and exploiting predicate push-down tricks including segmental statistics, dictionaries, and Bloom filters. Column-stores have additional innate benefits in this space compared to row-centric storage systems. By storing columnar data in non-overlapping partitions of some size it is always possible to efficiently seek and retrieve target data in an embarrassingly parallel fashion. Depending on the query, this can enable the application of trivial data level parallelism paradigms to scale to any number of cores and compute nodes.

In most chapters of this work, I described simple, yet efficient, succinct representations of data and developed methods to directly query compressed representation in time proportional to its compressed size. Given the considerable sparsity of genetic variant datasets, I envision that approaches like this will eventually represent a general paradigm for the efficient computation of genetic data. Lastly, future formats will require efficient indexing methods to support the considerable volumes of data that will be generated. Fortunately, efficient data indexing is a mature topic in the computer science community but has enjoyed limited attention in the bioinformatics space.

I envision a paradigm shift in the near future in how genomics data is being processed and handled and hope that my work will contribute towards this effort in some part.

References

[1] Abadi, D. J., Madden, S. R., and Ferreira, M. (2006). Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, Chicago, IL, USA.

[2] Abadi, D. J., Myers, D. S., DeWitt, D. J., and Madden, S. R. (2007). Materialization strategies in a column-oriented dbms. In *ICDE*, pages 466–475, Istanbul, Turkey.

[3] Abdellaoui, A., Hottenga, J.-J., de Knijff, P., Nivard, M. G., Xiao, X., Scheet, P., Brooks, A., Ehli, E. A., Hu, Y., Davies, G. E., Hudziak, J. J., Sullivan, P. F., van Beijsterveldt, T., Willemsen, G., de Geus, E. J., Penninx, B. W. J. H., and Boomsma, D. I. (2013). Population structure, migration, and diversifying selection in the Netherlands. *European Journal of Human Genetics*, 21(11):1277–1285.

[4] Adams, D. R. and Eng, C. M. (2018). Next-Generation Sequencing to Diagnose Suspected Genetic Disorders. *New England Journal of Medicine*, 379(14):1353–1362.

[5] Adams, Mark D.et al. (2000). The Genome Sequence of Drosophila melanogaster. *Science*, 287(5461):2185–2195.

[6] Alexander, D. H., Novembre, J., and Lange, K. (2009). Fast model-based estimation of ancestry in unrelated individuals. *Genome Research*, 19(9):1655–64.

[7] Balasubramanian, S., Klenerman, D., and Bentley, D. (U.S. Patent US6787308B2, 2001-01-30, Solexa Ltd Great Britain, https://patents.google.com/patent/US6787308B2/en). Arrayed biomolecules and their use in sequencing.

[8] Balci, T., Hartley, T., Xi, Y., Dyment, D., Beaulieu, C., Bernier, F., Dupuis, L., Horvath, G., Mendoza-Londono, R., Prasad, C., Richer, J., Yang, X.-R., Armour, C., Bareke, E., Fernandez, B., McMillan, H., Lamont, R., Majewski, J., Parboosingh, J., Prasad, A., Rupar, C., Schwartzentruber, J., Smith, A., Tétreault, M., Innes, A., Boycott, K., Innes, A. M., and Boycott, K. M. (2017). Debunking Occam's razor: Diagnosing multiple genetic diseases in families by whole-exome sequencing. *Clinical Genetics*, 92(3):281–289.

[9] Beadle, G. W. (1932). A Possible Influence of the Spindle Fibre on Crossing-Over in Drosophila. *Proceedings of the National Academy of Sciences of the United States of America*, 18(2):160–5.

[10] Bentley, David R. et al. (2008). Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59.

[11] Birney, E., Vamathevan, J., and Goodhand, P. (2017). Genomics in healthcare: GA4GH looks to 2022. *bioRxiv*, page 203554.

[12] Bonfield, J. K., McCarthy, S. A., and Durbin, R. (2019). Crumble: Reference free lossy compression of sequence quality values. *Bioinformatics*.

[13] Bulik-Sullivan, B. K., Loh, P.-R., Finucane, H. K., Ripke, S., Yang, J., Patterson, N., Daly, M. J., Price, A. L., Neale, B. M., and Neale, B. M. (2015). LD Score regression distinguishes confounding from polygenicity in genome-wide association studies. *Nature Genetics*, 47(3):291–295.

[14] Bycroft, C., Freeman, C., Petkova, D., Band, G., Elliott, L. T., Sharp, K., Motyer, A., Vukcevic, D., Delaneau, O., O'Connell, J., Cortes, A., Welsh, S., Young, A., Effingham, M., McVean, G., Leslie, S., Allen, N., Donnelly, P., and Marchini, J. (2018). The UK Biobank resource with deep phenotyping and genomic data. *Nature*, 562(7726):203–209.

[15] Cantor, C. R. (1990). Orchestrating the Human Genome Project. *Science (New York, N.Y.)*, 248(4951):49–51.

[16] Chambi, S., Lemire, D., Kaser, O., and Godin, R. (2014). Better bitmap performance with roaring bitmaps. *arXiv*.

[17] Chandak, S., Tatwawadi, K., Ochoa, I., Hernaez, M., and Weissman, T. (2018). SPRING: a next-generation compressor for FASTQ data. *Bioinformatics*.

[18] Church, G. M. and Gilbert, W. (1984). Genomic sequencing. *Proceedings of the National Academy of Sciences of the United States of America*, 81(7):1991–5.

[19] Cockerham, C. C. (1969). Variance of gene frequencies. *Evolution*, 23(1):72–84.

[20] Cockerham, C. C. (1973). Analyses of gene frequencies. *Genetics*, 74(4):679–700.

[21] Collins, F. S. and Varmus, H. (2015). A New Initiative on Precision Medicine. *New England Journal of Medicine*, 372(9):793–795.

[22] Conomos, M., Reiner, A., Weir, B., and Thornton, T. (2016). Model-free Estimation of Recent Genetic Relatedness. *The American Journal of Human Genetics*, 98(1):127–148.

[23] Consortium, T. I. C. G. (2010). International network of cancer genome projects. *Nature*, 464(7291):993–998.

[24] Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. (2016). Binarized Neural Networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv*.

[25] Dai, B., Guo, R., Kumar, S., He, N., and Song, L. (2017). Stochastic generative hashing. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 913–922, International Convention Centre, Sydney, Australia. PMLR.

[26] Danecek, P., Auton, A., Abecasis, G., Albers, C. A., Banks, E., DePristo, M. A., Handsaker, R. E., Lunter, G., Marth, G. T., Sherry, S. T., McVean, G., and Durbin, R. (2011). The variant call format and VCFtools. *Bioinformatics*, 27(15):2156–2158.

[27] Danek, A. and Deorowicz, S. (2018). GTC: how to maintain huge genotype collections in a compressed form. *Bioinformatics*, 34(11):1834–1840.

[28] de Lange, K. M., Moutsianas, L., Lee, J. C., Lamb, C. A., Luo, Y., Kennedy, N. A., Jostins, L., Rice, D. L., Gutierrez-Achury, J., Ji, S.-G., Heap, G., Nimmo, E. R., Edwards, C., Henderson, P., Mowat, C., Sanderson, J., Satsangi, J., Simmons, A., Wilson, D. C., Tremelling, M., Hart, A., Mathew, C. G., Newman, W. G., Parkes, M., Lees, C. W., Uhlig, H., Hawkey, C., Prescott, N. J., Ahmad, T., Mansfield, J. C., Anderson, C. A., and Barrett, J. C. (2017). Genome-wide association study implicates immune activation of multiple integrin genes in inflammatory bowel disease. *Nature Genetics*, 49(2):256–261.

[29] Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA.

[30] Deorowicz, S. and Danek, A. (2018). GTShark: Genotype compression in large project. *bioRxiv*, page 494104.

[31] Deorowicz, S., Danek, A., and Grabowski, S. (2013). Genome compression: a novel approach for large collections. *Bioinformatics*, 29(20):2572–2578.

[32] Devlin, B. and Risch, N. (1995). A Comparison of Linkage Disequilibrium Measures for Fine-Scale Mapping. *Genomics*, 29(2):311–322.

[33] Duncan, Laramie et al. (2017). Significant Locus and Metabolic Genetic Correlations Revealed in Genome-Wide Association Study of Anorexia Nervosa. *American Journal of Psychiatry*, 174(9):850–858.

[34] Durbin, R. (2014). Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT). *Bioinformatics*, 30(9):1266–1272.

[35] Edwards, A. and Caskey, C. T. (1991). Closure strategies for random DNA sequencing. *Methods*, 3(1):41–47.

[36] Farwell, K. D., Shahmirzadi, L., El-Khechen, D., Powis, Z., Chao, E. C., Tippin Davis, B., Baxter, R. M., Zeng, W., Mroske, C., Parra, M. C., Gandomi, S. K., Lu, I., Li, X., Lu, H., Lu, H.-M., Salvador, D., Ruble, D., Lao, M., Fischbach, S., Wen, J., Lee, S., Elliott, A., Dunlop, C. L., and Tang, S. (2015). Enhanced utility of family-centered diagnostic exome sequencing with inheritance model–based analysis: results from 500 unselected families with undiagnosed genetic conditions. *Genetics in Medicine*, 17(7):578–586.

[37] Ferragina, P. and Manzini, G. (2000). Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 390–, Washington, DC, USA. IEEE Computer Society.

[38] Fosso Wamba, S., Akter, S., Edwards, A., Chopin, G., and Gnanzou, D. (2015). How 'big data' can make big impact: Findings from a systematic review and a longitudinal case study. *International Journal of Production Economics*, 165:234–246.

[39] Gao, Y. and Chen, W.-M. (2017). Family Relationship Inference Using Knights Landing Platform. In *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, pages 27–30. IEEE.

[40] Gaunt, T. R., Rodríguez, S., and Day, I. N. (2007). Cubic exact solutions for the estimation of pairwise haplotype frequencies: implications for linkage disequilibrium analyses and a web tool 'CubeX'. *BMC Bioinformatics*, 8(1):428.

[41] Gaziano, J. M., Concato, J., Brophy, M., Fiore, L., Pyarajan, S., Breeling, J., Whitbourne, S., Deen, J., Shannon, C., Humphries, D., Guarino, P., Aslan, M., Anderson, D., LaFleur, R., Hammond, T., Schaa, K., Moser, J., Huang, G., Muralidhar, S., Przygodzki, R., and O'Leary, T. J. (2016). Million Veteran Program: A mega-biobank to study genetic influences on health and disease. *Journal of Clinical Epidemiology*, 70:214–223.

[42] Geiringer, H. (1944). On the Probability Theory of Linkage in Mendelian Heredity. *The Annals of Mathematical Statistics*, 15(1):25–57.

[43] Goodwin, S., McPherson, J. D., and McCombie, W. R. (2016). Coming of age: ten years of next-generation sequencing technologies. *Nature Reviews Genetics*, 17(6):333–351.

[44] Haque, I. S., Pande, V. S., and Walters, W. P. (2011). Anatomy of High-Performance 2D Similarity Calculations. *Journal of Chemical Information and Modeling*, 51(9):2345–2351.

[45] Hedrick, P. W. (1987). Gametic Disequilibrium Measures: Proceed With Caution. *Genetics*, 117(2).

[46] Hill, W. G. (1974). Estimation of linkage disequilibrium in randomly mating populations. *Heredity*, 33(2):229–239.

[47] Hill, W. G. and Robertson, A. (1968). Linkage disequilibrium in finite populations. *Theoretical and Applied Genetics*, 38(6):226–231.

[48] Hsi-Yang Fritz, M., Leinonen, R., Cochrane, G., and Birney, E. (2011). Efficient storage of high throughput dna sequencing data using reference-based compression. *Genome Research*, 21(5):734–740.

[49] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. (2017). Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898.

[50] Hyde, C. L., Nagle, M. W., Tian, C., Chen, X., Paciga, S. A., Wendland, J. R., Tung, J. Y., Hinds, D. A., Perlis, R. H., and Winslow, A. R. (2016). Identification of 15 genetic loci associated with risk of major depression in individuals of European descent. *Nature Genetics*, 48(9):1031–1036.

[51] International Human Genome Sequencing Consortium (2001). Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921.

[52] Investigators, The All of Us Research Program (2019). The "All of Us" Research Program. *New England Journal of Medicine*, 381(7):668–676.

[53] Jain, M., Koren, S., Miga, K. H., Quick, J., Rand, A. C., Sasani, T. A., Tyson, J. R., Beggs, A. D., Dilthey, A. T., Fiddes, I. T., Malla, S., Marriott, H., Nieto, T., O'Grady, J., Olsen, H. E., Pedersen, B. S., Rhie, A., Richardson, H., Quinlan, A. R., Snutch, T. P., Tee, L., Paten, B., Phillippy, A. M., Simpson, J. T., Loman, N. J., and Loose, M. (2018). Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature Biotechnology*, 36(4):338–345.

[54] Jansen, P. R., Watanabe, K., Stringer, S., Skene, N., Bryois, J., Hammerschlag, A. R., de Leeuw, C. A., Benjamins, J. S., Muñoz-Manchado, A. B., Nagel, M., Savage, J. E., Tiemeier, H., White, T., Tung, J. Y., Hinds, D. A., Vacic, V., Wang, X., Sullivan, P. F., van der Sluis, S., Polderman, T. J. C., Smit, A. B., Hjerling-Leffler, J., Van Someren, E. J. W., and Posthuma, D. (2019). Genome-wide analysis of insomnia in 1,331,010 individuals identifies new risk loci and functional pathways. *Nature Genetics*, 51(3):394–403.

[55] Karczewski, Konrad J et al. (2019). Variation across 141,456 human exomes and genomes reveals the spectrum of loss-of-function intolerance across human protein-coding genes. *bioRxiv*, page 531210.

[56] Kelleher, J., Etheridge, A. M., and McVean, G. (2016). Efficient Coalescent Simulation and Genealogical Analysis for Large Sample Sizes. *PLOS Computational Biology*, 12(5):e1004842.

[57] Kelleher, J., Thornton, K., Ashander, J., and Ralph, P. (2018a). Efficient pedigree recording for fast population genetics simulation. *bioRxiv*, page 248500.

[58] Kelleher, J., Wong, Y., Albers, P., Wohns, A. W., and McVean, G. (2018b). Inferring the ancestry of everyone. *bioRxiv*, page 458067.

[59] Klein, R. J., Zeiss, C., Chew, E. Y., Tsai, J.-Y., Sackler, R. S., Haynes, C., Henning, A. K., SanGiovanni, J. P., Mane, S. M., Mayne, S. T., Bracken, M. B., Ferris, F. L., Ott, J., Barnstable, C., and Hoh, J. (2005). Complement Factor H Polymorphism in Age-Related Macular Degeneration. *Science*, 308(5720):385–389.

[60] Knuth, D. E. (2011). *The Art of Computer Programming: volume 4A: combinatorial algorithms, part 1.* Addison-Wesley, Boston, Massachusetts.

[61] Langley, S. A., Miga, K., Karpen, G., and Langley, C. H. (2018). Haplotypes spanning centromeric regions reveal persistence of large blocks of archaic DNA. *bioRxiv*, page 351569.

[62] Langmead, B. and Nellore, A. (2018). Cloud computing for genomic data analysis and collaboration. *Nature Reviews Genetics*, 19(4):208–219.

[63] Layer, R. M., Kindlon, N., Karczewski, K. J., Quinlan, A. R., and Quinlan, A. R. (2016). Efficient genotype compression and analysis of large genetic-variation data sets. *Nature Methods*, 13(1):63–65.

[64] Lee, James J et al. (2018). Gene discovery and polygenic prediction from a genome-wide association study of educational attainment in 1.1 million individuals. *Nature Genetics*, 50(8):1112–1121.

[65] Lemire, D., Kaser, O., and Aouiche, K. (2009). Sorting improves word-aligned bitmap indexes. *arXiv*.

[66] Lemire, D., Kaser, O., Kurz, N., Deri, L., O'Hara, C., Saint-Jacques, F., and Ssi-Yan-Kai, G. (2017). Roaring bitmaps: Implementation of an optimized software library. *arXiv*.

[67] Lemire, D., Ssi-Yan-Kai, G., and Kaser, O. (2016). Consistently faster and smaller compressed bitmaps with Roaring. *Software: Practice and Experience*, 46(11):1547–1569.

[68] Levy, S. E. and Myers, R. M. (2016). Advancements in Next-Generation Sequencing. *Annual Review of Genomics and Human Genetics*, 17(1):95–115.

[69] Lewontin, R. C. (1964). The interaction of selection and linkage. *Genetics*, 49(1).

[70] Lewontin, R. C. and Kojima, K.-i. (1960). The evolutionary dynamics of complex polymorphisms. *Evolution*, 14(4):458–472.

[71] Li, H. (2016). BGT: efficient and flexible genotype query across many samples. *Bioinformatics*, 32(4):590–592.

[72] Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., and Durbin, R. (2009). The Sequence Alignment/Map format and SAMtools. *Bioinformatics*.

[73] Li, Z., Chen, J., Yu, H., He, L., Xu, Y., Zhang, D., Yi, Q., Li, C., Li, X., Shen, J., Song, Z., Ji, W., Wang, M., Zhou, J., Chen, B., Liu, Y., Wang, J., Wang, P., Yang, P., Wang, Q., Feng, G., Liu, B., Sun, W., Li, B., He, G., Li, W., Wan, C., Xu, Q., Li, W., Wen, Z., Liu, K., Huang, F., Ji, J., Ripke, S., Yue, W., Sullivan, P. F., O'Donovan, M. C., and Shi, Y. (2017). Genome-wide association analysis identifies 30 new susceptibility loci for schizophrenia. *Nature Genetics*, 49(11):1576–1583.

[74] Lin, M. F., Bai, X., Salerno, W. J., and Reid, J. G. (2019). Sparse project vcf: efficient encoding of population genotype matrices. *bioRxiv*.

[75] Lippert, C., Listgarten, J., Davidson, R. I., Baxter, J., Poon, H., Kadie, C. M., and Heckerman, D. (2013). An exhaustive epistatic snp association analysis on expanded wellcome trust data. *Scientific reports*, 3:1099.

[76] Lippert, C., Listgarten, J., Liu, Y., Kadie, C. M., Davidson, R. I., and Heckerman, D. (2011). FaST linear mixed models for genome-wide association studies. *Nature Methods*, 8(10):833–835.

[77] Liu, Mengzhen et al. (2019). Association studies of up to 1.2 million individuals yield new insights into the genetic etiology of tobacco and alcohol use. *Nature Genetics*, 51(2):237–244.

[78] Loh, P.-R., Baym, M., and Berger, B. (2012). Compressive genomics. *Nature Biotechnology*, 30(7):627–630.

[79] Loh, P.-R., Tucker, G., Bulik-Sullivan, B. K., Vilhjálmsson, B. J., Finucane, H. K., Salem, R. M., Chasman, D. I., Ridker, P. M., Neale, B. M., Berger, B., Patterson, N., and Price, A. L. (2015). Efficient Bayesian mixed-model analysis increases association power in large cohorts. *Nature Genetics*, 47(3):284–290.

[80] MacArthur, D. G., Manolio, T. A., Dimmock, D. P., Rehm, H. L., Shendure, J., Abecasis, G. R., Adams, D. R., Altman, R. B., Antonarakis, S. E., Ashley, E. A., Barrett, J. C., Biesecker, L. G., Conrad, D. F., Cooper, G. M., Cox, N. J., Daly, M. J., Gerstein, M. B., Goldstein, D. B., Hirschhorn, J. N., Leal, S. M., Pennacchio, L. A., Stamatoyannopoulos, J. A., Sunyaev, S. R., Valle, D., Voight, B. F., Winckler, W., and Gunter, C. (2014). Guidelines for investigating causality of sequence variants in human disease. *Nature*, 508(7497):469–476.

[81] MacArthur, J., Bowler, E., Cerezo, M., Gil, L., Hall, P., Hastings, E., Junkins, H., McMahon, A., Milano, A., Morales, J., Pendlington, Z. M., Welter, D., Burdett, T., Hindorff, L., Flicek, P., Cunningham, F., and Parkinson, H. (2017). The new NHGRI-EBI Catalog of published genome-wide association studies (GWAS Catalog). *Nucleic Acids Research*, 45(D1):D896–D901.

[82] Mäkinen, V. and Norri, T. (2019). Applying the Positional Burrows–Wheeler Transform to all-pairs Hamming distance. *Information Processing Letters*, 146:17–19.

[83] Manichaikul, A., Mychaleckyj, J. C., Rich, S. S., Daly, K., Sale, M., and Chen, W.-M. (2010). Robust relationship inference in genome-wide association studies. *Bioinformatics*, 26(22):2867–2873.

[84] Martín, G. M. (1979). Range encoding: an algorithm for removing redundancy from a digitised message.

[85] McVean, G. (2009). A Genealogical Interpretation of Principal Components Analysis. *PLoS Genetics*, 5(10):e1000686.

[86] Milne, Roger L et al. (2017). Identification of ten variants associated with risk of estrogen-receptor-negative breast cancer. *Nature Genetics*, 49(12):1767–1778.

[87] Mittag, F., Römer, M., and Zell, A. (2015). Influence of feature encoding and choice of classifier on disease risk prediction in genome-wide association studies. *PloS one*, 10(8):e0135832.

[88] Muła, W., Kurz, N., and Lemire, D. (2016). Faster population counts using avx2 instructions. *arXiv*.

[89] Myers, E. W., Sutton, G. G., Delcher, A. L., Dew, I. M., Fasulo, D. P., Flanigan, M. J., Kravitz, S. A., Mobarry, C. M., Reinert, K. H., Remington, K. A., Anson, E. L., Bolanos, R. A., Chou, H. H., Jordan, C. M., Halpern, A. L., Lonardi, S., Beasley, E. M., Brandon, R. C., Chen, L., Dunn, P. J., Lai, Z., Liang, Y., Nusskern, D. R., Zhan, M., Zhang, Q., Zheng, X., Rubin, G. M., Adams, M. D., and Venter, J. C. (2000). A whole-genome assembly of Drosophila. *Science (New York, N.Y.)*, 287(5461):2196–204.

[90] Network, T. C. G. A. R., Weinstein, J. N., Collisson, E. A., Mills, G. B., Shaw, K. R. M., Ozenberger, B. A., Ellrott, K., Shmulevich, I., Sander, C., and Stuart, J. M. (2013). The Cancer Genome Atlas Pan-Cancer analysis project. *Nature Genetics 2013 45:10*.

[91] Nickalls, R. W. D. (1993). A New Approach to Solving the Cubic: Cardan's Solution Revealed. *The Mathematical Gazette*, 77(480):354.

[92] Payne, A., Holmes, N., Rakyan, V., and Loose, M. (2018). Whale watching with BulkVis: A graphical viewer for Oxford Nanopore bulk fast5 files. *bioRxiv*, page 312256.

[93] Posey, J. E., Harel, T., Liu, P., Rosenfeld, J. A., James, R. A., Coban Akdemir, Z. H., Walkiewicz, M., Bi, W., Xiao, R., Ding, Y., Xia, F., Beaudet, A. L., Muzny, D. M., Gibbs, R. A., Boerwinkle, E., Eng, C. M., Sutton, V. R., Shaw, C. A., Plon, S. E., Yang, Y., and Lupski, J. R. (2017). Resolution of Disease Phenotypes Resulting from Multilocus Genomic Variation. *New England Journal of Medicine*, 376(1):21–31.

[94] Pritchard, J. K., Stephens, M., and Donnelly, P. (2000). Inference of population structure using multilocus genotype data. *Genetics*, 155(2):945–959.

[95] Purcell, S., Neale, B., Todd-Brown, K., Thomas, L., Ferreira, M. A., Bender, D., Maller, J., Sklar, P., de Bakker, P. I., Daly, M. J., and Sham, P. C. (2007). PLINK: A Tool Set for Whole-Genome Association and Population-Based Linkage Analyses. *The American Journal of Human Genetics*, 81(3):559–575.

[96] Quick, C., Fuchsberger, C., Taliun, D., Abecasis, G., Boehnke, M., and Kang, H. M. (2019). emeraLD: rapid linkage disequilibrium estimation with massive datasets. *Bioinformatics*, 35(1):164–166.

[97] Rang, F. J., Kloosterman, W. P., and de Ridder, J. (2018). From squiggle to basepair: computational approaches for improving nanopore sequencing read accuracy. *Genome Biology*, 19(1):90.

[98] Retterer, K., Juusola, J., Cho, M. T., Vitazka, P., Millan, F., Gibellini, F., Vertino-Bell, A., Smaoui, N., Neidich, J., Monaghan, K. G., McKnight, D., Bai, R., Suchy, S., Friedman, B., Tahiliani, J., Pineda-Alvarez, D., Richard, G., Brandt, T., Haverfield, E., Chung, W. K., and Bale, S. (2016). Clinical application of whole-exome sequencing across clinical indications. *Genetics in Medicine*, 18(7):696–704.

[99] Rhoads, A. and Au, K. F. (2015). PacBio Sequencing and Its Applications. *Genomics, Proteomics & Bioinformatics*, 13(5):278–289.

[100] Rissanen, J. J. (1976). Generalized Kraft Inequality and Arithmetic Coding. *IBM Journal of Research and Development*, 20(3):198–203.

[101] Roberts, L. (1990). The genetic map is back on track after delays. *Science (New York, N.Y.)*, 248(4957):805.

[102] Rubin, G M et al. (2000). Comparative genomics of the eukaryotes. *Science (New York, N.Y.)*, 287(5461):2204–15.

[103] Sanger, F., Coulson, A., Hong, G., Hill, D., and Petersen, G. (1982). Nucleotide sequence of bacteriophage λ DNA. *Journal of Molecular Biology*, 162(4):729–773.

[104] Sanger, F., Nicklen, S., and Coulson, A. R. (1977). DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences of the United States of America*, 74(12):5463–7.

[105] Sanyal, A., Kusner, M., Gascon, A., and Kanade, V. (2018). TAPAS: Tricks to Accelerate (encrypted) Prediction As a Service. In *Proceedings of the 35th International Conference on Machine Learning*, pages 4497–4506.

[106] Saunders, G., Baudis, M., Becker, R., Beltran, S., Béroud, C., Birney, E., Brooksbank, C., Brunak, S., Van den Bulcke, M., Drysdale, R., Capella-Gutierrez, S., Flicek, P., Florindi, F., Goodhand, P., Gut, I., Heringa, J., Holub, P., Hooyberghs, J., Juty, N., Keane, T. M., Korbel, J. O., Lappalainen, I., Leskosek, B., Matthijs, G., Mayrhofer, M. T., Metspalu, A., Navarro, A., Newhouse, S., Nyrönen, T., Page, A., Persson, B., Palotie, A., Parkinson, H., Rambla, J., Salgado, D., Steinfelder, E., Swertz, M. A., Valencia, A., Varma, S., Blomberg, N., and Scollen, S. (2019). Leveraging European infrastructures to access 1 million human genomes by 2022. *Nature Reviews Genetics*, pages 1–9.

[107] Schaid, D. J., Chen, W., and Larson, N. B. (2018). From genome-wide associations to candidate causal variants by statistical fine-mapping. *Nature Reviews Genetics*, 19(8):491–504.

[108] Seeburg, P. H., Shine, J., Martial, J. A., Ullrich, A., Goodman, H. M., and Baxter, J. D. (1977). Nucleotide sequence of a human gene coding for a polypeptide hormone. *Transactions of the Association of American Physicians*, 90:109–16.

[109] Segura, V., Vilhjálmsson, B. J., Platt, A., Korte, A., Seren, Ü., Long, Q., and Nordborg, M. (2012). An efficient multi-locus mixed-model approach for genome-wide association studies in structured populations. *Nature Genetics*, 44(7):825–830.

[110] Shamseldin, H. E., Maddirevula, S., Faqeih, E., Ibrahim, N., Hashem, M., Shaheen, R., and Alkuraya, F. S. (2017). Increasing the sensitivity of clinical exome sequencing through improved filtration strategy. *Genetics in Medicine*, 19(5):593–598.

[111] Sinsheimer, R. L. (1989). The Santa Cruz Workshop—May 1985. *Genomics*, 5(4):954–956.

[112] Slatkin, M. (2008). Linkage disequilibrium — understanding the evolutionary past and mapping the medical future. *Nature Reviews Genetics*, 9(6):477–485.

[113] Slatkin, M. and Excoffier, L. (1996). Testing for linkage disequilibrium in genotypic data using the Expectation-Maximization algorithm. *Heredity*, 76(4):377–383.

[114] Stark, Z., Dolman, L., Manolio, T. A., Ozenberger, B., Hill, S. L., Caulfied, M. J., Levy, Y., Glazer, D., Wilson, J., Lawler, M., Boughtwood, T., Braithwaite, J., Goodhand, P., Birney, E., and North, K. N. (2019). Integrating Genomics into Healthcare: A Global Responsibility. *American Journal of Human Genetics*, 104(1):13–20.

[115] Stark, Z., Schofield, D., Alam, K., Wilson, W., Mupfeki, N., Macciocca, I., Shrestha, R., White, S. M., and Gaff, C. (2017). Prospective comparison of the cost-effectiveness of clinical whole-exome sequencing with that of usual care overwhelmingly supports early use and reimbursement. *Genetics in Medicine*, 19(8):867–874.

[116] Stephens, Z. D., Lee, S. Y., Faghri, F., Campbell, R. H., Zhai, C., Efron, M. J., Iyer, R., Schatz, M. C., Sinha, S., and Robinson, G. E. (2015). Big Data: Astronomical or Genomical? *PLOS Biology*, 13(7):e1002195.

[117] Strauss, E. C., Kobori, J. A., Siu, G., and Hood, L. E. (1986). Specific-primer-directed DNA sequencing. *Analytical Biochemistry*, 154(1):353–360.

[118] Sud, A., Kinnersley, B., and Houlston, R. S. (2017). Genome-wide association studies of cancer: current insights and future perspectives. *Nature Reviews Cancer*, 17(11):692–704.

[119] Tajima, F. (1989). Statistical method for testing the neutral mutation hypothesis by dna polymorphism. *Genetics*, 123(3):585–595.

[120] Talbert, P. B. and Henikoff, S. (2010). Centromeres Convert but Don't Cross. *PLoS Biology*, 8(3):e1000326.

[121] Taliun, Daniel et al. (2019). Sequencing of 53,831 diverse genomes from the NHLBI TOPMed Program. *bioRxiv*, page 563866.

[122] Tarailo-Graovac, Majaet al. (2016). Exome Sequencing and the Management of Neurometabolic Disorders. *New England Journal of Medicine*, 374(23):2246–2255.

[123] The 1000 Genomes Consortium (2015). A global reference for human genetic variation. *Nature*, 526(7571):68–74.

[124] The CARDIoGRAMplusC4D Consortium (2015). A comprehensive 1000 Genomes–based genome-wide association meta-analysis of coronary artery disease. *Nature Genetics*, 47(10):1121–1130.

[125] The Haplotype Reference Consortium (2016). A reference panel of 64,976 haplotypes for genotype imputation. *Nature Genetics*, 48(10):1279–1283.

[126] Thornton, T., Tang, H., Hoffmann, T., Ochs-Balcom, H., Caan, B., and Risch, N. (2012). Estimating Kinship in Admixed Populations. *The American Journal of Human Genetics*, 91(1):122–138.

[127] Turnbull, C., Scott, R. H., Thomas, E., Jones, L., Murugaesu, N., Pretty, F. B., Halai, D., Baple, E., Craig, C., Hamblin, A., Henderson, S., Patch, C., O'Neill, A., Devereaux, A., Smith, K., Martin, A. R., Sosinsky, A., McDonagh, E. M., Sultana, R., Mueller, M., Smedley, D., Toms, A., Dinh, L., Fowler, T., Bale, M., Hubbard,

T., Rendon, A., Hill, S., and Caulfield, M. J. (2018). The 100 000 Genomes Project: Bringing whole genome sequencing to the NHS. *BMJ*.

[128] Venter, J C et al. (2001). The sequence of the human genome. *Science (New York, N.Y.)*, 291(5507):1304–51.

[129] Voight, B. F. and Pritchard, J. K. (2005). Confounding from Cryptic Relatedness in Case-Control Association Studies. *PLoS Genetics*, 1(3):e32.

[130] Wang, J., Zaitlen, N. A., Wade, C. M., Kirby, A., Heckerman, D., Daly, M. J., and Eskin, E. (2002). An estimator for pairwise relatedness using molecular markers. *Genetics*, 160(3):1203–15.

[131] Warren, Jr., H. S. (2007). The quest for an accelerated population count. In Wilson, G. and Oram, A., editors, *Beautiful Code: Leading Programmers Explain How They Think*, pages 147–160. chapter 10, O'Reilly Media, Sebastopol, California.

[132] Watson, J. D. and Crick, F. H. C. (1953). Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid. *Nature*, 171(4356):737–738.

[133] Weber, J. L. and Myers, E. W. (1997). Human whole-genome shotgun sequencing. *Genome Research*, 7(5):401–9.

[134] Weir, B. S. and Cockerham, C. C. (1984). Estimating Fi-statistics for the analysis of population structure. *Evolution*, 38(6):1358–1370.

[135] Wilkes, M. V., Wheeler, D. J., and Gill, S. (1957). *The Preparation of Programs for an Electronic Digital Computer*. Addison-Wesley Publishing, Boston, USA, second edition.

[136] Wu, K., Otoo, E. J., and Shoshani, A. (2006). Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(1):1–38.

[137] Wu, T. D. and Nacu, S. (2010). Fast and SNP-tolerant detection of complex variants and splicing in short reads. *Bioinformatics*, 26(7):873–881.

[138] Yang, Y., Muzny, D. M., Reid, J. G., Bainbridge, M. N., Willis, A., Ward, P. A., Braxton, A., Beuten, J., Xia, F., Niu, Z., Hardison, M., Person, R., Bekheirnia, M. R., Leduc, M. S., Kirby, A., Pham, P., Scull, J., Wang, M., Ding, Y., Plon, S. E., Lupski, J. R., Beaudet, A. L., Gibbs, R. A., and Eng, C. M. (2013). Clinical Whole-Exome Sequencing for the Diagnosis of Mendelian Disorders. *New England Journal of Medicine*, 369(16):1502–1511.

[139] Yang, Y., Muzny, D. M., Xia, F., Niu, Z., Person, R., Ding, Y., Ward, P., Braxton, A., Wang, M., Buhay, C., Veeraraghavan, N., Hawes, A., Chiang, T., Leduc, M., Beuten, J., Zhang, J., He, W., Scull, J., Willis, A., Landsverk, M., Craigen, W. J., Bekheirnia, M. R., Stray-Pedersen, A., Liu, P., Wen, S., Alcaraz, W., Cui, H., Walkiewicz, M., Reid, J., Bainbridge, M., Patel, A., Boerwinkle, E., Beaudet, A. L., Lupski, J. R., Plon, S. E., Gibbs, R. A., and Eng, C. M. (2014). Molecular Findings Among Patients Referred for Clinical Whole-Exome Sequencing. *JAMA*, 312(18):1870.

[140] Yengo, L., Sidorenko, J., Kemper, K. E., Zheng, Z., Wood, A. R., Weedon, M. N., Frayling, T. M., Hirschhorn, J., Yang, J., and Visscher, P. M. (2018). Meta-analysis of genome-wide association studies for height and body mass index in 700000 individuals of European ancestry. *Human Molecular Genetics*, 27(20):3641–3649.

[141] Yu, J., Pressoir, G., Briggs, W. H., Vroh Bi, I., Yamasaki, M., Doebley, J. F., McMullen, M. D., Gaut, B. S., Nielsen, D. M., Holland, J. B., Kresovich, S., and Buckler, E. S. (2006). A unified mixed-model method for association mapping that accounts for multiple levels of relatedness. *Nature Genetics*, 38(2):203–208.

[142] Yu, Y. W., Yorukoglu, D., Peng, J., and Berger, B. (2015). Quality score compression improves genotyping accuracy. *Nature Biotechnology*.

[143] Zeng, Z. B., Delbruck, M., and Woolliams, J. A. (1994). Precision mapping of quantitative trait loci. *Genetics*, 136(4):1457–68.

[144] Zhang, D., Yang, J., Ye, D., and Hua, G. (2018). LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks. In Ferrari, V., Hebert, M., Sminchisescu, C., and Weiss, Y., editors, *Computer Vision – ECCV 2018*, pages 373–390, Cham. Springer International Publishing.

[145] Zhang, Z., Ersoz, E., Lai, C.-Q., Todhunter, R. J., Tiwari, H. K., Gore, M. A., Bradbury, P. J., Yu, J., Arnett, D. K., Ordovas, J. M., and Buckler, E. S. (2010). Mixed linear model approach adapted for genome-wide association studies. *Nature Genetics*, 42(4):355–360.

[146] Zhao, Wei et al. (2017). Identification of new susceptibility loci for type 2 diabetes and shared etiological pathways with coronary heart disease. *Nature Genetics*, 49(10):1450–1457.

[147] Zheng, X., Gogarten, S. M., Lawrence, M., Stilp, A., Conomos, M. P., Weir, B. S., Laurie, C., and Levine, D. (2017). SeqArray—a storage-efficient high-performance data format for WGS variant calls. *Bioinformatics*, 33(15):2251–2257.

[148] Zhou, X. and Stephens, M. (2012). Genome-wide efficient mixed-model analysis for association studies. *Nature Genetics*, 44(7):821–824.

[149] Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343.

[150] Zukowski, M., Heman, S., Nes, N., and Boncz, P. (2006). Super-scalar ram-cpu cache compression. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA. IEEE Computer Society.

Supplemental Information: Chapter 3

.1 Figures

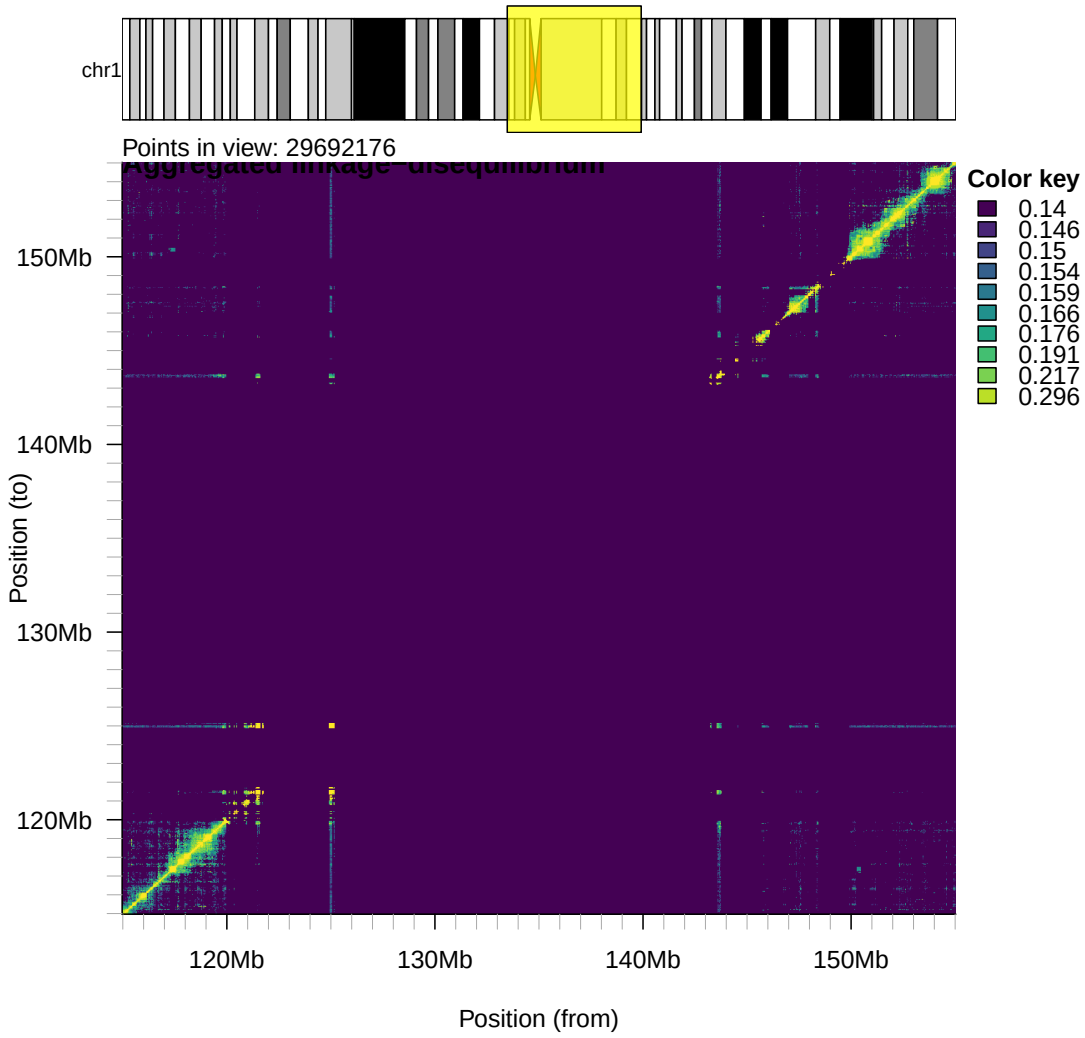


Figure 1 Centromeric LD block for chromosome 1.

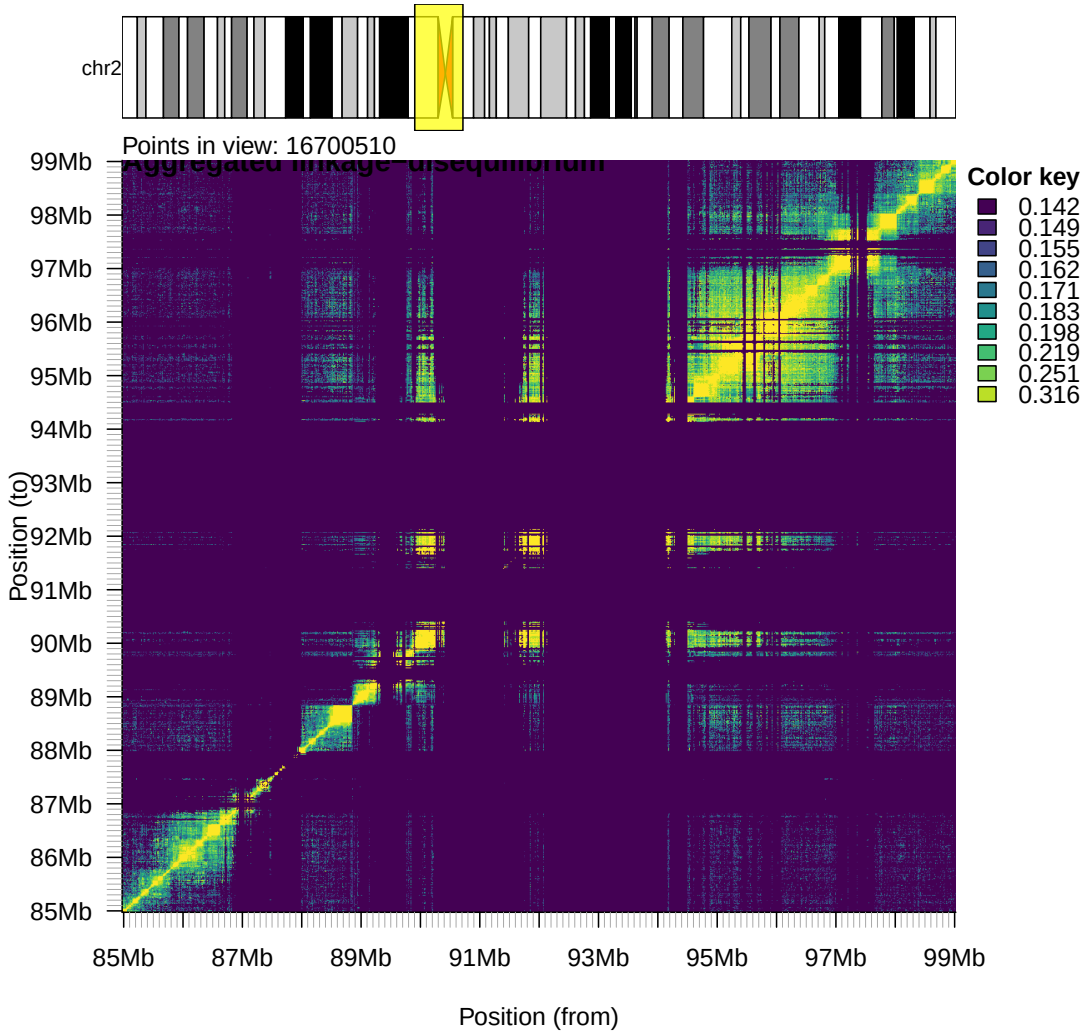


Figure 2 Centromeric LD block for chromosome 2.

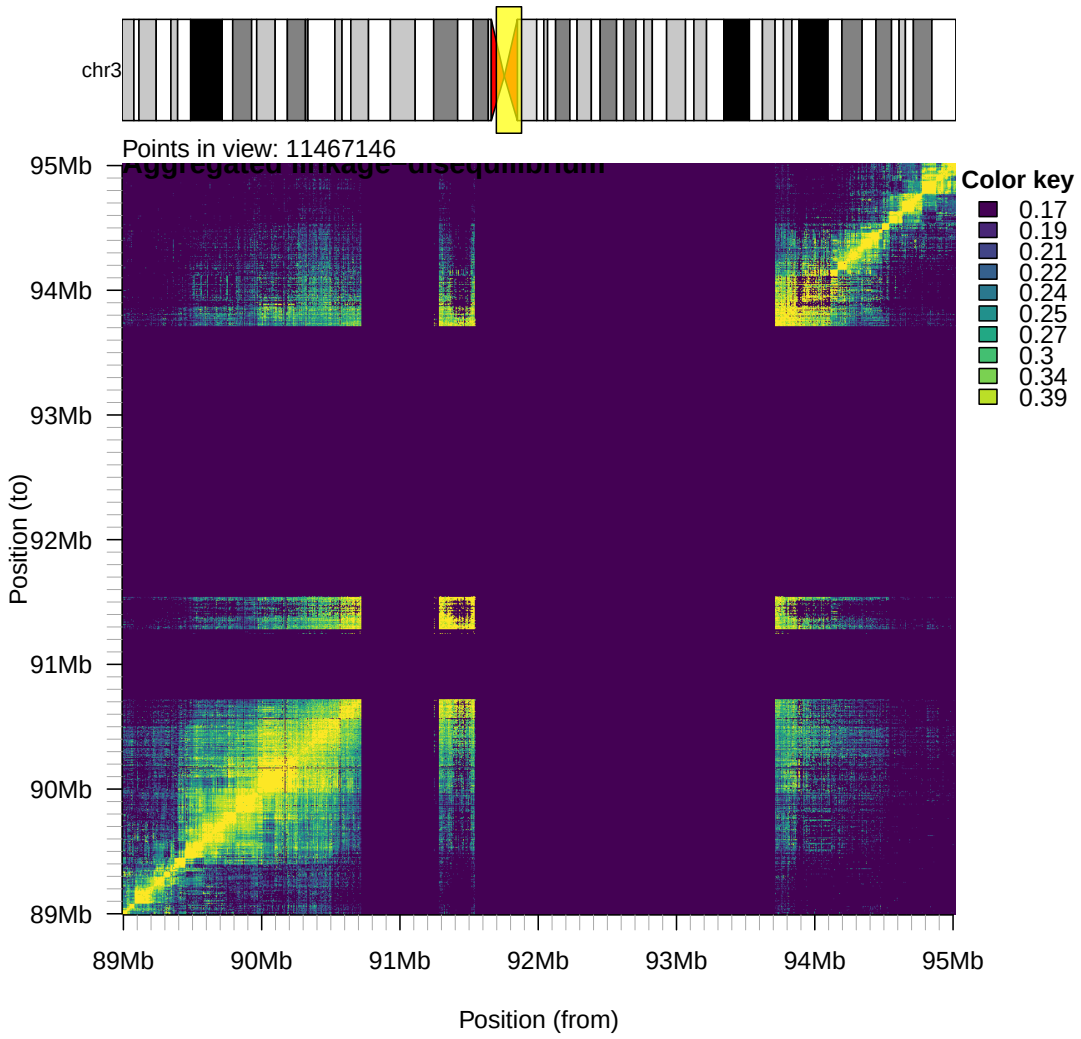


Figure 3 Centromeric LD block for chromosome 3.

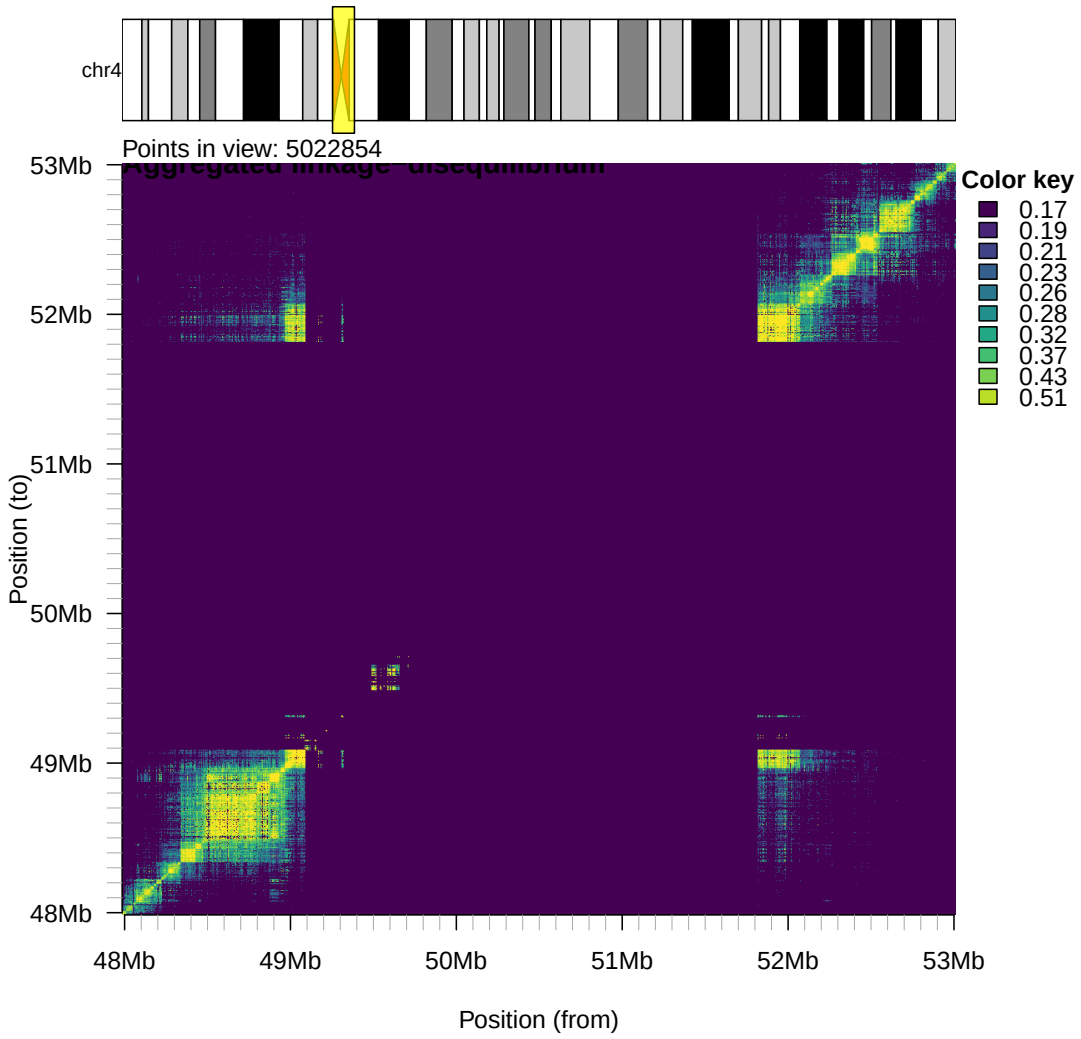


Figure 4 Centromeric LD block for chromosome 4.

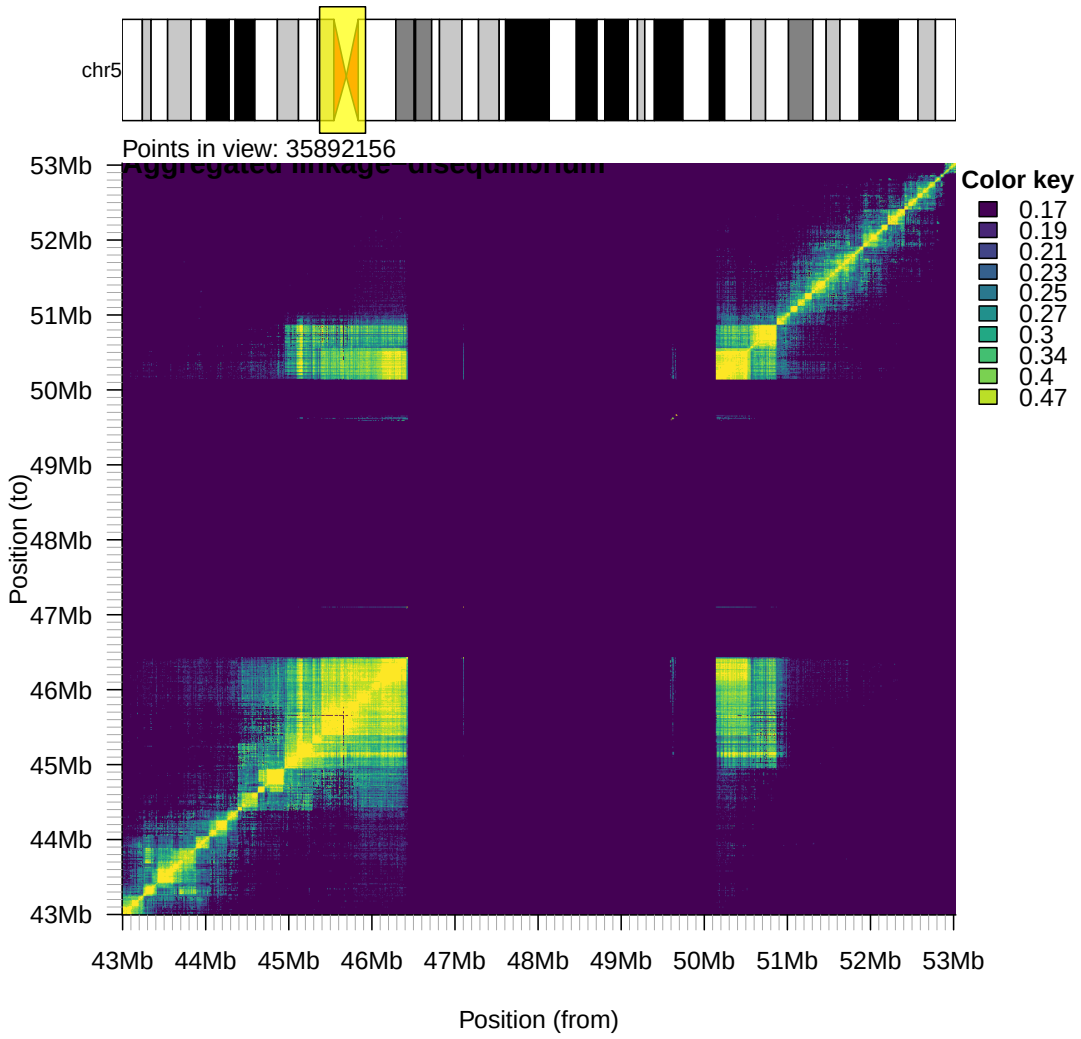


Figure 5 Centromeric LD block for chromosome 5.

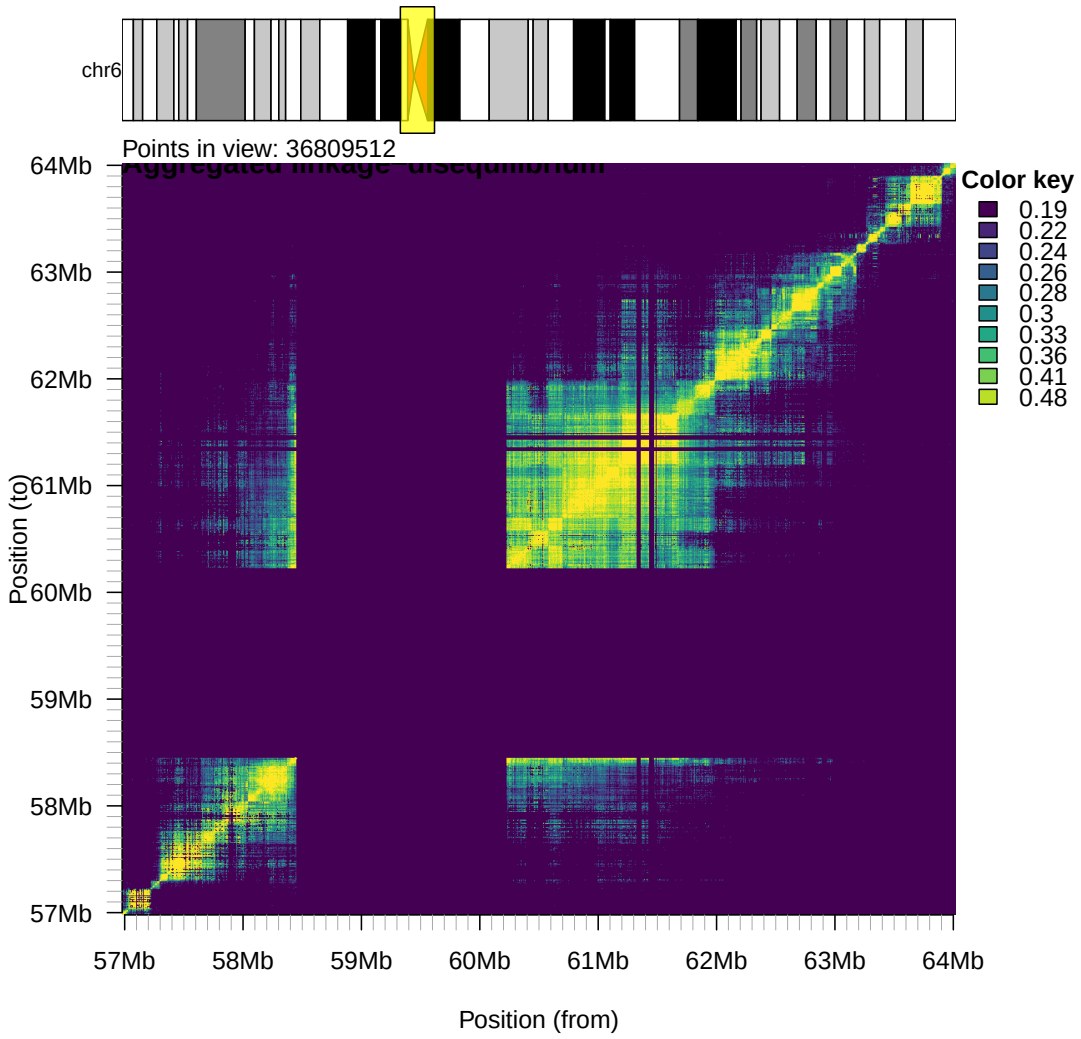


Figure 6 Centromeric LD block for chromosome 6.

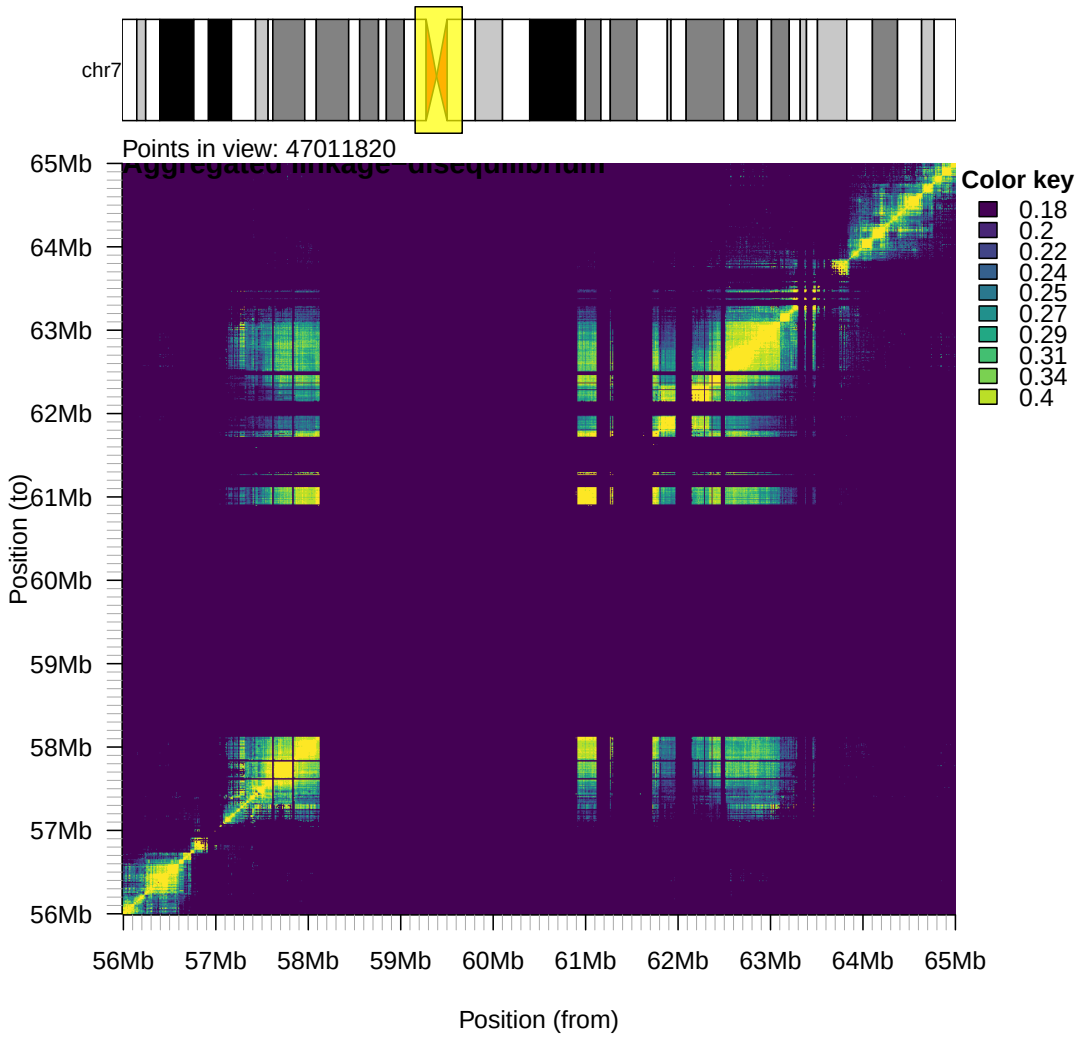


Figure 7 Centromeric LD block for chromosome 7.

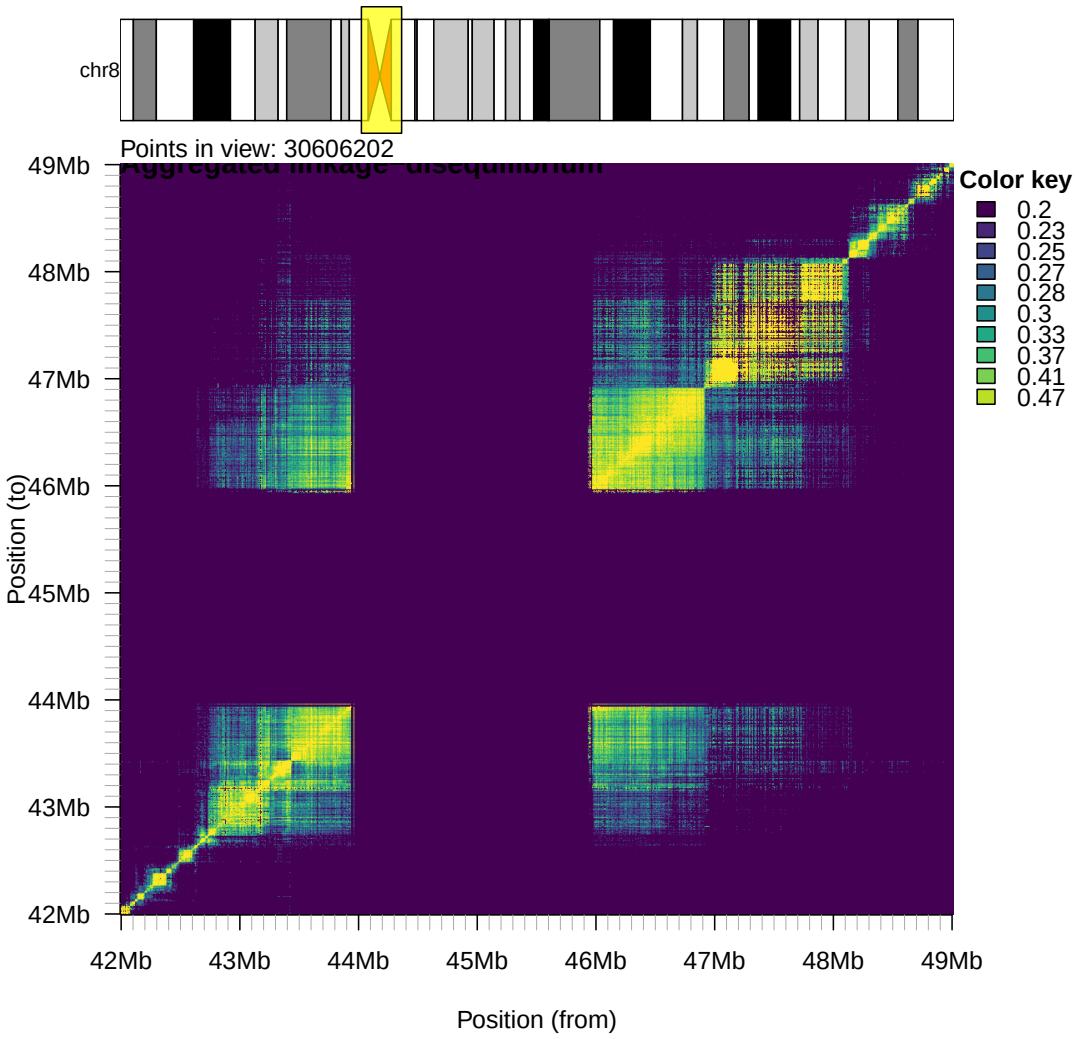


Figure 8 Centromeric LD block for chromosome 8.

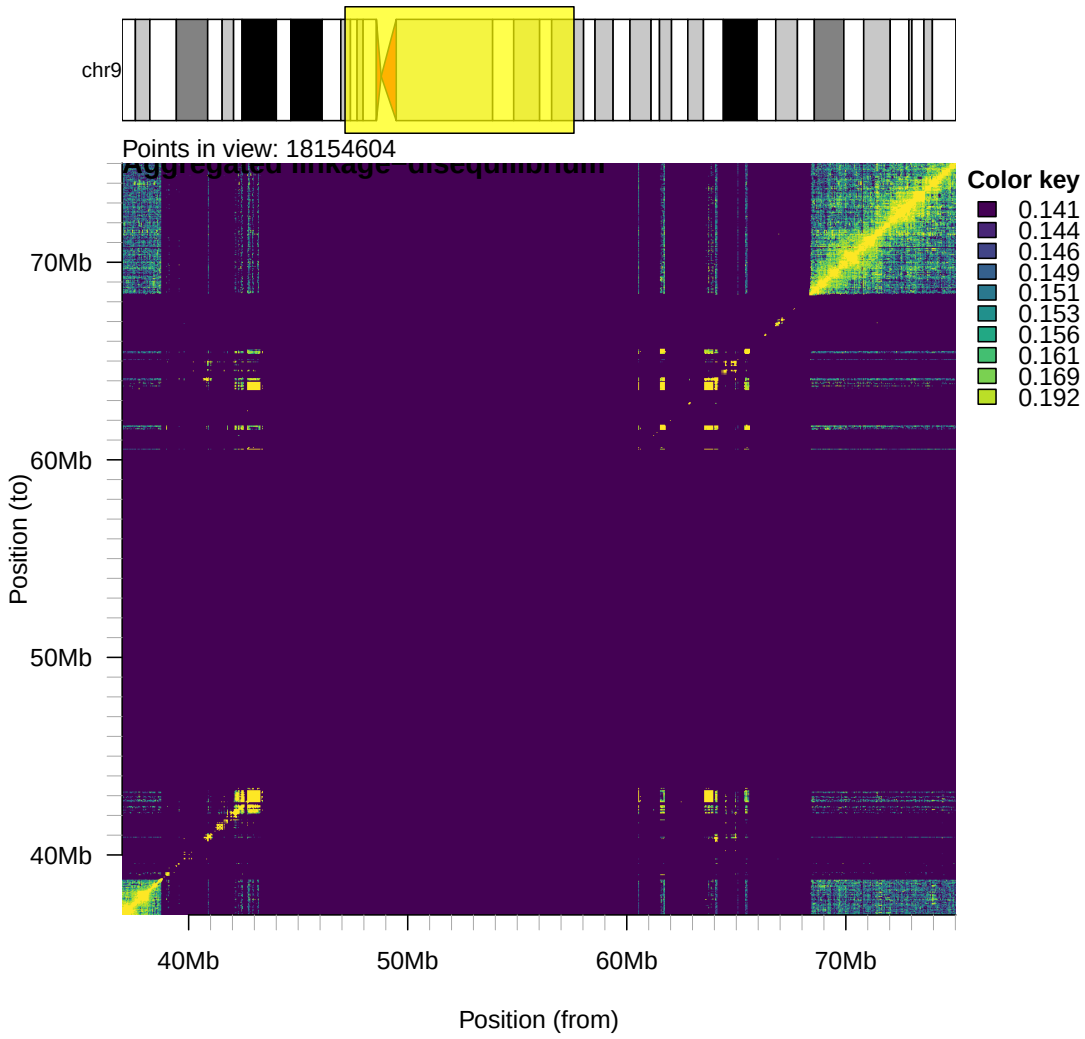


Figure 9 Centromeric LD block for chromosome 9.

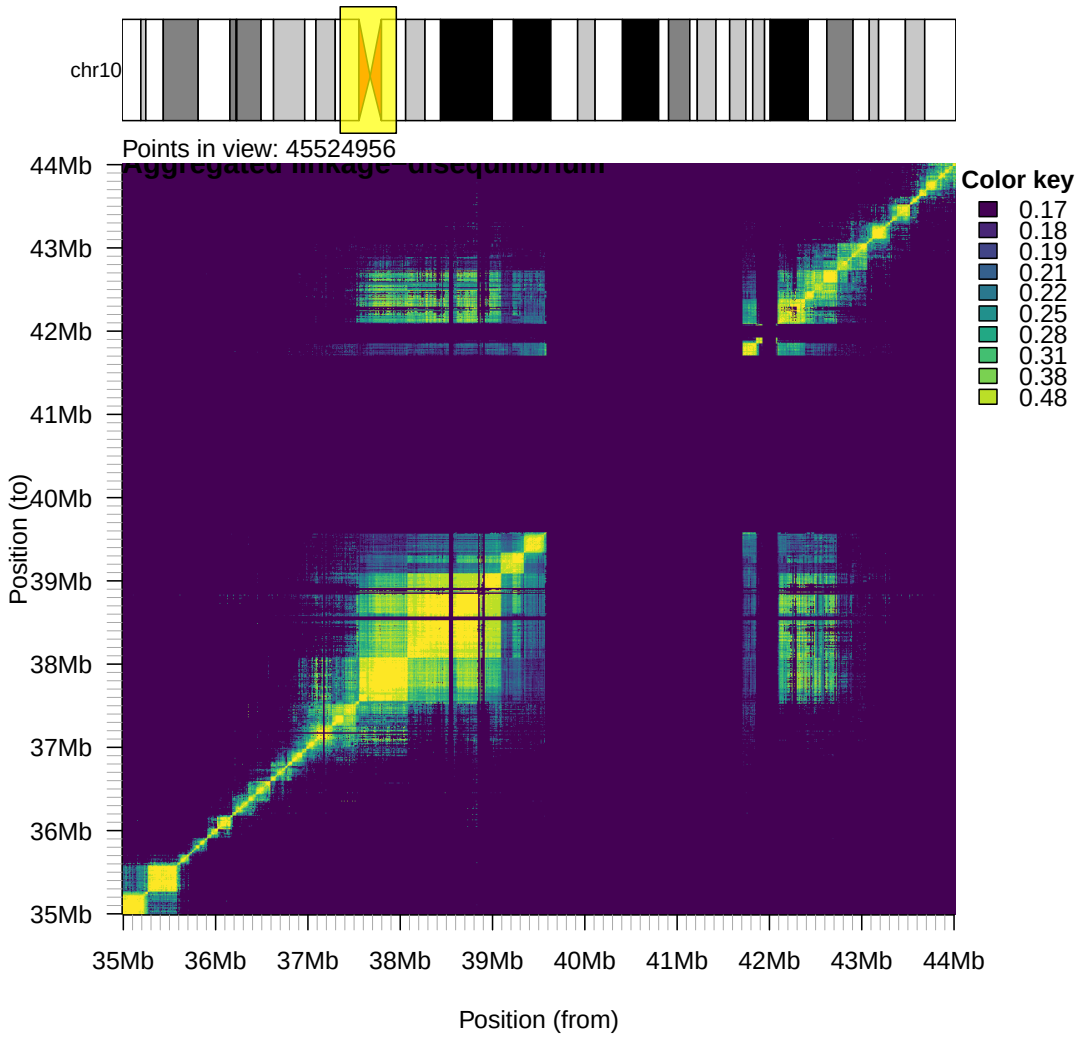


Figure 10 Centromeric LD block for chromosome 10.

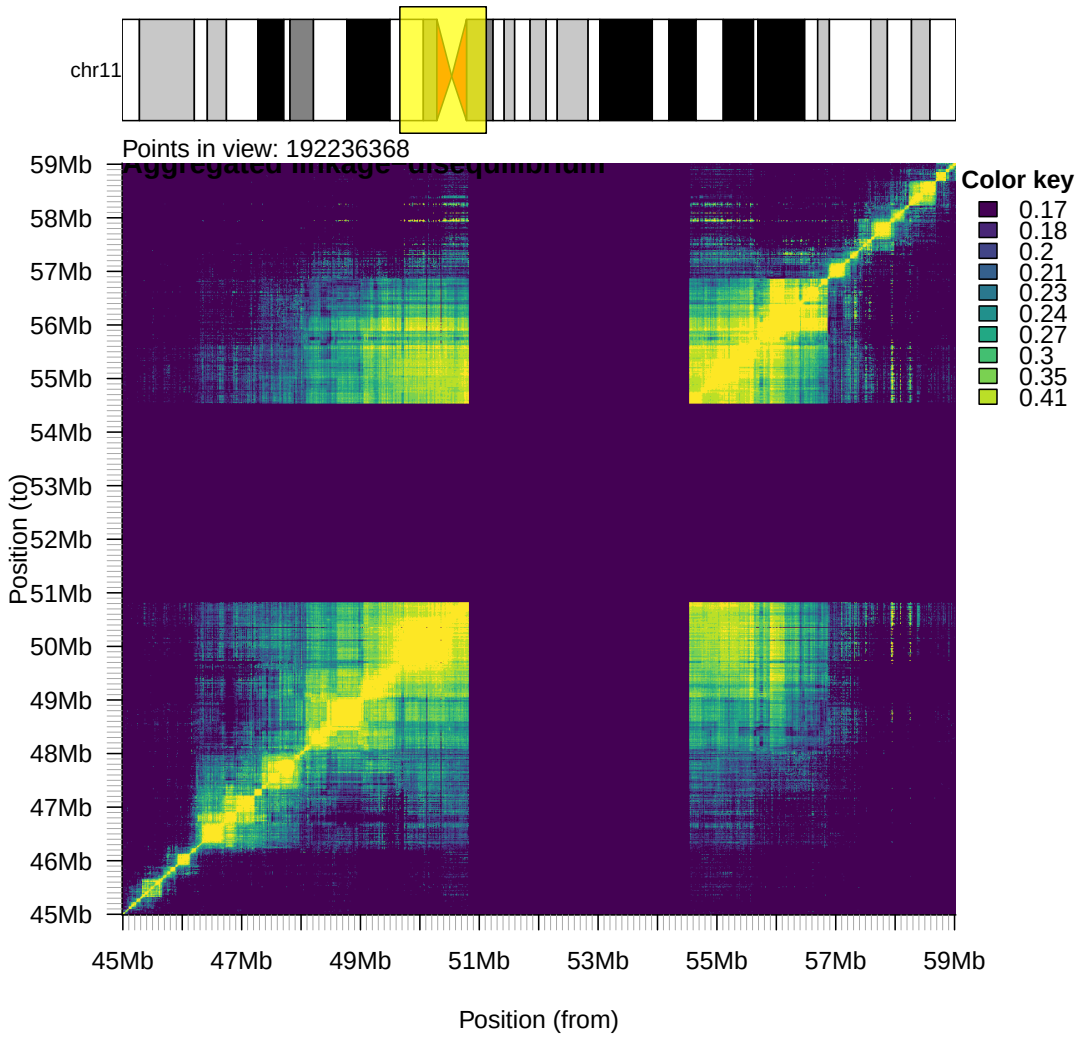


Figure 11 Centromeric LD block for chromosome 11.

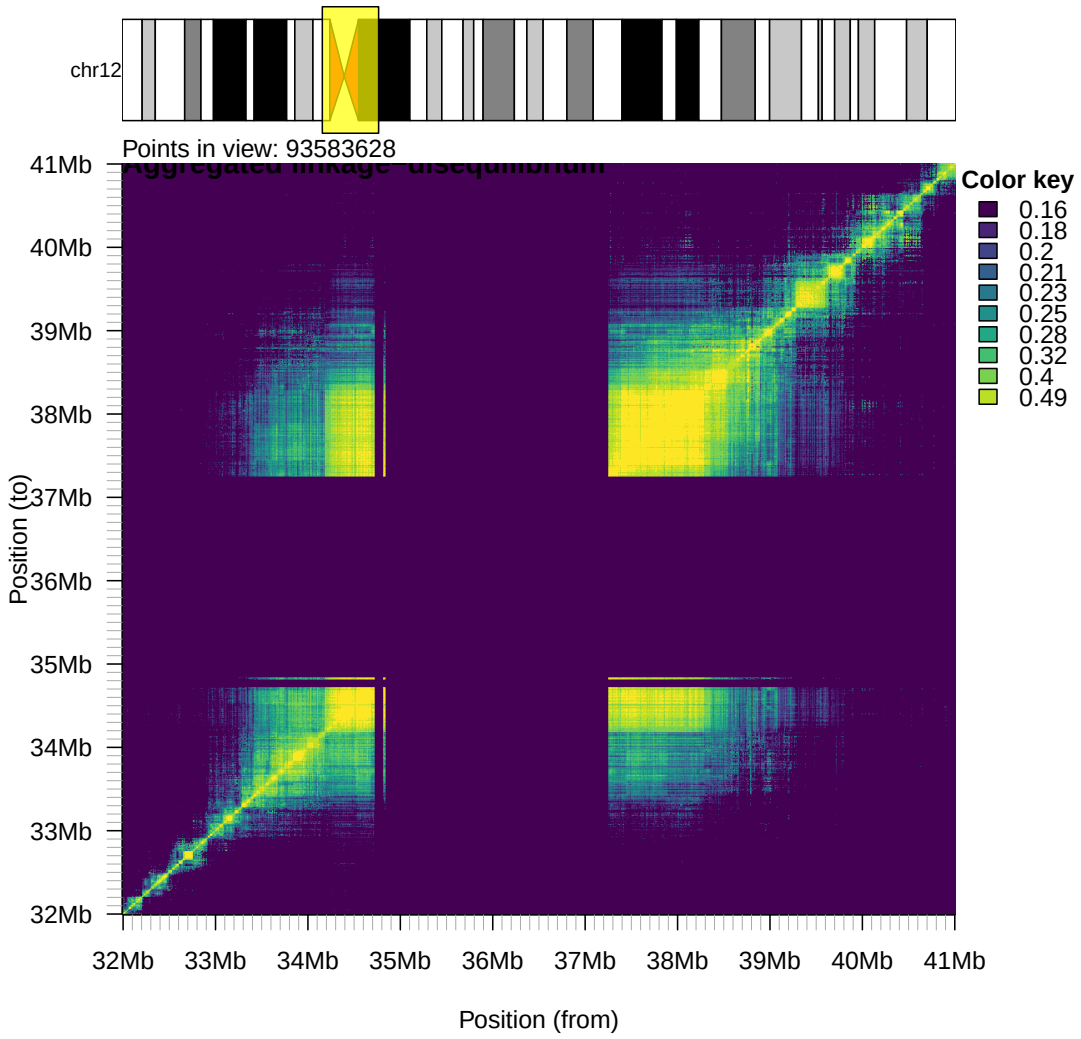


Figure 12 Centromeric LD block for chromosome 12.

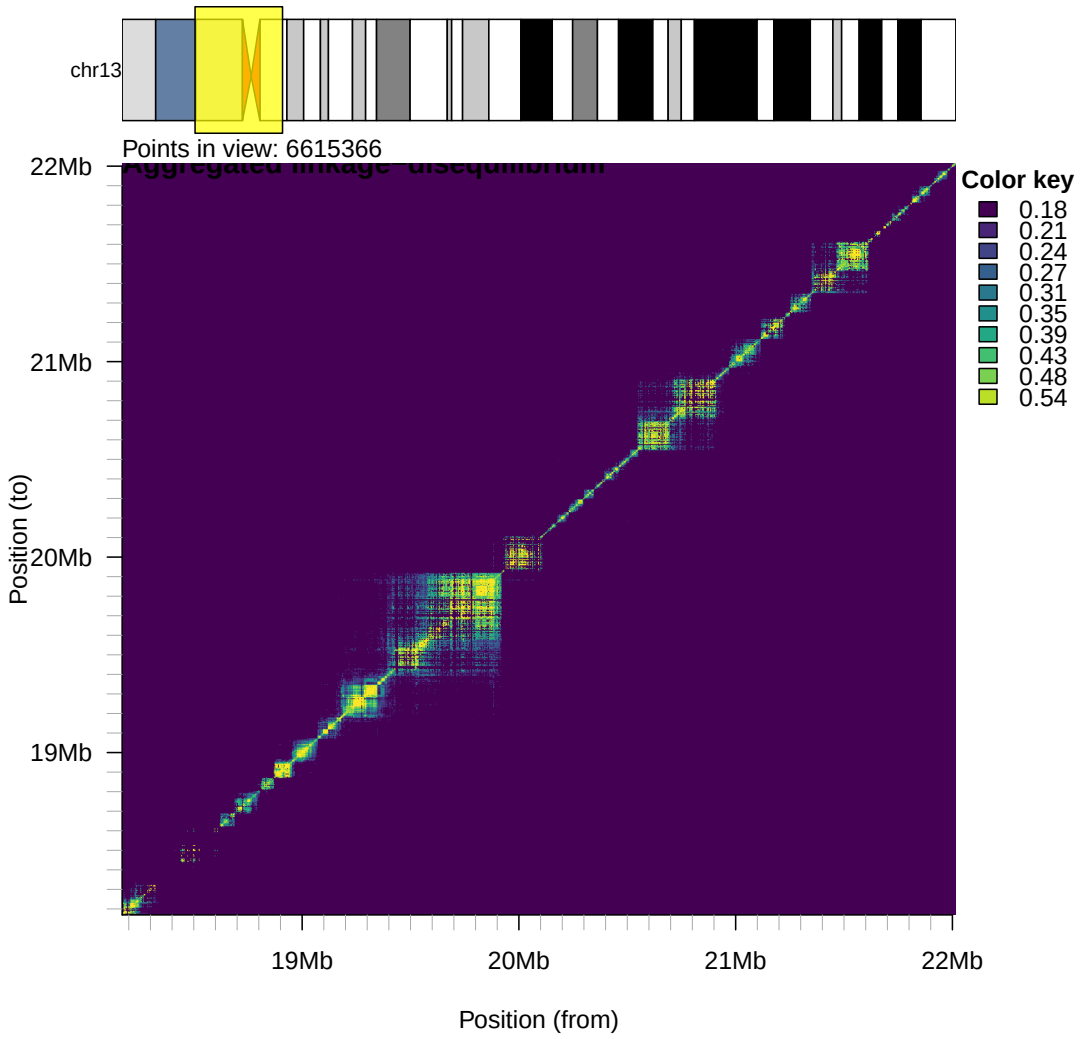


Figure 13 Centromeric LD block for chromosome 13.

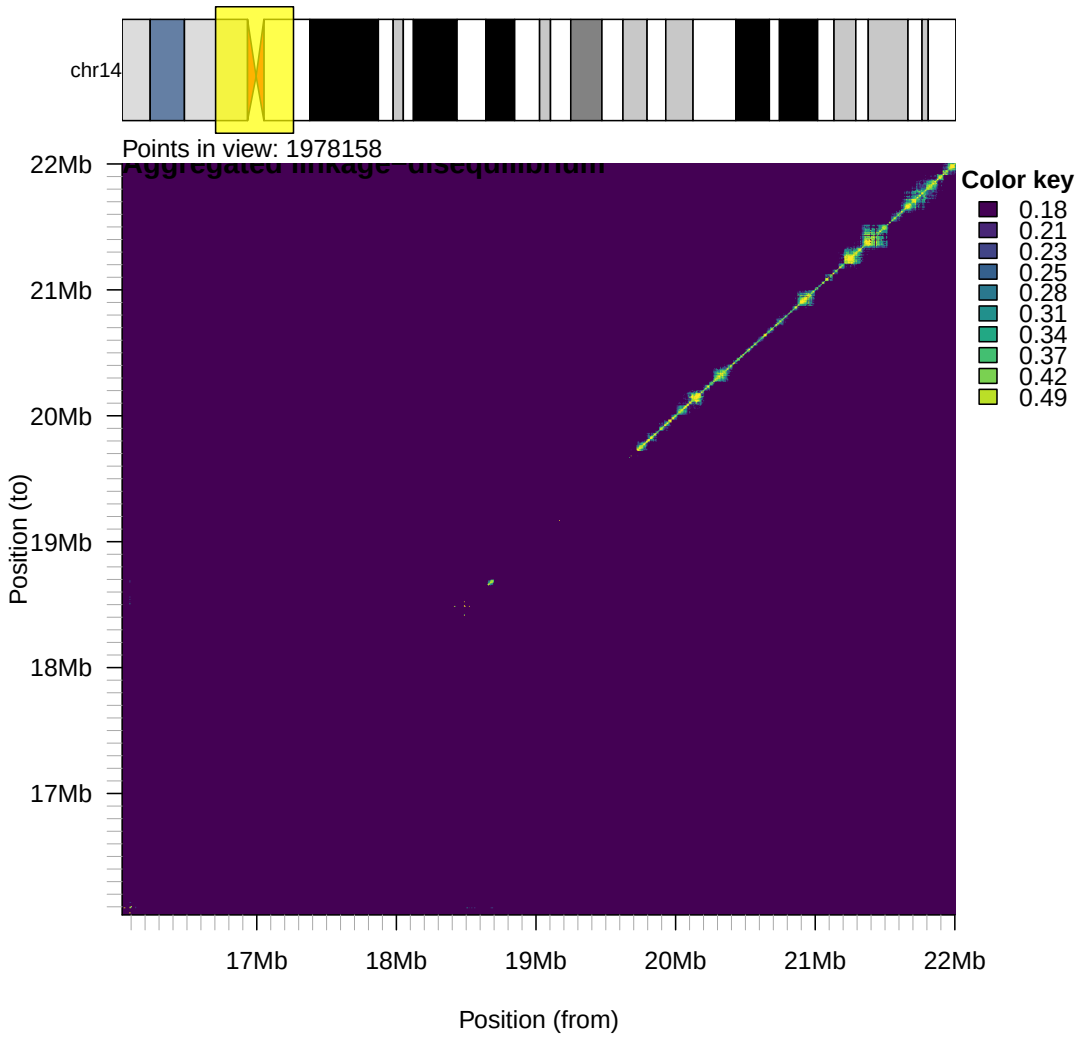


Figure 14 Centromeric LD block for chromosome 14.

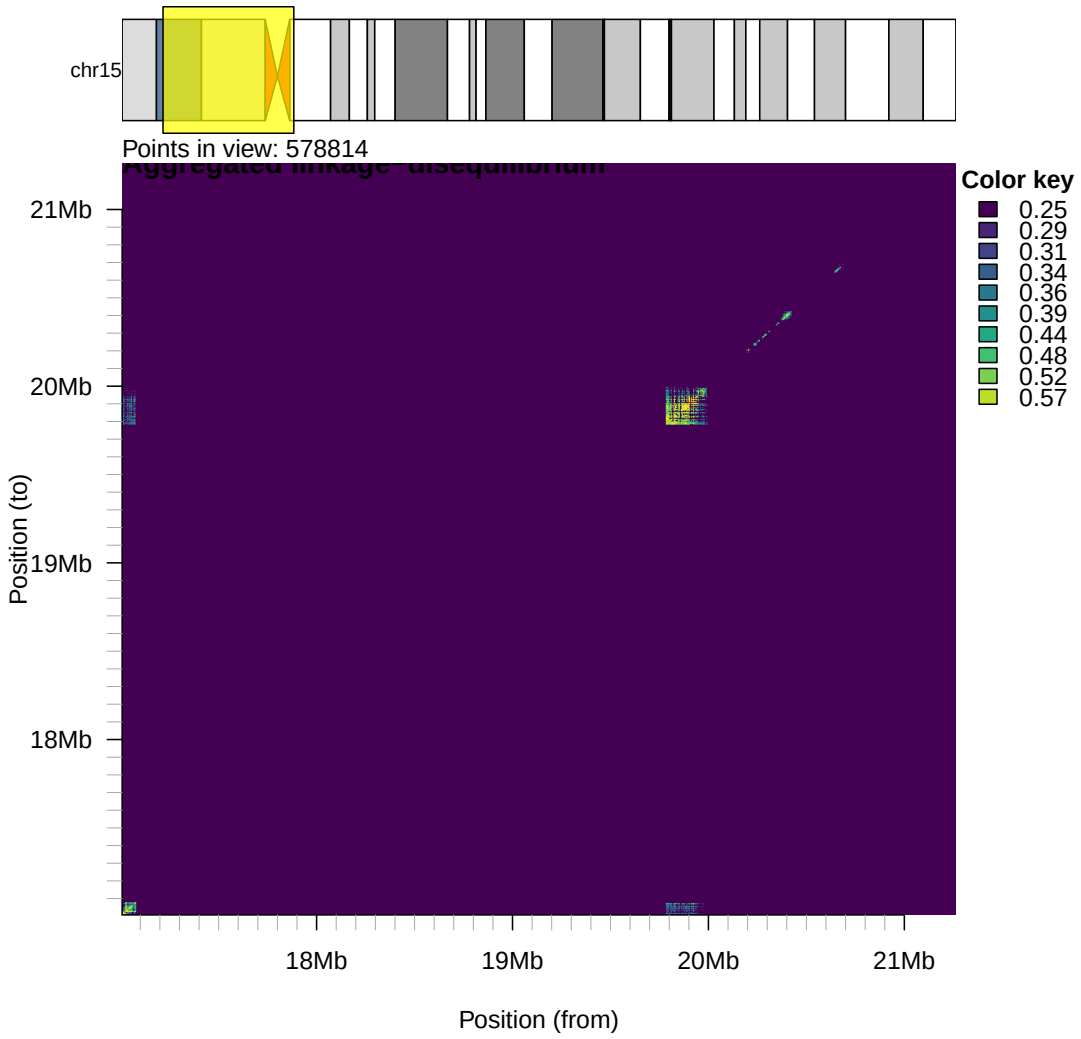


Figure 15 Centromeric LD block for chromosome 15.

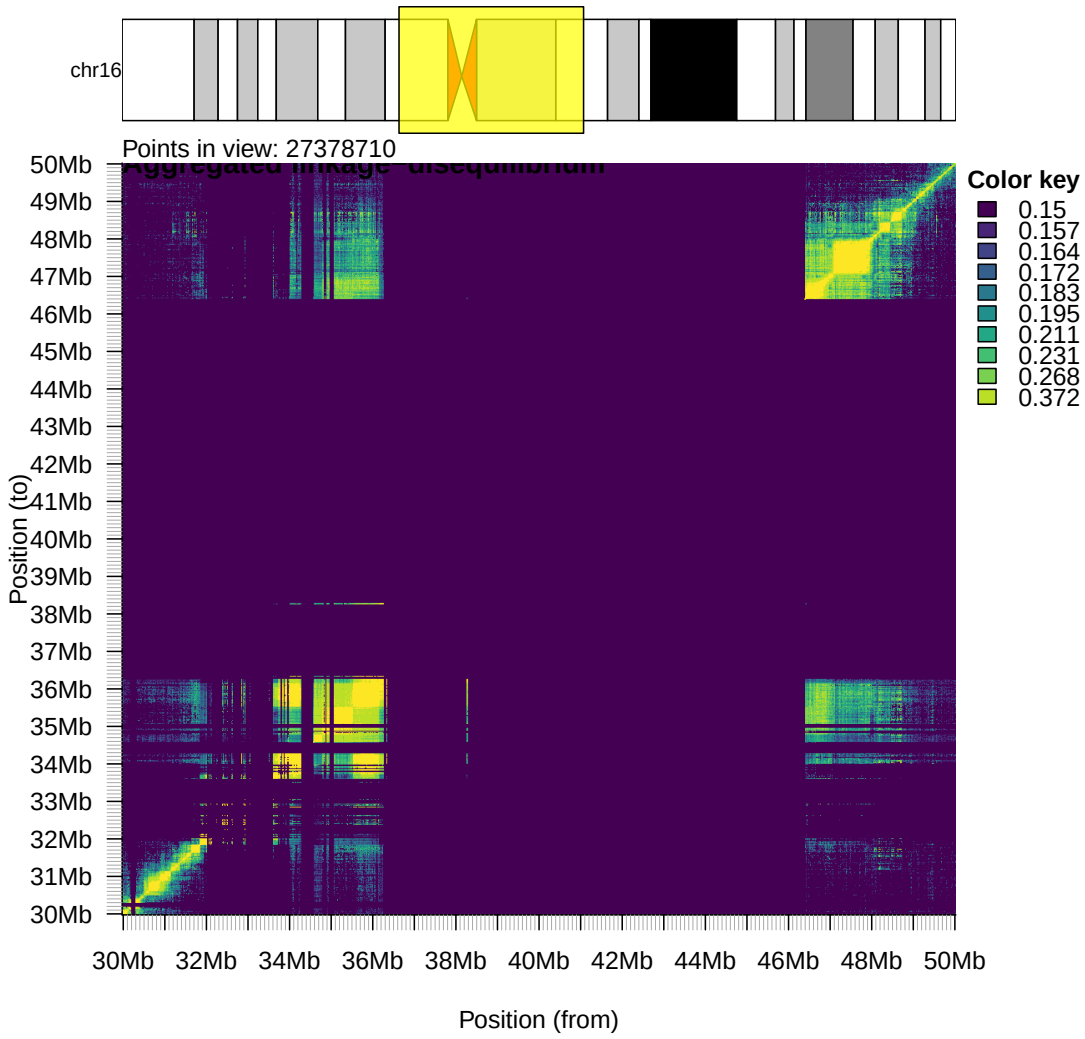


Figure 16 Centromeric LD block for chromosome 16.

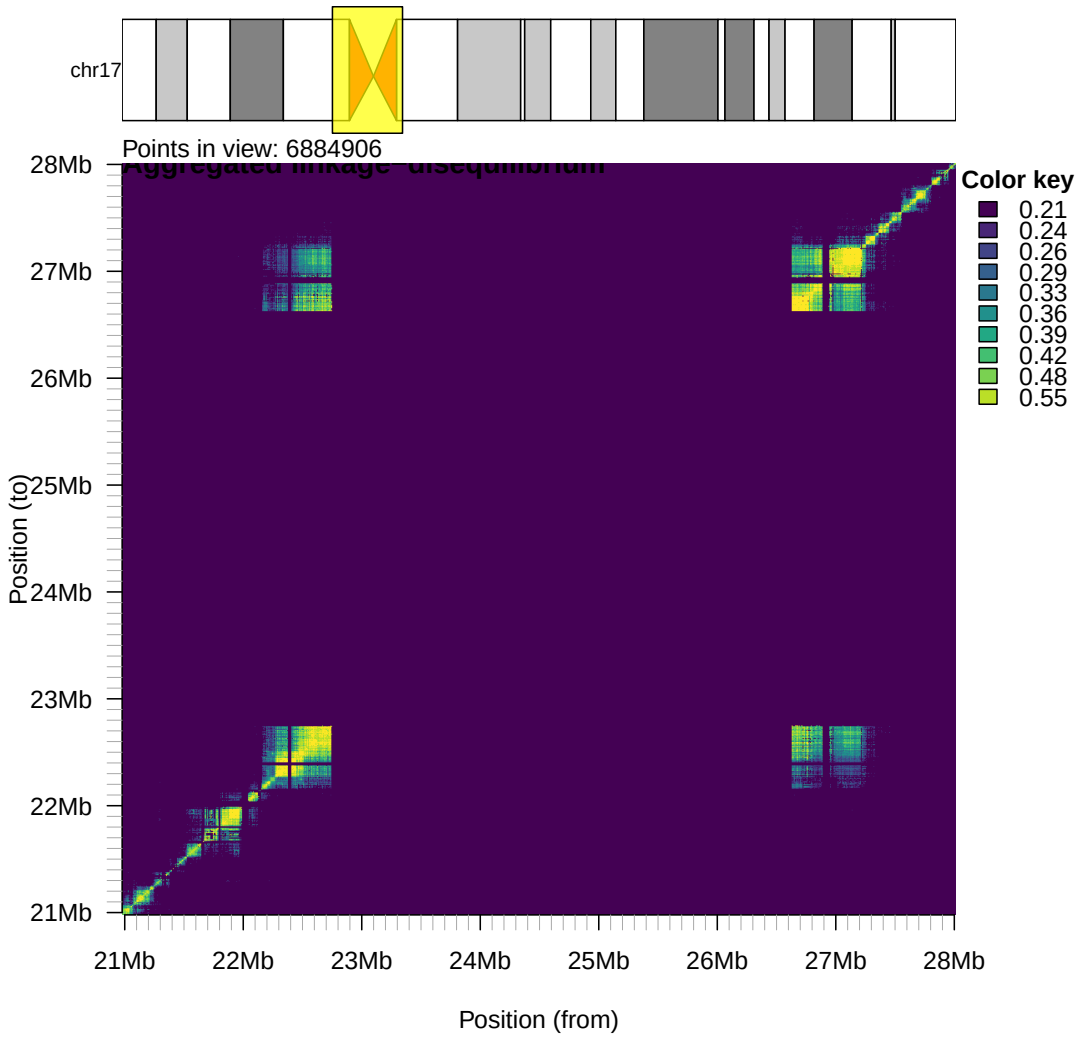


Figure 17 Centromeric LD block for chromosome 17.

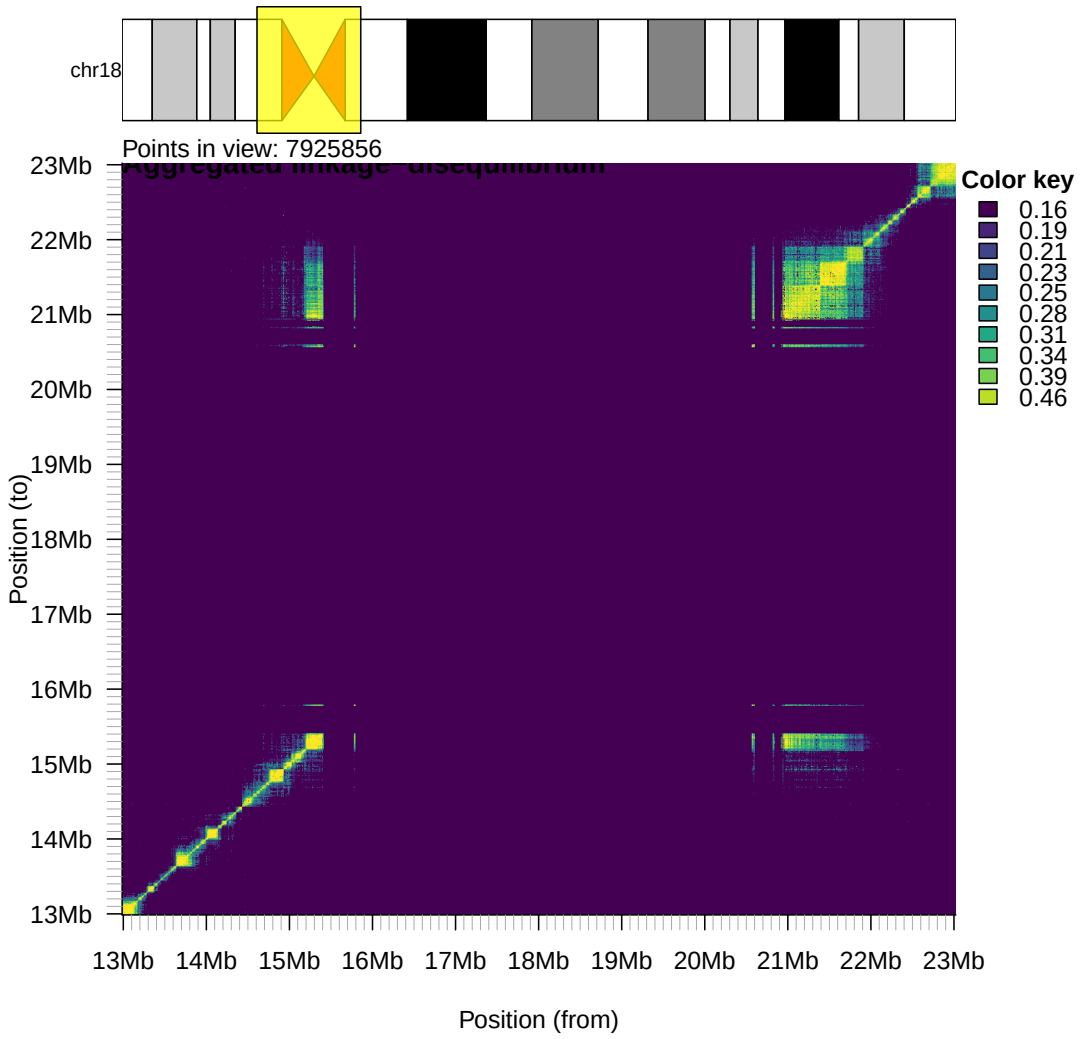


Figure 18 Centromeric LD block for chromosome 18.

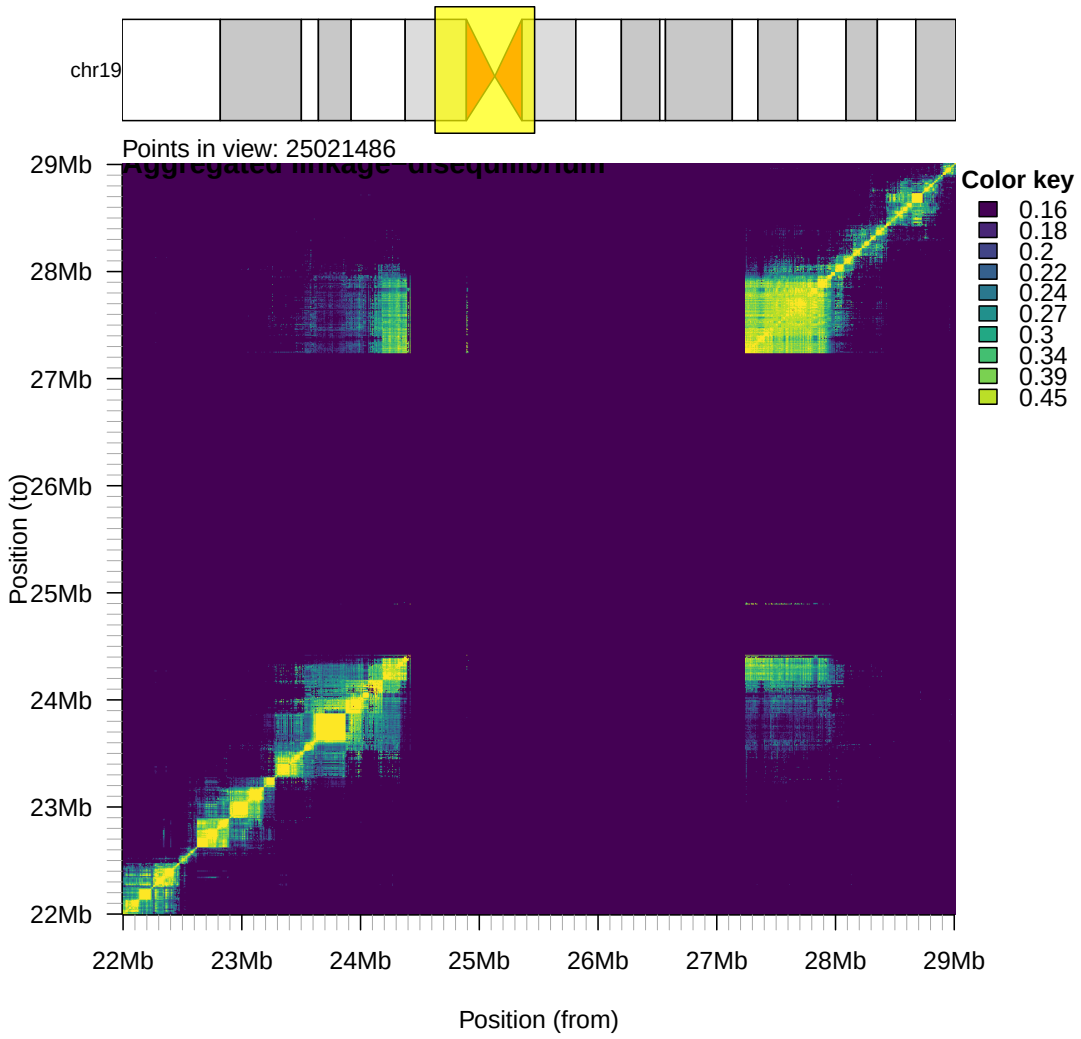


Figure 19 Centromeric LD block for chromosome 19.

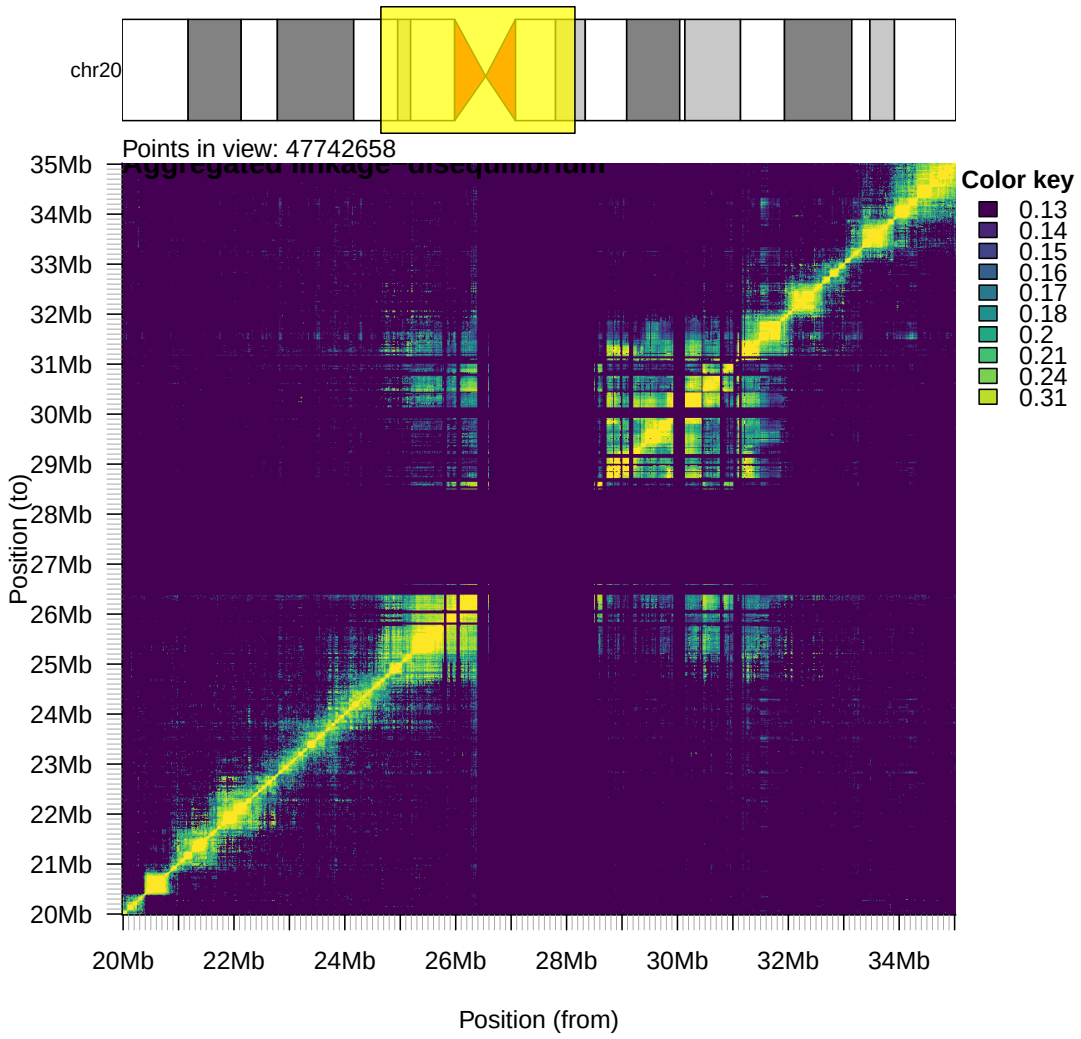


Figure 20 Centromeric LD block for chromosome 20.

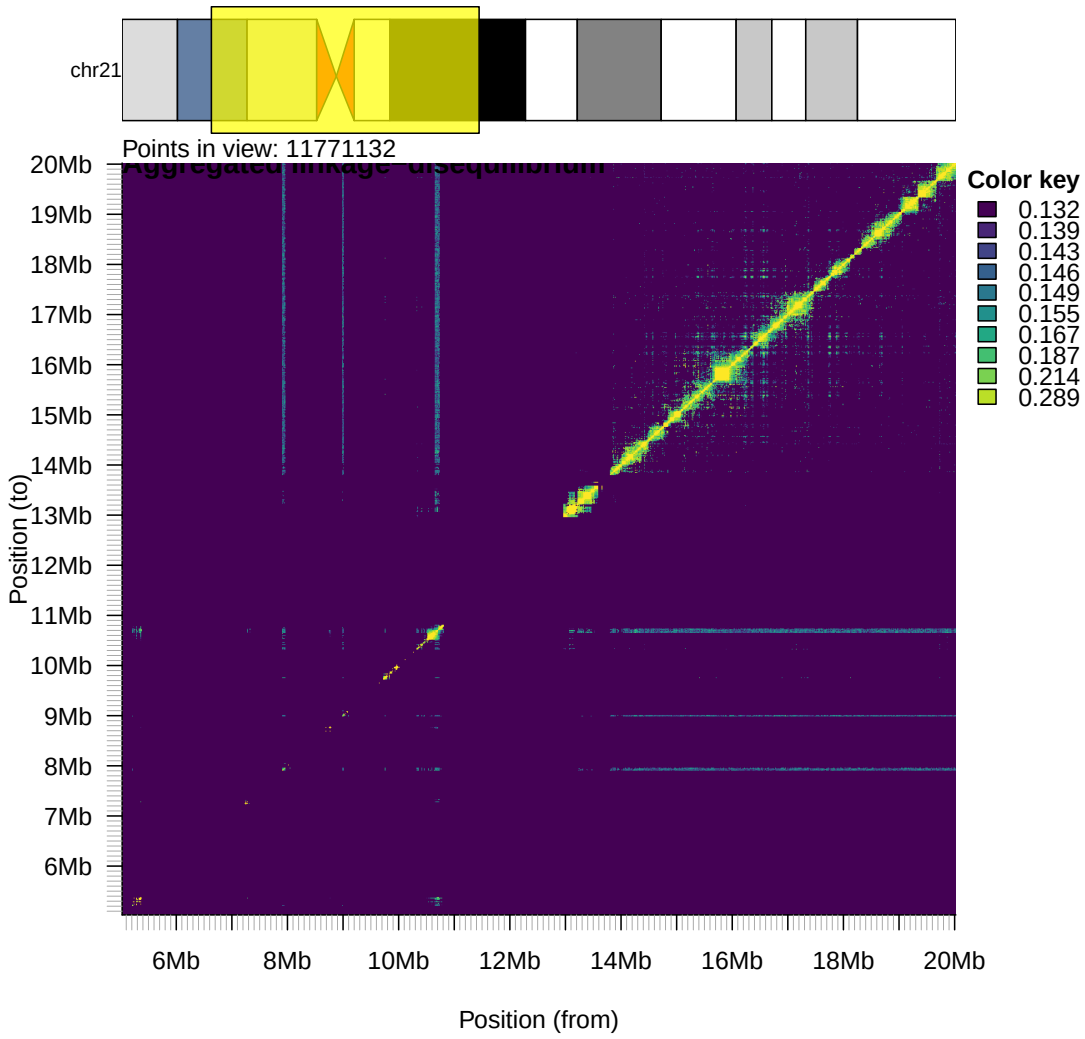


Figure 21 Centromeric LD block for chromosome 21.

.2 Workflow

In this section I describe reproducible code for computing the statistics for 1000 Genomes populations and super-populations as described in Chapter 3 on a *nix machine with the programming language `R` available.

.2.1 Slicing populations

Download the panel description table from ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/integrated_call_samples_v3.20130502.ALL.panel and the chromosome datasets ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/ALL.chr*.phase3_shapeit2_mvncall_integrated_v5a.20130502.genotypes.vcf.gz*. Partitioning the panels (super-populations and populations) table into sub-tables can be done using R:

```
panel <- read.delim("integrated_call_samples_v3.20130502.ALL.panel")
pops <- levels(panel$pop)
for(i in 1:length(pops)){
  write.table(panel[panel$pop==pops[i],1,drop=FALSE],
              file = paste0("1kgp3_pop__", pops[i], ".txt"),
              quote = F,row.names = F,col.names = F)
}

super_pops <- levels(panel$super_pop)
for(i in 1:length(super_pops)){
  write.table(panel[panel$super_pop==super_pops[i],1,drop=FALSE],
              file = paste0("1kgp3_super_pop__", super_pops[i], ".txt"),
              quote = F,row.names = F,col.names = F)
}
```

Slice the target bcf files using the sample names described in the super population and population files

```
for j in {1..22}; do \
chr=$j;
for i in {ACB,ASW,BEB,CDX,CEU,CHB,CHS,CLM,ESN,FIN,GBR,GIH,GWD,IBS,ITU,JPT,KHV, \
          LWK,MSL,MXL,PEL,PJL,PUR,STU,TSI,YRI}; \
do bcftools view ALL.chr${chr}.phase3*.bcf \
  -S 1kgp3_pop__$i.txt -O b -o 1kgp3_chr${chr}\__pop_$i.bcf; \
done; \
done

for j in {1..22}; do \
chr=$j;
for i in {AFR,AMR,EAS,EUR,SAS}; \
do bcftools view ALL.chr${chr}.phase3*.bcf \
  -S 1kgp3_super_pop__$i.txt -O b -o 1kgp3_chr${chr}\__super_pop_$i.bcf; \
done; \
done
```

Supplemental Information: Analysing population-scaled sequence variant data

Table 1 Field statistics for gnomad 2.1.1 whole-genome sequence variants. Target dataset was downloaded from https://storage.googleapis.com/gnomad-public/release/2.1.1/vcf/genomes/gnomad.genomes.r2.1.1.sites.1.vcf.bgz. Abbreviations: I, Integer; F, Float; S: String; NA, Not Applicable; Comp., Compressed size in MB; Ucomp., Uncompressed size in MB; Scomp., Compressed size of strides in MB; Sucomp., Uncompressed size of strides in MB; uBcf, uncompressed Bcf.

Field	Type	Comp.	Ucomp.	Scomp.	Ucomp.	Fold	uBcf
PPA	NA	0.00	0.00	0.00	0.00	0.0	0.00
CONTIG	NA	0.08	0.08	0.00	0.00	1.0	0.00
POSITION	NA	22.94	162.97	0.00	0.00	7.1	0.00
REFALT	NA	0.00	0.00	0.00	0.00	0.0	0.00
CONTROLLER	NA	3.49	40.74	0.00	0.00	11.7	0.00
QUALITY	NA	66.57	81.48	0.00	0.00	1.2	0.00
NAMES	NA	95.05	223.27	5.57	20.37	2.3	0.00
ALLELES	NA	24.87	136.88	0.00	0.00	5.5	0.00
ID_INFO	NA	12.13	20.56	0.00	0.00	1.7	0.00
ID_FORMAT	NA	0.08	0.08	0.00	0.00	1.0	0.00
ID_FILTER	NA	3.14	20.37	0.00	0.00	6.5	0.00
GT_INT8	NA	0.00	0.00	0.00	0.00	0.0	0.00
GT_INT16	NA	0.00	0.00	0.00	0.00	0.0	0.00
GT_INT32	NA	0.00	0.00	0.00	0.00	0.0	0.00
GT_INT64	NA	0.00	0.00	0.00	0.00	0.0	0.00
GT_S_INT8	NA	0.00	0.00	0.00	0.00	0.0	0.00
GT_S_INT16	NA	0.00	0.00	0.00	0.00	0.0	0.00
GT_S_INT32	NA	0.00	0.00	0.00	0.00	0.0	0.00
GT_S_INT64	NA	0.00	0.00	0.00	0.00	0.0	0.00
GT_N_INT8	NA	0.00	0.00	0.00	0.00	0.0	0.00
GT_N_INT16	NA	0.00	0.00	0.00	0.00	0.0	0.00

GT_N_INT32	NA	0.00	0.00	0.00	0.00	0.0	0.00
GT_N_INT64	NA	0.00	0.00	0.00	0.00	0.0	0.00
GT_SUPPORT	NA	0.00	0.00	0.00	0.00	0.0	0.00
GT_PLOIDY	NA	0.08	0.08	0.00	0.00	1.0	0.00
INFO:AC	Integer	16.44	40.74	0.00	0.00	2.5	22.16
INFO:AN	Integer	25.06	40.74	0.00	0.00	1.6	40.72
INFO:AF	Float	55.11	81.46	0.00	0.00	1.5	81.46
INFO:rf_tp_probability	Float	65.90	81.48	0.00	0.00	1.2	81.48
INFO:FS	Float	54.01	81.48	0.00	0.00	1.5	81.48
INFO:InbreedingCoeff	Float	40.86	81.48	0.00	0.00	2.0	81.48
INFO:MQ	Float	11.90	81.48	0.00	0.00	6.9	81.48
INFO:MQRankSum	Float	62.46	81.33	0.00	0.00	1.3	81.33
INFO:QD	Float	53.72	81.48	0.00	0.00	1.5	81.48
INFO:ReadPosRankSum	Float	62.05	81.32	0.00	0.00	1.3	81.32
INFO:SOR	Float	56.88	81.48	0.00	0.00	1.4	81.48
INFO:VQSR_POSITIVE_TRAIN_SITE	Flag	0.00	0.00	0.00	0.00	0.0	0.00
INFO:VQSR_NEGATIVE_TRAIN_SITE	Flag	0.00	0.00	0.00	0.00	0.0	0.00
INFO:BaseQRankSum	Float	65.07	81.33	0.00	0.00	1.2	81.33
INFO:ClippingRankSum	Float	61.56	81.33	0.00	0.00	1.3	81.33
INFO:DP	Integer	56.89	81.48	0.00	0.00	1.4	81.44
INFO:VQSLOD	Float	56.13	81.32	0.00	0.00	1.4	81.32
INFO:VQSR_culprit	String	4.98	45.20	2.59	20.30	9.1	45.20
INFO:segdup	Flag	0.00	0.00	0.00	0.00	0.0	0.00
INFO:lcr	Flag	0.00	0.00	0.00	0.00	0.0	0.00
INFO:decoy	Flag	0.00	0.00	0.00	0.00	0.0	0.00
INFO:nonpar	Flag	0.00	0.00	0.00	0.00	0.0	0.00
INFO:rf_positive_label	Flag	0.00	0.00	0.00	0.00	0.0	0.00
INFO:rf_negative_label	Flag	0.00	0.00	0.00	0.00	0.0	0.00
INFO:rf_label	String	0.93	6.48	0.00	0.00	7.0	6.48
INFO:rf_train	Flag	0.00	0.00	0.00	0.00	0.0	0.00
INFO:transmitted_singleton	Flag	0.00	0.00	0.00	0.00	0.0	0.00
INFO:variant_type	String	5.69	78.14	3.79	20.37	13.7	78.14
INFO:allele_type	String	4.42	61.11	0.00	0.01	13.8	61.11
INFO:n_alt_alleles	Integer	3.03	20.37	0.00	0.00	6.7	20.37
INFO:was_mixed	Flag	0.00	0.00	0.00	0.00	0.0	0.00
INFO:has_star	Flag	0.00	0.00	0.00	0.00	0.0	0.00
INFO:pab_max	Float	32.40	81.31	0.00	0.00	2.5	81.31
INFO:gq_hist_alt_bin_freq	String	75.27	820.22	5.58	20.37	10.9	820.22
INFO:gq_hist_all_bin_freq	String	618.38	1,497.74	10.18	20.37	2.4	1,497.74
INFO:dp_hist_alt_bin_freq	String	100.83	823.51	5.43	20.37	8.2	823.51
INFO:dp_hist_alt_n_larger	Integer	1.44	20.80	0.00	0.00	14.5	20.40
INFO:dp_hist_all_bin_freq	String	405.23	1,248.63	6.70	20.37	3.1	1,248.63
INFO:dp_hist_all_n_larger	Integer	1.90	21.00	0.00	0.00	11.1	20.64

INFO:ab_hist_alt_bin_freq	String	111.88	825.32	5.38	20.37	7.4	825.32
INFO:AC_nfe_seu	Integer	3.17	20.37	0.00	0.00	6.4	20.37
INFO:AN_nfe_seu	Integer	3.57	20.37	0.00	0.00	5.7	20.37
INFO:AF_nfe_seu	Float	5.88	80.43	0.00	0.00	13.7	80.43
INFO:nhomalt_nfe_seu	Integer	1.70	20.25	0.00	0.00	11.9	20.37
INFO:controls_AC_afr_male	Integer	9.22	40.71	0.00	0.00	4.4	21.03
INFO:controls_AN_afr_male	Integer	13.74	40.74	0.00	0.00	3.0	40.62
INFO:controls_AF_afr_male	Float	19.64	81.42	0.00	0.00	4.1	81.42
INFO:controls_nhomalt_afr_male	Integer	3.51	39.11	0.00	0.00	11.2	20.57
INFO:non_topmed_AC_amr	Integer	6.47	40.08	0.00	0.00	6.2	20.81
INFO:non_topmed_AN_amr	Integer	10.82	40.74	0.00	0.00	3.8	40.51
INFO:non_topmed_AF_amr	Float	13.08	81.41	0.00	0.00	6.2	81.41
INFO:non_topmed_nhomalt_amr	Integer	2.52	34.03	0.00	0.00	13.5	20.51
INFO:AC_raw	Integer	17.11	40.74	0.00	0.00	2.4	22.39
INFO:AN_raw	Integer	5.02	40.74	0.00	0.00	8.1	40.74
INFO:AF_raw	Float	32.03	81.48	0.00	0.00	2.5	81.48
INFO:nhomalt_raw	Integer	6.15	40.73	0.00	0.00	6.6	21.05
INFO:AC_fin_female	Integer	5.83	40.59	0.00	0.00	7.0	20.95
INFO:AN_fin_female	Integer	14.14	40.74	0.00	0.00	2.9	40.49
INFO:AF_fin_female	Float	11.59	81.37	0.00	0.00	7.0	81.37
INFO:nhomalt_fin_female	Integer	2.72	38.08	0.00	0.00	14.0	20.60
INFO:non_neuro_AC_asj_female	Integer	2.86	20.37	0.00	0.00	7.1	20.37
INFO:non_neuro_AN_asj_female	Integer	3.11	20.37	0.00	0.00	6.6	20.37
INFO:non_neuro_AF_asj_female	Float	5.12	81.21	0.00	0.00	15.9	81.21
INFO:non_neuro_nhomalt_asj_female	Integer	1.62	20.25	0.00	0.00	12.5	20.37
INFO:non_neuro_AC_afr_male	Integer	9.53	40.72	0.00	0.00	4.3	21.08
INFO:non_neuro_AN_afr_male	Integer	14.06	40.74	0.00	0.00	2.9	40.63
INFO:non_neuro_AF_afr_male	Float	20.42	81.42	0.00	0.00	4.0	81.42
INFO:non_neuro_nhomalt_afr_male	Integer	3.60	39.42	0.00	0.00	11.0	20.59
INFO:AC_afr_male	Integer	12.63	40.74	0.00	0.00	3.2	21.53
INFO:AN_afr_male	Integer	18.16	40.74	0.00	0.00	2.2	40.69
INFO:AF_afr_male	Float	29.83	81.44	0.00	0.00	2.7	81.44
INFO:nhomalt_afr_male	Integer	4.34	40.48	0.00	0.00	9.3	20.74
INFO:AC_afr	Integer	14.28	40.74	0.00	0.00	2.9	21.78
INFO:AN_afr	Integer	20.84	40.74	0.00	0.00	2.0	40.70
INFO:AF_afr	Float	35.95	81.45	0.00	0.00	2.3	81.45
INFO:nhomalt_afr	Integer	4.73	40.61	0.00	0.00	8.6	20.82
INFO:non_neuro_AC_afr_female	Integer	9.76	40.73	0.00	0.00	4.2	21.12
INFO:non_neuro_AN_afr_female	Integer	14.03	40.74	0.00	0.00	2.9	40.64
INFO:non_neuro_AF_afr_female	Float	20.90	81.42	0.00	0.00	3.9	81.42
INFO:non_neuro_nhomalt_afr_female	Integer	3.65	39.62	0.00	0.00	10.8	20.60
INFO:non_topmed_AC_amr_female	Integer	5.16	36.57	0.00	0.00	7.1	20.65
INFO:non_topmed_AN_amr_female	Integer	8.84	40.74	0.00	0.00	4.6	40.28

INFO:non_topmed_AF_amr_female	Float	10.18	81.38	0.00	0.00	8.0	81.38
INFO:non_topmed_nhomalt_amr_female	Integer	2.14	20.35	0.00	0.00	9.5	20.43
INFO:non_topmed_AC_oth_female	Integer	5.77	38.93	0.00	0.00	6.7	20.72
INFO:non_topmed_AN_oth_female	Integer	10.99	40.74	0.00	0.00	3.7	40.43
INFO:non_topmed_AF_oth_female	Float	11.80	81.39	0.00	0.00	6.9	81.39
INFO:non_topmed_nhomalt_oth_female	Integer	2.39	28.81	0.00	0.00	12.1	20.47
INFO:AC_eas_female	Integer	4.82	39.14	0.00	0.00	8.1	20.74
INFO:AN_eas_female	Integer	8.70	40.74	0.00	0.00	4.7	40.46
INFO:AF_eas_female	Float	9.00	81.33	0.00	0.00	9.0	81.33
INFO:nhomalt_eas_female	Integer	2.25	31.27	0.00	0.00	13.9	20.49
INFO:AC_afr_female	Integer	11.71	40.74	0.00	0.00	3.5	21.40
INFO:AN_afr_female	Integer	17.07	40.74	0.00	0.00	2.4	40.67
INFO:AF_afr_female	Float	26.78	81.43	0.00	0.00	3.0	81.43
INFO:nhomalt_afr_female	Integer	4.12	40.36	0.00	0.00	9.8	20.69
INFO:non_neuro_AC_female	Integer	13.71	40.74	0.00	0.00	3.0	21.45
INFO:non_neuro_AN_female	Integer	21.26	40.74	0.00	0.00	1.9	40.70
INFO:non_neuro_AF_female	Float	36.59	81.45	0.00	0.00	2.2	81.45
INFO:non_neuro_nhomalt_female	Integer	4.53	40.57	0.00	0.00	9.0	20.82
INFO:controls_AC_afr	Integer	10.74	40.74	0.00	0.00	3.8	21.25
INFO:controls_AN_afr	Integer	15.95	40.74	0.00	0.00	2.6	40.66
INFO:controls_AF_afr	Float	23.92	81.43	0.00	0.00	3.4	81.43
INFO:controls_nhomalt_afr	Integer	3.89	40.11	0.00	0.00	10.3	20.65
INFO:AC_nfe_onf	Integer	7.67	40.66	0.00	0.00	5.3	21.00
INFO:AN_nfe_onf	Integer	13.29	40.74	0.00	0.00	3.1	40.65
INFO:AF_nfe_onf	Float	16.22	81.42	0.00	0.00	5.0	81.42
INFO:nhomalt_nfe_onf	Integer	2.87	38.61	0.00	0.00	13.4	20.63
INFO:controls_AC_fin_male	Integer	4.59	39.17	0.00	0.00	8.5	20.73
INFO:controls_AN_fin_male	Integer	9.83	40.74	0.00	0.00	4.1	40.16
INFO:controls_AF_fin_male	Float	8.72	81.25	0.00	0.00	9.3	81.25
INFO:controls_nhomalt_fin_male	Integer	2.26	30.74	0.00	0.00	13.6	20.48
INFO:non_neuro_AC_nfe_nwe	Integer	10.51	40.74	0.00	0.00	3.9	21.24
INFO:non_neuro_AN_nfe_nwe	Integer	17.89	40.74	0.00	0.00	2.3	40.69
INFO:non_neuro_AF_nfe_nwe	Float	24.80	81.44	0.00	0.00	3.3	81.44
INFO:non_neuro_nhomalt_nfe_nwe	Integer	3.35	40.23	0.00	0.00	12.0	20.77
INFO:AC_fin_male	Integer	5.69	40.55	0.00	0.00	7.1	20.92
INFO:AN_fin_male	Integer	14.05	40.74	0.00	0.00	2.9	40.48
INFO:AF_fin_male	Float	11.31	81.37	0.00	0.00	7.2	81.37
INFO:nhomalt_fin_male	Integer	2.68	37.64	0.00	0.00	14.1	20.59
INFO:AC_nfe_female	Integer	10.13	40.74	0.00	0.00	4.0	21.21
INFO:AN_nfe_female	Integer	19.72	40.74	0.00	0.00	2.1	40.68
INFO:AF_nfe_female	Float	24.57	81.43	0.00	0.00	3.3	81.43
INFO:nhomalt_nfe_female	Integer	3.34	40.14	0.00	0.00	12.0	20.76
INFO:AC_amr	Integer	6.58	40.15	0.00	0.00	6.1	20.82

INFO:AN_amr	Integer	10.91	40.74	0.00	0.00	3.7	40.52
INFO:AF_amr	Float	13.37	81.41	0.00	0.00	6.1	81.41
INFO:nhomalt_amr	Integer	2.54	34.41	0.00	0.00	13.5	20.52
INFO:non_topmed_AC_nfe_male	Integer	9.47	40.74	0.00	0.00	4.3	21.17
INFO:non_topmed_AN_nfe_male	Integer	19.05	40.74	0.00	0.00	2.1	40.68
INFO:non_topmed_AF_nfe_male	Float	22.33	81.43	0.00	0.00	3.6	81.43
INFO:non_topmed_nhomalt_nfe_male	Integer	3.25	39.97	0.00	0.00	12.3	20.73
INFO:AC_eas	Integer	6.51	40.58	0.00	0.00	6.2	20.93
INFO:AN_eas	Integer	12.34	40.74	0.00	0.00	3.3	40.60
INFO:AF_eas	Float	13.03	81.37	0.00	0.00	6.2	81.37
INFO:nhomalt_eas	Integer	2.65	38.13	0.00	0.00	14.4	20.61
INFO:nhomalt	Integer	5.68	40.71	0.00	0.00	7.2	21.00
INFO:non_neuro_AC_nfe_female	Integer	9.76	40.74	0.00	0.00	4.2	21.19
INFO:non_neuro_AN_nfe_female	Integer	19.56	40.74	0.00	0.00	2.1	40.68
INFO:non_neuro_AF_nfe_female	Float	23.48	81.43	0.00	0.00	3.5	81.43
INFO:non_neuro_nhomalt_nfe_female	Integer	3.28	40.06	0.00	0.00	12.2	20.74
INFO:non_neuro_AC_afr	Integer	11.49	40.74	0.00	0.00	3.5	21.37
INFO:non_neuro_AN_afr	Integer	16.80	40.74	0.00	0.00	2.4	40.67
INFO:non_neuro_AF_afr	Float	26.05	81.43	0.00	0.00	3.1	81.43
INFO:non_neuro_nhomalt_afr	Integer	4.08	40.33	0.00	0.00	9.9	20.68
INFO:controls_AC_raw	Integer	14.98	40.74	0.00	0.00	2.7	21.70
INFO:controls_AN_raw	Integer	4.33	40.73	0.00	0.00	9.4	40.74
INFO:controls_AF_raw	Float	26.80	81.48	0.00	0.00	3.0	81.48
INFO:controls_nhomalt_raw	Integer	5.09	40.67	0.00	0.00	8.0	20.88
INFO:controls_AC_male	Integer	12.39	40.74	0.00	0.00	3.3	21.31
INFO:controls_AN_male	Integer	19.82	40.74	0.00	0.00	2.1	40.69
INFO:controls_AF_male	Float	30.81	81.44	0.00	0.00	2.6	81.44
INFO:controls_nhomalt_male	Integer	4.27	40.40	0.00	0.00	9.5	20.77
INFO:non_topmed_AC_male	Integer	15.37	40.74	0.00	0.00	2.7	21.78
INFO:non_topmed_AN_male	Integer	22.46	40.74	0.00	0.00	1.8	40.71
INFO:non_topmed_AF_male	Float	43.21	81.45	0.00	0.00	1.9	81.45
INFO:non_topmed_nhomalt_male	Integer	5.07	40.66	0.00	0.00	8.0	20.89
INFO:controls_AC_nfe_female	Integer	7.27	40.68	0.00	0.00	5.6	21.02
INFO:controls_AN_nfe_female	Integer	17.93	40.74	0.00	0.00	2.3	40.63
INFO:controls_AF_nfe_female	Float	15.95	81.41	0.00	0.00	5.1	81.41
INFO:controls_nhomalt_nfe_female	Integer	2.93	38.95	0.00	0.00	13.3	20.65
INFO:non_neuro_AC_amr	Integer	5.83	39.10	0.00	0.00	6.7	20.73
INFO:non_neuro_AN_amr	Integer	9.58	40.74	0.00	0.00	4.3	40.45
INFO:non_neuro_AF_amr	Float	11.53	81.39	0.00	0.00	7.1	81.39
INFO:non_neuro_nhomalt_amr	Integer	2.37	29.55	0.00	0.00	12.5	20.47
INFO:non_neuro_AC_eas_female	Integer	4.82	39.14	0.00	0.00	8.1	20.74
INFO:non_neuro_AN_eas_female	Integer	8.70	40.74	0.00	0.00	4.7	40.46
INFO:non_neuro_AF_eas_female	Float	9.00	81.33	0.00	0.00	9.0	81.33

INFO:non_neuro_nhomalt_eas_female	Integer	2.25	31.27	0.00	0.00	13.9	20.49
INFO:AC_asj_male	Integer	3.58	20.37	0.00	0.00	5.7	20.50
INFO:AN_asj_male	Integer	4.63	20.37	0.00	0.00	4.4	40.05
INFO:AF_asj_male	Float	7.08	81.33	0.00	0.00	11.5	81.33
INFO:nhomalt_asj_male	Integer	1.94	20.34	0.00	0.00	10.5	20.37
INFO:controls_AC_nfe_male	Integer	7.60	40.70	0.00	0.00	5.4	21.05
INFO:controls_AN_nfe_male	Integer	17.91	40.74	0.00	0.00	2.3	40.65
INFO:controls_AF_nfe_male	Float	16.80	81.41	0.00	0.00	4.8	81.41
INFO:controls_nhomalt_nfe_male	Integer	2.98	39.19	0.00	0.00	13.1	20.66
INFO:non_neuro_AC_fin	Integer	5.26	40.37	0.00	0.00	7.7	20.86
INFO:non_neuro_AN_fin	Integer	11.66	40.74	0.00	0.00	3.5	40.36
INFO:non_neuro_AF_fin	Float	10.19	81.32	0.00	0.00	8.0	81.32
INFO:non_neuro_nhomalt_fin	Integer	2.53	36.18	0.00	0.00	14.3	20.55
INFO:AC_oth_female	Integer	5.82	39.04	0.00	0.00	6.7	20.73
INFO:AN_oth_female	Integer	11.07	40.74	0.00	0.00	3.7	40.44
INFO:AF_oth_female	Float	11.88	81.39	0.00	0.00	6.9	81.39
INFO:nhomalt_oth_female	Integer	2.41	29.45	0.00	0.00	12.2	20.47
INFO:controls_AC_nfe	Integer	8.99	40.74	0.00	0.00	4.5	21.17
INFO:controls_AN_nfe	Integer	20.60	40.74	0.00	0.00	2.0	40.67
INFO:controls_AF_nfe	Float	21.05	81.42	0.00	0.00	3.9	81.42
INFO:controls_nhomalt_nfe	Integer	3.24	39.95	0.00	0.00	12.3	20.73
INFO:controls_AC_oth_female	Integer	3.69	20.37	0.00	0.00	5.5	20.51
INFO:controls_AN_oth_female	Integer	7.46	20.37	0.00	0.00	2.7	39.70
INFO:controls_AF_oth_female	Float	7.69	81.31	0.00	0.00	10.6	81.31
INFO:controls_nhomalt_oth_female	Integer	1.95	20.34	0.00	0.00	10.4	20.37
INFO:controls_AC_asj	Integer	2.59	20.37	0.00	0.00	7.9	20.37
INFO:controls_AN_asj	Integer	2.91	20.37	0.00	0.00	7.0	20.37
INFO:controls_AF_asj	Float	4.38	80.75	0.00	0.00	18.4	80.75
INFO:controls_nhomalt_asj	Integer	1.45	20.05	0.00	0.00	13.8	20.37
INFO:non_neuro_AC_amr_male	Integer	4.21	20.37	0.00	0.00	4.8	20.53
INFO:non_neuro_AN_amr_male	Integer	5.62	20.37	0.00	0.00	3.6	39.95
INFO:non_neuro_AF_amr_male	Float	8.77	81.36	0.00	0.00	9.3	81.36
INFO:non_neuro_nhomalt_amr_male	Integer	2.03	20.35	0.00	0.00	10.0	20.37
INFO:controls_AC_nfe_nwe	Integer	5.80	40.13	0.00	0.00	6.9	20.82
INFO:controls_AN_nfe_nwe	Integer	11.18	40.74	0.00	0.00	3.6	40.45
INFO:controls_AF_nfe_nwe	Float	11.63	81.38	0.00	0.00	7.0	81.38
INFO:controls_nhomalt_nfe_nwe	Integer	2.46	34.65	0.00	0.00	14.1	20.53
INFO:AC_nfe_nwe	Integer	10.84	40.74	0.00	0.00	3.8	21.26
INFO:AN_nfe_nwe	Integer	18.04	40.74	0.00	0.00	2.3	40.69
INFO:AF_nfe_nwe	Float	26.04	81.44	0.00	0.00	3.1	81.44
INFO:nhomalt_nfe_nwe	Integer	3.40	40.28	0.00	0.00	11.8	20.78
INFO:controls_AC_nfe_seu	Integer	2.81	20.36	0.00	0.00	7.2	20.37
INFO:controls_AN_nfe_seu	Integer	2.96	20.37	0.00	0.00	6.9	20.37

INFO:controls_AF_nfe_seu	Float	4.86	80.22	0.00	0.00	16.5	80.22
INFO:controls_nhomalt_nfe_seu	Integer	1.53	20.16	0.00	0.00	13.1	20.37
INFO:non_neuro_AC_amr_female	Integer	4.76	33.30	0.00	0.00	7.0	20.59
INFO:non_neuro_AN_amr_female	Integer	8.23	40.73	0.00	0.00	4.9	40.22
INFO:non_neuro_AF_amr_female	Float	9.39	81.36	0.00	0.00	8.7	81.36
INFO:non_neuro_nhomalt_amr_female	Integer	2.09	20.35	0.00	0.00	9.7	20.41
INFO:non_neuro_AC_nfe_onf	Integer	7.20	40.61	0.00	0.00	5.6	20.96
INFO:non_neuro_AN_nfe_onf	Integer	12.29	40.74	0.00	0.00	3.3	40.64
INFO:non_neuro_AF_nfe_onf	Float	14.85	81.41	0.00	0.00	5.5	81.41
INFO:non_neuro_nhomalt_nfe_onf	Integer	2.79	38.00	0.00	0.00	13.6	20.61
INFO:non_topmed_AC_eas_male	Integer	5.74	40.33	0.00	0.00	7.0	20.86
INFO:non_topmed_AN_eas_male	Integer	10.65	40.74	0.00	0.00	3.8	40.56
INFO:non_topmed_AF_eas_male	Float	11.09	81.36	0.00	0.00	7.3	81.36
INFO:non_topmed_nhomalt_eas_male	Integer	2.49	36.44	0.00	0.00	14.7	20.56
INFO:controls_AC_amr_female	Integer	3.41	20.37	0.00	0.00	6.0	20.37
INFO:controls_AN_amr_female	Integer	4.70	20.37	0.00	0.00	4.3	20.37
INFO:controls_AF_amr_female	Float	6.58	80.92	0.00	0.00	12.3	80.92
INFO:controls_nhomalt_amr_female	Integer	1.76	20.28	0.00	0.00	11.6	20.37
INFO:non_neuro_AC_fin_male	Integer	4.59	39.17	0.00	0.00	8.5	20.73
INFO:non_neuro_AN_fin_male	Integer	9.83	40.74	0.00	0.00	4.1	40.16
INFO:non_neuro_AF_fin_male	Float	8.72	81.25	0.00	0.00	9.3	81.25
INFO:non_neuro_nhomalt_fin_male	Integer	2.26	30.74	0.00	0.00	13.6	20.48
INFO:AC_female	Integer	15.00	40.74	0.00	0.00	2.7	21.70
INFO:AN_female	Integer	22.58	40.74	0.00	0.00	1.8	40.71
INFO:AF_female	Float	41.98	81.45	0.00	0.00	1.9	81.45
INFO:nhomalt_female	Integer	4.94	40.65	0.00	0.00	8.2	20.88
INFO:non_neuro_AC_oth_male	Integer	5.07	35.79	0.00	0.00	7.1	20.63
INFO:non_neuro_AN_oth_male	Integer	8.97	40.74	0.00	0.00	4.5	40.34
INFO:non_neuro_AF_oth_male	Float	10.06	81.38	0.00	0.00	8.1	81.38
INFO:non_neuro_nhomalt_oth_male	Integer	2.18	20.36	0.00	0.00	9.4	20.43
INFO:non_topmed_AC_nfe_est	Integer	7.80	40.73	0.00	0.00	5.2	21.13
INFO:non_topmed_AN_nfe_est	Integer	19.58	40.74	0.00	0.00	2.1	40.66
INFO:non_topmed_AF_nfe_est	Float	17.06	81.41	0.00	0.00	4.8	81.41
INFO:non_topmed_nhomalt_nfe_est	Integer	3.14	39.82	0.00	0.00	12.7	20.71
INFO:non_topmed_AC_nfe_nwe	Integer	9.27	40.73	0.00	0.00	4.4	21.15
INFO:non_topmed_AN_nfe_nwe	Integer	16.79	40.74	0.00	0.00	2.4	40.68
INFO:non_topmed_AF_nfe_nwe	Float	20.91	81.43	0.00	0.00	3.9	81.43
INFO:non_topmed_nhomalt_nfe_nwe	Integer	3.18	39.90	0.00	0.00	12.6	20.72
INFO:non_topmed_AC_amr_male	Integer	5.25	37.13	0.00	0.00	7.1	20.66
INFO:non_topmed_AN_amr_male	Integer	8.37	40.74	0.00	0.00	4.9	40.22
INFO:non_topmed_AF_amr_male	Float	10.21	81.38	0.00	0.00	8.0	81.38
INFO:non_topmed_nhomalt_amr_male	Integer	2.14	20.35	0.00	0.00	9.5	20.44
INFO:non_topmed_AC_nfe_onf	Integer	6.91	40.54	0.00	0.00	5.9	20.93

INFO:non_topmed_AN_nfe_onf	Integer	12.51	40.74	0.00	0.00	3.3	40.62
INFO:non_topmed_AF_nfe_onf	Float	14.22	81.41	0.00	0.00	5.7	81.41
INFO:non_topmed_nhomalt_nfe_onf	Integer	2.73	37.43	0.00	0.00	13.7	20.59
INFO:controls_AC_eas_male	Integer	4.92	39.36	0.00	0.00	8.0	20.75
INFO:controls_AN_eas_male	Integer	8.79	40.74	0.00	0.00	4.6	40.47
INFO:controls_AF_eas_male	Float	9.23	81.33	0.00	0.00	8.8	81.33
INFO:controls_nhomalt_eas_male	Integer	2.27	32.19	0.00	0.00	14.2	20.50
INFO:controls_AC_oth_male	Integer	3.79	20.37	0.00	0.00	5.4	20.48
INFO:controls_AN_oth_male	Integer	6.54	20.37	0.00	0.00	3.1	39.49
INFO:controls_AF_oth_male	Float	7.90	81.32	0.00	0.00	10.3	81.32
INFO:controls_nhomalt_oth_male	Integer	1.95	20.34	0.00	0.00	10.4	20.37
INFO:non_topmed_AC	Integer	16.71	40.74	0.00	0.00	2.4	22.11
INFO:non_topmed_AN	Integer	24.87	40.74	0.00	0.00	1.6	40.72
INFO:non_topmed_AF	Float	53.23	81.46	0.00	0.00	1.5	81.46
INFO:non_topmed_nhomalt	Integer	5.59	40.70	0.00	0.00	7.3	20.98
INFO:controls_AC_fin	Integer	5.25	40.37	0.00	0.00	7.7	20.86
INFO:controls_AN_fin	Integer	11.66	40.74	0.00	0.00	3.5	40.36
INFO:controls_AF_fin	Float	10.18	81.31	0.00	0.00	8.0	81.31
INFO:controls_nhomalt_fin	Integer	2.52	36.17	0.00	0.00	14.3	20.55
INFO:non_neuro_AC_nfe	Integer	12.19	40.74	0.00	0.00	3.3	21.36
INFO:non_neuro_AN_nfe	Integer	22.14	40.74	0.00	0.00	1.8	40.70
INFO:non_neuro_AF_nfe	Float	33.02	81.44	0.00	0.00	2.5	81.44
INFO:non_neuro_nhomalt_nfe	Integer	3.64	40.46	0.00	0.00	11.1	20.83
INFO:non_neuro_AC_fin_female	Integer	4.50	38.81	0.00	0.00	8.6	20.71
INFO:non_neuro_AN_fin_female	Integer	9.61	40.74	0.00	0.00	4.2	40.11
INFO:non_neuro_AF_fin_female	Float	8.50	81.27	0.00	0.00	9.6	81.27
INFO:non_neuro_nhomalt_fin_female	Integer	2.22	29.56	0.00	0.00	13.3	20.47
INFO:non_topmed_AC_nfe_seu	Integer	3.17	20.37	0.00	0.00	6.4	20.37
INFO:non_topmed_AN_nfe_seu	Integer	3.57	20.37	0.00	0.00	5.7	20.37
INFO:non_topmed_AF_nfe_seu	Float	5.88	80.43	0.00	0.00	13.7	80.43
INFO:non_topmed_nhomalt_nfe_seu	Integer	1.70	20.25	0.00	0.00	11.9	20.37
INFO:controls_AC_eas_female	Integer	4.20	35.88	0.00	0.00	8.5	20.64
INFO:controls_AN_eas_female	Integer	7.17	40.73	0.00	0.00	5.7	40.32
INFO:controls_AF_eas_female	Float	7.72	81.30	0.00	0.00	10.5	81.30
INFO:controls_nhomalt_eas_female	Integer	2.01	20.35	0.00	0.00	10.1	20.44
INFO:non_topmed_AC_asj	Integer	3.32	20.37	0.00	0.00	6.1	20.41
INFO:non_topmed_AN_asj	Integer	4.50	20.37	0.00	0.00	4.5	38.41
INFO:non_topmed_AF_asj	Float	6.45	81.26	0.00	0.00	12.6	81.26
INFO:non_topmed_nhomalt_asj	Integer	1.81	20.32	0.00	0.00	11.2	20.37
INFO:controls_AC_nfe_onf	Integer	4.83	37.20	0.00	0.00	7.7	20.66
INFO:controls_AN_nfe_onf	Integer	9.22	40.74	0.00	0.00	4.4	40.22
INFO:controls_AF_nfe_onf	Float	9.39	81.34	0.00	0.00	8.7	81.34
INFO:controls_nhomalt_nfe_onf	Integer	2.09	20.35	0.00	0.00	9.8	20.44

INFO:non_neuro_AC	Integer	15.87	40.74	0.00	0.00	2.6	21.78
INFO:non_neuro_AN	Integer	24.51	40.74	0.00	0.00	1.7	40.71
INFO:non_neuro_AF	Float	48.37	81.45	0.00	0.00	1.7	81.45
INFO:non_neuro_nhomalt	Integer	5.13	40.67	0.00	0.00	7.9	20.93
INFO:non_topmed_AC_nfe	Integer	11.47	40.74	0.00	0.00	3.6	21.32
INFO:non_topmed_AN_nfe	Integer	22.00	40.74	0.00	0.00	1.9	40.69
INFO:non_topmed_AF_nfe	Float	30.11	81.44	0.00	0.00	2.7	81.44
INFO:non_topmed_nhomalt_nfe	Integer	3.56	40.39	0.00	0.00	11.4	20.81
INFO:non_topmed_AC_raw	Integer	17.45	40.74	0.00	0.00	2.3	22.33
INFO:non_topmed_AN_raw	Integer	4.95	40.74	0.00	0.00	8.2	40.74
INFO:non_topmed_AF_raw	Float	32.39	81.48	0.00	0.00	2.5	81.48
INFO:non_topmed_nhomalt_raw	Integer	6.06	40.73	0.00	0.00	6.7	21.03
INFO:non_neuro_AC_nfe_est	Integer	7.61	40.73	0.00	0.00	5.4	21.11
INFO:non_neuro_AN_nfe_est	Integer	19.71	40.74	0.00	0.00	2.1	40.65
INFO:non_neuro_AF_nfe_est	Float	16.61	81.40	0.00	0.00	4.9	81.40
INFO:non_neuro_nhomalt_nfe_est	Integer	3.10	39.72	0.00	0.00	12.8	20.70
INFO:non_topmed_AC_oth_male	Integer	5.56	37.90	0.00	0.00	6.8	20.68
INFO:non_topmed_AN_oth_male	Integer	9.72	40.74	0.00	0.00	4.2	40.38
INFO:non_topmed_AF_oth_male	Float	11.15	81.38	0.00	0.00	7.3	81.38
INFO:non_topmed_nhomalt_oth_male	Integer	2.25	20.36	0.00	0.00	9.1	20.45
INFO:AC_nfe_est	Integer	7.81	40.73	0.00	0.00	5.2	21.13
INFO:AN_nfe_est	Integer	19.55	40.74	0.00	0.00	2.1	40.66
INFO:AF_nfe_est	Float	17.09	81.41	0.00	0.00	4.8	81.41
INFO:nhomalt_nfe_est	Integer	3.14	39.82	0.00	0.00	12.7	20.71
INFO:non_topmed_AC_afr_male	Integer	12.57	40.74	0.00	0.00	3.2	21.53
INFO:non_topmed_AN_afr_male	Integer	18.26	40.74	0.00	0.00	2.2	40.68
INFO:non_topmed_AF_afr_male	Float	29.68	81.44	0.00	0.00	2.7	81.44
INFO:non_topmed_nhomalt_afr_male	Integer	4.33	40.48	0.00	0.00	9.4	20.73
INFO:AC_eas_male	Integer	5.78	40.34	0.00	0.00	7.0	20.86
INFO:AN_eas_male	Integer	10.68	40.74	0.00	0.00	3.8	40.56
INFO:AF_eas_male	Float	11.23	81.36	0.00	0.00	7.2	81.36
INFO:nhomalt_eas_male	Integer	2.49	36.51	0.00	0.00	14.6	20.56
INFO:controls_AC_eas	Integer	5.57	40.23	0.00	0.00	7.2	20.85
INFO:controls_AN_eas	Integer	10.15	40.74	0.00	0.00	4.0	40.55
INFO:controls_AF_eas	Float	10.73	81.35	0.00	0.00	7.6	81.35
INFO:controls_nhomalt_eas	Integer	2.45	35.99	0.00	0.00	14.7	20.55
INFO:non_neuro_AC_eas_male	Integer	5.78	40.34	0.00	0.00	7.0	20.86
INFO:non_neuro_AN_eas_male	Integer	10.68	40.74	0.00	0.00	3.8	40.56
INFO:non_neuro_AF_eas_male	Float	11.23	81.36	0.00	0.00	7.2	81.36
INFO:non_neuro_nhomalt_eas_male	Integer	2.49	36.51	0.00	0.00	14.6	20.56
INFO:non_neuro_AC_asj_male	Integer	3.47	20.37	0.00	0.00	5.9	20.47
INFO:non_neuro_AN_asj_male	Integer	4.10	20.37	0.00	0.00	5.0	39.96
INFO:non_neuro_AF_asj_male	Float	6.77	81.32	0.00	0.00	12.0	81.32

INFO:non_neuro_nhomalt_asj_male	Integer	1.91	20.34	0.00	0.00	10.6	20.37
INFO:controls_AC_oth	Integer	4.88	37.05	0.00	0.00	7.6	20.66
INFO:controls_AN_oth	Integer	10.81	40.74	0.00	0.00	3.8	40.26
INFO:controls_AF_oth	Float	9.77	81.36	0.00	0.00	8.3	81.36
INFO:controls_nhomalt_oth	Integer	2.14	20.36	0.00	0.00	9.5	20.44
INFO:AC_nfe	Integer	12.60	40.74	0.00	0.00	3.2	21.39
INFO:AN_nfe	Integer	22.30	40.74	0.00	0.00	1.8	40.70
INFO:AF_nfe	Float	34.87	81.44	0.00	0.00	2.3	81.44
INFO:nhomalt_nfe	Integer	3.70	40.50	0.00	0.00	10.9	20.84
INFO:non_topmed_AC_female	Integer	14.76	40.74	0.00	0.00	2.8	21.67
INFO:non_topmed_AN_female	Integer	22.45	40.74	0.00	0.00	1.8	40.71
INFO:non_topmed_AF_female	Float	40.61	81.45	0.00	0.00	2.0	81.45
INFO:non_topmed_nhomalt_female	Integer	4.89	40.64	0.00	0.00	8.3	20.87
INFO:non_neuro_AC_asj	Integer	3.69	20.37	0.00	0.00	5.5	20.54
INFO:non_neuro_AN_asj	Integer	4.66	20.37	0.00	0.00	4.4	40.20
INFO:non_neuro_AF_asj	Float	7.34	81.34	0.00	0.00	11.1	81.34
INFO:non_neuro_nhomalt_asj	Integer	1.99	20.35	0.00	0.00	10.2	20.37
INFO:non_topmed_AC_eas_female	Integer	4.78	38.98	0.00	0.00	8.2	20.73
INFO:non_topmed_AN_eas_female	Integer	8.61	40.74	0.00	0.00	4.7	40.45
INFO:non_topmed_AF_eas_female	Float	8.91	81.33	0.00	0.00	9.1	81.33
INFO:non_topmed_nhomalt_eas_female	Integer	2.23	30.38	0.00	0.00	13.6	20.49
INFO:non_neuro_AC_raw	Integer	16.61	40.74	0.00	0.00	2.5	21.96
INFO:non_neuro_AN_raw	Integer	4.65	40.73	0.00	0.00	8.8	40.74
INFO:non_neuro_AF_raw	Float	29.48	81.48	0.00	0.00	2.8	81.48
INFO:non_neuro_nhomalt_raw	Integer	5.54	40.71	0.00	0.00	7.4	20.96
INFO:non_topmed_AC_eas	Integer	6.46	40.57	0.00	0.00	6.3	20.93
INFO:non_topmed_AN_eas	Integer	12.13	40.74	0.00	0.00	3.4	40.60
INFO:non_topmed_AF_eas	Float	12.90	81.37	0.00	0.00	6.3	81.37
INFO:non_topmed_nhomalt_eas	Integer	2.64	38.06	0.00	0.00	14.4	20.61
INFO:non_topmed_AC_fin_male	Integer	5.69	40.55	0.00	0.00	7.1	20.92
INFO:non_topmed_AN_fin_male	Integer	14.05	40.74	0.00	0.00	2.9	40.48
INFO:non_topmed_AF_fin_male	Float	11.31	81.37	0.00	0.00	7.2	81.37
INFO:non_topmed_nhomalt_fin_male	Integer	2.68	37.64	0.00	0.00	14.1	20.59
INFO:AC_fin	Integer	6.68	40.72	0.00	0.00	6.1	21.06
INFO:AN_fin	Integer	16.56	40.74	0.00	0.00	2.5	40.58
INFO:AF_fin	Float	13.70	81.40	0.00	0.00	5.9	81.40
INFO:nhomalt_fin	Integer	2.97	39.43	0.00	0.00	13.3	20.67
INFO:AC_nfe_male	Integer	10.88	40.74	0.00	0.00	3.7	21.26
INFO:AN_nfe_male	Integer	19.59	40.74	0.00	0.00	2.1	40.69
INFO:AF_nfe_male	Float	27.19	81.44	0.00	0.00	3.0	81.44
INFO:nhomalt_nfe_male	Integer	3.44	40.29	0.00	0.00	11.7	20.78
INFO:controls_AC_amr_male	Integer	3.51	20.37	0.00	0.00	5.8	20.37
INFO:controls_AN_amr_male	Integer	4.63	20.37	0.00	0.00	4.4	20.37

INFO:controls_AF_amr_male	Float	6.85	80.86	0.00	0.00	11.8	80.86
INFO:controls_nhomalt_amr_male	Integer	1.78	20.31	0.00	0.00	11.4	20.37
INFO:controls_AC_afr_female	Integer	8.76	40.68	0.00	0.00	4.6	20.96
INFO:controls_AN_afr_female	Integer	13.05	40.74	0.00	0.00	3.1	40.60
INFO:controls_AF_afr_female	Float	18.37	81.41	0.00	0.00	4.4	81.41
INFO:controls_nhomalt_afr_female	Integer	3.37	38.41	0.00	0.00	11.4	20.55
INFO:controls_AC_amr	Integer	4.09	20.37	0.00	0.00	5.0	20.53
INFO:controls_AN_amr	Integer	5.93	20.37	0.00	0.00	3.4	39.63
INFO:controls_AF_amr	Float	8.48	81.03	0.00	0.00	9.6	81.03
INFO:controls_nhomalt_amr	Integer	1.94	20.34	0.00	0.00	10.5	20.37
INFO:AC_asj_female	Integer	3.01	20.37	0.00	0.00	6.8	20.37
INFO:AN_asj_female	Integer	3.50	20.37	0.00	0.00	5.8	20.37
INFO:AF_asj_female	Float	5.45	81.23	0.00	0.00	14.9	81.23
INFO:nhomalt_asj_female	Integer	1.68	20.29	0.00	0.00	12.0	20.37
INFO:non_neuro_AC_eas	Integer	6.51	40.58	0.00	0.00	6.2	20.93
INFO:non_neuro_AN_eas	Integer	12.34	40.74	0.00	0.00	3.3	40.60
INFO:non_neuro_AF_eas	Float	13.03	81.37	0.00	0.00	6.2	81.37
INFO:non_neuro_nhomalt_eas	Integer	2.65	38.13	0.00	0.00	14.4	20.61
INFO:non_neuro_AC_male	Integer	14.21	40.74	0.00	0.00	2.9	21.48
INFO:non_neuro_AN_male	Integer	21.11	40.74	0.00	0.00	1.9	40.70
INFO:non_neuro_AF_male	Float	38.58	81.45	0.00	0.00	2.1	81.45
INFO:non_neuro_nhomalt_male	Integer	4.61	40.60	0.00	0.00	8.8	20.85
INFO:AC_asj	Integer	4.09	31.90	0.00	0.00	7.8	20.57
INFO:AN_asj	Integer	6.44	40.70	0.00	0.00	6.3	40.27
INFO:AF_asj	Float	7.71	81.34	0.00	0.00	10.5	81.34
INFO:nhomalt_asj	Integer	2.02	20.35	0.00	0.00	10.1	20.40
INFO:controls_AC_nfe_est	Integer	7.57	40.72	0.00	0.00	5.4	21.11
INFO:controls_AN_nfe_est	Integer	19.93	40.74	0.00	0.00	2.0	40.65
INFO:controls_AF_nfe_est	Float	16.50	81.40	0.00	0.00	4.9	81.40
INFO:controls_nhomalt_nfe_est	Integer	3.10	39.71	0.00	0.00	12.8	20.70
INFO:non_topmed_AC_asj_female	Integer	2.87	20.37	0.00	0.00	7.1	20.37
INFO:non_topmed_AN_asj_female	Integer	3.31	20.37	0.00	0.00	6.2	20.37
INFO:non_topmed_AF_asj_female	Float	5.18	81.15	0.00	0.00	15.7	81.15
INFO:non_topmed_nhomalt_asj_female	Integer	1.60	20.24	0.00	0.00	12.6	20.37
INFO:non_topmed_AC_oth	Integer	6.98	40.30	0.00	0.00	5.8	20.85
INFO:non_topmed_AN_oth	Integer	12.74	40.74	0.00	0.00	3.2	40.56
INFO:non_topmed_AF_oth	Float	14.72	81.41	0.00	0.00	5.5	81.41
INFO:non_topmed_nhomalt_oth	Integer	2.72	35.53	0.00	0.00	13.1	20.54
INFO:non_topmed_AC_fin_female	Integer	5.83	40.59	0.00	0.00	7.0	20.95
INFO:non_topmed_AN_fin_female	Integer	14.14	40.74	0.00	0.00	2.9	40.49
INFO:non_topmed_AF_fin_female	Float	11.59	81.37	0.00	0.00	7.0	81.37
INFO:non_topmed_nhomalt_fin_female	Integer	2.72	38.08	0.00	0.00	14.0	20.60
INFO:AC_oth	Integer	7.12	40.39	0.00	0.00	5.7	20.87

INFO:AN_oth	Integer	13.07	40.74	0.00	0.00	3.1	40.58
INFO:AF_oth	Float	14.97	81.41	0.00	0.00	5.4	81.41
INFO:nhomalt_oth	Integer	2.75	36.00	0.00	0.00	13.1	20.55
INFO:non_neuro_AC_nfe_male	Integer	10.50	40.74	0.00	0.00	3.9	21.24
INFO:non_neuro_AN_nfe_male	Integer	19.18	40.74	0.00	0.00	2.1	40.69
INFO:non_neuro_AF_nfe_male	Float	25.63	81.44	0.00	0.00	3.2	81.44
INFO:non_neuro_nhomalt_nfe_male	Integer	3.39	40.23	0.00	0.00	11.9	20.77
INFO:controls_AC_female	Integer	11.75	40.74	0.00	0.00	3.5	21.25
INFO:controls_AN_female	Integer	19.81	40.74	0.00	0.00	2.1	40.68
INFO:controls_AF_female	Float	28.79	81.44	0.00	0.00	2.8	81.44
INFO:controls_nhomalt_female	Integer	4.12	40.28	0.00	0.00	9.8	20.74
INFO:non_topmed_AC_fin	Integer	6.68	40.72	0.00	0.00	6.1	21.06
INFO:non_topmed_AN_fin	Integer	16.56	40.74	0.00	0.00	2.5	40.58
INFO:non_topmed_AF_fin	Float	13.70	81.40	0.00	0.00	5.9	81.40
INFO:non_topmed_nhomalt_fin	Integer	2.97	39.43	0.00	0.00	13.3	20.67
INFO:non_topmed_AC_nfe_female	Integer	9.52	40.74	0.00	0.00	4.3	21.17
INFO:non_topmed_AN_nfe_female	Integer	19.28	40.74	0.00	0.00	2.1	40.67
INFO:non_topmed_AF_nfe_female	Float	22.68	81.43	0.00	0.00	3.6	81.43
INFO:non_topmed_nhomalt_nfe_female	Integer	3.27	40.01	0.00	0.00	12.2	20.74
INFO:controls_AC_asj_male	Integer	2.11	20.37	0.00	0.00	9.7	20.37
INFO:controls_AN_asj_male	Integer	2.22	20.36	0.00	0.00	9.2	20.37
INFO:controls_AF_asj_male	Float	3.20	80.19	0.00	0.00	25.1	80.19
INFO:controls_nhomalt_asj_male	Integer	1.19	19.50	0.00	0.00	16.4	20.37
INFO:non_topmed_AC_asj_male	Integer	2.98	20.37	0.00	0.00	6.8	20.37
INFO:non_topmed_AN_asj_male	Integer	3.72	20.37	0.00	0.00	5.5	20.37
INFO:non_topmed_AF_asj_male	Float	5.50	81.21	0.00	0.00	14.8	81.21
INFO:non_topmed_nhomalt_asj_male	Integer	1.66	20.25	0.00	0.00	12.2	20.37
INFO:non_neuro_AC_oth	Integer	6.27	39.87	0.00	0.00	6.4	20.79
INFO:non_neuro_AN_oth	Integer	11.73	40.74	0.00	0.00	3.5	40.52
INFO:non_neuro_AF_oth	Float	12.96	81.40	0.00	0.00	6.3	81.40
INFO:non_neuro_nhomalt_oth	Integer	2.55	32.98	0.00	0.00	12.9	20.50
INFO:AC_male	Integer	15.79	40.74	0.00	0.00	2.6	21.84
INFO:AN_male	Integer	22.74	40.74	0.00	0.00	1.8	40.71
INFO:AF_male	Float	45.82	81.45	0.00	0.00	1.8	81.45
INFO:nhomalt_male	Integer	5.18	40.67	0.00	0.00	7.9	20.92
INFO:controls_AC_fin_female	Integer	4.49	38.79	0.00	0.00	8.6	20.71
INFO:controls_AN_fin_female	Integer	9.61	40.74	0.00	0.00	4.2	40.11
INFO:controls_AF_fin_female	Float	8.48	81.25	0.00	0.00	9.6	81.25
INFO:controls_nhomalt_fin_female	Integer	2.21	29.48	0.00	0.00	13.3	20.47
INFO:controls_AC_asj_female	Integer	2.36	20.37	0.00	0.00	8.6	20.37
INFO:controls_AN_asj_female	Integer	2.50	20.37	0.00	0.00	8.2	20.37
INFO:controls_AF_asj_female	Float	3.68	80.47	0.00	0.00	21.9	80.47
INFO:controls_nhomalt_asj_female	Integer	1.32	19.88	0.00	0.00	15.0	20.37

INFO:AC_amr_male	Integer	5.34	37.46	0.00	0.00	7.0	20.67
INFO:AN_amr_male	Integer	8.41	40.74	0.00	0.00	4.8	40.26
INFO:AF_amr_male	Float	10.46	81.39	0.00	0.00	7.8	81.39
INFO:nhomalt_amr_male	Integer	2.16	20.35	0.00	0.00	9.4	20.44
INFO:AC_amr_female	Integer	5.25	37.06	0.00	0.00	7.1	20.66
INFO:AN_amr_female	Integer	8.97	40.74	0.00	0.00	4.5	40.32
INFO:AF_amr_female	Float	10.36	81.38	0.00	0.00	7.9	81.38
INFO:nhomalt_amr_female	Integer	2.16	20.35	0.00	0.00	9.4	20.44
INFO:AC_oth_male	Integer	5.74	38.70	0.00	0.00	6.7	20.71
INFO:AN_oth_male	Integer	10.03	40.74	0.00	0.00	4.1	40.44
INFO:AF_oth_male	Float	11.55	81.39	0.00	0.00	7.0	81.39
INFO:nhomalt_oth_male	Integer	2.37	27.65	0.00	0.00	11.6	20.46
INFO:non_neuro_AC_nfe_seu	Integer	2.79	20.36	0.00	0.00	7.3	20.37
INFO:non_neuro_AN_nfe_seu	Integer	2.92	20.37	0.00	0.00	7.0	20.37
INFO:non_neuro_AF_nfe_seu	Float	4.86	80.21	0.00	0.00	16.5	80.21
INFO:non_neuro_nhomalt_nfe_seu	Integer	1.52	20.15	0.00	0.00	13.2	20.37
INFO:non_topmed_AC_afr_female	Integer	11.66	40.74	0.00	0.00	3.5	21.39
INFO:non_topmed_AN_afr_female	Integer	17.04	40.74	0.00	0.00	2.4	40.67
INFO:non_topmed_AF_afr_female	Float	26.61	81.43	0.00	0.00	3.1	81.43
INFO:non_topmed_nhomalt_afr_female	Integer	4.11	40.36	0.00	0.00	9.8	20.69
INFO:non_topmed_AC_afr	Integer	14.23	40.74	0.00	0.00	2.9	21.77
INFO:non_topmed_AN_afr	Integer	20.34	40.74	0.00	0.00	2.0	40.70
INFO:non_topmed_AF_afr	Float	35.67	81.45	0.00	0.00	2.3	81.45
INFO:non_topmed_nhomalt_afr	Integer	4.71	40.61	0.00	0.00	8.6	20.82
INFO:controls_AC	Integer	14.18	40.74	0.00	0.00	2.9	21.55
INFO:controls_AN	Integer	22.59	40.74	0.00	0.00	1.8	40.70
INFO:controls_AF	Float	38.24	81.45	0.00	0.00	2.1	81.45
INFO:controls_nhomalt	Integer	4.72	40.61	0.00	0.00	8.6	20.85
INFO:non_neuro_AC_oth_female	Integer	5.01	35.57	0.00	0.00	7.1	20.63
INFO:non_neuro_AN_oth_female	Integer	9.85	40.74	0.00	0.00	4.1	40.30
INFO:non_neuro_AF_oth_female	Float	10.09	81.37	0.00	0.00	8.1	81.37
INFO:non_neuro_nhomalt_oth_female	Integer	2.15	20.36	0.00	0.00	9.5	20.43
INFO:non_topmed_faf95_amr	Float	12.52	81.48	0.00	0.00	6.5	81.48
INFO:non_topmed_faf99_amr	Float	12.52	81.48	0.00	0.00	6.5	81.48
INFO:faf95_afr	Float	26.28	81.48	0.00	0.00	3.1	81.48
INFO:faf99_afr	Float	25.57	81.48	0.00	0.00	3.2	81.48
INFO:controls_faf95_afr	Float	21.69	81.48	0.00	0.00	3.8	81.48
INFO:controls_faf99_afr	Float	21.70	81.48	0.00	0.00	3.8	81.48
INFO:faf95_amr	Float	9.78	81.48	0.00	0.00	8.3	81.48
INFO:faf99_amr	Float	9.78	81.48	0.00	0.00	8.3	81.48
INFO:faf95_eas	Float	9.38	81.48	0.00	0.00	8.7	81.48
INFO:faf99_eas	Float	9.37	81.48	0.00	0.00	8.7	81.48
INFO:faf95	Float	36.17	81.48	0.00	0.00	2.3	81.48

INFO:faf99	Float	34.87	81.48	0.00	0.00	2.3	81.48
INFO:non_neuro_faf95_afr	Float	22.92	81.48	0.00	0.00	3.6	81.48
INFO:non_neuro_faf99_afr	Float	22.86	81.48	0.00	0.00	3.6	81.48
INFO:non_neuro_faf95_amr	Float	11.33	81.48	0.00	0.00	7.2	81.48
INFO:non_neuro_faf99_amr	Float	11.31	81.48	0.00	0.00	7.2	81.48
INFO:controls_faf95_nfe	Float	17.60	81.48	0.00	0.00	4.6	81.48
INFO:controls_faf99_nfe	Float	17.53	81.48	0.00	0.00	4.6	81.48
INFO:non_topmed_faf95	Float	37.40	81.48	0.00	0.00	2.2	81.48
INFO:non_topmed_faf99	Float	36.04	81.48	0.00	0.00	2.3	81.48
INFO:non_neuro_faf95_nfe	Float	24.38	81.48	0.00	0.00	3.3	81.48
INFO:non_neuro_faf99_nfe	Float	23.51	81.48	0.00	0.00	3.5	81.48
INFO:non_neuro_faf95	Float	34.63	81.48	0.00	0.00	2.4	81.48
INFO:non_neuro_faf99	Float	33.42	81.48	0.00	0.00	2.4	81.48
INFO:non_topmed_faf95_nfe	Float	22.68	81.48	0.00	0.00	3.6	81.48
INFO:non_topmed_faf99_nfe	Float	22.08	81.48	0.00	0.00	3.7	81.48
INFO:controls_faf95_eas	Float	10.45	81.48	0.00	0.00	7.8	81.48
INFO:controls_faf99_eas	Float	10.45	81.48	0.00	0.00	7.8	81.48
INFO:faf95_nfe	Float	21.05	81.48	0.00	0.00	3.9	81.48
INFO:faf99_nfe	Float	20.30	81.48	0.00	0.00	4.0	81.48
INFO:non_topmed_faf95_eas	Float	11.97	81.48	0.00	0.00	6.8	81.48
INFO:non_topmed_faf99_eas	Float	11.98	81.48	0.00	0.00	6.8	81.48
INFO:controls_faf95_amr	Float	8.60	81.48	0.00	0.00	9.5	81.48
INFO:controls_faf99_amr	Float	8.59	81.48	0.00	0.00	9.5	81.48
INFO:non_neuro_faf95_eas	Float	12.18	81.48	0.00	0.00	6.7	81.48
INFO:non_neuro_faf99_eas	Float	12.16	81.48	0.00	0.00	6.7	81.48
INFO:non_topmed_faf95_afr	Float	30.32	81.48	0.00	0.00	2.7	81.48
INFO:non_topmed_faf99_afr	Float	29.70	81.48	0.00	0.00	2.7	81.48
INFO:controls_faf95	Float	30.37	81.48	0.00	0.00	2.7	81.48
INFO:controls_faf99	Float	29.61	81.48	0.00	0.00	2.8	81.48
INFO:controls_popmax	String	5.77	30.71	0.00	0.00	5.3	30.71
INFO:controls_AC_popmax	Integer	9.21	20.47	0.00	0.00	2.2	11.19
INFO:controls_AN_popmax	Integer	13.29	20.47	0.00	0.00	1.5	20.32
INFO:controls_AF_popmax	Float	29.13	40.94	0.00	0.00	1.4	40.94
INFO:controls_nhomalt_popmax	Integer	3.92	20.18	0.00	0.00	5.1	10.57
INFO:popmax	String	8.90	56.20	0.00	0.00	6.3	56.20
INFO:AC_popmax	Integer	14.18	37.47	0.00	0.00	2.6	20.29
INFO:AN_popmax	Integer	25.24	37.47	0.00	0.00	1.5	37.43
INFO:AF_popmax	Float	50.81	74.94	0.00	0.00	1.5	74.94
INFO:nhomalt_popmax	Integer	5.14	37.37	0.00	0.00	7.3	19.22
INFO:age_hist_het_bin_freq	String	59.17	404.73	3.67	20.37	6.8	404.73
INFO:age_hist_het_n_smaller	Integer	12.28	40.74	0.00	0.00	3.3	21.41
INFO:age_hist_het_n_larger	Integer	3.76	20.37	0.00	0.00	5.4	20.37
INFO:age_hist_hom_bin_freq	String	16.83	395.01	2.07	20.35	23.5	395.01

INFO:age_hist_hom_n_smaller	Integer	4.36	40.47	0.00	0.00	9.3	20.75
INFO:age_hist_hom_n_larger	Integer	2.08	20.35	0.00	0.00	9.8	20.37
INFO:non_neuro_popmax	String	7.71	45.93	0.00	0.00	6.0	45.93
INFO:non_neuro_AC_popmax	Integer	11.84	30.62	0.00	0.00	2.6	16.51
INFO:non_neuro_AN_popmax	Integer	19.61	30.62	0.00	0.00	1.6	30.57
INFO:non_neuro_AF_popmax	Float	41.23	61.24	0.00	0.00	1.5	61.24
INFO:non_neuro_nhomalt_popmax	Integer	4.54	30.46	0.00	0.00	6.7	15.73
INFO:non_topmed_popmax	String	8.38	50.98	0.00	0.00	6.1	50.98
INFO:non_topmed_AC_popmax	Integer	13.38	33.99	0.00	0.00	2.5	18.50
INFO:non_topmed_AN_popmax	Integer	22.74	33.99	0.00	0.00	1.5	33.94
INFO:non_topmed_AF_popmax	Float	46.70	67.97	0.00	0.00	1.5	67.97
INFO:non_topmed_nhomalt_popmax	Integer	5.01	33.88	0.00	0.00	6.8	17.47
INFO:vep	String	541.64	17,580.78	15.89	38.92	32.5	17,580.78